What is the difference between traditional graphics APIs and new low-level APIs?

- High-level APIs like OpenGL are quite easy to use, because the developer declares what they want to do and how they want to do it, and the driver handles the rest.

- 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.
What is Vulkan?

It is a multiplatform API that allows developers to prepare high-performance graphics applications like games, CAD tools, benchmarks, and so forth.

Vulkan API in some way inherits OpenGL but it was built on API Mantle from AMD.
What is Vulkan?

Address the needs for graphics developers involved with:

- Video Games
- Engineering visualization
- Simulation
- VR and other graphic intensive applications

Support for graphic and compute functionality
Available on current hardware
What Vulkan is not

1. Vulkan is not an extension of OpenGL
2. It is not backwards compatible to either OpenGL, OpenGLES, OpenCL or any other existing API
3. No one to one mapping of API calls with OpenGL
4. It is not a state machine like OpenGL is
5. It does not hide everything that the GPU is doing
Why do we need it?

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.
Why do we need it?

The Need for Vulkan

Ground-up design of a modern open standard API for driving high-efficiency graphics and compute on GPUs used across diverse devices.

In the twenty two years since OpenGL was invented - the architecture of GPUs and platforms has changed radically.

GPUs being used for graphics, compute and vision processing on a rapidly increasing diversity of platforms - increasing the need for cross-platform standards.
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.
Do we need any window or separate display?

Of course not, you can easily set the way of translation results or even do not translate them at all.
Let’s start from simple things. As Vulkan was created by Khronous Group its syntax is very similar to OpenGL, but through all API is used the prefix “vk”.

For example:

- Functions: vkDOSomething()
- Names of structures: VkSomething
- Constant expressions: VK_SOMETHING
- There are also exist special command functions to which prefix “Cmd” is added: vkCmdJustDoIt()
First steps and main determinations

There are two main things:

- Device (Device will do every command that it was asked to execute)
- Host (Sends commands to device)
Interesting fact!

In Vulkan you always have to fill in some data structure to create an object. This data is a characteristics of your object and can also contain information about your application, version of the API and other info.
Layered Architecture

Allows developer to add and remove layers as desired
There are no hard checks of incoming data for correctness in Vulkan. So the most common use of layers is to check data and monitor the work of Vulkan fin appropriate way.
Extensions

As you can guess from its name they extend the functionality of the Vulkan. For example extension “debug report” will return errors from all layers.
But what exact information can we collect (if there are no errors or they are not dangerous for our app)?

This info:
- supported formats
- memory
- queue families.
Queues and Queues’ Families

Our opportunities:
- Draw
- Do Computations
- Copy Data
- Do sparse memory management
Commands and their execution

All commands are collected in special container – command buffer.
Command buffer can be of two types:

1. **Primary** – is sent directly to the queue
2. **Secondary** – can not be sent, it is run in the Primary Buffer
Important tips!

- Host does not wait till the end of command use or command buffer execution (But it can be always make to wait if you do it by yourself)
- After sending command buffers to the queue control then immediately is given to App.
Synchronization
(There are 4 primitive ways of synchronization)
Fence

The easiest way. It lets host wait for execution of all commands in buffer. (The most popular way of synchronization)
Semaphore

The way of synchronization inside device. Its state can never be seen or waited on host or inside command buffer, but we can set what semaphore should signal after execution all commands in buffer and what semaphore has to wait before starting execution commands in buffer. Not all buffer will wait but only a part of it.
Element of very accurate synchronization. Signal can be sent both from host or device, the same is with waiting for signal process. Event describes interdependence of two sets of commands (before and after) in buffer. And for event there is a special state which allows wait for host.
Barrier

Can be used only in device, to be more clear it is used in command buffer. This way of synchronization decelerate relation between first and second commands sets. There is also permitted to set memory barriers, which can be of 3 types:

1. Global barrier
2. Buffer barrier
3. Image barrier

They will protect data from writing while reading the same data or wise versa.
In Vulkan there are two conveyors:

1. **Graphical**
2. **Executable** (results can be then sent to the graphical conveyor)

For the Conveyor you can create cash, which can then be used by other conveyors even after reloading of the App.
Render pass, graphic pipeline and framebuffer

The graphical conveyer is needed to use drawing commands.

To start rendering process we need to set:
1. Render Pass
2. Subpasses
3. Attachments

After the start of the rendering process we can draw. We also can switch between subpasses. After the end of the drawing process we can end the rendering process.
Memory and resources management

The memory in Vulkan is shared by the host. If we need to put image or other data in to the device – memory is allocated. To allocate memory these steps are needed:

1. Resource of needed size is created
2. Resource’s requirements are then sent to the memory
3. Needed memory is allocated
4. Resource is then associated with allocated memory

And only after these steps data can be transferred to the resource.
Vulkan supports 6 types of shaders:

- Vertex
- Tessellation control
- Tessellation analysis
- Geometric
- Fragmented (also pixel)
- Computational
Windows and Displays

To show something in window or display – we need specific addings.
Working method looks nearly like this:

1. **We request index of the free image**
2. **Call command buffer, which will copy the result from the Framebuffer into this image**
3. **Send the command “send image” to the commands’ queue**
As a result, we will have two semaphore:

1. First gives info that image is available for copying
2. Second one will say that image is ready to be shown
Difference between Vulkan and OpenGL

### Ground-up Explicit API Redesign

<table>
<thead>
<tr>
<th>OpenGL</th>
<th>Vulkan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originally architected for graphics workstations with direct renderers and split memory</td>
<td>Matches architecture of modern platforms including mobile platforms with unified memory, tiled rendering</td>
</tr>
<tr>
<td>Driver does lots of work: state validation, dependency tracking, error checking. Limits and randomizes performance</td>
<td>Explicit API – the application has direct, predictable control over the operation of the GPU</td>
</tr>
<tr>
<td>Threading model doesn’t enable generation of graphics commands in parallel to command execution</td>
<td>Multi-core friendly with multiple command buffers that can be created in parallel</td>
</tr>
<tr>
<td>Syntax evolved over twenty years – complex API choices can obscure optimal performance path</td>
<td>Removing legacy requirements simplifies API design, reduces specification size and enables clear usage guidance</td>
</tr>
<tr>
<td>Shader language compiler built into driver. Only GLSL supported. Have to ship shader source</td>
<td>SPIR-V as compiler target simplifies driver and enables front-end language flexibility and reliability</td>
</tr>
<tr>
<td>Despite conformance testing developers must often handle implementation variability between vendors</td>
<td>Simpler API, common language front-ends, more rigorous testing increase cross vendor functional/performance portability</td>
</tr>
</tbody>
</table>
Difference between Vulkan and OpenGL

https://www.youtube.com/watch?v=rvCD9FaTKCA
Thank you for attention!