Graphics API

- OpenGL Graphics Pipeline
- Computer Graphics From Scratch
  - Raytracing
  - Rasterization
- Vulkan
  - Getting Started with Vulkan
  - Vulkan & OpenGL Differences (Shaded Triangle)

OpenGL Graphics Pipeline

Graphics Pipeline is an abstract model that describes sequence of steps needed to render a 3D scene.

asynchronous operation - The CPU sends rendering commands to the GPU, which then perform rendering operations while the CPU continues with other tasks

VRAM - Memory core on the GPU which stores buffers

- **image buffer** - Has back image buffer, where the GPU renders the scene, and the front image buffer, where the exact pixel data this is visible to the viewport. Buffer Swap is performed when the back image buffer is done rendering and ready to be displayed by the front image buffer.
- **Depth Buffer / Z- Buffer** - Stores values how far the pixel lies in every pixel im the image buffer. Used to perform hidden surface elimination, by allowing pixel drawn that if its depth is less than the pixel already in the image buffer
- **Stencil Buffer** -
- **Texture Maps** - Images applied to surface of an object. Can include othe maps such as bump map. Takes up largest amount of VRAM

0. Core Concepts and Vocabulary

rendering - Generate two-dimensional images of 3D scenes

shading - The darkness of an object not in direct light

shadows - the silhouette of one object's shape on the surface of another object

frustrum - Region contained within the truncated pyramid shape outlined in white indicates the space visible to the camera.

pixel - specify colors using triples of floating-point numbers between 0 and 1 to represent the amount of red, green, and blue light present in a color; a value of 0 represents no amount of that color is present, while a value of 1 represents that color at full intensity

- **raster** - rendered scene via an array of pixels (picture elements) which will be displayed on a screen, arranged in a 2D grid
- **resolution** - the number of pixels in the raster, the more it is the higher the quality
- **precision** - the number of bits used for each pixel as each bit has two possible values (0 or 1), the number of colors that can be displayed

buffer (data buffer/buffer memory) is a part of a computer's memory that serves as temporary storage for data while it is being moved from one location to another.

- **frame buffer** - Pixel data is stored in a region of memory. A framebuffer may contain multiple buffers that store diferent types of data for each pixel.
  - **color buffer** - located in frame buffer which stores RGB values. Need this at minimum. Alpha value can also be stored
  - **depth buffer** - located in frame buffer, which stores distances from points on scene objects to the virtual camera. Depth values are used to determine whether the various points on each object are in front of or behind other objects (from the camera’s perspective), and thus whether they will be visible when the scene is rendered.
  - **stencil buffer** - store values used in generating advanced effects, such as shadows, reffections, or portal rendering.

1. Application Stage

Initializing the window where the rendered graphics will be displayed.

- Reading data required for the rendering process and sending to the GPU, such as
  - **vertex attributes**, describes appearance of geometric shapes rendered, stored as in **vertex buffer objects** (VBO)
  - images to be applied to surfaces, stored in **texture buffers**
  - source code for **vertex shader** and **fragment shader** programs, sent to GPU to be complied and loaded.
- Loop that re-renders the scene repeatedly, like 60 fps
- Monitoring hardware for user inputs, handled by the CPU
- **Vertex Array Objects**, manages the associations and whether they are turned on and off, between attributes data stored in VBOs and attribute variables in the vertex shader program

2. Geometry Processing

Determining the final position of each vertex of the geometric shapes to be rendered, implemented by a program called the vertex shader

mesh - a collection of points (vertices) that are grouped into lines or triangles to make a shape of a geometric object

- **vertex** - a point with a data structure holding properties or attributes that are specific to rendering.
  - 3D position of the corresponding point. Mandatory
  - color to be used when rendering the point. Optional

    - **texture coordinates** (or UV coordinates) - indicates a point in an image that is mapped to the vertex. Optional
    - **normal vector** - indicates the direction perpendicular to a surface, used for lighting calculations. Optional

Vertex shader is applied to each of the vertices to determine the final position each point being rendered, which is typically calculated from a series of transformations:

- **model transformation** - the collection of points defining the intrinsic shape of an object may be translated, rotated, and scaled so that the object appears to have a particular location, orientation, and size with respect to a 3D world.
    - **world space** - coordinates expressed from this frame of reference are said to be in world space
    - **virtual camera** - camera with its own position and orientation in the virtual world.
    - **view transformation** - In order to render the world from the virtual camera's point of view, the coordinates of each object in the world must be converted to a frame of reference relative to the camera itself.
    - **view space** (camera/eye space) - coordinates after view transformation
        - **projection transformation -** clipping points outside the view space
            - **clip space** - points outside the view space
            - perspective projection
            - orthographic projection

In addition to these transformation calculations, the vertex shader may perform additional calculations and send additional information to the **fragment shader** as needed.

The end of geometry processing

3. Rasterization

geometric primitive - How the vertices is connected to produce a shape. OpenGL supports ten of these types, including Points, lines, or triangles, which consist of sets of 1, 2, or 3 points.

Rasterization - Process of filling in the horizontal spans of pixels belonging to a geometric primitive.

primitive assembly - process of grouping points to geometric primitives

Once the geometric primitives have been assembled, the next step is to determine which pixels correspond to the interior of each geometric primitive. Since pixels are discrete units, they will typically only approximate the continuous nature of a geometric shape, and a criterion must be given to clarify which pixels are in the interior.

Three simple criteria could be

1. The entire pixel area is contained within the shape
2. The center point of the pixel is contained within the shape
3. Any part of the pixel is contained within the shape

fragment - For each pixel corresponding to the interior of a shape

- raster position / pixel position - data stored in a fragment
  - **depth** - stored in fragment when in 3D. Needed when points on different geometric objects would overlap from the perspective of the viewer. When this happens, the associated fragments would correspond to the same pixel, and the depth value determines which fragment's data should be used when rendering this pixel
  - Additional data may be assigned to each vertex, such as a color, and passed along from the vertex shader to the fragment shader. In this case, a new data field is added to each fragment.
  - **interpolated** - from the values at the vertices: calculated using a weighted average, depending on the distance from the interior point to each vertex. Optional

4. Pixel Processing

This stage determinea the final color of each pixel, storing this data in the color buffer within the frame buffer.

During the first part, a program called the fragment shader is applied to each of the fragments to calculate their final color. This calculation involves various data stored in each fragment, in combination with global data available during rendering, such as

- base color applied to the entire shape
- colors stored in each fragment (interpolated from vertex colors)
- textures, where colors are sampled from locations specified by UV coordinates
- light sources, whose relative position and/or orientation may lighten or darken the color, depending on the direction the surface is facing at a point, specified by normal vector

The GPU handles the following:

- Depth values stored in each fragment are used to resolve visibility issues in a 3D scene, determining which parts of objects are blocked from view by other objects. After the color of a fragment has been calculated, the fragment's depth value will be compared to the value currently stored in the depth buffer at the corresponding pixel coordinates. If the fragment's depth value is smaller than the depth buffer value, then the corresponding point is closer to the viewer than any that were previously processed, and the fragment's color will be used to overwrite the data currently stored in the color buffer at the corresponding pixel coordinates.
- Transparency using alpha values stored in the color of each fragment, determining how much color blend with another color. All opaque objects must be rendered first (in any

order), followed by transparent objects ordered from farthest to closest with respect to the virtual camera.

Computer Graphics From Scratch

Raytracing

Rasterization

Color

Lines

1. draw line by using Interpolate to compute values of a linear function.

Filled Triangles

draw lines to make wireframe of a triangle using 3 2D Vertices with connecting points

1. Sort the points
2. compute the x coordinate of the triangle edges
3. concatenate the short sides
4. determine which is ledt annd which is right
5. draw the horizontal segments

Shaded Triangles

Vulkan

vulkan low level modern graphics api

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