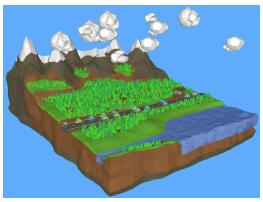
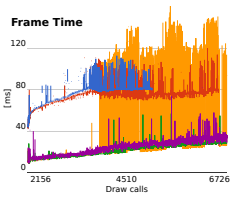
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2016
http://excel.fit.vutbr.cz

# Vulkan based render toolkit

Matěj Mainuš*






**Abstract**
This paper presents results of the experiments with next-generation rendering API Vulkan. These experiments focused on performance gain of techniques based on batching, parallel rendering, staging buffers, effective descriptors binding, and memory pool allocations. The measured results of the reference render system implementation show performance gain of used methods over baseline implementation. This paper could be useful for engineers which need to design Vulkan based render system targeted to real time rendering.

**Keywords:** Vulkan — Realtime rendering

**Supplementary Material:** Source code

*xmainu00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

## 1. Introduction

The goal of this project was demonstrate the features of new rendering API Vulkan. This API is sometimes called as redesigned or low level OpenGL. Vulkan was designed from scratch at purpose to fit modern PC, console, and GPGPU architecture. Vulkan belongs to group of graphic API family called ”next-generation”. This APIs are based on concepts described in Section 2.

Section 3 describes the design of render toolkit with techniques that needs to be implemented in almost all render systems using next generation rendering APIs, which aims to real time rendering. Simple and naive render system implementation could leads to slow, expensive and inefficient rendering due to bad utilization of the GPU architecture due to missing optimization performed by driver in prior-gen APIs.

The result of this project are benchmark results of an dynamic scene with many dynamic objects. These results show advantages of Vulkan and implementation efficiency. The tests are described in detail in Section 4.

## 2. Previous works

Evolution in GPU hardware leading from fixed graphics pipeline via programmable pipeline into GPGPU, requires changes in rendering APIs. In case of OpenGL, many changes are performed, but some of concepts survived and limit the performance of OpenGL. This led to release of new rendering API called Vulkan. It is designed with these key features [1]:

**Context less** Vulkan drops global state machine, which didn’t allow parallel API access. This is an important feature to maximize CPU and GPU utilization.

**Zero driver overhead** Vendor driver implementation did perform any state or parameter checks or any optimization. It is application responsibility to call methods in the right context and optimize the call chain. For example the application could not destroy buffer when the GPU uses them. The application also has to manage execution and memory dependencies to avoid memory conflicts.

**Layers** Part of API call chain that could be unloaded without functionality effect. Layers are mostly used to check the state and function parameters during development. In production build they could be removed [2].

**Explicit operations** Vulkan does not hide implementation of queues, devices, pipelines, command buffers, descriptor sets, render passes ... like OpenGL. Instead of is composed of entities that abstract render process [3]. It also does not hide shader compilation process, because load shaders from SPIR-V bytecode.

**Direct memory management** Application could manage memory allocations for images and buffers directly on targeted GPU heaps. It could preallocate or reuse allocated memory, it has full control over memory management process [4].

Unfortunately, Vulkan does not guarantee incredible performance. Clear driver implementation and missing optimization like in OpenGL requires big emphasis to optimize the application rendering process. First important method is **draw call batching** [5]. This method clusters objects to be rendered by pipeline configuration, object data, and rendered mesh. By clusters are generated command buffers with minimal context switching (pipeline, descriptor set or vertex buffer). CommandBuffer is a list of GPU commands which are executed after the application pushes the command buffer into the device work queue. Command buffer creation could also be done in parallel [6]. Draw command batching method should be used with hierarchical descriptor set binding [7]. Descriptor sets are structures which describe binding between shaders and buffers or images.

**Memory management** is the next part where to optimize. Key approach is to allocate memory for performance critical data (like vertexies, textures) in most powerful heaps. But mostly this memory is not accessible from CPU. It is time for staging buffers. It is buffer bound to GPU memory allocated on the heap which is accessible from CPU. Then the application copies data from RAM into staging buffer, and creates a command buffer with copy operation that transfers data from the staging buffer into the destination buffer [4]. Both data transfers also should be done in batches and staging buffers memory should be allocated from memory pool. To copy large amount of data from CPU to GPU, only GPU memory mapping is supported. It is inefficient in case of frequent usage. It is recommended to map a large GPU region at application start and then flush dirty regions, or use coherent GPU memory. In case of GPU images, the staging buffer should be used to transfer image data layout into GPU implementation optimal layout.

**Memory pool preallocation** is another method that prevents frame drops. In this case, application for each heap preallocate some amount of memory which provides to GPU objects requiring small amount of GPU memory to bind own buffers or images [4]. In some Vulkan implementations, this method could save GPU memory in case when application allocates many small memory regions, because implementation could allocate memory aligned by page size.

## 3. Design

For demonstration purposes of Vulkan API and methods described in Section 2, I created a render toolkit and benchmark application. The toolkit consists of a render and core module, that handles events, manages application and toolkit life cycle and contains scene components.

The render toolkit frame loop consists of three major steps. An event phase, update phase and render phase. In the event phase, the render toolkit kernel process user events by registered event handlers. Then render loop continues to update phase. This step updates all active scene object components in all active scenes. The toolkit scene consists of objects that build up a scene graph. Each object has a child and components such as mesh renderer, transform, camera, components related to rendering like mesh, material, texture, have reference to render system specific implementation. The update traversal is single threaded, so components does not need guarded access by another components (scripts). After update traversal core module delegate work to a render module. This frame loop process is shown in figure 1.

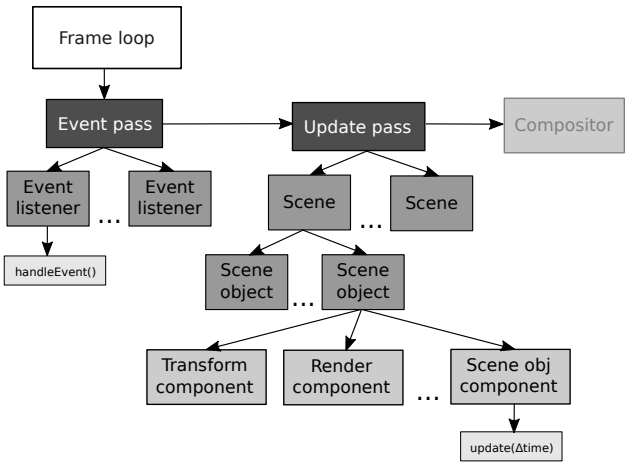



**Figure 1.** Application frame loop. It handle events at the beginning, then update active scene object components in active scenes.

The rendering module creates compositor that manages the draw process. The compositor creates a render pass with subpasses and a render worker. This worker creates several render threads, where each one initialize command and descriptor pools to avoid expensive locking during rendering process. Also creates a instance of staging buffer pool, which is going to be described in detail later. Before the drawing process, the compositor executes scene traversal process to build render tree.

Each render component from present scene that will pass render conditions and going to be rendered is added to the render list. In second step this list is sorted by render hash. This hash has to meet a condition of norm where larger distance means more difficult context or data switch. For example, renderers with same pipeline but different mesh has smaller distance then with the different materials but same mesh. It is similar with shared materials, which is closer then different one. This approach is preliminary for batching. Then by hash difference and crowdedness is created two level deep render tree. Each render node contains a reference to similar render contexts. The render tree generation process shows also figure 2.

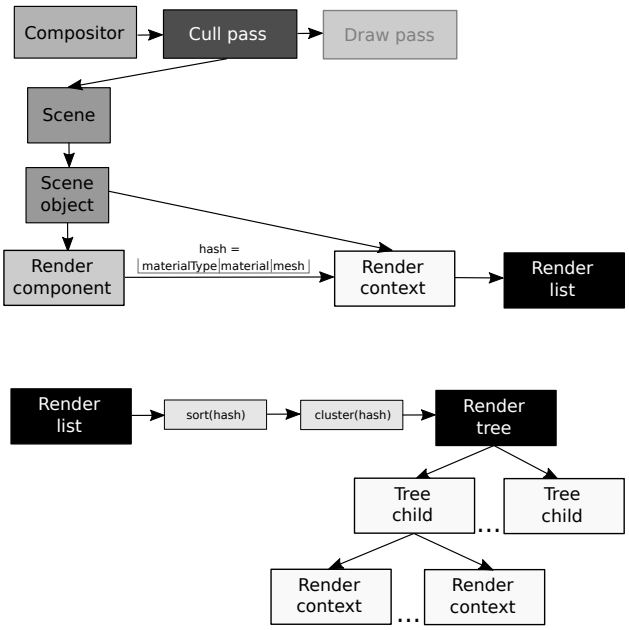



**Figure 2.** Render tree process generation. Render worker process all active render components in scene and creates list, then sort them and cluster into the render tree.

Render process starts in compositor which resets render worker and his threads. Then, for each render pass executes a worker that joins render threads that work parallel with the render tree. Each thread fetches a render node, processes them and fetches another. Render node process shows figure 3. This the render node draw phase consists of four steps. At the beginning threads prepare the phase. There they allocate internal draw structure and secondary draw and update command buffers. In the next steps, on each renderer from render node performs data upload to GPU, sets the pipeline, binds images, buffers, samplers and records the draw call into draw command buffer. The data binding via descriptor set is hierarchical, by frequency of rebind. For example viewport and scene buffers are bound once for each secondary draw command buffer. Pipeline configuration is rebound by different material type. Then material or mesh data are rebound by material/mesh instance, so many mesh renderers could share same instance of material or mesh and then could save performance. At last, scene object data is bound. The descriptor sets updates are executed in batch by the worker thread cycle.

Data are uploaded to GPU via staging memory pools. This pool preallocates a set of buffers called pages, and allocate new one if does not have enough free space. These pages are constantly mapped into the CPU accessible memory, and flushed before the primary copy command enqueuing. Then the render components copy data into the mapped page. Into render worker copy command buffer inserts copy commands that transfers data from staging buffer into component private memory, which is allocated on CPU inaccessible GPU memory heap. These secondary copy command buffers are executed by primary command buffer before any draw commands in each render pass. The draw and update command buffer dependencies shows figure 4.

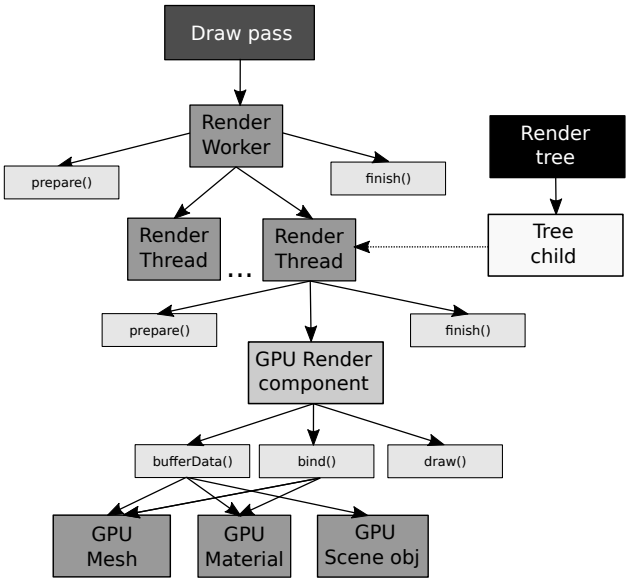



**Figure 3.** Draw traversal. The tree nodes is consumed by the render threads. The figure shows also draw phase actions.

Component private buffers or images could be

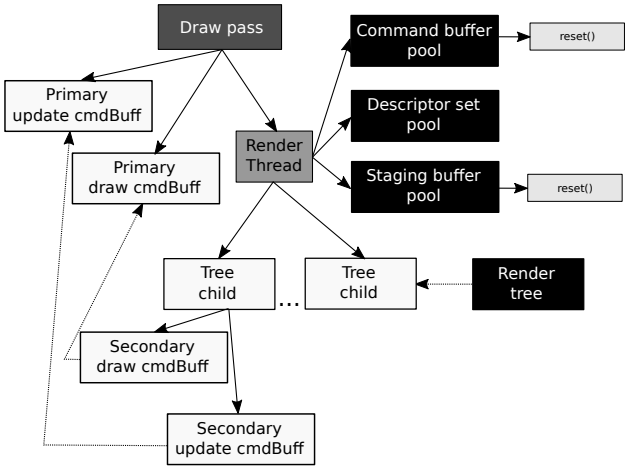



**Figure 4.** Command buffers and command, descriptor and staging pools ownership and dependency graph.

bound to inherent memory blocks or to memory pages from global memory pool. Memory manager creates a pool for each memory type, where it preallocate some pages for each one by estimated usage. Then, for each request, memory pool iterates over allocated pages and find one where it could suballocate the required amount of memory for buffer or image.

Entities like meshes, materials or textures are divided to the core part and the render implementation. The core part stores user data (like material attributes, or mesh info). The renderer part cares about GPU buffer management and data upload, when data are dirty. Upload process with staging buffers describes figure 5. Material type also defines pipeline configuration and pipeline layout. These layouts are shared between material instances.

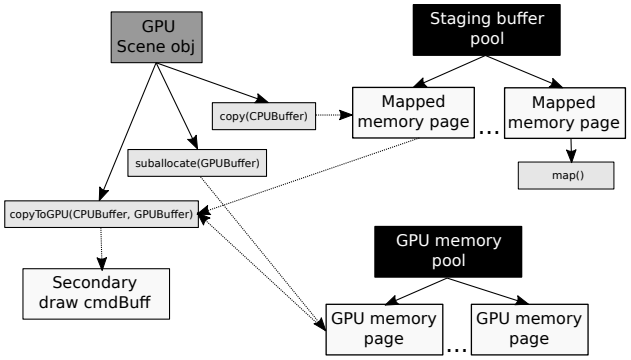


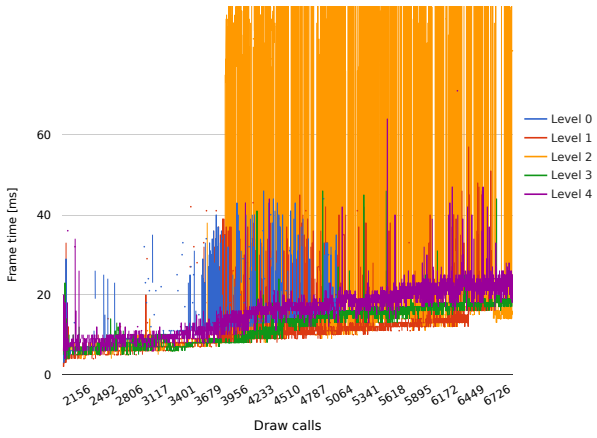

**Figure 5.** Usage of the memory pool and staging buffer in the GPUScene object.

## 4. Tests

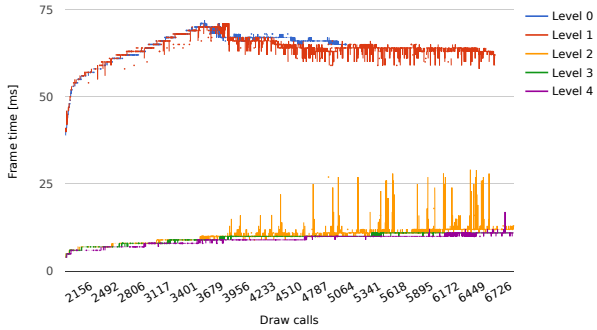
Reference implementation of the render toolkit was tested by the benchmark app. The application using toolkit build dynamic scene ( fig. 7, where objects are created and destroyed. The scene objects could be split into categories like static objects, dynamic objects (which changes transform or material properties), or mutable object, which changes material or mesh during application runtime. The test was designed to measure dependency between rendered objects count and GPU/CPU frame time in the high dynamic scene. In this test case, the application updates, allocates and frees a lot of the GPU memory, shares resources like a material properties, textures and meshes between many objects. The scene objects have multiple different materials, that shows advantages or disadvantages of the context switch minimization optimization. The application uses simple shaders because render quality and shader optimization methods was out of scope of this project.

Figure 6 and Table 1 show dependency between CPU/GPU frame time and render object count with different levels of the optimization:

**Level 0** No optimization enabled
**Level 1** Parallel rendering (5 threads)
**Level 2** Level 1 + staging buffers
**Level 3** Level 2 + memory pool
**Level 4** Level 3 + minimal pipeline state change




**(a)** CPU frame time




**(b)** GPU frame time

**Figure 6.** The frame time dependency of the rendered objects count and the level of optimization.

The tests were executed on a laptop with CPU Intel Core i7-5500U @ 2,4GHz with GPU AMD Radeon

| | 1k | 2k | 3k | 4k | 5k | 6k | 7k |
|---|---|---|---|---|---|---|---|
| Level 0 | 6.2 | 9.1 | 14.8 | 16.7 | | | |
| Level 1 | 4.9 | 6.2 | 11.6 | 13.3 | 13.1 | 16.7 | |
| Level 2 | 6.5 | 7.4 | 27.9 | 49.7 | 46.9 | 42.0 | 37.0 |
| Level 3 | 6.7 | 7.6 | 8.2 | 13.2 | 16.3 | 18.0 | 19.9 |
| Level 4 | 8.8 | 8.7 | 11.9 | 16.9 | 19.8 | 22.9 | 25.3 |

**(a)** CPU frame times.

| | 1k | 2k | 3k | 4k | 5k | 6k | 7k |
|---|---|---|---|---|---|---|---|
| Level 0 | 48.3 | 61.0 | 68.1 | 66.3 | | | |
| Level 1 | 49.4 | 61.2 | 67.8 | 64.3 | 63.7 | 63.2 | |
| Level 2 | 5.7 | 7.7 | 9.7 | 10.5 | 11.7 | 13.1 | 14.0 |
| Level 3 | 5.5 | 7.5 | 9.3 | 10.0 | 10.6 | 11.0 | 11.9 |
| Level 4 | 5.2 | 7.0 | 8.6 | 9.4 | 10.0 | 10.8 | 11.4 |

**(b)** GPU frame times.

**Table 1.** Average CPU/GPU frame times in milliseconds grouped by thousands of the draw calls for each optimization level.

R5 M240. The results in case of the Level 0 shows that GPU access to main memory is bottleneck of the baseline solution. The parallel rendering gain is significant from large object count, but the gain is not negligible. The Level 2 bottleneck is GPU memory allocation/free. On the CPU side the peaks are probably related with a GPU memory fragmentation on a small mobile GPU memory heap. This issue was not preset in case of Level 0 and 1, because the driver allocates memory on larger host memory heap. The Level 3 had suboptimal performance characteristic on CPU but not on GPU, which has to change render states more frequently than in case of Level 4, but the Level 4 was not optimal in CPU case, due to sorting operations.

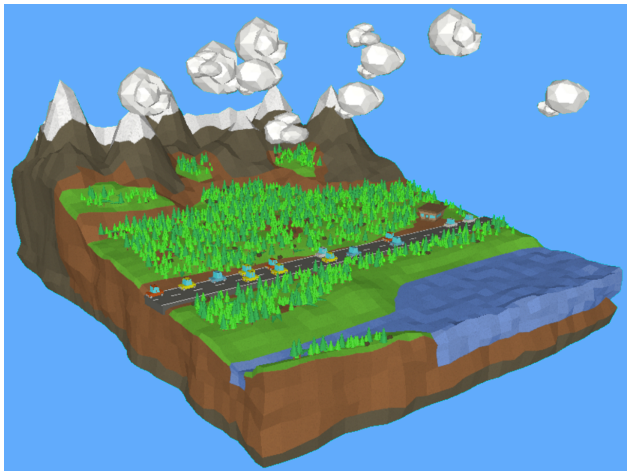


**Figure 7.** Benchmark scene.

## 5. Conclusions

This paper describes the design of render system suitable for Vulkan API. This render system includes several optimization techniques, which help to reach optimal performance and minimal overhead with complex scenes. It is achieved by parallel rendering, with draw calls batched by pipeline and input data, preallocation strategies which allocates important memory on most powerful GPU memory heap, staging buffers and hierarchical descriptors binding. With the optimization methods the render toolkit is able to render complex and high dynamic scene in real time.

Future development of the render toolkit should focus on Vulkan implementation of other techniques used in nowadays rendering such as post processing, occlusion culling, virtual textures, level of details, global illumination methods, etc...


## Acknowledgements

I would like to thank Prof. Adam Herout, Ph.D. for his help and valuable advice with my diploma thesis, on which this paper is based.


## References


[1] The Khronos Vulkan Working Group. Vulkan specification. online, Mar 2016. https://www.khronos.org/registry/vulkan/specs/1.0-wsi_extensions/xhtml/vkspec.html.

[2] Daniel Rákos. Using the vulkan validation layers. online, Mar 2016. http://gpuopen.com/using-the-vulkan-validation-layers/.

[3] Tobias Hector. Vulkan: Explicit operation and consistent frame times. online, Mar 2016. http://blog.imgtec.com/powervr/vulkan-explicit-operation-and-consistent-frame-times.

[4] Chris Hebert and Christoph Kubisch. Vulkan memory management. online, Mar 2016. https://developer.nvidia.com/vulkan-memory-management.

[5] Christoph Kubisch. Vulkan & opengl threaded cad scene sample. online, Mar 2016. https://developer.nvidia.com/vulkan-opengl-threaded-cad-scene-sample.

[6] Ashley Smith. Gnomes per second in vulkan and opengl es. online, Mar 2016. http://blog.imgtec.com/powervr/gnomes-per-second-in-vulkan-and-opengl-es.

[7] Christoph Kubisch. Vulkan shader resource binding. online, Mar 2016. https://developer.nvidia.com/vulkan-shader-resource-binding.