Vulkan

Maik Klein
University of Koblenz
Email: maikklein@uni-koblenz.de

I. Introduction

Vulkan is a low level graphics and compute API, designed to transfer more control to the developer. It has a unified API for desktop, mobile, console, and embedded [2, p. 12].
The context in OpenGL can not be accessed across threads at the same time but Vulkan was designed from the ground up to support multithreading. In many cases, Vulkan does not do any synchronization on its own but instead exposes synchronization primitives to the developer. Additionally Vulkan introduced the concept of command buffers which are able to record graphics and compute commands across threads. These command buffers can then be submitted to the GPU.
Although OpenGL 4.5 introduced direct state access, it still relies on global state. Because of the global state it is hard for the driver to guess the intentions of the developer. In Vulkan, everything is explicit and there are no run time validation checks but validation checks can still be enabled with the validation layer.
Vulkan does not hide information from the developer. For example OpenGL exposes 24-bit textures while modern GPUs do not natively support such a format. This might result in the driver emulating the feature with a 32-bit texture. In Vulkan the developer has to check explicitly if the format is actually supported by the underlying hardware. The developer is responsible to fall back to a different format if the requested format is not available [1, Chapter 30].
GPUs might have additional hardware which excel at very specific tasks. Modern GPUs commonly ship with separate hardware that is very good for moving data from RAM to GPU memory [3, p. 60]. Vulkan exposes such hardware through specialized queues. The developer is responsible for finding and using these queues.
Vulkan shifts responsibility from the driver to the developer. This results in a less complex driver and allows the developer to better express their intents.

II. Hardware support

Vulkan is a crossplatform API, and is available on the following platforms. Linux, Windows 7, 8, 10, Android 6.0+, Nintendo Switch, Nvidia Shield, [20] And these hardware vendors offer official drivers for the follwing operating systems.

- AMD: Linux and Windows [17]
- Nvidia: Linux and Windows [16]
- ARM: Linux and Android [20]
- Intel: Windows [20]
- Imagination: Android [20]
- Qualcomm: Android [20]
- VeriSilicon: Android [20]

Officially MacOS and iOS do not support Vulkan [8, p.429], but Vulkan can be emulated on those platforms with MoltenVK [25].
Additionally it should be noted that RADV, an open source driver for AMD on Linux, effectively passes the conformance tests for Vulkan [18].

III. Who is using Vulkan in production?

Vulkan is already used in production by various companies
Games:

- Dota2 [22]
- Talons Principle [9]
- Doom [21]
- Ashes of Singularity [23]
- Mad Max [24]
- Star Citizen *will exclusively use Vulkan* [10]

Game Engines:

- Unreal Engine 4 [11]
- CryEngine 5.4 [13]
- Unity3D [12]
- Source2 [15]
- Xenko [14]

IV. Reason to and not to use Vulkan

A. Reasons to use Vulkan

- Multithreading: If the application wants to use more than one core for graphics and compute commands [5, p. 8].
- Keeping the GPU busy: It can be hard for a single threaded application to keep the GPU busy at all times. Because Vulkan supports multithreading, it is possible to interleave frames to better utilize the GPU [5, p. 8].
- Avoiding hitches: Vulkan is an explicit API and therefore can avoid some pitfalls of OpenGL. For example, in OpenGL some pipelines might only be created right before they are needed, which might cause the performance to drop. In Vulkan, pipelines have to be created explicitly, which means that the developer controls when such a performance hit is acceptable [5, p. 8].
- CPU limited applications: Vulkan has a much lower CPU overhead than previous graphics APIs and therefore CPU limited applications might benefit from using Vulkan [5, p. 8].

B. Reasons not to use Vulkan

- Single threaded applications: While Vulkan still has benefits for single threaded applications, single threaded applications might not benefit as much from using Vulkan compared to OpenGL [5, p. 9].

- GPU limited applications: If the application already uses the GPU to its full extent, then Vulkan might not improve the performance [5, p. 9].
- Portability: Vulkan exposes hardware capabilities and it is the job of the developer to implement a fast path for the specific target hardware. This might add additional complexity that the developer does not want to manage [5, p. 9].
- Requires additional thought: Porting a D3D11/OpenGL application directly to Vulkan might not result in a net benefit. Additional thought is required to take advantage of the explicit API [5, p. 9].

V. Overview of the eco-system

- LunarG Vulkan SDK: [28] Includes Vulkan documentation, samples and demos. It provides Vulkan runtime libraries, a Vulkan loader, Validation layers, Vulkan trace tools and SPIR-V tools.
- Glslang [26]: The official reference compiler for SPIR-V.
- SPIRV-Cross [27]: Exposes a reflection API for SPIR-V and can cross compile SPIR-V into HLSL, GLSL, and MSL.
- Renderdoc [29]: A free and open source debugger for Vulkan.
- SaschaWillems's Vulkan Samples [30]: Extensive list of documented Vulkan samples.
- Gpuinfo [32]: A public Vulkan hardware database. This is useful to get an overview of which hardware capabilities are available on consumer hardware.

VI. Highlevel overview

"VkInstance" serves as an entry point for the Vulkan API. It exposes functions to find physical devices, create logical devices. "VkPhysicalDevice" usually represents a single device in a system. It is possible to query various information about a physical device such as memory porperties, supported image formats, driver version, device name, API version and so on. With a physical device a logical device can be created. A logical device "VkDevice" offers functionality to interact with a device. It can allocate memory, create buffers, create command buffers. More information in section VIII.
"VkBuffer" can store data in an linear array and which can be bound by graphics or compute pipelines inside a command buffer. Unlike in OpenGL, memory allocation and buffers are seperate concepts. Memory allocation happens through "vkAllocateMemory()", and this allocation can then be bound to a buffer. It is possible to suballocate many buffers into one big allocation. More information in section X.
"VkDescriptorSet" describes a list of resources that are used by a shader. A shader can have more than one descriptor set. More information in section XII.
A "VkRenderPass" describes how the rendering is going to happen. A render pass consist of a series subpasses [38]. A "VkFramebuffer" is used to store the information from a renderpass in images through images views. A frame buffer can be specified when a render pass is started. Drawing can only happen inside a render pass. More information in section XIII.
A "VkPipeline" contains all the state that is required for drawing or computations. Vulkan differentiates between graphics and compute pipelines. It stores state such as the shader modules, render and subpasses, depth, blending, multisampling, tesselation and so on. The state is mostly immutable which means that the state can not be changed after a pipeline is created. More information in section XIV.
"VkCommandBuffer" is able to record commands and can then be executed by a "VkQueue". Command buffers offer functionality for binding a pipeline, drawing, binding buffers, binding desciprtor sets and more. The main benefit of command buffers is that they can be recorded on multiple threads. More information in section IX.
Vulkan also offers extensions and layers. More information in section XVI.

VII. Loading Vulkan

A. Loading Entry, Instance and Device level functions

After the Vulkan library has been loaded into memory, "vkGetInstanceProcAddr" can be resolved. This function can be used to resolve the entry level functions.

```

vkGetInstanceProcAddr(NULL, "vkCreateInstance");
vkGetInstanceProcAddr(NULL, "
    vkEnumerateInstanceExtensionProperties");
vkGetInstanceProcAddr(NULL, "
    vkEnumerateInstanceLayerProperties");
```

After "vkCreateInstance" has successfully been resolved, it can be used to resolve instance level functions.

```

vkGetInstanceProcAddr(instance, "vkCreateDevice");
vkGetInstanceProcAddr(instance,
                      "vkGetDeviceProcAddr");
...
```

After "vkCreateDevice" and "vkGetDeviceProcAddr" have been resolved, device level functions can be loaded.

```
vkGetDeviceProcAddr(device, "vkGetDeviceQueue");
vkGetDeviceProcAddr(device, "vkQueueSubmit");
...
```

It should be noted that it is also possible to resolve device level function pointers from "vkGetInstanceProcAddr" but this will introduce one level of indirection [31].

VIII. Instances and Devices

Instances can be created with "vkCreateInstance". When the instance has been created, it is then possible to create a logical device with "vkCreateDevice". Before a logical device can be created, the developer has to find a suitable physical device. All physical devices that have Vulkan support can be found with "vkEnumeratePhysicalDevices". A physical device exposes all available hardware capabilities.

The developer might want to check at this time which queues and device features are available for a particular device.

For example if the application intents to use graphics commands and to display the results onto a surface, the developer has to find one or more queues that support these two features. “vkGetPhysicalDeviceQueueFamilyProperties” can be used to get an array of “VkQueueFamilyProperties”.

“queueFlags” contains information of the capabilities that this queue supports. The “VK_QUEUE_GRAPHICS_BIT” will be set if the queue supports graphics commands. “vkGetPhysicalDeviceSurfaceSupportKHR” can be used to find a queue family index that supports presentation. A queue family index is the index of a “VkQueueFamilyProperties” object inside the array that is returned from “vkGetPhysicalDeviceQueueFamilyProperties” starting from 0. A queue is allowed to have all “VkQueueFlagBits” set. In fact all modern Nvidia GPUs currently expose 16 queues that support graphics, compute, transfer, and sparse binding commands. Specialized hardware can be found by looking at queues that do not support every feature. Most modern GPUs expose a specialized queue that excels at moving data from RAM to GPU memory. Such a queue might only have the “VK_QUEUE_TRANSFER_BIT” set.

Before the device can be created, the developer has to specify which and how many queues the application should use.

Lastly “vkGetPhysicalDeviceFeatures” can be used to retrieve the supported features for a specific physical device. These features can then be enabled before the logical device is created.

IX. CommandPools and CommandBuffers

Command buffers are objects used to record commands which can be submitted to a queue. [1, Chapter 5] Command buffers are allocated from a “VkCommandPool”. A command buffer can be recorded with “vkCmd*” commands called in between “vkBeginCommandBuffer” and “vkEndCommandBuffer”. Then the command buffer can be submitted to a queue with “vkQueueSubmit”. It is important to know that command buffers are implicit externally synchronized, which means that they are tied to a specific pool. In practice this means that it is not allowed to concurrently record two command buffers, that are allocated from the same pool, on two different threads. To avoid synchronization, the developer is allowed to create one pool per thread.

A draw call might look like this: [34]

- Begin Renderpass
- Bind Vertex/Index
- Set Viewport
- Bind Pipeline
- Bind DescriptorSet
- Draw
- End Renderpass

While it is possible to create a command buffer for every draw call, a better approach would be to record every draw call directly inside the command buffer. This is why Vulkan differentiates between primary and secondary command buffers. Secondary command buffers are more limited and can not be submitted to a queue directly but they can be executed by a primary command buffer with “vkCmdExecuteCommands”. One approach would be to only create one primary command buffer, create N secondary command buffers on different threads and then execute all secondary command buffers in the primary command buffer.

Secondary command buffer:

- Bind DescriptorSet
- Bind Vertex/Index
- Set Viewport
- Bind Pipeline
- Bind DescriptorSet
- Draw

Primary command buffer:

- Begin Renderpass
- Execute N secondary command buffer
- End Renderpass

A command buffer can either be freed with “vkFreeCommandBuffers”, reset individually with “vkResetCommandBuffer” or every command buffer inside a pool can be reset with “vkResetCommandPool”. It should be noted that a pool needs to be created with the “VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT” flag if the application wants to free a command buffer individually.

X. Buffer and Allocation

Buffers in Vulkan can be created by filling out the “VkBufferCreateInfo” struct and calling “vkCreateBuffer”. “size” is the size of the buffer in bytes. It should be noted that “vkCreateBuffer” does not do any allocation. Any combination of bits can be specified for “usage”, but at least one of the bits must be set in order to create a valid buffer.

After the buffer has been created, “vkGetBufferMemoryRequirements” can be used to retrieve the memory requirements of that buffer.

memoryTypeBits is a bitmask and contains one bit set for every supported memory type for the resource. Bit i is set if and only if the memory type i in the “VkPhysicalDeviceMemoryProperties” structure for the physical device is supported for the resource [1, Chapter 10].

```
// Find a memory type in "memoryTypeBits" that
    includes all of "properties"
int32_t FindProperties(uint32_t memoryTypeBits,
    VkMemoryPropertyFlags properties)
{
    for (int32_t i = 0; i < memoryTypeCount; ++i)
    {
        if ((memoryTypeBits & (1 << i)) &&
            ((memoryTypes[i].propertyFlags &
    properties) == properties))
            return i;
    }
    return -1;
}

// Try to find an optimal memory type, or if it does
    not exist
// find any compatible memory type
VkMemoryRequirements memoryRequirements;
vkGetImageMemoryRequirements(device, image, &
    memoryRequirements);
```

```
int32_t memoryType = FindProperties(
      memoryRequirements.memoryTypeBits,
      optimalProperties);
if (memoryType == -1)
      memoryType = FindProperties(memoryRequirements.
      memoryTypeBits, requiredProperties);
```

[1, Chapter 10]

After the correct memory type index has been found, “vkAllocateMemory” can be used to allocate memory. If the memory was created with the “HOST_VISIBLE” flag, the memory can then be mapped with “vkMapMemory”. It should be noted at “vkMapMemory” is externally synchronized and the application should not try to write to memory that is currently in use. The memory can then be unmapped with “vkUnmapMemory”. If the “VK_MEMORY_PROPERTY_HOST_COHERENT_BIT” was not set, the application might need to call “vkFlushMappedMemoryRanges” and “vkInvalidateMapped-MemoryRanges” to make host writes visible to the device or device writes visible to the host.
“vkBindBufferMemory” can be used to bind the memory to a “VkBuffer”. Device local buffers usually yield better performance when read by the GPU. To create a device local buffer from a host visible buffer, the developer has to create a new buffer. The host visible buffer should have been created with the “VK_BUFFER_USAGE_TRANSFER_SRC_BIT” flag and the device local buffer should be created with the “VK_BUFFER_USAGE_TRANSFER_DST_BIT” flag. The allocation for the device local buffer should be found with the “VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT” flag. The host local buffer can then be copied to the device local buffer with “vkCmdCopyBuffer”.
It should be noted that one buffer can also consist of multiple buffers. For example it is common for a vertex and index buffer to belong together. Instead of creating two different buffers and two different allocations, it is possible to create one buffer and one allocation and then use “vkCmdBindVertexBuffers” / “vkCmdBindIndexBuffer” with the correct offset. Alternatively it is also possible to create two buffers that share one allocation. Additionally the developer should also know that there is a limit of how many allocations can be made. This value can be found in “VkPhysicalDeviceLimits::maxMemoryAllocationCount”. This means that it is recommended to allocate big and to sub-allocate buffers into one allocation.

XI. Images

To create a device local image, the application has to create a host visible buffer which contains the image data. An image can be created with “vkCreateImage”. The initial layout has to be “VK_IMAGE_LAYOUT_PREINITIALIZED” or “VK_IMAGE_LAYOUT_UNDEFINED”. The application also needs to verify if the requested format is supported by the underlying hardware. This can be done with “vkGetPhysicalDeviceFormatProperties”. The application then has to find the correct memory index for the image. The correct memory requirements can be found with “vkGetImageMemoryRequirements”. Before copying the data, the application should transition the image layout to “VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL”. Then the data can be copied with “vkCmd-CopyBufferToImage”. After the data has been copied, the image layout should be changed to “VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL” if the image should be read by a shader.
An “VkImageView” can then be created with “vkCreateImageView” which can be passed into a “VkDescriptorImageInfo” alongside a sampler object which was created with “vkCreateSampler”. Then the “VkDe-scriptorImageInfo” can be written to a descriptor set with “VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER”.

XII. Resource Descriptors

Descriptors allow shaders to access buffer and images resources. Often developers want to update shader resources at different frequencies [33].

```
// example for typical loops in rendering
for each view {
  bind view resources          // camera,
    environment...
  for each shader {
    bind shader pipeline
    bind shader resources      // shader control
    values
    for each material {
      bind material resources  // material
    parameters and textures
      for each object {
        bind object resources  // object transforms
        draw object
      }
    }
  }
}
```

Descriptor sets allow shader resources to be grouped. For example the application might create 3 different descriptor sets. One descriptor set for static scene data such as lights, one descriptor set for material data like image samplers for roughness and metallic maps for a PBR material, and one descriptor set for object transforms. Every descriptor set can be updated individually. Descriptor sets are allocated from descriptor pools, similar to how command buffers are allocated from commands pools. “VkPhysicalDeviceLimits::maxBoundDescriptorSets” is the maximum amount of descriptor sets that can be bound at the same time.
Vulkan also introduced “push constants”, which allow the application to send data directly to the shader without needing to use a descriptor set. “VkPhysicalDevice-Limits::maxPushConstantsSize” is the maximum size of the data that can be send to the shader with push constants. The minimum size for “VkPhysicalDeviceLimits::maxPushConstantsSize” is 128 bytes, this is the size for two 4x4 f32 matrices [1, Table 37. Required Limits].

XIII. Renderpass

A render pass represents a collection of attachments, subpasses, and dependencies between the subpasses, and

describes how the attachments are used over the course of the subpasses [1, Chapter 7].
A renderpass can be used inside a command buffer with "vkCmdBeginRenderPass" and "vkCmdEndRenderPass".
Sometimes the application might want to use the output of one rendering job as the input of another rendering job. A common example of this is deferred rendering where the application writes diffuse color, specular color, depth, normal information into separate textures. These textures are then used as the input of another rendering job to light the scene, but rendering to separate textures is slow [35, Page. 17]. Subpasses are able to express dependencies between different rendering jobs and allow the driver to produce a better mapping to the hardware. Subpasses can be executed inside a renderpass with "vkCmdNextSubpass". If the images are created with "VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT", the driver might not write to the texture at all and it can directly pass the results to the next subpass. The downside is that the render job can only access the current pixel.

XIV. Pipelines

Vulkan supports two types of pipelines, a compute and a graphics pipeline. OpenGL has no real concept of pipelines and render state might change from draw call to draw call. Vulkan encapsulates the state in a pipeline which has to be explicitly created with "vkCreategraphicsPipelines" or "vkCreateComputePipelines". Pipeline creation can be cached with a "VkPipelineCache" object. Pipelines are mostly immutable and can not change state after they have been created, unless the state was defined to be dynamic at pipeline creation time. Pipelines can be bound with "vkCmdBindPipeline" inside a render pass.

XV. Swapchain

The swapchain is a device level extension which allows images to be presented to a surface. A swapchain can be created with "vkCreateSwapchainKHR". The presentable images are owned by the swapchain and the "minImageCount" has to be specified before the swapchain can be created. "vkGetPhysicalDeviceSurfaceCapabilitiesKHR" can be used to retrieve information of the minimum and maximum number of images the underlying hardware supports. "vkGetPhysicalDeviceSurfacePresentModesKHR" can be used to retrieve the available present modes. The images inside the swapchain can be accessed with "vkGetSwapchainImagesKHR". These images can be used to create images views which can then be used to create framebuffers. A renderpass is able to render to a "VkFramebuffer". "vkAquireNextImageKHR" can be used to retrieve the index of framebuffer, which can then be used inside a renderpass. After the draw call has finished, "vkQueuePresentKHR" can present the image to a surface. If the application wants to resize the framebuffers, a new swapchain has to be created.

XVI. Extensions and Layers

Vulkan offers two ways to add functionality, Extensions and Layers. Layers can not add or modify Vulkan commands but can insert themselves into the call chain. An example for a layer would be the "standard validation layer". This layer is able to validate API calls during development. An extension is allowed to add functionality. [1, Chaper 30.1] [31]

Commonly used extensions:

- VK_KHR_surface abstracts over native surface or window objects. This extension is commonly used with the following platform specific extensions a) VK_KHR_win32_surface, b) VK_KHR_xlib_surface,
- VK_KHR_swapchain provides the ability to present rendering results to a surface.
- VK_EXT_debug_report is able to output warnings, errors, performance warnings, and general information. This extension is commonly used with the "VK_LAYER_LUNARG_standard_validation" layer.

Instance extensions, and layers can be enabled at instance creation, they can be explicitly enabled in the "VkInstanceCreateInfo" struct.

Device level extensions can be enabled in the "VkDeviceCreateInfo" struct.

The naming convention for extensions is "VK_VENDORNAME_extension_name". The vendor name for extensions developed by multiple vendors is "EXT". Additionally the vendor name for experimental extensions is "KHX". An example for such a extension would be "VK_KHX_device_group" which is able to group multiple physical devices into one logical device.
Previously Vulkan distinguished between instance and device level layers, but this feature has been deprecated and all layers can now be enabled at instance creation time [1, Chapter 30.1.1].

A complete list of extensions can be found at [1, Appendix C: Layers & Extensions].

XVII. Threading Behavior

All functions in Vulkan can be called concurrently from different threads but certain parameters or components of parameters are defined to be "externally synchronized".This means that the caller must guarantee that no more than one thread is using such a parameter at a given time [1, Chapter 2.5] Additionally there are also functions with implicit externally synchronized parameters. For example command buffers that are allocated from a "VkCommandPool" are tied to this specific pool and therefore share the same threading behavior.

XVIII. Synchronization

Vulkan offers 4 synchronization primitives, fences, semaphores, events, and pipeline barriers. Fences are used for communication with the host. One example would be "vkQueueSubmit". After the application has submitted work to the queue, a fence can be used with "vkGetFenceStatus" to check if the work has been completed. Additionally it is also possible to wait for the work to complete with "vkWaitForFences".
A semaphore is used to synchronize dependencies on the GPU and it can only be in two states, signaled or unsignaled.

One example would be to synchronize the draw call with the presentation. Semaphores do not need to be reset manually.
Events provide a fine-grained synchronization primitive which can be signaled either within a command buffer or by the host, and can be waited upon within a command buffer or queried on the host [1, Chapter 6].
Pipeline barriers also provide synchronization control within a command buffer, but at a single point, rather than with separate signal and wait operations. One example of a barrier would be the “VkImageMemoryBarrier” which can be used to transition images to different image layouts.

XIX. Conclusion

Vulkan delivers a low level, explicit and cross platform API that does not hide important details from the developer and therefore makes the API more transparent. It is already used in production by various games and game engines, has driver support from all major hardware vendors and simplifies shader backends by providing a low level shading language. But other options still exist.

- OpenGL: OpenGL supports all platforms that Vulkan currently supports but can also be run in the browser and on iOS and macOS. Modern OpenGL can still be a good choice, especially with AZDO [37] but it does not have the same level of support for multithreading that Vulkan has.
- Direct3D 12: Direct3D 12 has a very similar API compared to Vulkan but only runs on Windows 10. Because D3D 12 only supports Windows 10, Star Citizen might exclusively use Vulkan. But because the APIs are so similar, it should not be too hard to find a common abstraction.
- Metal: Metal is a graphics and compute API from Apple. It only supports iOS, macOS, and tvOS. It also is not as low level as Vulkan and Direct3D 12. MoltenVK [25] is a library that uses Metal to emulate Vulkan on macOS and iOS.

The spec [1] for Vulkan is very readable and the standard validation layer does a very good job of finding errors and reporting errors in a very understandable way. Often the error messages even quote the spec to better explain the error message.
Vulkan has a pretty steep learning curve and it can be very tedious to render a simple triangle because of its explicit API. But after rendering a simple triangle, it is not hard to implement more complex tasks because rendering a triangle already requires the developer to understand a big part of the API. Rendering a triangle requires roughly 1000 LOC, and the quake 3 vulkan renderer [39] only requires 3200 LOC.

References

[1] https://www.khronos.org/registry/vulkan/specs/1.0-extensions/html/vkspec.html Accessed on 16.07.2016

[2] https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf Accessed on 16.07.2016

[3] http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2016/03/d3d12_vulkan_lessons_learned.pdf Accessed on 16.07.2016

[4] https://github.com/KhronosGroup/Vulkan-LoaderAndValidationLayers/blob/master/loader/LoaderAndLayerInterface.md#device-related-objects Accessed on 16.07.2016

[5] http://developer.download.nvidia.com/gameworks/events/GDC2016/Vulkan_Essentials_GDC16_tlorach.pdf Accessed on 16.07.2016

[6] https://developer.nvidia.com/transitioning-opengl-vulkan Accessed on 16.07.2016

[7] https://developer.nvidia.com/transitioning-opengl-vulkan and http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2016/03/d3d12_vulkan_lessons_learned.pdf Accessed on 16.07.2016

[8] https://www.amazon.com/Vulkan-Programming-Guide-Official-Learning/dp/0134464540 Accessed on 16.07.2016

[9] http://www.croteam.com/house-interview-vulkan/ Accessed on 16.07.2016

[10] https://forums.robertsspaceindustries.com/discussion/comment/7581676/#Comment_7581676 Accessed on 16.07.2016

[11] https://www.unrealengine.com/en-US/blog/unreal-engine-4-15-released Accessed on 16.07.2016

[12] https://blogs.unity3d.com/2017/03/31/5-6-is-now-available-and-completes-the-unity Accessed on 16.07.2016

[13] https://www.cryengine.com/roadmap Accessed on 16.07.2016

[14] http://xenko.com/ Accessed on 16.07.2016

[15] http://www.valvesoftware.com/news/?id=16000 Accessed on 16.07.2016

[16] https://developer.nvidia.com/vulkan-driver Accessed on 16.07.2016

[17] http://www.amd.com/en-us/innovations/software-technologies/technologies-gaming/vulkan Accessed on 16.07.2016

[18] http://airlied.livejournal.com/83102.html Accessed on 16.07.2016

[19] https://developer.android.com/ndk/guides/graphics/index.html Accessed on 16.07.2016

[20] https://www.khronos.org/conformance/adopters/conformant-products Accessed on 16.07.2016

[21] http://doom.com/en-us/ Accessed on 16.07.2016

[22] http://www.valvesoftware.com/games/dota2.html Accessed on 16.07.2016

[23] http://www.ashesofthesingularity.com/ Accessed on 16.07.2016

[24] http://madmaxgame.com Accessed on 16.07.2016

[25] MoltenVk https://moltengl.com/moltenvk/ Accessed on 16.07.2016

[26] https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/ Accessed on 16.07.2016

[27] https://github.com/KhronosGroup/SPIRV-Cross Accessed on 16.07.2016

[28] https://www.lunarg.com/vulkan-sdk/ Accessed on 16.07.2016

[29] https://renderdoc.org/ Accessed on 16.07.2016

[30] https://github.com/SaschaWillems/Vulkan Accessed on 16.07.2016

[31] http://gpuopen.com/using-the-vulkan-validation-layers/ Accessed on 16.07.2016

[32] http://vulkan.gpuinfo.org/ Accessed on 16.07.2016

[33] https://developer.nvidia.com/vulkan-shader-resource-binding Accessed on 16.07.2016

[34] https://developer.nvidia.com/engaging-voyage-vulkan Accessed on 16.07.2016

[35] https://www.khronos.org/assets/uploads/developers/library/2016-vulkan-devday-uk/6-Vulkan-subpasses.pdf Accessed on 16.07.2016

[36] https://www.khronos.org/registry/spir-v/specs/1.2/SPIRV.html Accessed on 16.07.2016

[37] https://www.khronos.org/assets/uploads/developers/library/2014-gdc/Khronos-OpenGL-Efficiency-GDC-Mar14.pdf Accessed on 16.07.2016

[38] https://renderdoc.org/vulkan-in-30-minutes.html Accessed on 16.07.2016

[39] https://medium.com/@kennyalive/quake-3-vulkanized-245cc349fdcf Accessed on 16.07.2016