

# Vulkan

vulkan low level modern graphics api

- [Getting Started with Vulkan](#)
- [Vulkan & OpenGL Differences \(Shaded Triangle\)](#)

# Getting Started with Vulkan

Install Vulkan SDK:

<https://vulkan.lunarg.com/>

# Vulkan & OpenGL

## Differences (Shaded Triangle)

### 1. Explicit Control

- **Vulkan:** Provides explicit control over GPU resources and operations. You need to manage and allocate resources like memory, command buffers, and synchronization primitives directly.
- **OpenGL:** Abstracts much of this complexity. It handles resource management for you, making it easier for developers but less flexible for advanced use cases.

### 2. Command Buffers

- **Vulkan:** Uses command buffers to record rendering commands before submitting them to the GPU. You can record commands once and execute them multiple times, allowing for better performance optimization.
- **OpenGL:** Commands are issued immediately and not recorded for later execution. This can be less efficient, especially in complex rendering scenarios.

### 3. Multiple Queues

- **Vulkan:** Supports multiple queues for different operations (graphics, compute, transfer). You can use these queues in parallel to optimize performance.
- **OpenGL:** Generally operates on a single command queue, meaning that all rendering commands are submitted sequentially.

### 4. Pipeline Creation

- **Vulkan:** Requires explicit pipeline creation for each shader stage and configuration. This process can be cumbersome but allows for fine-tuned optimization and custom behavior.

- **OpenGL:** Simplifies pipeline management. You can bind shaders and set states with fewer API calls, which makes setup quicker and more straightforward.

## 5. Synchronization

- **Vulkan:** Provides detailed synchronization control using semaphores and fences. This allows you to manage resource access and rendering operations more precisely.
- **OpenGL:** Uses simpler synchronization mechanisms. It abstracts the synchronization process, which can lead to issues like implicit synchronization overhead.

## 6. Resource Binding

- **Vulkan:** Requires explicit binding of resources (like buffers and textures) to the pipeline, which can lead to better performance through optimization.
- **OpenGL:** Uses a more implicit model for resource binding, where resources can be bound and unbound more flexibly but can introduce overhead.

## 7. Shader Modules

- **Vulkan:** Utilizes shader modules that compile GLSL (or SPIR-V) into an intermediate representation. This approach provides more control over shader compilation and linking.
- **OpenGL:** Shaders are compiled and linked at runtime, which is easier but provides less flexibility in optimizing shader performance.

## 8. Render Passes

- **Vulkan:** Uses render passes to define the structure of rendering operations, allowing for more control over how framebuffer attachments are managed and used.
- **OpenGL:** Does not have an explicit concept of render passes; instead, it relies on simpler framebuffer attachments and operations.

## Summary

- **Vulkan** provides more control, flexibility, and optimization opportunities compared to OpenGL, but at the cost of complexity. This makes Vulkan better suited for high-performance applications, while OpenGL is often preferred for simpler applications due to its ease of use and abstraction.
- If you're setting up a shaded triangle in Vulkan, you'll need to manage more details, such as command buffer creation, resource binding, and synchronization, which are mostly

handled automatically by OpenGL.

# Shaded Triangle Steps

## OpenGL Steps

```
+-----+
|   OpenGL Application   |
| (Setup and Initialization) |
+-----+
|
| v
+-----+
|   Create Shader Program   |
| GLuint shaderProgram = glCreateProgram(); |
| glAttachShader(shaderProgram, vertexShader); |
| glAttachShader(shaderProgram, fragmentShader); |
| glLinkProgram(shaderProgram); |
+-----+
|
| v
+-----+
|   Setup Vertex Array Object   |
| GLuint VAO; |
| glGenVertexArrays(1, &VAO); |
| glBindVertexArray(VAO); |
| glGenBuffers(1, &VBO); |
| glBindBuffer(GL_ARRAY_BUFFER, VBO); |
| glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); |
| glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0); |
| glEnableVertexAttribArray(0); |
+-----+
|
| v
+-----+
|   Draw Call   |
```

```
|   glUseProgram(shaderProgram);   |
|   glBindVertexArray(VAO);       |
|   glDrawArrays(GL_TRIANGLES, 0, 3);|
```

```
+-----+
```

```
|
v
```

```
+-----+
```

```
|   Framebuffer   |
|   SwapBuffers(window);   |
```

```
+-----+
```

## Vulkan Steps

```
+-----+
```

```
|   Vulkan Application   |
|   (Setup and Initialization) |
```

```
+-----+
```

```
|
v
```

```
+-----+
```

```
|   Create Instance   |
|   vkCreateInstance(&instanceInfo, &instance); |
```

```
+-----+
```

```
|
v
```

```
+-----+
```

```
|   Create Device   |
|   vkCreateDevice(instance, &deviceCreateInfo, &device); |
```

```
+-----+
```

```
|
v
```

```
+-----+
```

```
|   Create Shader Modules   |
|   vkCreateShaderModule(device, &vertShaderInfo, &vertShaderModule); |
|   vkCreateShaderModule(device, &fragShaderInfo, &fragShaderModule); |
```

```
+-----+
```

```
|
v
```

```
+-----+
```

```

|   Create Graphics Pipeline   |
|   vkCreateGraphicsPipelines(device, &pipelineInfo, &pipeline); |
+-----+
|
|   v
+-----+
|   Create Vertex Buffer   |
|   vkCreateBuffer(device, &bufferCreateInfo, &vertexBuffer); |
|   vkBindBufferMemory(device, vertexBuffer, vertexBufferMemory); |
+-----+
|
|   v
+-----+
|   Allocate Command Buffer   |
|   vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer); |
+-----+
|
|   v
+-----+
|   Record Command Buffer   |
|   vkBeginCommandBuffer(commandBuffer, &beginInfo); |
|   vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline); |
|   vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets); |
|   vkCmdDraw(commandBuffer, 3, 1, 0, 0); |
|   vkEndCommandBuffer(commandBuffer); |
+-----+
|
|   v
+-----+
|   Submit Command Buffer   |
|   vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE); |
+-----+
|
|   v
+-----+
|   Render Pass   |
|   vkBeginRenderPass(commandBuffer, &renderPassInfo); |
+-----+
|
|   v

```

```
+-----+
|      Framebuffer      |
| vkEndRenderPass(commandBuffer); |
| vkQueuePresentKHR(presentQueue, &presentInfo); |
+-----+
```

## 1. Initialize Vulkan

- **Create a Vulkan Instance:**

- This is the starting point for any Vulkan application and contains information about the application and the Vulkan API version.
- Use `vkCreateInstance` to create a Vulkan instance.

## 2. Select a Physical Device

- **Enumerate Physical Devices:**

- Find available physical devices (GPUs) on the system.
- Use `vkEnumeratePhysicalDevices`

- **Select a Suitable Device:**

- Choose a physical device that supports the features and queues you need (like graphics, compute, etc.).

## 3. Create a Logical Device

- **Create Logical Device:**

- Create a logical device that allows your application to interact with the physical device. Specify the queue types needed, such as a graphics queue.
- Use `vkCreateDevice`

## 4. Create a Swap Chain

- **Choose Swap Chain Parameters:**

- Determine the swap chain's surface format, presentation mode, and extent.

- **Create Swap Chain:**

- Create the swap chain, which handles presenting images to the screen.
- Use `vkCreateSwapchainKHR`

## 5. Create Image Views

- **Create Image Views:**

- For each image in the swap chain, create an image view using `vkCreateImageView`. This allows Vulkan to access the images in the swap chain.

## 6. Create Render Pass



- **Define Render Pass:**
  - Create a render pass that defines how framebuffer attachments (color, depth, etc.) are used during rendering.
  - Use `vkCreateRenderPass`

## 7. Create Framebuffers

- **Create Framebuffers:**
  - For each image view in the swap chain, create a framebuffer using `vkCreateFramebuffer`. This links the image views with the render pass.

## 8. Create Shaders

- **Load Shader Code:**
  - Load the vertex and fragment shader code, typically written in GLSL or HLSL.
- **Create Shader Modules:**
  - Create shader modules for both the vertex and fragment shaders.
  - Use `vkCreateShaderModule`

## 9. Create Graphics Pipeline

- **Define Graphics Pipeline:**
  - Use `vkCreateGraphicsPipelines` to create the graphics pipeline.
  - This involves specifying the shader stages, fixed-function state (like viewport, rasterization, blending), and the render pass.

## 10. Create Vertex Buffer

- **Create Buffer:**
  - Create a vertex buffer that holds the triangle's vertex data (positions, colors, etc.).
  - Use `vkCreateBuffer`
- **Allocate Memory:**
  - Allocate memory for the vertex buffer using `vkAllocateMemory` and bind it using `vkBindBufferMemory`.
- **Copy Data:**
  - Map the memory, copy the vertex data into it, and unmap the memory.

## 11. Create Command Buffers

- **Allocate Command Buffers:**
  - Allocate command buffers for recording commands.
  - Use `vkAllocateCommandBuffers`

## 12. Record Commands

- **Begin Command Buffer:**

- Start recording commands in the command buffer.
  - Use `vkBeginCommandBuffer`
- **Begin Render Pass:**
  - Use `vkCmdBeginRenderPass`
- **Bind Graphics Pipeline:**
  - Use `vkCmdBindPipeline`
- **Bind Vertex Buffer:**
  - Use `vkCmdBindVertexBuffers`
- **Draw Command:**
  - Use `vkCmdDraw` to issue the draw command for the triangle.
- **End Render Pass:**
  - Use `vkCmdEndRenderPass` to finish the render pass.
- **End Command Buffer:**
  - Use `vkEndCommandBuffer` to finalize the command buffer.

## 13. Submit Command Buffer

- **Submit to Queue:**
  - Submit the recorded command buffer to the graphics queue for execution.
    - Use `vkQueueSubmit`

## 14. Presenting the Image

- **Present the Frame:**
  - Present the rendered image from the swap chain to the screen.
    - Use `vkQueuePresentKHR`

## 15. Cleanup

- **Cleanup Resources:**
  - Destroy resources in the reverse order of their creation, such as pipelines, framebuffers, swap chains, and the Vulkan instance.

### RESOURCES

<https://edw.is/learning-vulkan/#what-i-gained-from-switching-to-vulkan>