Tampere University of Applied Sciences

3D graphics renderer for Web Applications using Web Assembly and WGPU

Minh Dang

BACHELOR'S THESIS
January 2024

Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Bachelor's Degree Programme in Software Engineering

DANG, MINH:
3D graphics renderer for Web Applications using Web Assembly and WGPU

Bachelor's thesis 42 pages, appendices 2 pages
January 2024

The primary objective of this thesis was to develop a new 3D graphics engine based on a cross-platform contemporary graphics API to replace recent web graphics solutions. Furthermore, it could be used as a starter for becoming a back-end renderer for desktop and mobile applications.

To achieve the outcomes, the engine was built on Rust and WGPU to take advantage of multi-platform support, memory safety, and a web-friendly ecosystem. The solution could be separated into two parts: an engine as the backend renderer and an application as the output presentation. The engine had been developed according to WGPU and WebGPU documentation. It has essential pipeline configurations and supports input buffers of vertex, uniform, and image.

From the application perspective, it could run on modern desktop operating systems (Window, Linux, and MacOS), and the Web via Web Assembly. Moreover, the window management (Winit) used in the application is the most advanced library in the market. For instance, it could support inputs from keyboard, mouse, and touchscreen. Hence, there was not a hindrance with camera controls when ported on different platforms. Furthermore, an asset importer has been added to the application. By having its support, numerous assets could be seamlessly rendered using the built-in asset importer.

At the beginning of the thesis, it started with an essential acknowledgment of Rust and WGPU. Although there was some experience with computer graphics, working with a new programming language and its memory-borrowing concept still created a significant impact on the thesis process. Besides the fact, the overall outcome was satisfied with the high compatibility of Web Frameworks and excellent performance when running on the Web.

Key words: WGPU, WebGPU, Winit, cross-platform, graphics, renderer

CONTENTS

1 INTRODUCTION 5
2 REQUIREMENTS 6
  2.1 Architecture 6
  2.2 Features 6
3 ACKNOWLEDGEMENT 8
  3.1 WGPU 8
  3.2 Rust 9
    3.2.1 Rust History 9
    3.2.2 Cargo 10
  3.3 Rendering Flow 11
  3.4 Graphics Pipelines 12
  3.5 Vertex Specification 14
  3.6 Textures 17
  3.7 Camera 21
    3.7.1 Orthographic and Perspective projection 21
    3.7.2 Arcball Camera Systems 23
  3.8 GLTF modelling format 26
4 IMPLEMENTATION 28
  4.1 Source code and library management 28
  4.2 Application Event Flow 29
  4.3 Control camera with mouse cursor position 30
  4.4 Fetching GLTF model on cloud storage 32
  4.5 Working with React Web Application 33
5 QUALITY ASSURANCE 36
  5.1 Renderdoc 36
6 DISCUSSION 37
REFERENCES 39
APPENDICES 41

ABBREVIATIONS AND TERMS

GPU	Graphics Processing Unit
iGPU	Integrated Graphics Processing Unit
dGPU	Dedicated Graphics Processing Unit
3D	Three-dimensional
API	Application Programming Interface
URL	Uniform Resource Locator
WASM	Web Assembly
OS	Operating System
WebGL	Web Graphics Library
OpenGL/OGL	Open Graphics Library
GLTF	Graphics Library Transmission Format
Metal	Apple Graphics Library
Direct3D	Microsoft Graphics Library
UV	Texture Coordinate
FOVY	Vertical Field of View
VRAM	Video Random-Access Memory
UBO	Uniform Buffer Object
VBO	Vertex Buffer Object
JSON	JavaScript Object Notation
NPM	JavaScript Package Manager

1 INTRODUCTION

In recent years, the demand for high-performance and cross-platform 3D Graphics Engines continues to grow rapidly for applications such as video games, virtual simulations, and metaverse concepts. Consequently, a graphics library for multiple platforms becomes imperative. However, the current high-level 3D solutions for web development have not adapted to the mentioned trend. By mostly using a rendering backend based on WebGL API, applications can only commute with the hardware via an old low-level Graphics API called OpenGL. Hence, they usually serve deficient performance, and low efficiency with a lack of new features. Fortunately, a state-of-the-art technology called WebGPU has been promised a bright future by replacing old technologies on the market.

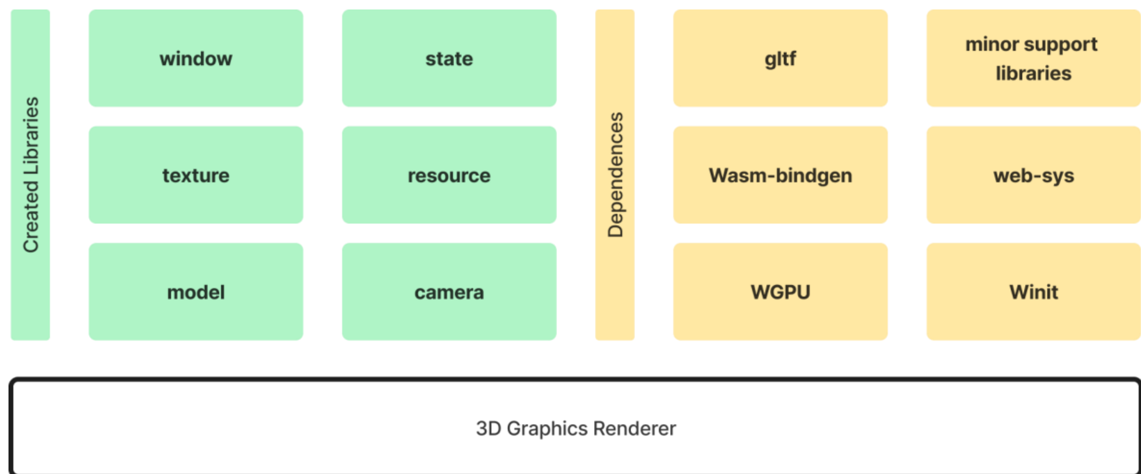
This thesis focuses on the creation of a 3D rendering viewer running on the web service (built by React web framework). It will apply some essential rendering techniques to provide a robust and versatile solution for users aiming to observe immersive 3D assets. Therefore, the primary objective was not to use any enterprise graphics engines such as Unity or Unreal Engine, but to build a basic 3D engine on top of WGPU graphics API.

The motivation for this project was to develop a new 3D renderer for web applications, which replaces the popular 3D Viewer based on old technologies. The thesis will cover features, including all essential techniques, such as basic vertex rendering, an arc ball camera system, and fetching remote GLTF model files via an URL.

Developing in this manner presents inevitable challenges, such as a lack of support for development tools and the absence of a preview scene like what is available on commercial engines.

2 REQUIREMENTS

2.1 Architecture



PICTURE 1. Software architecture

In picture 1, the graphics renderer would be built depending on WGPU which is a native core of WebGPU. It supports the renderer to construct the graphics pipeline using numerous modern graphics APIs. To handle hardware inputs and manage system windows, the author decided to use Winit which fully supports Rust. About the web migration, wasm-bindgen and web-sys are good cargo packages that help the project to be built to web target effortlessly. Finally, the GLTF package is a support library for importing GLTF-format files. Besides the dependencies, the project also includes built-in support libraries to control data flows, rendering resources, and object systems.

2.2 Features

The project will be separated into two parts: the engine and a web application for presentation. In theory, the graphics pipeline operations would be started in WGPU internal code. Hence, the primary purposes of the engine are processing asset data (meshes and textures), configuring the pipeline, handling inputs from physical gestures, and controlling presenting view angles.

The renderer should have a reliable performance with low latency. It should have good logic for view control. Moreover, there is no issue with rendering models

and texture mapping. Finally, the support for web frameworks needs to be initiated via binding project source code to a Web Assembly package.

3 ACKNOWLEDGEMENT

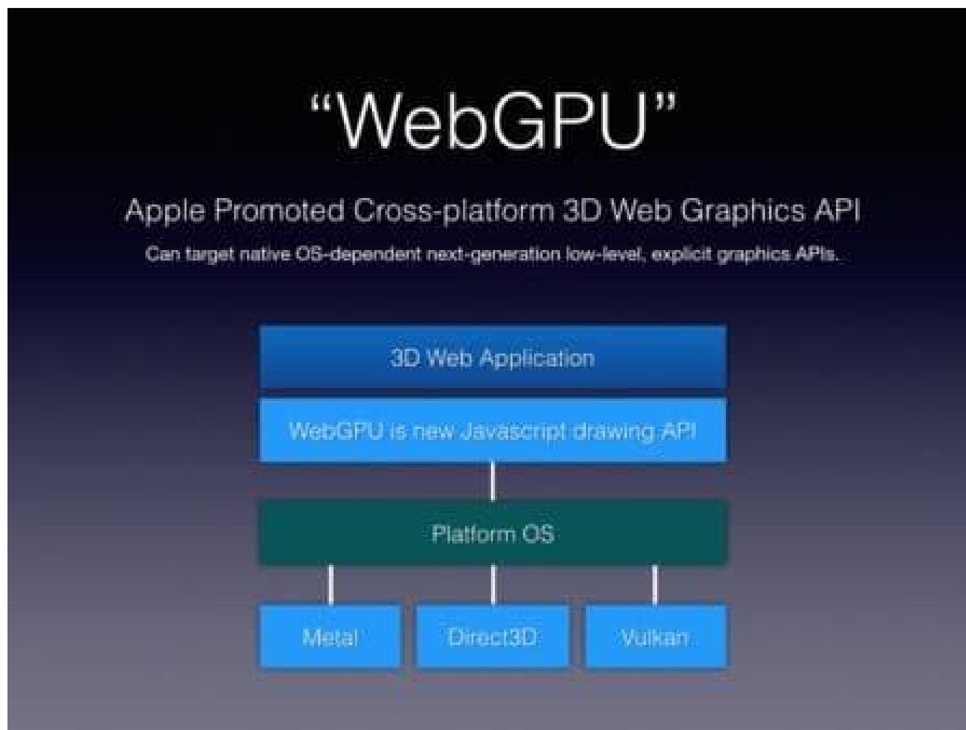
3.1 WGPU

In general, graphics hardware needs to be controlled by data and configuration. Then, the process of GPU will present frames on a display. There are numerous APIs and workloads behind the curtain, however, the ones taking the responsibility to control GPU rendering are graphics APIs. For instance, OpenGL, WebGL, and Direct3D have been standards for graphics development in the early 21st century and recently, some noticeable ones are coming with state-of-the-art technologies, such as Metal, WebGPU, and Vulkan could change the game graphics industry.

Working with graphics APIs requires tremendous effort and time for configuration and programming, especially Vulkan which is too complex for the lack of experienced developers.

Came from the inspiration of creating a Graphics API that works on any system and builds fully on Rust. The WGPU was born, and it started to be a new 3D standard for desktop, mobile, and web applications.

PICTURE 2. WebGL API (Frausto-Robledo, A. (2017, February 24))



PICTURE 3. WebGPU API (Frausto-Robledo, A. (2017, February 24))

Before WebGPU was created, all 3D graphics web views were created using WebGL API. As can be seen clearly, WebGL has been useful for accessing low-level graphics API, in this case, it is OpenGL. However, many new graphics APIs were inevitably created recently. Hence, it made WebGL and OpenGL is not enough anymore. Therefore, WebGPU API presents a new method for accessing contemporary low-level graphics API, such as Metal, Direct 3D, and Vulkan. As a result, developers can now create complex rendered images in the browser with new advanced features. (Next-generation 3D graphics on the web. (2017, April 5))

3.2 Rust

3.2.1 Rust History

Rust was found as a personal project by Graydon Hoare (an employee at Mozilla Research) in 2006. Then, Mozilla began sponsoring the project in 2009. The modern Rust compiler was successfully compiled in 2011 and the first stable release of the language was in May 2015. Rust has been used in numerous companies

including Amazon, Discord, Dropbox, Google, Meta, and Microsoft. (Thompson, C. (2023, February 15))

The reason Rust has excelled rapidly nowadays is due to its benefit of memory safety. For instance, Rust's ownership system and borrow checker force developers to follow strict principles of memory handling. Without a garbage collector for memory management, Rust can run extremely fast and avoid bugs coming from null pointers like on C and C++.

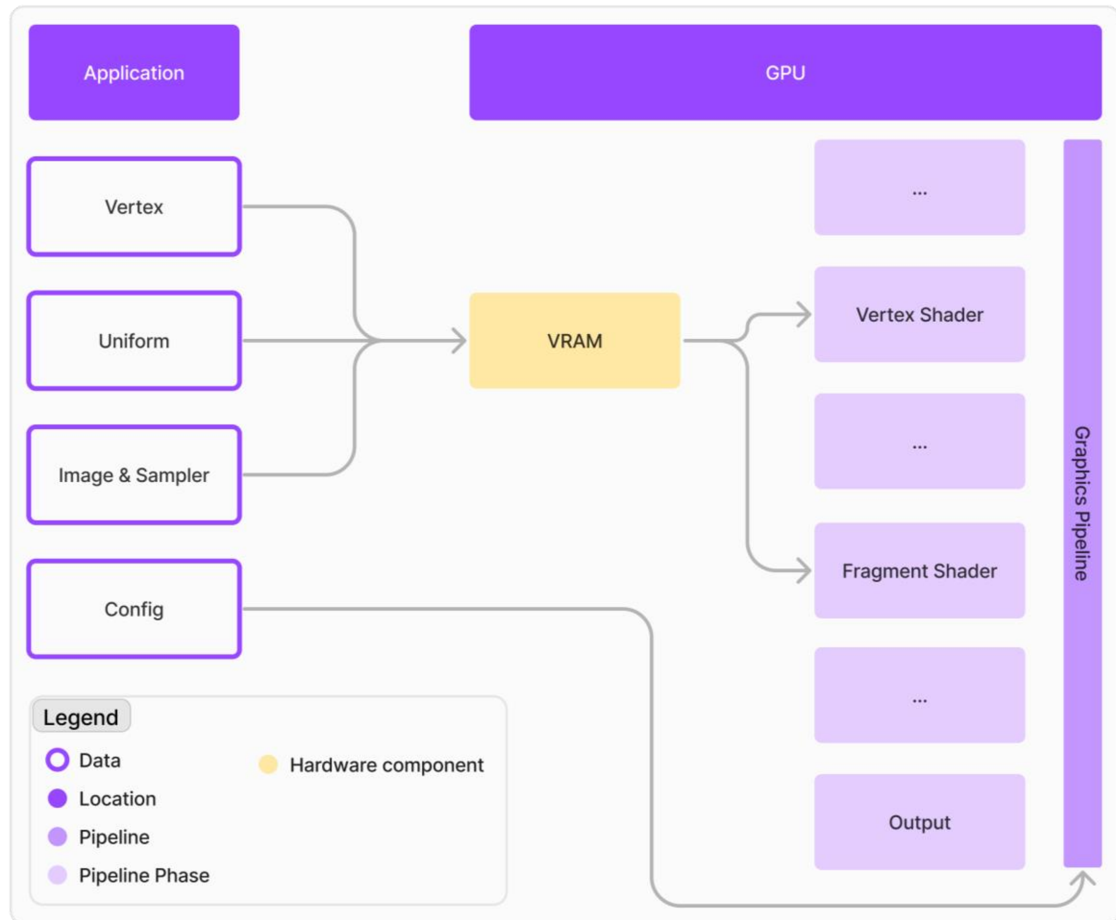
3.2.2 Cargo

Cargo is Rust package management. Like the famous Node Package Management, it controls package dependencies in a Rust project by downloading and compiling those into a suitable architect-developing environment. It requires creating a "Cargo.toml" file to configure a rust project.

```
// Cargo.toml
[package]
name = "core-renderer"
version = "0.0.3"

# Rust Version
edition = "2021"

[dependencies]
winit = "0.28"
wgpu = "0.18"
```

FIGURE 4. An example of a Cargo.toml file

3.3 Rendering Flow



PICTURE 5. An Illustration of Rendering flows

In theory, developers cannot program and access GPU directly using normal development processes. It can be explained that the GPU uses a different memory called VRAM. Hence, they need to follow some robust steps before making the hardware rendering images. For instance, the data needed to be gathered and processed. Then, it will need to be stored in the VRAM. There are data structures supported in most modern graphics libraries:

- Vertex Data includes values of Point Position, Color, or UV.
- Uniform Data includes sets of uniform attributes. For instance, it declares a structure including groups of data types which is handy for working with custom-structed data working as an object constructor.
- Image and Sampler are used for bitmap or image data.

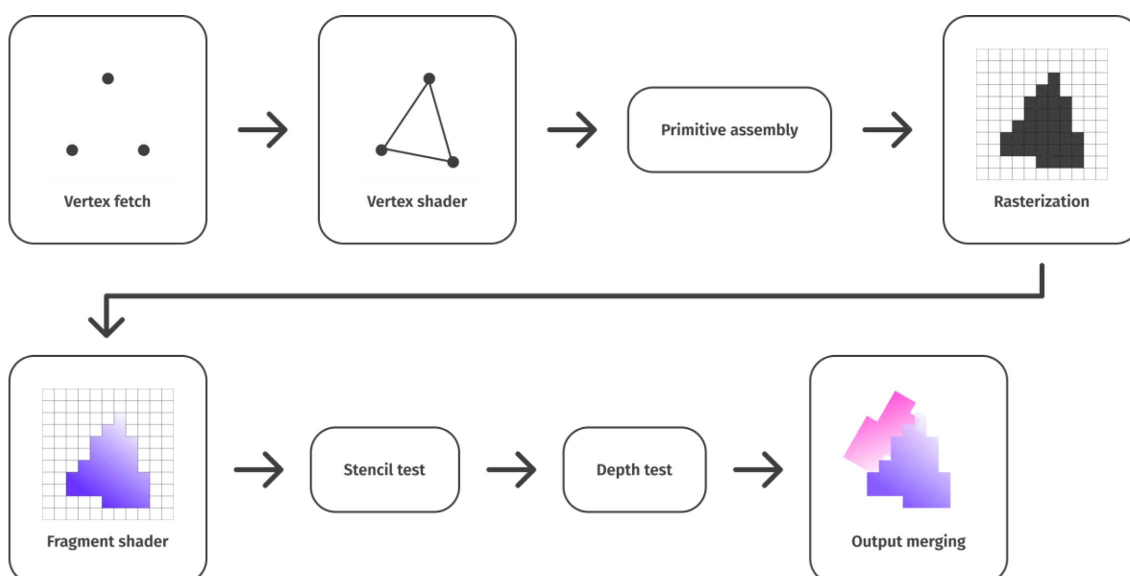
After finishing binding work, the GPU can finally collect data from two programmable phases which are vertex shader and fragment shader. Unlike the mentioned

phase, most others cannot be programmed. Fortunately, they can be adjusted by sets of configurations and descriptions. If all requirements are satisfied, the pipeline should run smoothly and return a complete image as a buffer.

3.4 Graphics Pipelines

In theory, a graphics pipeline is a workflow in a rendering process that creates a framebuffer in the finish. The pipeline structure is shared to all graphics APIs on the market. It can be explained easily that the pipeline serves as a path or an instruction which reflects separate ways of how GPU can render a frame to a screen.

As a result, developers can manipulate the process by adding data as buffers.



PICTURE 6. WGPU Pipeline. (Gfx-Rs)

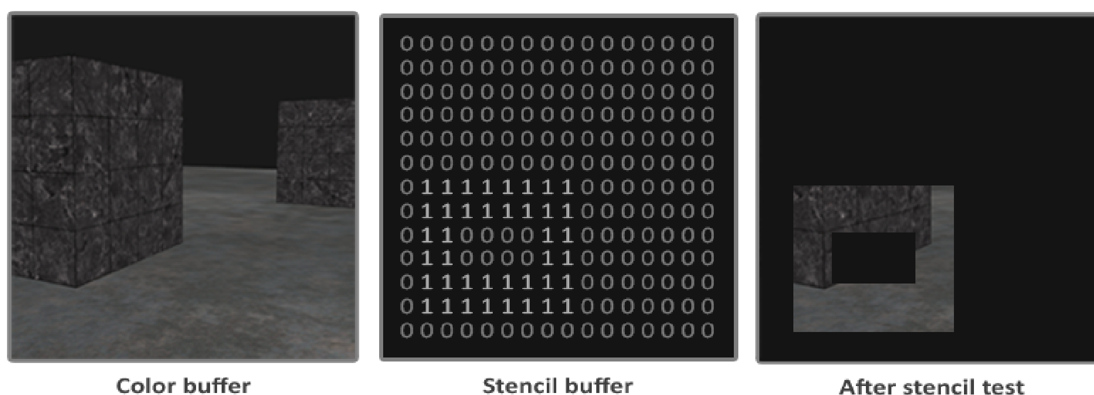
In WGPU, the pipeline could be divided into seven phases, vertex fetching, vertex shader processing, primitive assembly, rasterization, fragment shader processing, stencil-depth tests, and output merging. Only vertex shader and fragment shader could be programmed from the application side. Other phases support passing configurations for options regarding WGPU features.

As a starting point, in vertex fetch, a set of positions will be stored in the memory with a buffer address. It will send buffers and addresses to the vertex shader

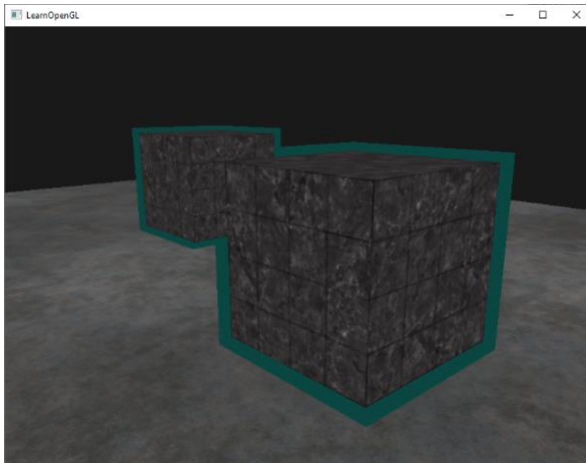
phase. With the data from VRAM, the phase will define a calculation for space positions of vertices. Fortunately, this phase could be programmed by a shading programming language, hence, it could be helpful if developers want to pass some additional attributes into the calculation, such as camera or lighting positions. In primitive assembly, it converts a vertex stream into a sequence of base primitives.

After processing raw vertices data, the pipeline still requires one more mandatory process, which is fragment shader. In addition, it displays fragments in a colorful texture or canvas. However, before going to the fragment shader, the primitives need to be broken down into fragments thanks to Rasterization, a connection phase.

As a result, two more processes need to be executed, which are testing and output merging. In the testing process, tests of stencil and depth are usually mandatory in most graphics' pipelines, additionally, it will also include a scissor test if clipping fragments outside a rectangular area is required. For instance, the stencil test will remove fragments that are disabled according to a stencil buffer. It is useful when outlining an object.

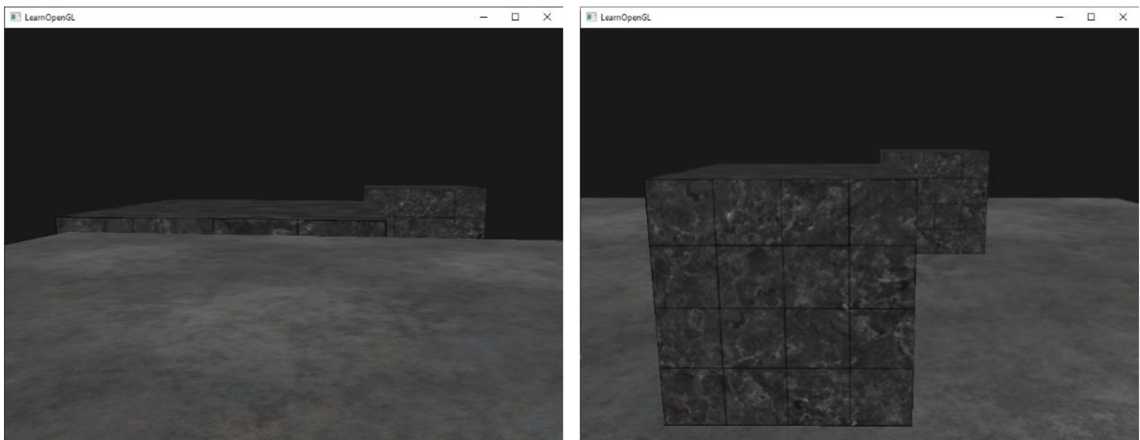


PICTURE 7. Example of Stencil testing (LearnOpenGL - Stencil testing)



PICTURE 8. Example of Outlining objects (LearnOpenGL - Stencil testing)

In 3D rendering, the depth test is always added due to removing overlapped artifacts which must render behind other ones. Finally, if all tests pass, the fragments will be combined with other results to present on a surface.



PICTURE 9. Example of Depth test (LearnOpenGL – Depth Test)

3.5 Vertex Specification

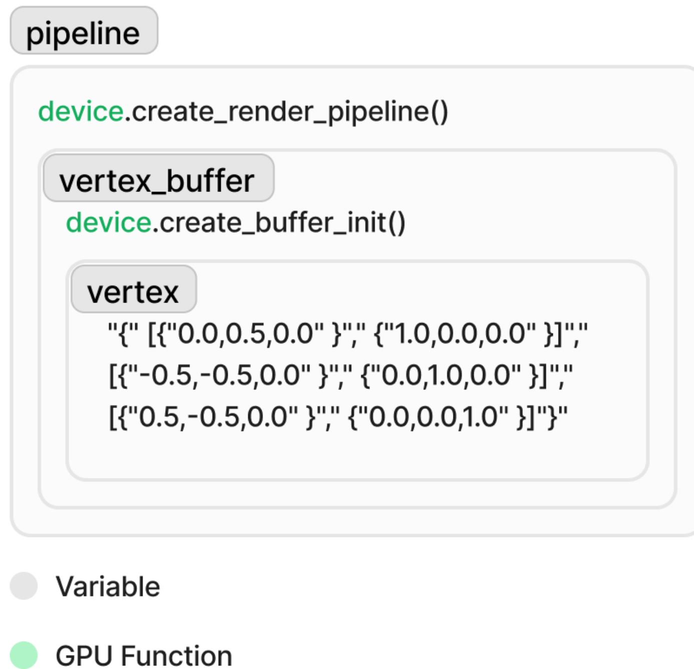
In theory, a chunk of data can be passed to the pipeline via the vertex shader. Furthermore, the way for the GPU to understand which data in memory is used for rendering depends on the stream of vertices. It could include the position of points, normal values, or even texture coordinates. For instance, this is how to draw a triangle with specific color inputs.

{[{0.0, 0.5, 0.0}, {1.0, 0.0, 0.0}],

[{-0.5, -0.5, 0.0}, {0.0, 1.0, 0.0}],

[{0.5, -0.5, 0.0}, {0.0, 0.0, 1.0}]}

Each object includes three attributes for presenting x, y, and z positions in 3D space and three attributes for RGB float-type color values starting from 0.0 to 1.0 equivalent to eight-bit color (0 to 255).



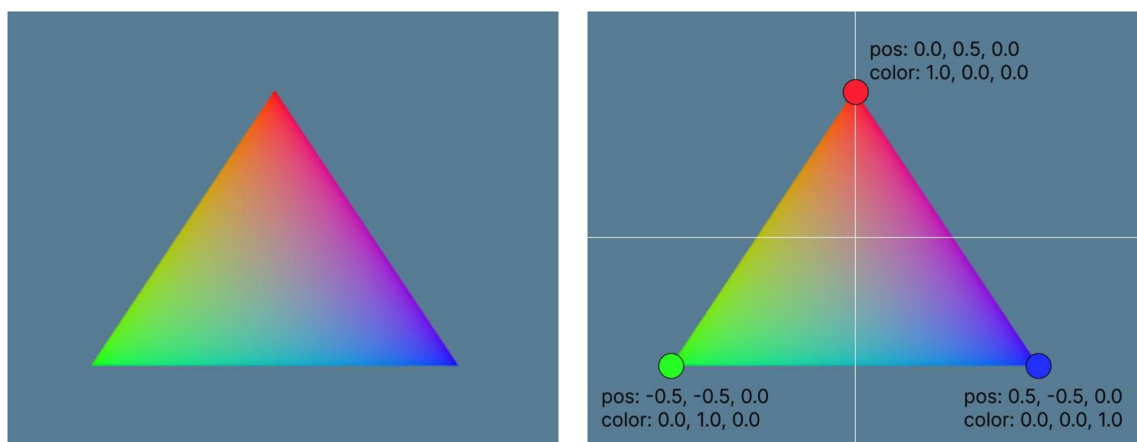
PICTURE 10. Creating a vertex buffer.

In WGPU, the implementation of vertex input requires small preparations beforehand. Primarily, the vertex attribute should be initiated and created via the "device.create_buffer_init()" method. Furthermore, the pipeline needs to be noticed by adding a new buffer layout description. Those operations needed to be ready before starting the pipeline. They can be prepared in the graphics engine, by calling WGPU-supported functions regarding buffer binding. Finally, the result should present a shape depending on the input vertices with a basic vertex shader.

```
// Vertex shader
struct VertexInput {
    @location(0) position: vec3<f32>,
    @location(1) color: vec3<f32>,
};

struct VertexOutput {
    @builtin(position) clip_position: vec4<f32>,
    @location(0) color: vec3<f32>,
};

@vertex
fn vs_main(
    model: VertexInput,
) -> VertexOutput {
    var out: VertexOutput;
    out.color = model.color;
    out.clip_position = vec4<f32>(model.position, 1.0);
    return out;
}

// Fragment shader
@fragment
fn fs_main(in: VertexOutput) -> @location(0) vec4<f32> {
    return vec4<f32>(in.color, 1.0);
}
```

PICTURE 11. Shader script used for drawing vertices from buffers.

In picture 11, there are two scripts for vertex and fragment shader. With the scripts, those shaders can be programmed from the engine side. For instance, the vertex shader gets vertex buffers including position and color for each vertex. They are already processed carefully from the engine side. It just needs to gather those data in the correct locations according to earlier binding layouts. After processing memory values from vertex buffers, it will pass them into the rasterization phase before turning into blank fragments. Finally, the fragment shader will return the correct color of each pixel to the drawing surface.



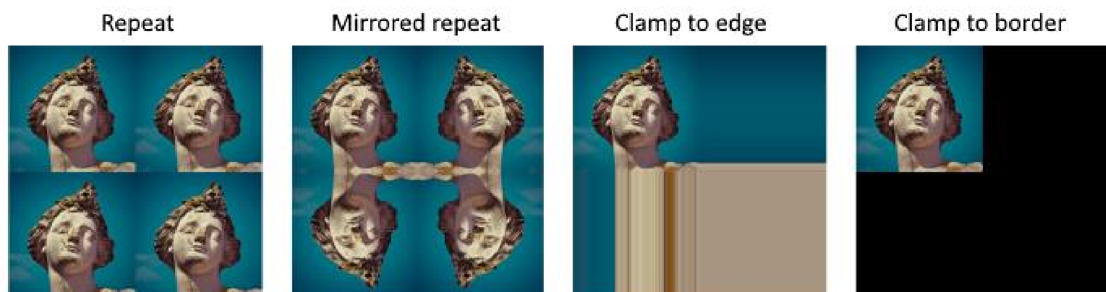
PICTURE 12. A result of a simple triangle primitive. (captured from the project engine)

In picture 12, the color of each pixel has been filled according to the position and color blending between points. Color blending in fragment shader is an internal feature in the graphics pipeline that is supported in all graphics APIs on the market.

3.6 Textures

Nowadays, supporting a feature to map texture including image and bitmap has been mandatory in most contemporary graphics engines. In WGPU, mapping textures seems straightforward. First, a texture needs to be loaded on the memory then it can be reused as bytes. It could be in any format, such as JPEG, PNG, or even Bitmap. Afterward, the image could also be converted into an array of pixels with RGBA values. Secondly, a primary device could create a Texture format according to the mentioned values via the “device.create_texture()” method. Finally, the texture needs to be written into the system queue, including, an image texture, image size, and pixel data. It helps the system understand how to copy data precisely.

To make GPU understand the imported data from WGPU, it needs to have a texture view and a sampler. Normally, the texture views have alike settings, then it just needs to be set as the default description. However, the samplers take a bigger role in the texture mapping purpose. Besides image-format data, the process also needs data to coordinate the mapping area. If the texture coordinate is outside the image, it will need some options to fill the blank pixels. For instance, the address mode offers four options including ClampToEdge, Repeat, MirrorRepeat, and ClampToBorder. (Overvoorde, A)

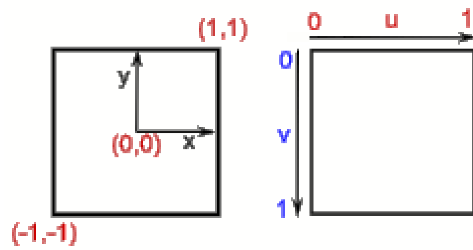


PICTURE 13. Examples of Address Mode (Overvoorde, A).

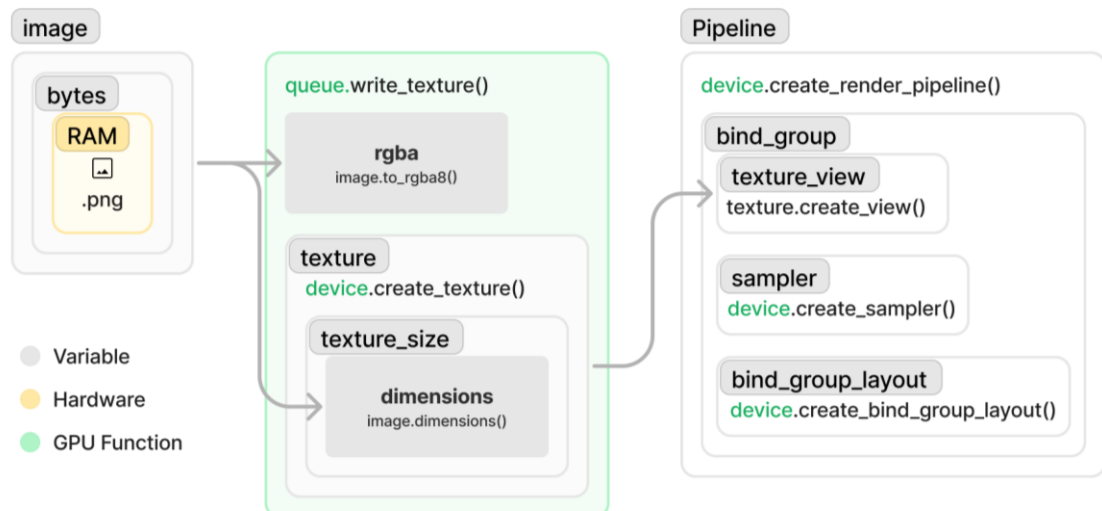
To make it easier to understand, let's start mapping a picture on a rectangular surface. In this example, it needs to have vertex positions and texture coordinates.

$$\{[\{1.0, 1.0, 0.0\}, \{1.0, 0.0\}],$$
$$[\{1.0, -1.0, 0.0\}, \{1.0, 1.0\}],$$
$$[\{-1.0, -1.0, 0.0\}, \{0.0, 1.0\}]\}$$
$$[\{-1.0, 1.0, 0.0\}, \{0.0, 0.0\}]\}$$

Each object includes three attributes for presenting x, y, and z positions in 3D space and two attributes for float-type texture coordinates starting from 0.0 to 1.0 equivalent to eight-bit color (0 to 255).



PICTURE 14. Coordinate Systems in WGPU – left one is rendering position and right one is texture mapping position. (Gfx-Rs)



PICTURE 15. Setup and configuration for texture mapping via WGPU related functions.

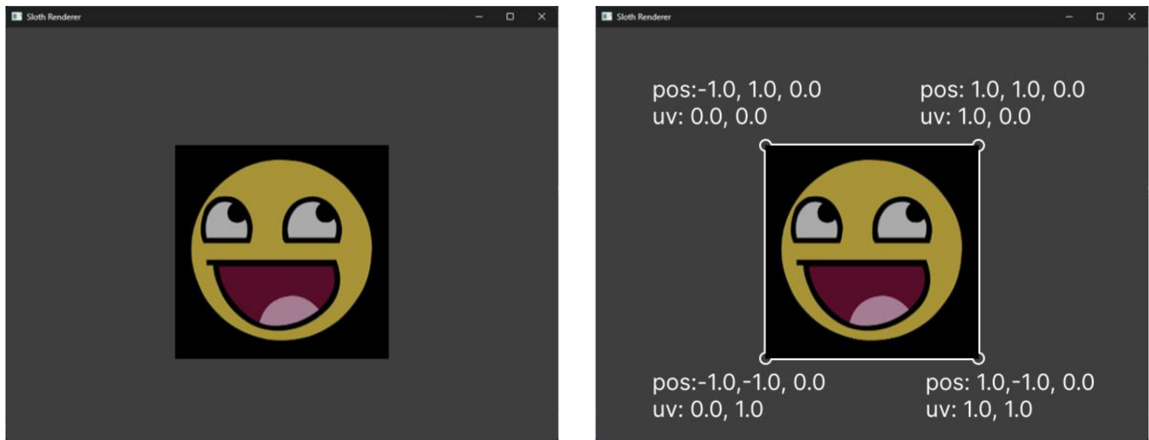
In picture 15, the pipeline needs an extra texture binding group with a texture binding group layout which will be placed in the Pipeline description. They can be

done easily by calling methods from the device and queue which relates to GPU functions. Like vertex buffer, texture mapping resources needed to be ready before running the pipeline. They can be prepared in graphics engine, by calling WGPU supported functions regarding texture mapping. With a suitable shader script, it can be seen clearly that the texture has been mapped correctly on the primitives.

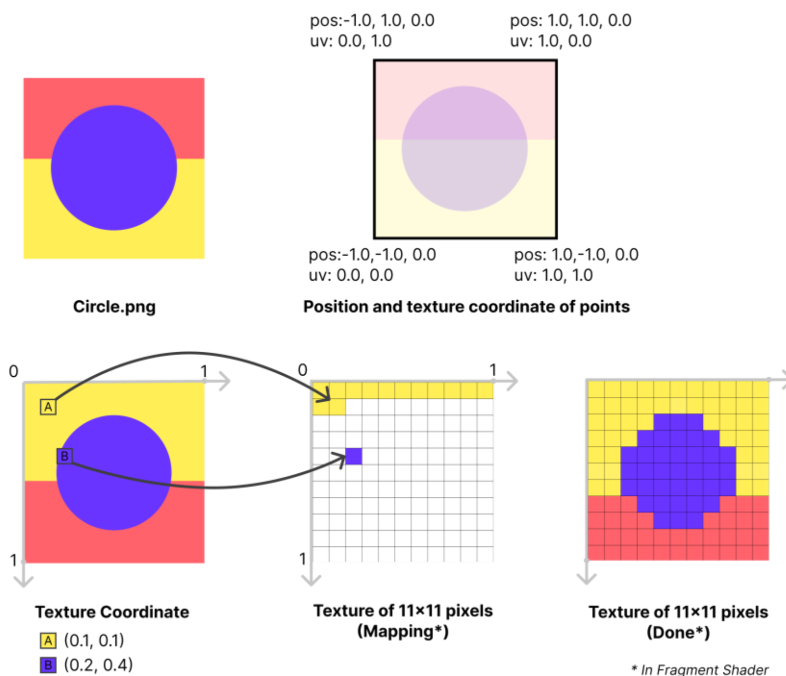
```
struct VertexInput {
    @location(0) position: vec3<f32>,
    @location(1) tex_coords: vec2<f32>,
}

struct VertexOutput {
    @builtin(position) clip_position: vec4<f32>,
    @location(0) tex_coords: vec2<f32>,
}

@vertex
fn vs_main(
    model: VertexInput,
) -> VertexOutput {
    var out: VertexOutput;
    out.tex_coords = model.tex_coords;
    out.clip_position = vec4<f32>(model.position, 1.0);
    return out;
}

@group(0) @binding(0)
var t_diffuse: texture_2d<f32>;
@group(0) @binding(1)
var s_diffuse: sampler;

@fragment
fn fs_main(in: VertexOutput) -> @location(0) vec4<f32> {
    return textureSample(t_diffuse, s_diffuse, in.tex_coords);
}
```

PICTURE 16. Shader script has texture support.

In picture 16, the vertex shader pushed out the new texture coordinate value to the fragment shader. With the new data, fragment color can be calculated according to texture with sampler and coordinate values. As a result, each pixel of texture will be blended on the rectangular primitive.



PICTURE 17. A result of a simple texture mapping. (captured from the project engine)



PICTURE 18. How image is mapping on a surface texture in fragment shader.

In picture 18, the image has a red part on the top. Normally, the texture coordinates (UV) of four points would be like in picture 16. However, in this case, they are swapped upside down. Then, the pipeline maps a 180-rotated version of the original image. The result will present a red part on the bottom.

3.7 Camera

3.7.1 Orthographic and Perspective projection

From a camera perspective, there are two unique projection matrices. They are Orthographic and Perspective. Each is supposed to work on a niche style of view. For instance, an orthographic projection matrix is described as a cube-style frustum box that presents primitives or parts of primitives inside a cube space, other ones will be clipped.

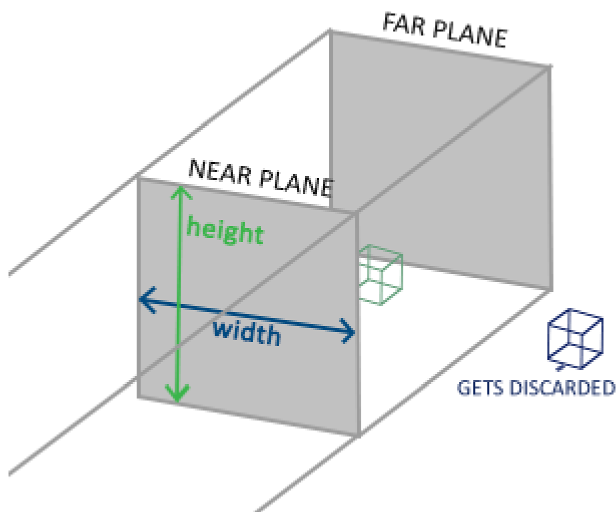


FIGURE 19. The orthographic frustum (LearnOpenGL - Coordinate Systems)

On the other hand, perspective view is more noticeable in 3D rendering due to the relation of depth and object scale. For instance, with trapezoidal prism frustum, it can work like a mechanical camera with rear and front parts.

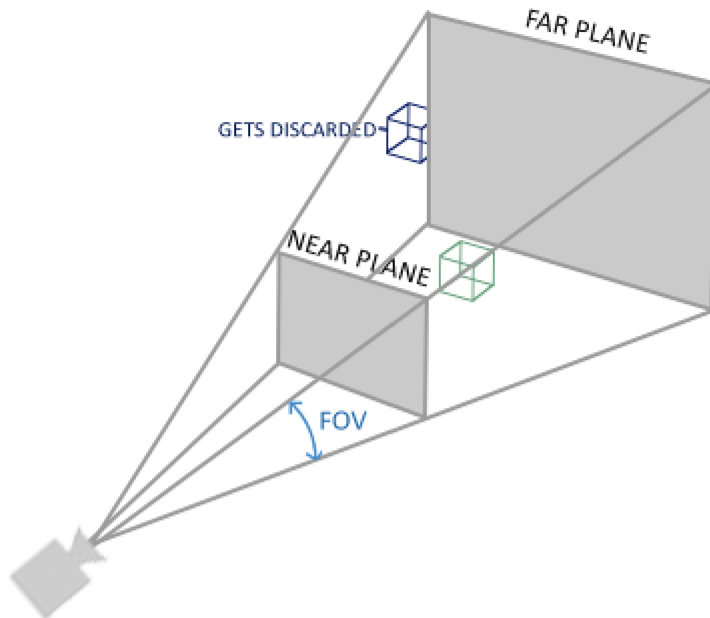


FIGURE 20. The perspective frustum (LearnOpenGL - Coordinate Systems)

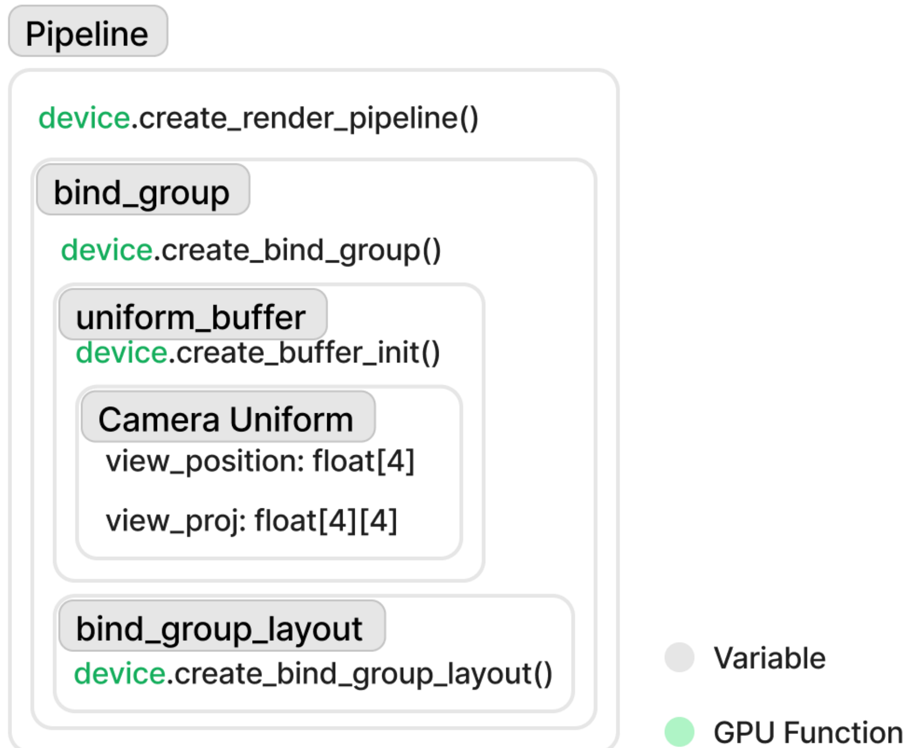
To calculate the camera matrix from a perspective projection, there are four elements including the vertical field of depth (FOVY), Aspect ratio, Near, and Far which impact on perspective view.

- The FOVY is the value that impacts the view angle of the camera on the Y axis.
- The Aspect ratio is equal to a division of width and height of view.
- The Near value is the length of the near plane on the Z axis.
- The Far value is the length of a far plane on the Z axis.

```
cgmath::perspective(Deg(self.fovy), self.aspect, self.znear, self.zfar);
```

PICTURE 21. calculating perspective matrix in WGPU from camera attributes.

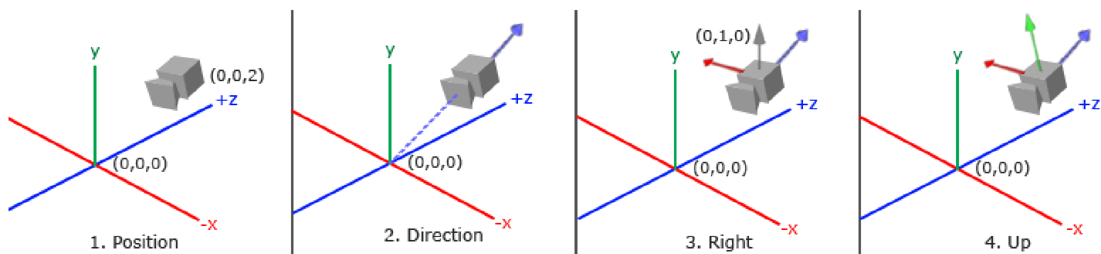
It can be seen clearly that the matrix needs to be served differently than other data types. Due to the volatility of the mentioned matrices, it should be technically used for passing specific types, such as matrices, structure of types, and so on. Fortunately, as mentioned before, there is a technique called uniform buffer object which provides user-defined data to the shader. For instance, to serve a pipeline with a Camera Uniform, the uniform buffer needs to be initiated with uniform usage. Afterward, like other types of buffers, it must be bound before providing them into a suitable pipeline.



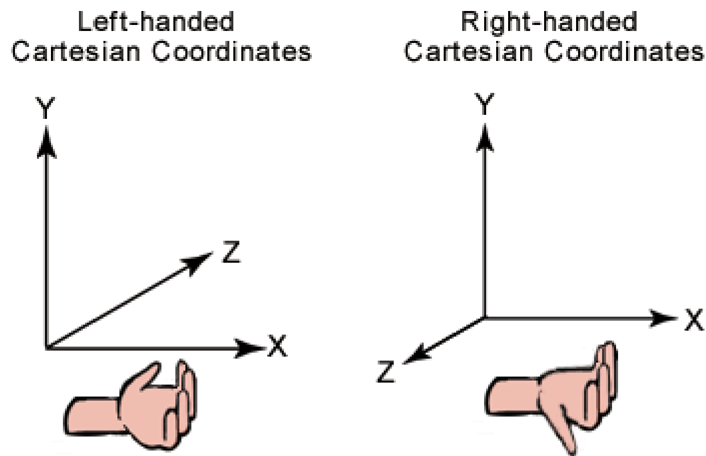
PICTURE 22. Camera Uniform Structure.

3.7.2 Arcball Camera Systems

Because of the purpose of 3D rendering, the camera should be calculated into a perspective matrix in this project. Additionally, it will be controlled by an Arcball camera system which calculates position in a spherical route. The camera will be controlled by mouse input.



PICTURE 23. Camera view attributes using Euler angles and right-handed Cartesian coordinate system. (LearnOpenGL - Camera)



PICTURE 24. Cartesian coordinate systems (Stevewhims)

In this thesis, the camera motion needs to qualify two factors - position and rotation angles. Therefore, it should move in a spherical area and be always looking at the origin point. In rotation perspective, the camera as an object needs to rotate itself around up and right vectors whenever it is moving around the target Y axis and X axis respectively (Eberly, D. 1999, December 1). Hence, it always looks at the original point. Fortunately, the math library used in this project also includes the method for calculating a matrix when a vector is looking at a point.

```
cgmath::Matrix4::look_at_rh(camera_position, target_point, up_vector);
```

PICTURE 25. “Look at” method for right-handed coordinate.

On the other hand, the position camera needs to be calculated actively during the engine runtime. Furthermore, Euler angles take a huge advantage in calculating the position of an orbital point around the origin point. A basic camera should have four attributes including one position point, three vectors of direction, right, and up. However, it does not need to save all four attributes and they can be calculated by eye, target, and up. For instance, the position point is the eye point, the direction is equal to the subtraction of target and eye values, and finally, the right value can be calculated by the cross product of up and the view direction. (Marie. (2019, November 30))

$$\vec{V} = normalize(P_{target} - P_{camera})$$

$\vec{V}$ is the vector of direction, P_{camera} is the eye position of the camera in 3D space, and P_{target} is the point the camera is looking at in this case is 0.0, 0.0, 0.0.

$$\vec{R} = normalize(\vec{V} \times \vec{U})$$

$\vec{R}$, $\vec{V}$, and $\vec{U}$ are vectors of right, direction, and up respectively.

First, the position and pivot vectors of the camera need to be calculated, those attributes are the position (eye) point and target point.

$$position = \begin{bmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ 1 \end{bmatrix}, pivot = \begin{bmatrix} x_{target} \\ y_{target} \\ z_{target} \\ 1 \end{bmatrix}$$

$$\theta_x = (horizontal\ position_{current} - \ horizontal\ position_{last}) \cdot \frac{2\pi}{width}$$

$$\theta_y = (vertical\ position_{current} - \ vertical\ position_{last}) \cdot \frac{\pi}{height}$$

θ_x and θ_y are theta values of vertical and horizontal rotating angles.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos\theta \cdot x + \sin\theta \cdot z \\ \mathrm{y} \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \cos\theta \cdot x + \sin\theta \cdot z \\ \mathrm{z} \\ 1 \end{pmatrix}$$

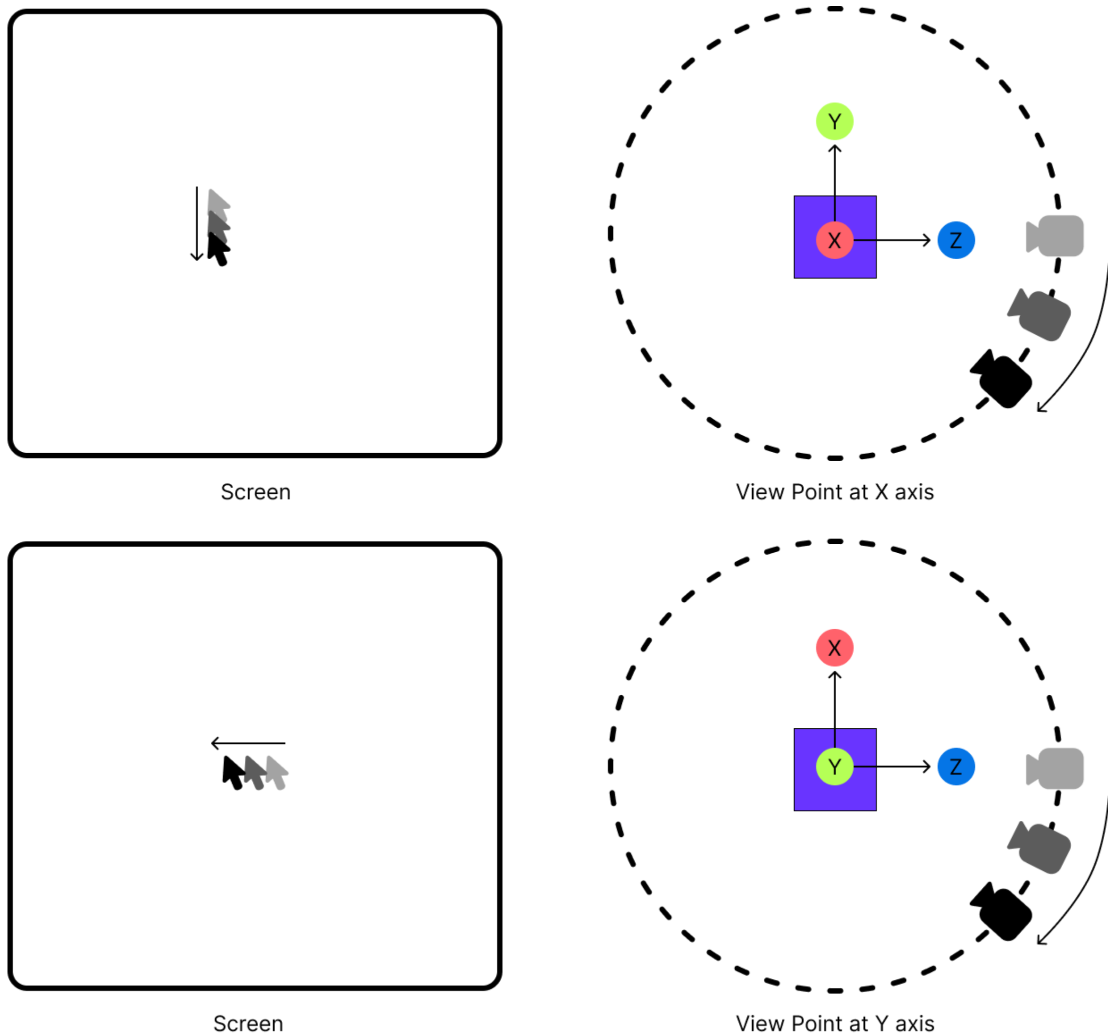
R_x, R_y, and R_z are rotation matrices around X axis, Y axis, and Z axis, respectively. (LearnOpenGL - Camera)

$$\mathrm{position} = R_x \cdot (\mathrm{position} - \mathrm{pivot}) + \mathrm{pivot}$$

$$\mathrm{final_pos} = R_y \cdot (\mathrm{position} - \mathrm{pivot}) + \mathrm{pivot}$$

Second, the first rotating position will be solved by multiplying the X rotation matrix and subtraction (position and pivot), then adding that result with the pivot value.

Finally, the second rotating position is equal to the multiply of the Y rotation matrix and subtraction of (position and pivot). The final position is completed by adding them to the last result.



PICTURE 26. Camera position when user drags mouse in vertical direction and horizontal direction.

3.8 GLTF modelling format

In theory, the structure of a GLTF file contains one JSON file (.gltf), one binary file (.bin), and some texture image files. The JSON files define the address and length of each data buffer in the Binary file.

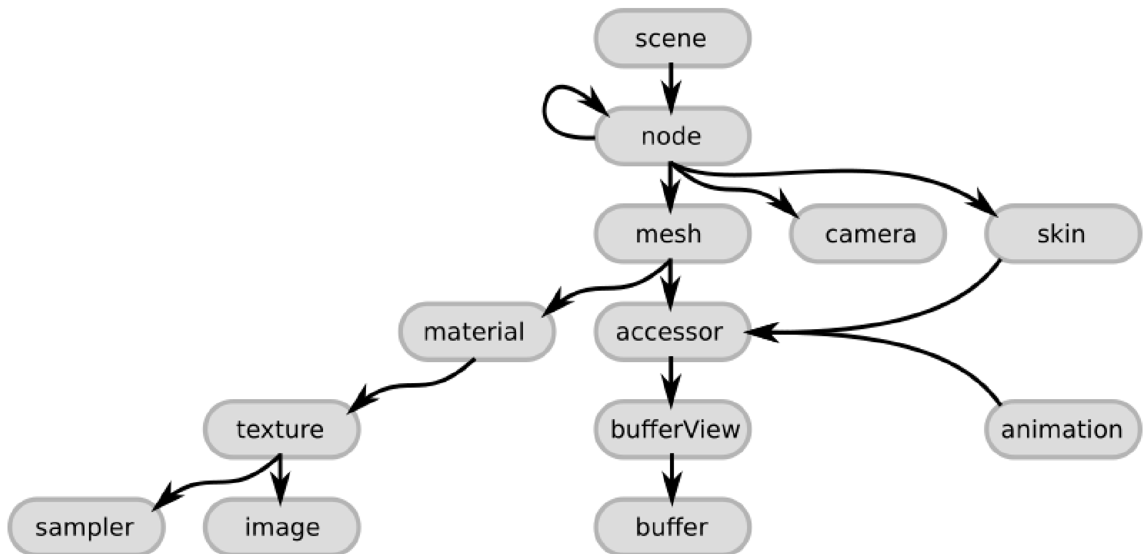
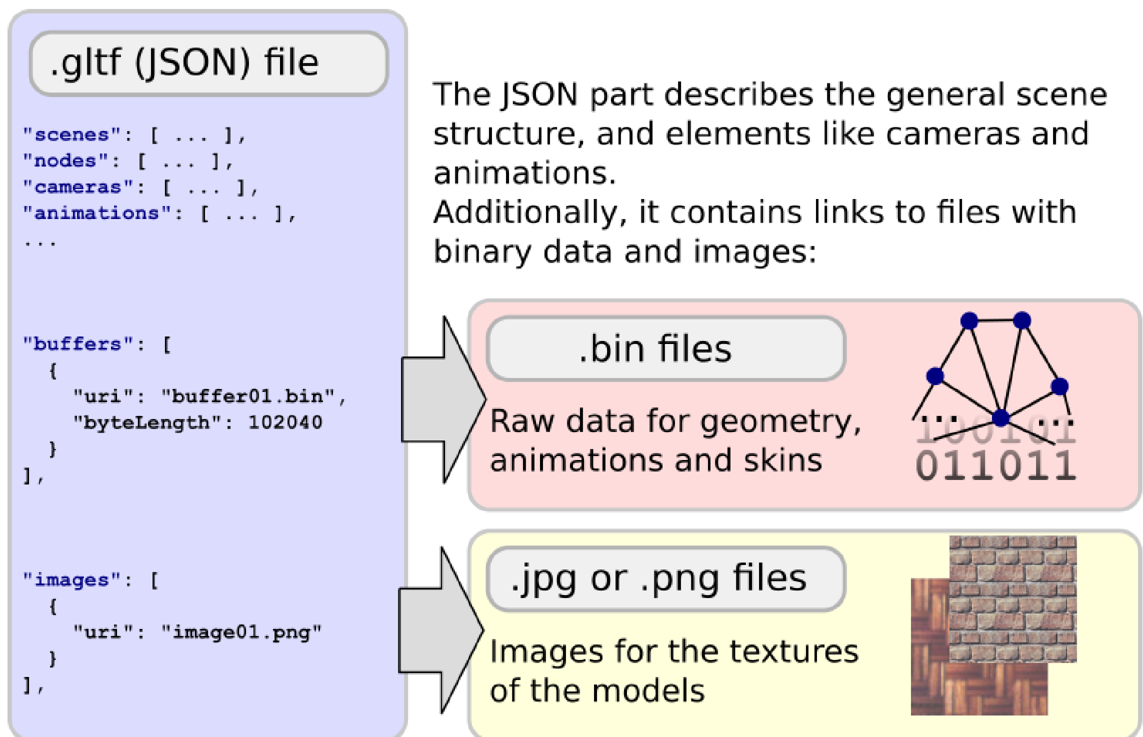


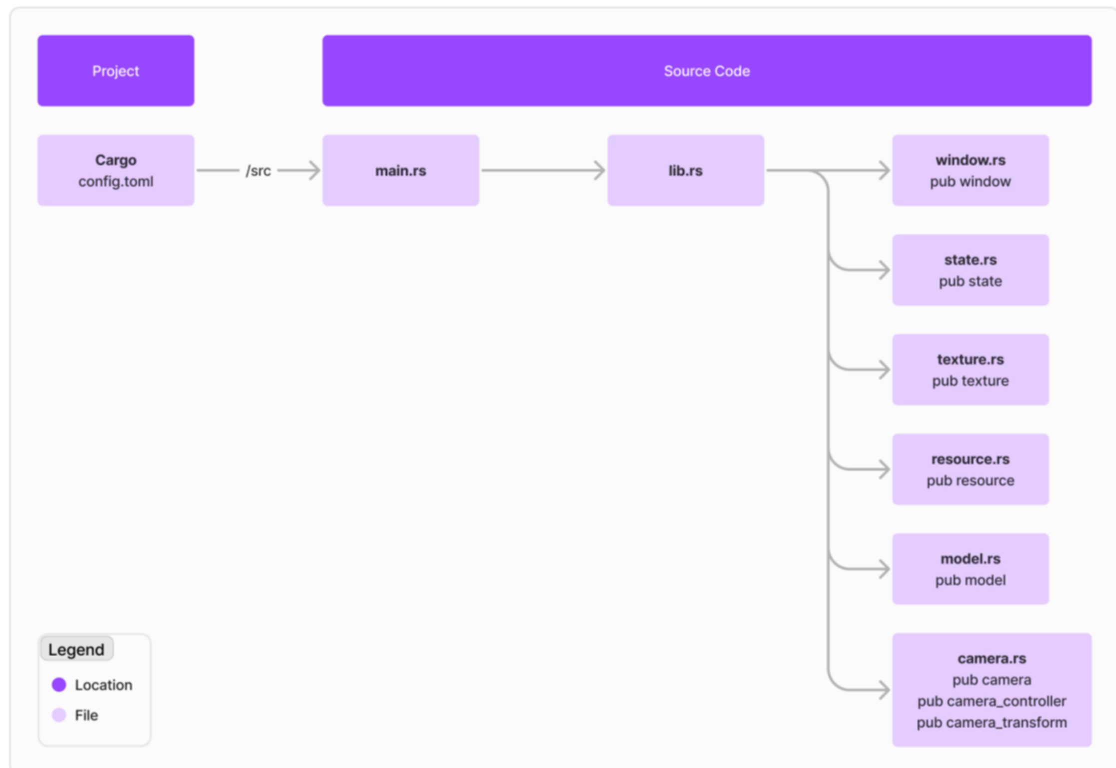
FIGURE 27. The GLTF JSON structure (GLTF-Tutorials)



PICTURE 28. The GLTF structure (GLTF-Tutorials)

4 IMPLEMENTATION

4.1 Source code and library management



PICTURE 29. Simplified Project Structure

The structure of the project seems quite simple. In any rust project, it needs a cargo configuration file (Config.toml) for installing dependencies. In the directory of source code, it has main.rs as a starting point and lib.rs for controlling internal submodules as smaller library files. There is a role of each file:

- "window.rs" contains the "run" method which initiates the state, event loop, and window.
- "state.rs" includes all methods for the event system in the application.
- "texture.rs" supports a shorter way of handling texture mapping operations.
- "resource.rs" is a helper library that includes some methods used for fetching and processing data on the cloud.
- "model.rs" takes a key role in creating Model objects and operating drawing commands from the resources.
- "camera.rs" is built for the Camera data structure and its control methods.

4.2 Application Event Flow

Primarily, the projection targets of this solution are desktop operating systems. Hence, a library for window management is required. Besides, the application also needs to have a state system for handling application events. In this project, Winit was chosen because of its compatibility with cross-platform devices, even on the web. It is just used for window management, graphics presentation, and input handling only, without any further high-level implementations. (Rust-Windowing)

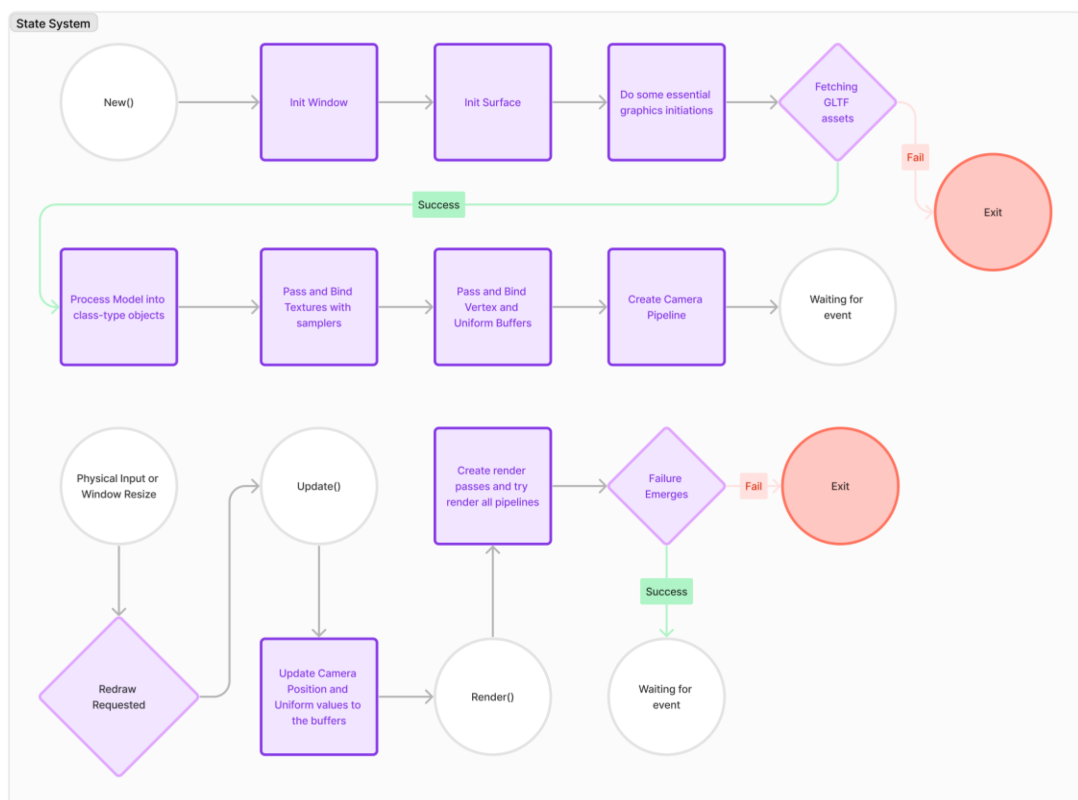


FIGURE 30. Event Flowchart

According to figure 31, to create a surface for drawing rendering frames, an instance should be initiated to gather supporting backend graphics APIs from the current device. Afterward, there is a mandatory fact that the surface needs to live with the generated window lifetime. Additionally, there is also an adapter that opts for a preferred power mode for the application. For instance, in some computers

having dGPU and iGPU, the OS will always prioritize running applications with the most power-efficient hardware.

In the render method, a request is made for a rendering destination as a new Surface Texture. Then, it will be stored in output. After having a texture for the surface, its view can be created with a default setting. Besides this fact, there are two more important ones. For instance, the GPU needs a command request, then, an encoder with a render pass is required. Finally, the render method needs to be added to the event loop for handling the redraw request.

4.3 Control camera with mouse cursor position

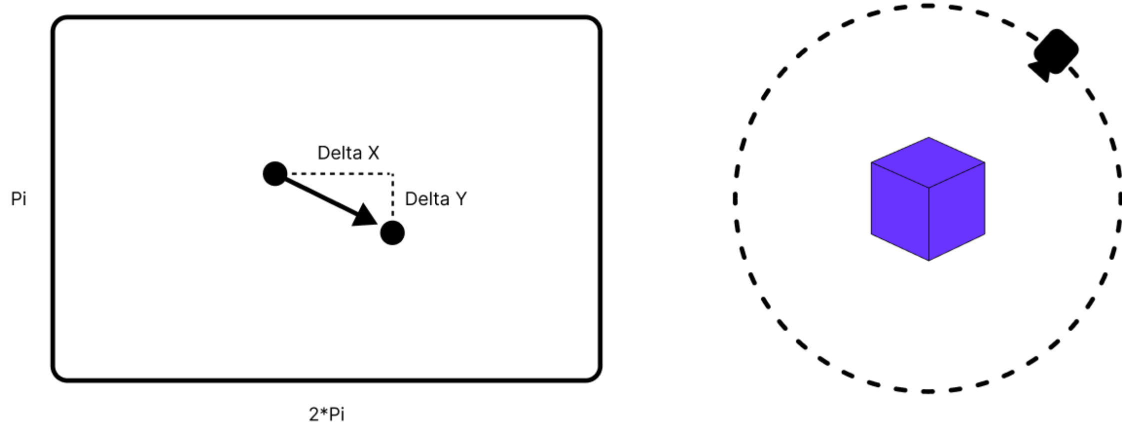


Figure 31. Gesture input to control the arc ball camera.

Fortunately, Winit supports methods for handling physical inputs such as mouse, keyboard, or even touchscreen.

```
match event {
    WindowEvent::CursorMoved { position, .. } => {
        if self.is_left_mouse_pressed {
            let delta_x = position.x as f64;
            let delta_y = position.y as f64;
            camera.update_mouse_position((delta_x as f32, delta_y as f32));
        }
        camera.set_last_mouse_position(Some((*position).into()));

        true
    }
    WindowEvent::MouseInput { state, button, .. } => {
        if *button == MouseButton::Left {
            self.is_left_mouse_pressed = *state == ElementState::Pressed;
        }
        true
    }
    ...
}
```

PICTURE 32. Example of handling mouse inputs.

When the mouse is moving, the window event will return the cursor position. Then, according to the received data, it will be stored in the camera object via a setter method called “set_last_mouse_position()”. Moreover, after having a new position, the camera update method will be instantly called.

Firstly, according to what was explained in the theoretical part, the calculation starts with calculating the pivot and position of the eye and target.

Secondly, the horizontal step must be calculated by the subtraction of current mouse's horizontal position from the last one, and then multiplying it with a division of two PI and width values. The vertical one can be solved by subtraction of the current mouse's vertical position from the last one. However, it just needs to multiply with the division of PI and height value.

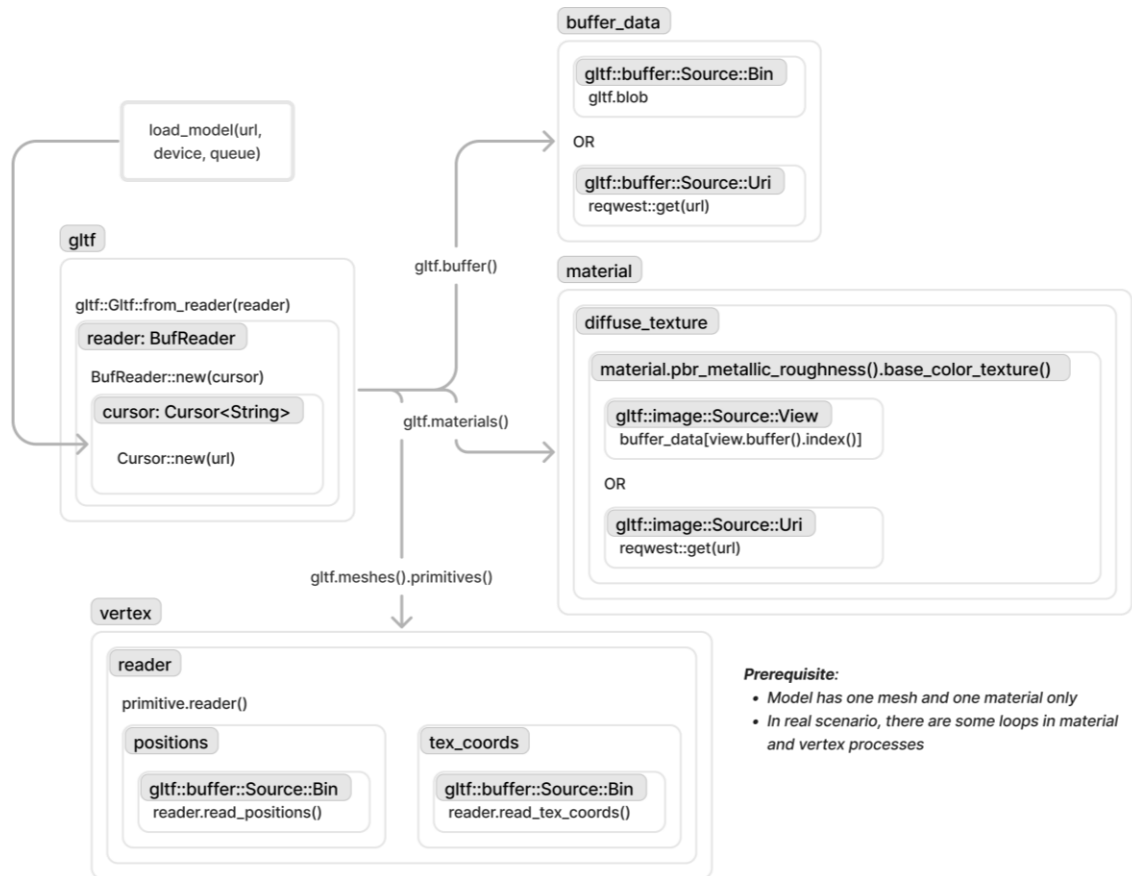
Thirdly, the delta angle of x and y will be calculated depending on the subtraction of the new position and the last position. Finally, the camera position will be calculated based on rotation matrices of the up vector and the right vector.

```
pub fn update_mouse_position(&mut self, new_mouse_pos: (f32, f32)) {
    if let Some((last_x, last_y)) = self.last_mouse_pos {
        let (width, height) = self.view_port.unwrap_or((1.0, 1.0));

        let mut position = vec4(self.eye.x, self.eye.y, self.eye.z, 1.0);
        let pivot = vec4(self.target.x, self.target.y, self.target.z, 1.0 as f32);

        let step_angle_x = 2.0 * PI as f32 / width;
        let step_angle_y = PI as f32 / height;

        let mut theta_x = (new_mouse_pos.0 - last_x) * step_angle_x;
        let mut theta_y = (new_mouse_pos.1 - last_y) * step_angle_y;

        let rotation_matrix_x = Matrix4::from_axis_angle(self.up, Rad(theta_x));
        position = (rotation_matrix_x * (position - pivot)) + pivot;

        let rotation_matrix_y = Matrix4::from_axis_angle(self.get_right_vector(), Rad(theta_y));
        let final_pos = (rotation_matrix_y * (position - pivot)) + pivot;

        self.set_camera_view(
            Point3::new(final_pos.x, final_pos.y, final_pos.z),
            self.target,
            self.up,
        );
    }

    self.last_mouse_pos = Some(new_mouse_pos);
}
```

PICTURE 33. Updating camera position.

4.4 Fetching GLTF model on cloud storage

In this thesis, the requirement is to render meshes without lighting. Hence, there are only three values that need to be collected from the model, which are the positions of a points, texture coordinates, and color texture (albedo).



PICTURE 34. Loading data from file using GLTF library based on Rust.

Primarily, a link or path needs to be passed into a model-loading method. Then, before processing data, it needs a GLTF reader which points to the passed link. When it is ready, buffer data can be recalled by one of two scenarios. If it is a URL, it will be fetched by the “reqwest::get(URL)” which is an HTTP client. If it is a directory path, the binary will automatically be stored in the library. Then, it just needs to be assigned with the "gltf.blob". Like processing binary, if the material is stored on the cloud, the data will be fetched. However, the material can be collected by accessing buffer data due to it is already stored. From the vertex perspective, it needs another reader to travel through all points. Unlike material, the values of points are only placed in binary. Hence, positions and texture coordinates can be accessed via “read_position()” and “read_tex_coords()” methods.

4.5 Working with React Web Application

The destination for this thesis is running on a web framework. It can be solved by a recent technology called Web Assembly. In 2015, it was announced and created a massive impact on web applications at that time. Recently, there has been

a contemporary feature that allows developers to bind fully or partially Rust source code for web applications. Hence, it can be built into a package containing JavaScript, TypeScript, and WASM. The package can be imported into various Web Frameworks such as React or Vue.

To define a line of code only running on WASM target, the project needs to add wasm-bindgen library.

```
[target.'cfg(target_arch = "wasm32")'.dependencies]
wasm-bindgen = "0.2"
```

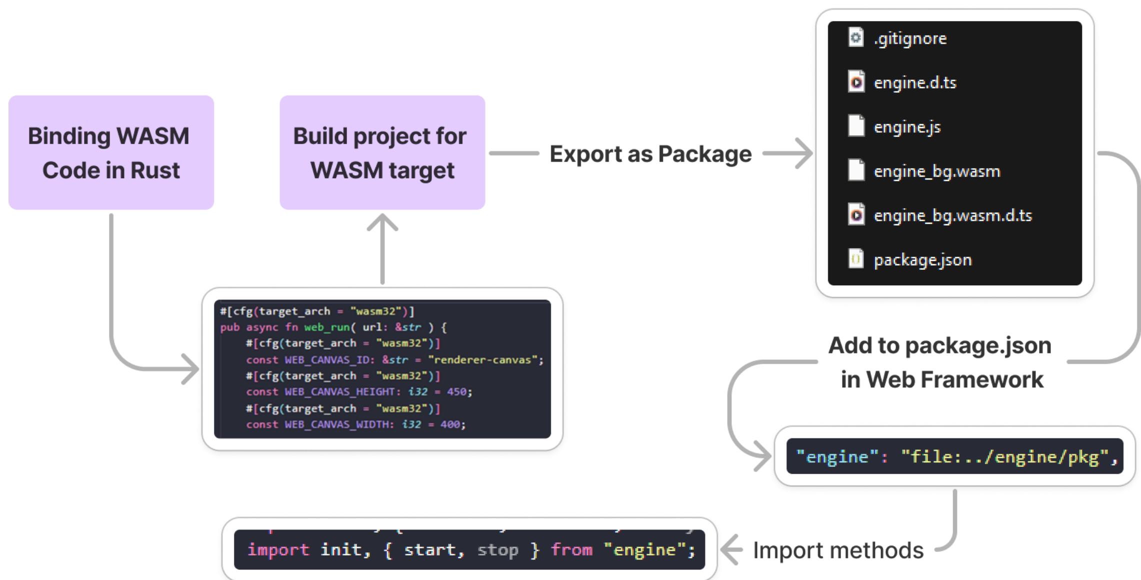
PICTURE 35. Adding wasm-bindgen into Cargo.toml file.

After having the required dependency, it is required to set a configuration of target architecture to “wasm32” by adding “#[cfg(target_arch = "wasm32")]”.

```
#[cfg(target_arch = "wasm32")]
pub async fn web_run( url: &str ) {
    #[cfg(target_arch = "wasm32")]
    const WEB_CANVAS_ID: &str = "renderer-canvas";
    #[cfg(target_arch = "wasm32")]
    const WEB_CANVAS_HEIGHT: i32 = 450;
    #[cfg(target_arch = "wasm32")]
    const WEB_CANVAS_WIDTH: i32 = 400;
```

PICTURE 36. web_run method and those variables work only when it is built for WASM target.

Finally, the project can be built and implemented into various recent web frameworks. Additionally, to make it easier, this thesis used a tool named wasm-pack. It supports building Rust project to WASM package working with NPM. With wasm-pack support, the project is easily placed inside any package.json files.



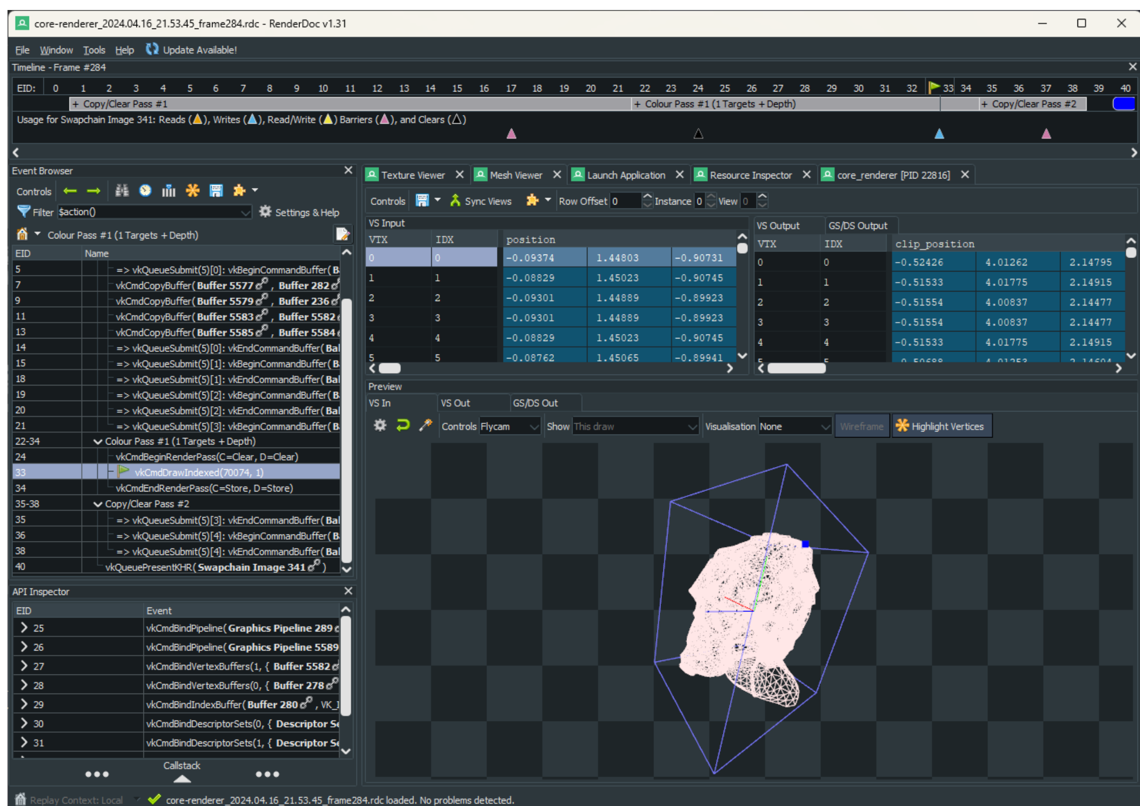
PICTURE 37. Implementing the project to React Framework.

5 QUALITY ASSURANCE

5.1 Renderdoc

RenderDoc is a free open-source tool for graphics developers. It is widely used in many industries including VFX - Visual Effects, Fiction Film, Animation, and Video Game. Using this tool, developers can investigate events when a frame is rendered, such as buffer values, images, pipeline timestamps, and so on. (Baldurk. (n.d.))

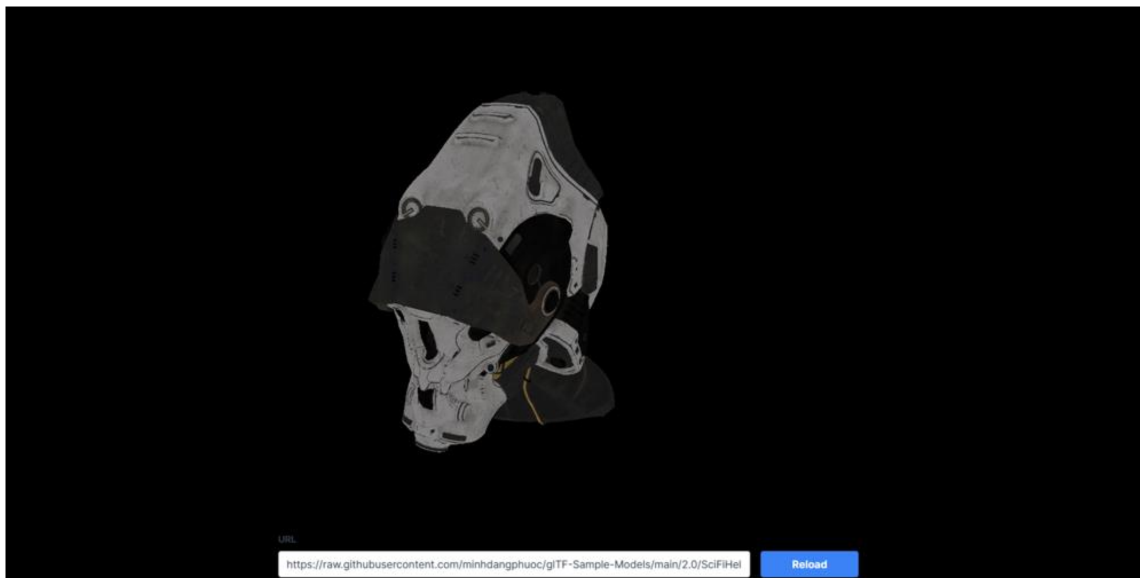
Unfortunately, RenderDoc only works with desktop platforms including Window (Direct3D, Vulkan, and OpenGL), Linux (Vulkan and OpenGL), and MacOS (OpenGL, Vulkan and Metal). Hence, it is only used for testing primary features, engine output, and improving optimization.



PICTURE 38. Mesh Viewer Pane.

6 DISCUSSION

The engine achieved exceptional performance with a clean visual and fast responsiveness, displaying the advantages of WGPU's ability to work across multiple platforms. Additionally, the GLTF Loader is capable of loading models from both local and cloud sources. (Appendix 1)



PICTURE 39. Screenshot of web live demonstration.

However, the camera system could be improved. It experiences issues when the up vector and view direction are parallel, often leading to a Gimbal lock, which is a disadvantage of the Euler rotation theorem. Moreover, implementing a simple lighting system could enhance the visual experience.

Besides the rendering minuses, the event system could be improved in the future. For instance, there were a lot of stop points in the flow. Currently, it could serve as a shortcut to come through the memory-borrow and mutation concepts of Rust. For instance, the application should not break because of process failures. If this improvement is applied in the future, the application will achieve the concurrency milestone. Hence, it will improve the applicability of the solution.

Furthermore, this thesis could serve as valuable material for developers taking advantage of WGPU and its features, especially, its ability to run on the web via WebGPU. It is still early to predict that it will replace commercial engines in the

market, but it is poised to have a significant and positive impact on the current landscape of graphics development. Additionally, computer graphics should receive more attention from developers, as WGPU could direct a bright future, attracting more enthusiasts due to its flexibility and simplicity.

REFERENCES

Frausto-Robledo, A. (2017, February 24). Apple's WebGPU standard proposal-aiming at common access to explicit graphics. Architosh. https://architosh.com/2017/02/apples-webgpu-standard-proposal-aiming-at-common-access-to-explicit-graphics/

Next-generation 3D graphics on the web. (2017, April 5). WebKit. https://webkit.org/blog/7380/next-generation-3d-graphics-on-the-web/

Thompson, C. (2023, February 15). How Rust went from a side project to the world's most-loved programming language. MIT Technology Review. https://www.technologyreview.com/2023/02/14/1067869/rust-worlds-fastest-growing-programming-language/

Gfx-Rs. (n.d.). WGPU/etc/big-picture.png at trunk · gfx-rs/WGPU. GitHub. https://github.com/gfx-rs/WGPU/blob/trunk/etc/big-picture.png

Ninomiya, K., Jones, B., & Blandy, J. (Eds.). (2024, February 5). WebGPU. W3.org. https://www.w3.org/TR/webgpu/#gpurenderpipeline

Primitive assembly. Primitive Assembly - OpenGL Wiki. (n.d.). https://www.khronos.org/opengl/wiki/Primitive_Assembly

Vertex Specification - OpenGL Wiki. (n.d.). https://www.khronos.org/opengl/wiki/Vertex_Specification

Stencil test - OpenGL Wiki. (n.d.). https://www.khronos.org/opengl/wiki/Stencil_Test

LearnOpenGL - Stencil testing. (n.d.). Retrieved April 14, 2024, from https://learnopengl.com/Advanced-OpenGL/Stencil-testing

LearnOpenGL - Depth testing. (n.d.). Retrieved April 14, 2024, from https://learnopengl.com/Advanced-OpenGL/Depth-testing

Depth Test - OpenGL Wiki. (n.d.). https://www.khronos.org/opengl/wiki/Depth_Test

Overvoorde, A. (n.d.). Image view and sampler - Vulkan Tutorial. https://vulkan-tutorial.com/Texture_mapping/Image_view_and_sampler

Gfx-Rs. (n.d.). GitHub - gfx-rs/wgpu: Cross-platform, safe, pure-rust graphics api. GitHub. Retrieved April 13, 2024, from https://github.com/gfx-rs/wgpu

LearnOpenGL - Coordinate Systems. (n.d.). https://learnopengl.com/Getting-started/Coordinate-Systems

LearnOpenGL - Camera. (n.d.). https://learnopengl.com/Getting-started/Camera

Stevewhims. (n.d.). Coordinate Systems - UWP applications. Microsoft Learn. Retrieved April 13, 2024, from https://learn.microsoft.com/en-us/windows/uwp/graphics-concepts/coordinate-systems

Eberly, D. (1999, December 1). Euler angle formulas. Euler Angle Formulas. https://www.geometrictools.com/Documentation/EulerAngles.pdf

Marie. (2019, November 30). How to implement a simple Arcball Camera. A Slice of Rendering. https://asliceofrendering.com/camera/2019/11/30/Arcball-Camera/

GLTF-Tutorials. (n.d.). GLTF-Tutorials. https://github.khronos.org/GLTF-Tutorials/gltfTutorial/gltfTutorial_002_BasicGltfStructure.html

Rust-Windowing. (n.d.-b). GitHub - rust-windowing/winit: Window handling library in pure Rust. GitHub. Retrieved April 13, 2024, from https://github.com/rust-windowing/winit

Baldurk. (n.d.). Baldurk/renderdoc: RenderDoc is a stand-alone graphics debugging tool. GitHub. https://github.com/baldurk/renderdoc

APPENDICES

Appendix 1 Related Resources

- Source Code: github.com/minhdangphuoc/thesis-3d-web-renderer
- Web Demonstration: thesis-3d-web-renderer.vercel.app
- GLTF Sample Models: github.com/KhronosGroup/GLTF-Sample-Models

Appendix 2 RenderDoc Screenshots

- Swapchain and Depth Output views
- Image Buffer view