Dang Bao Nghi Nguyen

WEBGPU - THE NEW POWERFUL SUCCESSOR, THE FUTURE OF WEB API

Bachelor's thesis

Information Technology

Bachelor of Engineering

2022

South-Eastern Finland
University of Applied Sciences

Author (authors) Nghi Nguyen	**Degree title** Bachelor of Engineering	**Time** May 2022
Thesis title WebGPU - The New Powerful Successor, The Future of Web API		35 pages 0 pages of appendices
Commissioned by		
Supervisor Timo Hynninen		

Abstract

The objective of this thesis was to create an application by using new Web API technology called WebGPU to show how well and optimize this API compared to WebGL. The project was designed to have two end results application to show what this new type of WebGPU technology is capable of and deliver us a glimpse of beneficial from WebGPU when using for other purposes.

The introduction of the APIs laid the foundation for many acomplishments, and web platform was one of which increased drastically. In addition, with the most recent powerful smart devices, that inspired the path for developers to utilize this opportunity to design and implement so many great things.

Keywords

TypeScript, JavaScript, WebGPU, Pipeline, Matrix

CONTENTS

1 INTRODUCTION

The main goal of this thesis is to create a simple project to demonstrate new Web API called WebGPU that will help reduce the workload of system by focusing on the GPU resouce. Thoughout this thesis, the project will be built using new WebGPU technology instead of WebGL or WebGL2 that has been around for more than 10 years. This new technology is going to be the next future in graphical design method which will help reduce time, resouce and cost efficiency.

2 THEORY

The introduction of JavaScript, TypeScript, WebGPU and WebGPU Shading Language are crucial for creating the project. Alongside with compute Pipeline, GPU Buffer are also deeply studied.

2.1 WebGPU

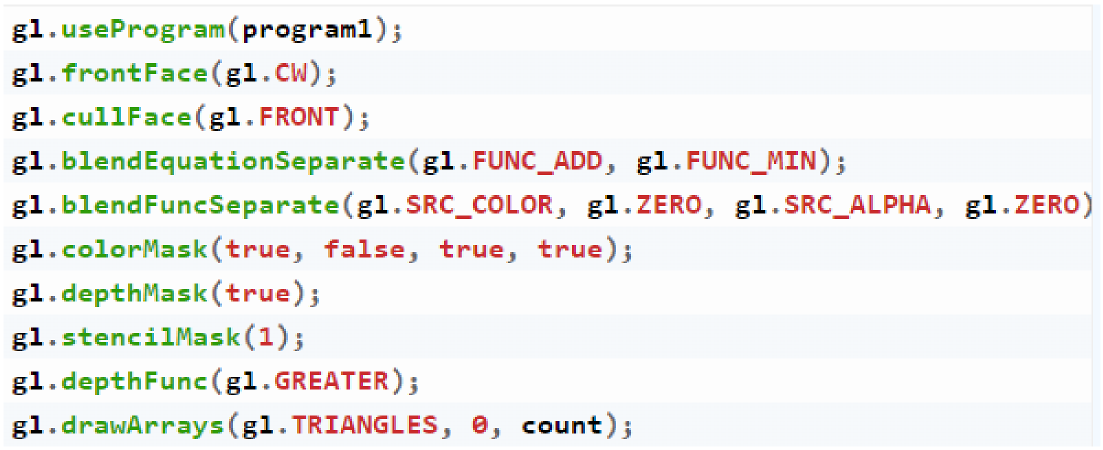
Graphic Processing Unit (GPU) is an electronic subsystem within a computer that was originally specialized for processing graphics. However, in the past 10 years, it has evolved towards a more flexible architecture allowing developers to implement many types of algorithms, not just render 3D graphics, while taking advantage of the unique architecture of the GPU. These capabilities are referred to as GPU Compute, and using a GPU as a coprocessor for general-purpose scientific computing is called general-purpose GPU (GPGPU) programming.

Nowadays, with the development of AI, machine learning, and Virtual Reality GPU Compute has contributed significantly to the recent machine learning boom. The convolution neural networks and other models can take advantage of the architecture to run more efficiently on GPUs. With the current Web Platform lacking in GPU Compute capabilities, the W3C's "GPU for the Web" Community Group is designing an API to expose the modern GPU APIs that are available on most current devices. This API is called WebGPU, this WebGPU name is not official since this is still under development by many developers around the world. WebGPU is a low-level API, like WebGL. It is very powerful and quite verbose, as seen.

2.2 WebGPU or WebGL?

There are many things to be considered when choosing between WebGPU and WebGL. **WebGL**, is based on OpenGL, involves lots of individual function calls to change individual settings. **WebGPU** on the other hand creates groups of settings the application will use in advance. Then at runtime it can switch between entire groups of settings with a single function call, which is much faster. It also organizes all the settings according to how modern GPUs work, allowing applications to work more efficiently with hardware.

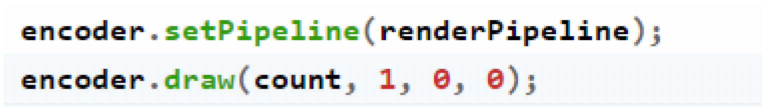
A single object will make good of example for better understanding during the code. In programming, having less code means running faster, In Figure 1, WebGL uses more codes to just render one object.



```
gl.useProgram(program1);
gl.frontFace(gl.CW);
gl.cullFace(gl.FRONT);
gl.blendEquationSeparate(gl.FUNC_ADD, gl.FUNC_MIN);
gl.blendFuncSeparate(gl.SRC_COLOR, gl.ZERO, gl.SRC_ALPHA, gl.ZERO)
gl.colorMask(true, false, true, true);
gl.depthMask(true);
gl.stencilMask(1);
gl.depthFunc(gl.GREATER);
gl.drawArrays(gl.TRIANGLES, 0, count);
```

Figure 1. WebGL code

In Figure 2, doing the same task but with only just a few lines of codes



```
encoder.setPipeline(renderPipeline);
encoder.draw(count, 1, 0, 0);
```

Figure 2. WebGPU code

2.3 WebGPU Shading Language

WebGPU Shading Language (WGSL) is the shader language for WebGPU. WGSL's development focuses on getting it to easily convert into the shader language corresponding to the backend; for example, SPIR-V for Vulkan, MSL for Metal, HLSL for DX12, and GLSL for OpenGL. Despite WebGPU was suggested by Apple in 2017 and still under developing up until now, based on this evidence, WebGPU Shading Language is be considered fairly new to the community.

2.4 JavaScript Language

JavaScript often known as JS is a high-level programming language used by 97% of websites, alongside HTML and CSS mainly use for web decoration. JavaScript is the dominant client-side scripting language of the Web, scripts are embedded in or included from HTML documents and interact with the DOM. All major web browsers have a built-in JavaScript engine that executes the code on the user's device. More than 80% of websites use third-party JavaScript library or framework to build their client-side, when talking about JavaScript library, one most commontly known is the **jQuery** used by over 75% of websites. Since the late 2000s JavaScript engine has been embedded in a variety of other software systems, one can be listed as **server-side**. Nowadays **server-side** started to grow with the creation of **Node.js**.

One thing related to JavaScript is a misconception between JavaScript and Java. Even though they both have a C-like syntax, have almost the same name, and both appeared in 1995, Java was developed by **James Gosling** of Sun Microsystems and JavaScript by **Brendan Eich** of Netscape Communications. Despite the similarities of these two programming languages, their differences are truly prominent, Java has static typing, while JavaScript's typing is dynamic. Java is loaded from compiled bytecode, while JavaScript is loaded as human-readable source code. Java's objects are class-based, while JavaScript's are prototype-based. Finally, Java did not support functional programming until Java 8, while JavaScript has done so from the beginning, being influenced by Scheme.

2.5 TypeScript

Another programming language, which was originally built based on JavaScript syntax itself is TypeScript. TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor, it is designed for the development of large applications and transpiles to JavaScript. Catch errors early in your editor.

TypeScript contains type information of existing JavaScript libraries. much like C++ header files can describe the structure of existing object files. This enables other programs to use the values defined in the files as if they were statically typed TypeScript entities. There are many third-party libraries like jQuery, MongoDB, and D3.js, TypeScript headers for the Node.js basic modules are also available, allowing development of Node.js programs within TypeScript. TypeScript is included as a first-class programming language in Microsoft Visual Studio 2013 Update 2 and later, alongside C# and other Microsoft languages.

2.6 Compute Pipeline

The Direct3D compute pipeline is designed to handle calculations that can be done mostly in parallel with the graphics pipeline. There are only a few steps in the compute pipeline, with data flowing from input to output through the programmable compute shader stage.

A compute shader provides high-speed general purpose computing and takes advantage of the large numbers of parallel processors on the graphics processing unit (GPU). The compute shader provides memory sharing and thread synchronization features to allow more effective parallel programming methods. The input can be one, two or three-dimensional in nature, determining the number of invocations of the compute shader to execute. Output data from the compute shader, which can be highly varied, can be synchronized with the graphics rendering pipeline when the computed data is required.

2.7 Vertex, fragment

A vertex is a point in 3d space (can also be 2d). These vertices are then bundled in groups of 2s to form lines and/or 3s to form triangles.

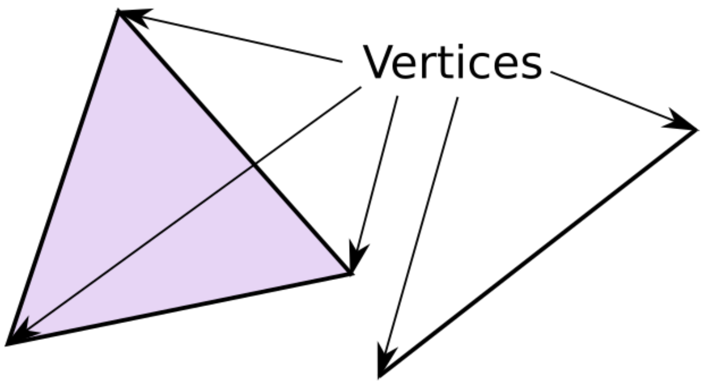


Figure 3. Vertices

Most modern rendering uses triangles to make all shapes, from simple shapes as cubes to complex ones as people. These triangles are stored as vertices which are the points that make up the corners of the triangles.

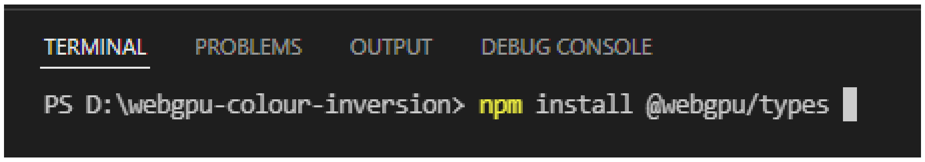
We use a vertex shader to manipulate the vertices, to transform the shape to look the way we want it. The vertices are then converted into fragments. Every pixel in the result image gets at least one fragment. Each fragment has a color that will be copied to its corresponding pixel. The fragment shader decides what color the fragment will be.

3 IMPLEMENTATION

This section is the practical part of this thesis, showing the establishment of this application. The concepts from above Chapter 2 are used to support this process.

3.1 Color Inversion

In this project, in Figure 4 starting with basic typescript web project, which include an **index.html** and an **index.ts** files. Next, we install some packages with **npm**, and since this is about WebGPU API, by using the **npm** package management, we install the most crucial package **WebGPU Types**, where we can access the **WebGPU API**.

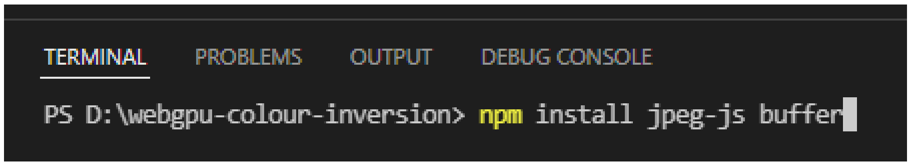


TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS D:\webgpu-colour-inversion> npm install @webgpu/types
```

Figure 4. Install WebGPU packages

Next, in Figure 5 we install the other two packages, called **jpeg-js** library, which decodes jpegs into an array with red, green, blue and alpha values for all pixels and encodes that array into a jpeg image. Then, the **buffer** library, plays as a dependency for the **jpeg.js** library.



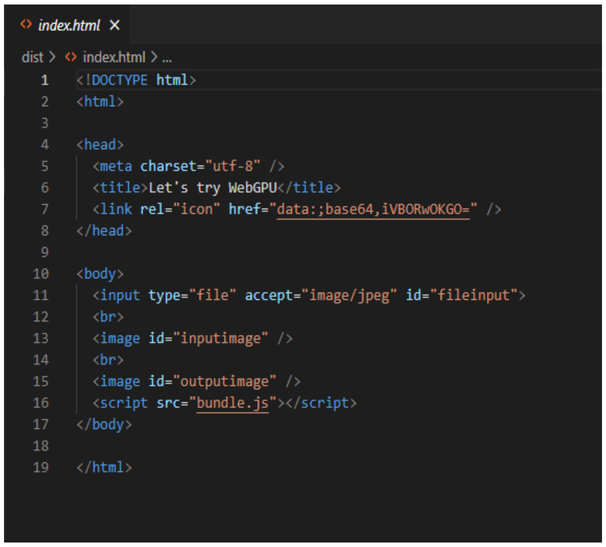
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
PS D:\webgpu-colour-inversion> npm install jpeg-js buffer
```

Figure 5. Install jpeg-js library

3.1.1 Create JPEG image and read Pixel Data Input

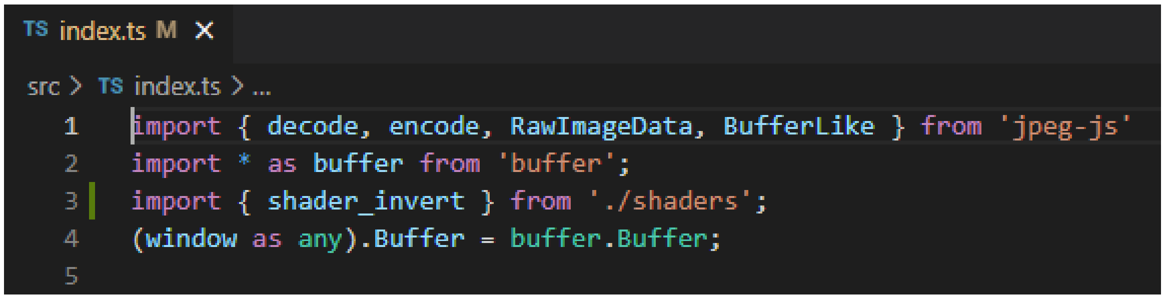
In Figure 6, now that we have installed every crucial packages, navigate to the **index.html,** here a button is created which will let us choose and input random jpeg images, we also create an input image where the original image will be stored and an output image where the image pixel colour has been reversed.



```html
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <title>Let's try WebGPU</title>
  <link rel="icon" href="data:;base64,iVBORw0KGO=" />
</head>

<body>
  <input type="file" accept="image/jpeg" id="fileinput">
  <br>
  <image id="inputimage" />
  <br>
  <image id="outputimage" />
  <script src="bundle.js"></script>
</body>

</html>
```

Figure 6. Create JPEG input and output machanism

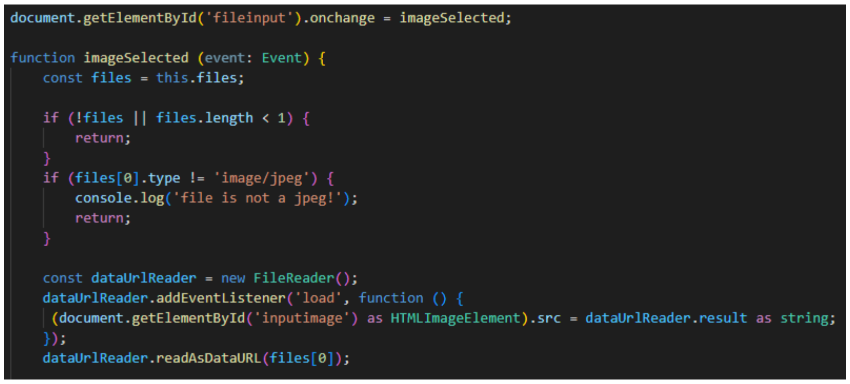
Next, move to the **index.ts** file, here from **jpeg-js** library, we import some methods, interfaces and a workaround so that **jpeg-js** library can also run in browsers, as shown in Figure 7.



```
TS index.ts M ×
src > TS index.ts > ...
import { decode, encode, RawImageData, BufferLike } from 'jpeg-js'
import * as buffer from 'buffer';
import { shader_invert } from './shaders';
(window as any).Buffer = buffer.Buffer;

```

Figure 7. Import libraries from jpeg-js

Figure 8 shows the creation of jpeg image file select function for the button.



```
document.getElementById('fileinput').onchange = imageSelected;

function imageSelected (event: Event) {
    const files = this.files;

    if (!files || files.length < 1) {
        return;
    }
    if (files[0].type != 'image/jpeg') {
        console.log('file is not a jpeg!');
        return;
    }

    const dataUrlReader = new FileReader();
    dataUrlReader.addEventListener('load', function () {
     (document.getElementById('inputimage') as HTMLImageElement).src = dataUrlReader.result as string;
    });
    dataUrlReader.readAsDataURL(files[0]);
```

Figure 8. Select function button

In Figure 9 we can see the image on the browsers.

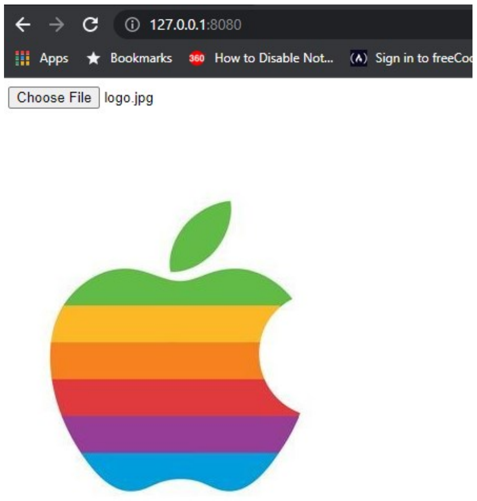
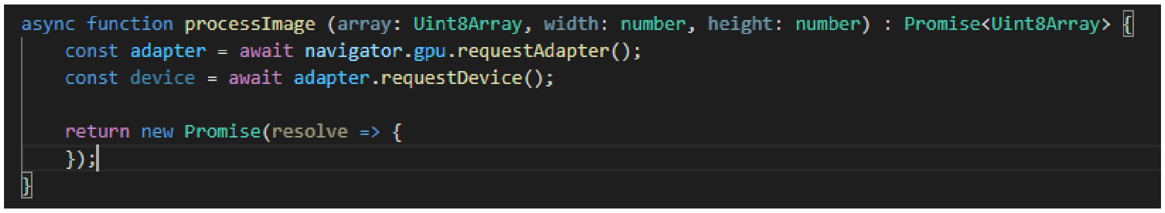


Figure 9. Display the result on browsers

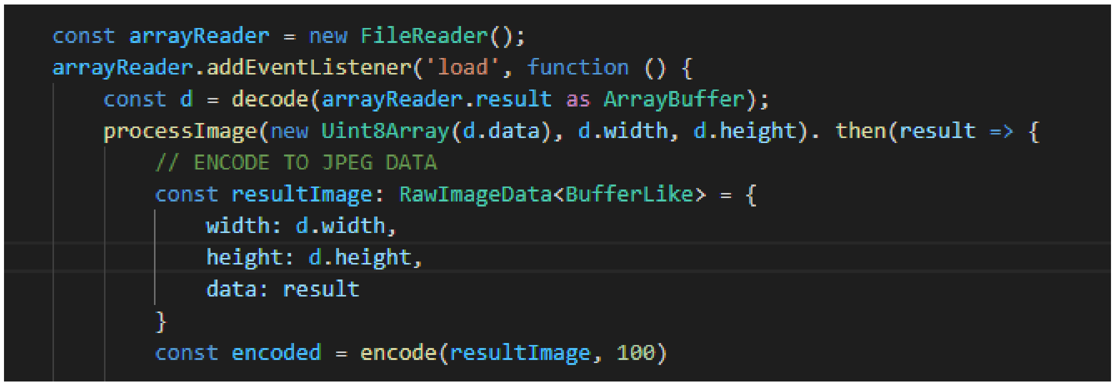
The successfully created function that can input and display an image on browsers. As shown in Figure 10 create file reading as an **array buffer** is necessary, the reason so that we can decode the pixel colours with the **jpeg.js**.



```
async function processImage (array: Uint8Array, width: number, height: number) : Promise<Uint8Array> {
    const adapter = await navigator.gpu.requestAdapter();
    const device = await adapter.requestDevice();

    return new Promise(resolve => {
    });
}
```

Figure 10. Raw image pixel data process

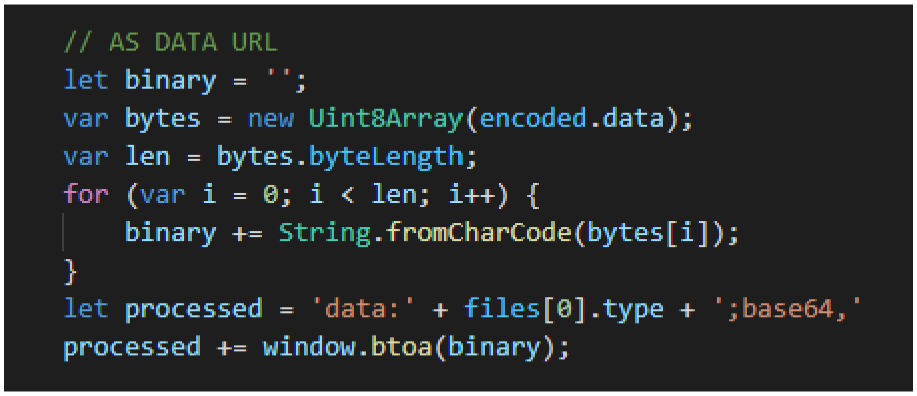
Now that an array buffer has been created, next step is to create a function which will process raw image pixel data with WebGPU as shown in Figure 11, and because the process has to be done before it can be displayed on the browsers, for that we use **async function.**



```
const arrayReader = new FileReader();
arrayReader.addEventListener('load', function () {
    const d = decode(arrayReader.result as ArrayBuffer);
    processImage(new Uint8Array(d.data), d.width, d.height). then(result => {
        // ENCODE TO JPEG DATA
        const resultImage: RawImageData<BufferLike> = {
            width: d.width,
            height: d.height,
            data: result
        }
        const encoded = encode(resultImage, 100)
```

Figure 11. Create array buffer

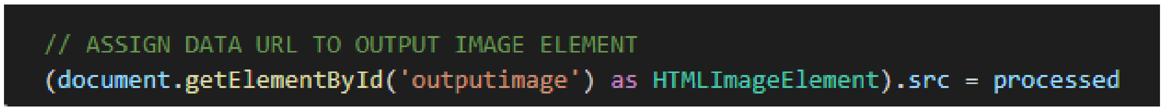
We then transform data as data URL in Figure 12.



```
// AS DATA URL
let binary = '';
var bytes = new Uint8Array(encoded.data);
var len = bytes.byteLength;
for (var i = 0; i < len; i++) {
    binary += String.fromCharCode(bytes[i]);
}
let processed = 'data:' + files[0].type + ';base64,'
processed += window.btoa(binary);
```

Figure 12. Transform data to URL

We can then assign the data URL to an HTML image element with the ID ouput image in Figure 13.

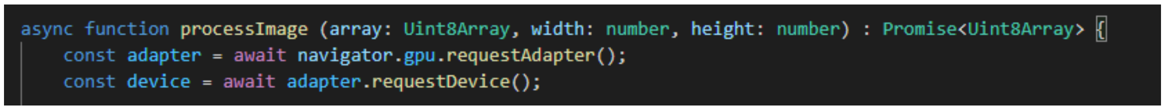


```
// ASSIGN DATA URL TO OUTPUT IMAGE ELEMENT
(document.getElementById('outputimage') as HTMLImageElement).src = processed
```

Figure 13. Assign data URL to HTML

3.1.2 Access GPU device

We now have prepare the input and the ouput of the jpeg image process, the next step is to access WebGPU API, but first thing to do is to get the GPU device as shown in Figure 14.



```
async function processImage (array: Uint8Array, width: number, height: number) : Promise<Uint8Array> {
    const adapter = await navigator.gpu.requestAdapter();
    const device = await adapter.requestDevice();
```

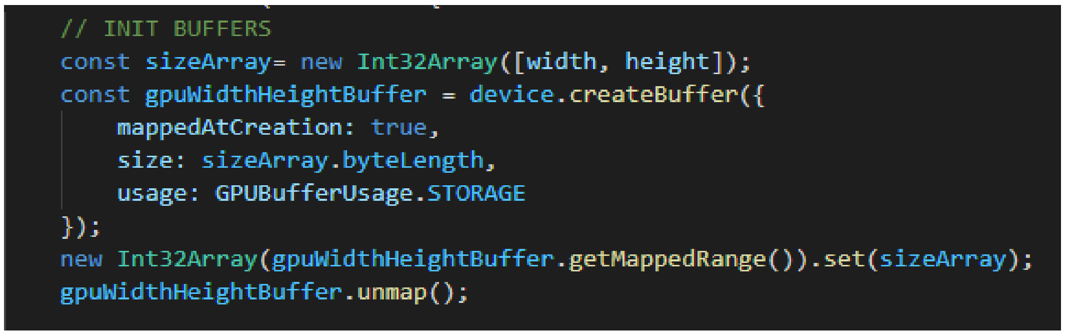
Figure 14. Access the GPU resouce

3.1.3 Buffer

Buffers are needed for WebGPU rendering, and buffer is a contiguous block of memory in the GPU that stores rendering data for a model. In this case, we create four buffers:

3.1.3.1 Width Height buffer

Figure 15 configures the Width Height buffer, which will contain the height and width of the image.

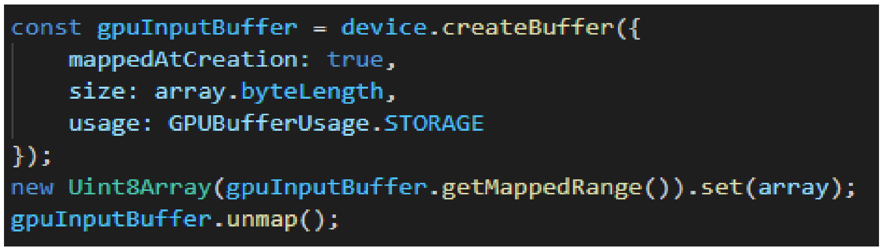


```
// INIT BUFFERS
const sizeArray= new Int32Array([width, height]);
const gpuWidthHeightBuffer = device.createBuffer({
    mappedAtCreation: true,
    size: sizeArray.byteLength,
    usage: GPUBufferUsage.STORAGE
});
new Int32Array(gpuWidthHeightBuffer.getMappedRange()).set(sizeArray);
gpuWidthHeightBuffer.unmap();
```

Figure 15. Width and Heigh buffer

3.1.3.2 Pixels buffer

This step will process the raw pixel rgba data as shown in Figure 16.

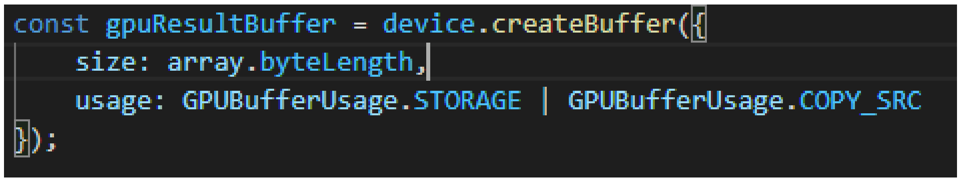


```
const gpuInputBuffer = device.createBuffer({
    mappedAtCreation: true,
    size: array.byteLength,
    usage: GPUBufferUsage.STORAGE
});
new Uint8Array(gpuInputBuffer.getMappedRange()).set(array);
gpuInputBuffer.unmap();
```

Figure 16. Pixel buffer

3.1.3.3 Result buffer

This section holds the above processed raw pixel rgba data.

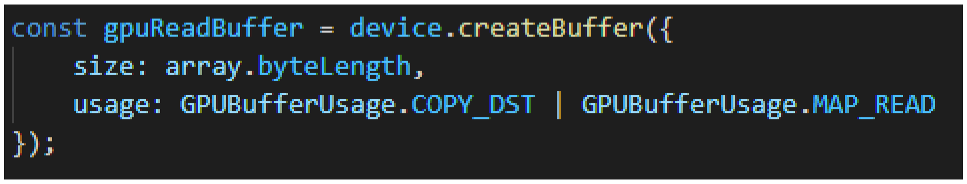


```
const gpuResultBuffer = device.createBuffer({
    size: array.byteLength,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC
});
```

Figure 17. Processed raw pixel containment

3.1.3.4 Read buffer

Figure 18 shows the result will be read by CPU.



```
const gpuReadBuffer = device.createBuffer({
    size: array.byteLength,
    usage: GPUBufferUsage.COPY_DST | GPUBufferUsage.MAP_READ
});
```

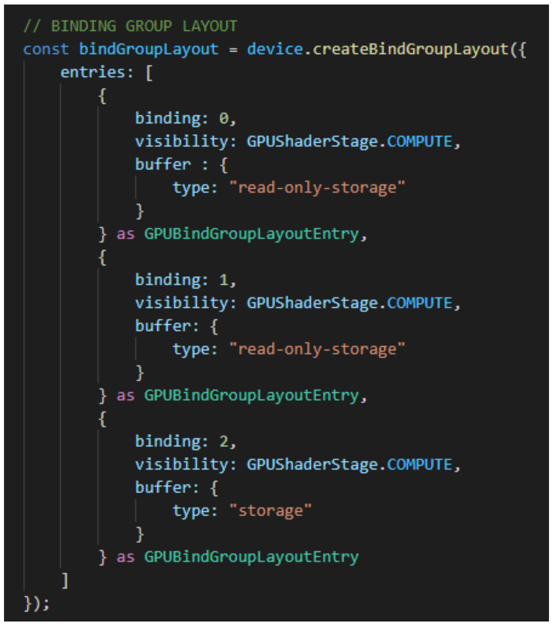
Figure 18. Reading the buffer

3.1.4 Bind Group Layout and Bind Group

In WebGPU, we do not set individual resources through an API, instead resources are bound in collections called bind groups.

3.1.4.1 Bind Group Layout

A bind group layout, which describes which stages the bind group's resources are visible to. The following configuration is shown in Figure 19.

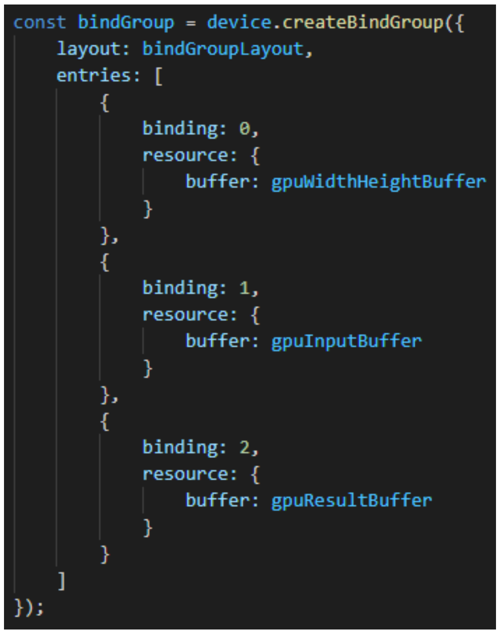


```
// BINDING GROUP LAYOUT
const bindGroupLayout = device.createBindGroupLayout({
    entries: [
        {
            binding: 0,
            visibility: GPUShaderStage.COMPUTE,
            buffer : {
                type: "read-only-storage"
            }
        } as GPUBindGroupLayoutEntry,
        {
            binding: 1,
            visibility: GPUShaderStage.COMPUTE,
            buffer: {
                type: "read-only-storage"
            }
        } as GPUBindGroupLayoutEntry,
        {
            binding: 2,
            visibility: GPUShaderStage.COMPUTE,
            buffer: {
                type: "storage"
            }
        } as GPUBindGroupLayoutEntry
    ]
});
```

Figure 19. Bind Group Layout

3.1.4.2 Bind Group

A bind group is an object that represents a collection of resources that can all be bound at once. This can be significantly more efficient than binding resources one at a time, and we can partition our bind groups by how often resource bindings change (per-frame, per-instance, etc.) to minimize the work done by the driver. The configuration is shown in Figure 20.



```
const bindGroup = device.createBindGroup({
    layout: bindGroupLayout,
    entries: [
        {
            binding: 0,
            resource: {
                buffer: gpuWidthHeightBuffer
            }
        },
        {
            binding: 1,
            resource: {
                buffer: gpuInputBuffer
            }
        },
        {
            binding: 2,
            resource: {
                buffer: gpuResultBuffer
            }
        }
    ]
});
```

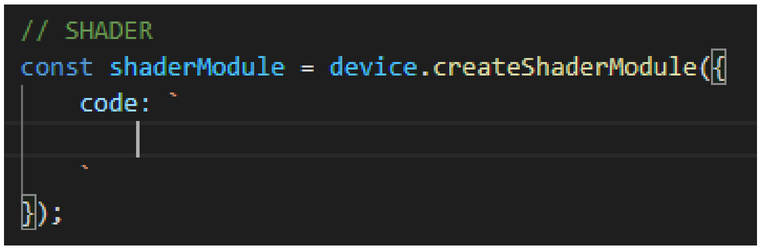
Figure 20. Bind Group

3.1.5 Shader Module and Compute Pipeline

We have created an input and output for the shader program, next is to create shader program module and compute pipeline.

3.1.5.1 Shader Module

In shader module, we will be providing shader language code, and this can be done by using the actual shading language code called WebGPU Shading Language – WGSL inside the **code** section, we can see in Figure 21.

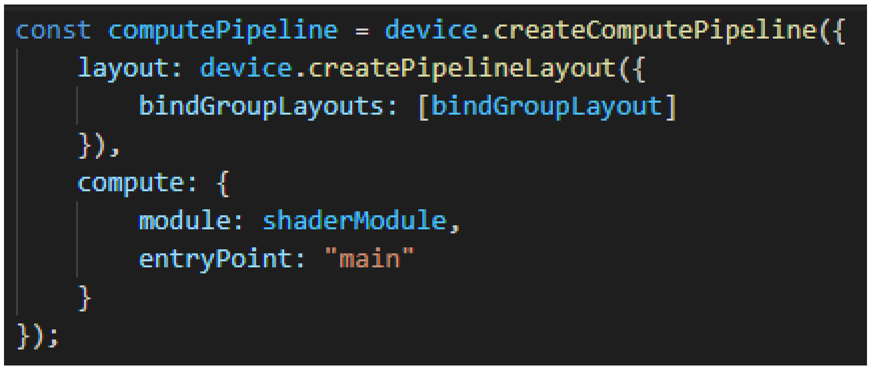


```
// SHADER
const shaderModule = device.createShaderModule({
    code: `

    `
});
```

Figure 21. Creating shader module

3.1.5.2 Compute Pipeline

Figure 22 creates compute pipeline is designed to handle calculation that can be done parallel with the graphics pipeline, with data flowing from input to output through the programmable compute shader stage. This can take advantage of the large numbers of parallel processors on the GPU, by providing memory sharing and thread synchronization.

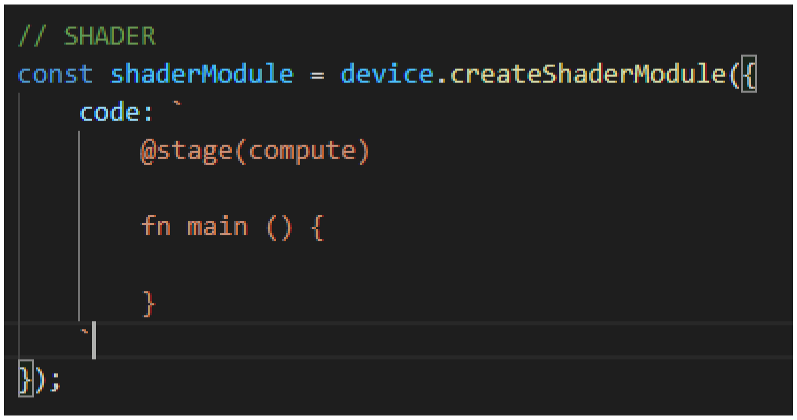


```
const computePipeline = device.createComputePipeline({
    layout: device.createPipelineLayout({
        bindGroupLayouts: [bindGroupLayout]
    }),
    compute: {
        module: shaderModule,
        entryPoint: "main"
    }
});
```

Figure 22. Compute Pipeline

3.1.6 WebGPU Shading Language – WGSL

WebGPU shading language is a shader language designed specifically for WebGPU to express the programs that run on the GPU. WGSL has two kinds of GPU commands, one of which is **draw command** to execute a **render pipeline** that contains inputs, outputs and attached resources. While the other is **dispatch command** executes a **compute pipeline** with only inputs and attached resources, both pipelines use shaders written in WGSL.

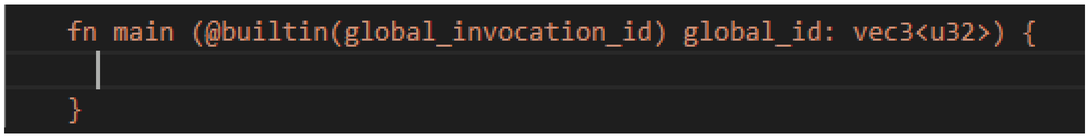


```
// SHADER
const shaderModule = device.createShaderModule({
    code: `
        @stage(compute)

        fn main () {

        }
    `
});
```

Figure 23. WebGPU Shading Language

In WGSL code in Figure 23, a **main** function has to be declaired for entry point of the shader program, we must also provide the **stage** of the main function which will be the compute stage since we have a **computePipeline.**

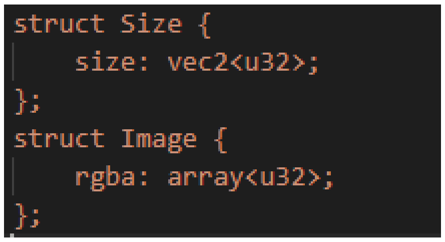
In **main** function we provide a built-in with a **global invocation id** and the variable name uder which we want to access this input built-in, in this case we use a three dimensional vector which contains einsteins's 32-bit value **global_id: vec3<u32>,** see Figure 24.



```
    fn main (@builtin(global_invocation_id) global_id: vec3<u32>) {

    }
```

Figure 24. A main function for WGSL

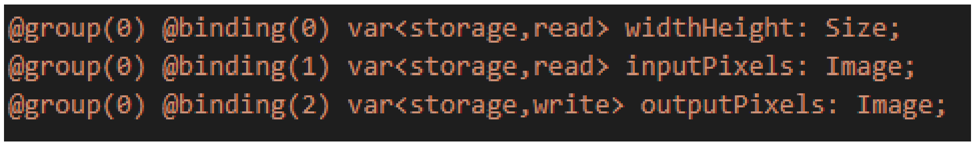
In Figure 25, we need to create two structures for the instance one for **size** which we will be using two-dimensional vector x, y and one for **image** that will hold an array of pixel values of the image.



```
struct Size {
    size: vec2<u32>;
};
struct Image {
    rgba: array<u32>;
};
```

Figure 25. Structure instance

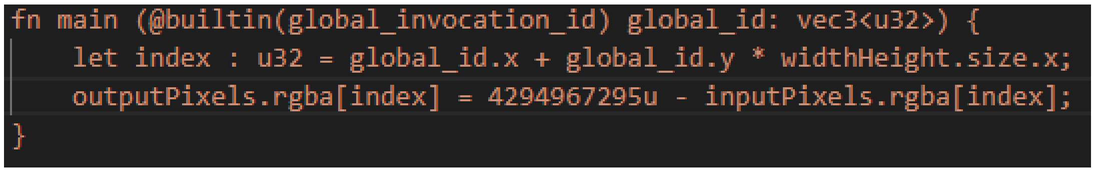
Now we can start mapping the buffers to these structs, for that, in Figure 26 we need to access the binding group that we already declared above.



```
@group(0) @binding(0) var<storage,read> widthHeight: Size;
@group(0) @binding(1) var<storage,read> inputPixels: Image;
@group(0) @binding(2) var<storage,write> outputPixels: Image;
```

Figure 26. Binding group access

We now have everything ready for the **main** function to work properly, as shown in Figure 27, navigate to the **main** function again we can now do the color inversion process.

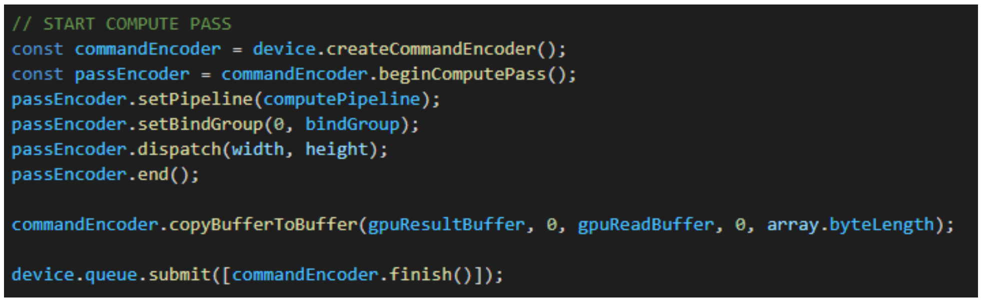


```
fn main (@builtin(global_invocation_id) global_id: vec3<u32>) {
    let index : u32 = global_id.x + global_id.y * widthHeight.size.x;
    outputPixels.rgba[index] = 4294967295u - inputPixels.rgba[index];
}
```

Figure 27. Color inversion

3.1.7 Compute Pass

The compute pass is where we will create and start the **pass encoder** based on the bind group and compute pipeline with bind group layout, refer to Figure 28.

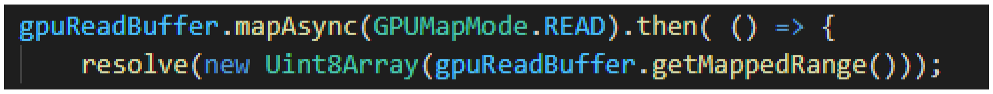


```
// START COMPUTE PASS
const commandEncoder = device.createCommandEncoder();
const passEncoder = commandEncoder.beginComputePass();
passEncoder.setPipeline(computePipeline);
passEncoder.setBindGroup(0, bindGroup);
passEncoder.dispatch(width, height);
passEncoder.end();

commandEncoder.copyBufferToBuffer(gpuResultBuffer, 0, gpuReadBuffer, 0, array.byteLength);

device.queue.submit([commandEncoder.finish()]);
```

Figure 28. Compute pass

3.1.8 Read the result

Lastly, in Figure 29 shows how to read the result from the read buffer.

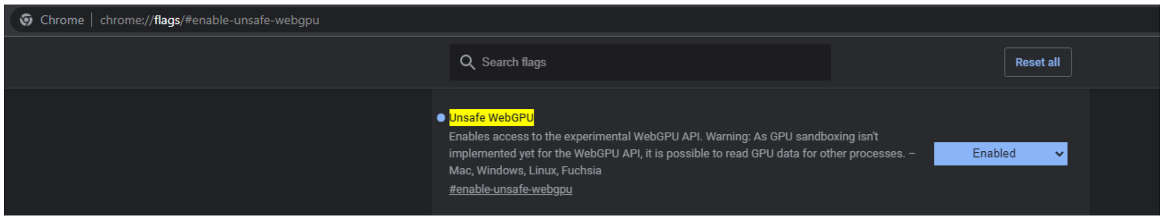


```
gpuReadBuffer.mapAsync(GPUMapMode.READ).then( () => {
    resolve(new Uint8Array(gpuReadBuffer.getMappedRange()));
```

Figure 29. Read result

3.1.9 Project Testing

The color inversion project is now ready to be tested, but first we need to install web browser that support WebGPU API, in this experiment, we use **chrome canary**, navigate to **chrome://flags/#enable-unsafe-webgpu** and enable **unsafe WebGPU** mode, as shown in Figure 30.



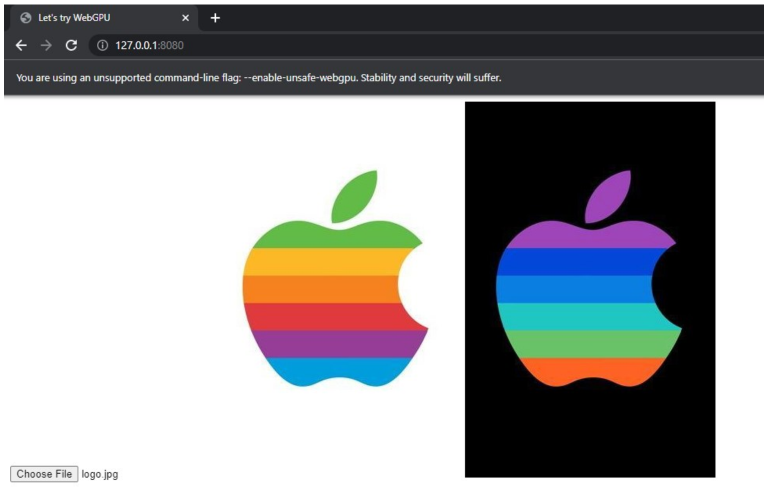
Chrome | chrome://flags/#enable-unsafe-webgpu

Unsafe WebGPU
Enables access to the experimental WebGPU API. Warning: As GPU sandboxing isn't implemented yet for the WebGPU API, it is possible to read GPU data for other processes. – Mac, Windows, Linux, Fuchsia
#enable-unsafe-webgpu

Enabled

Figure 30. Enable WebGPU flag

Next, we start up the project and access it with our localhost and choose the same jpeg image previously to see how it changes. In Figure 31, we can see the image on the right side with its original colors was inverted, meaning that we successfully create an application which GPU resource was being ultilized by the WebGPU API.



Let's try WebGPU

127.0.0.1:8080

You are using an unsupported command-line flag: --enable-unsafe-webgpu. Stability and security will suffer.

Figure 31. Final result

3.2 3D Object Rendering

WebGPU can do more than just color reversing, by using the GPU resource, it can be used to draw 3D objects, from simple triangles, cubes to an actual character or building.

3.2.1 Concept of Rendered 3D Object

The concept of rendering a 3D object, is to upload all data which does not change over time to the GPU memory, such as: **vertices, colors, textures**.

Most of the 3D graphics objects and all 3D graphics engines use **triangle parameters**, **triangle-list** to be specific. The reason is that the three vertices of a triangle are always in the plain, any more than three vertices would be considered to be **non-plain**, thus the render is ripple unless it was converted into a triangle. **Non-planar** is a degenerate and cannot be solved or rendered correctly, since three points are the minimum necessary to define a planar surface and any shapes can be simulated by using many triangles. In short, the triangle is the most important parameter in any 3D graphics engine.

Next is **renderer**, which has a function to draw a frame within the rendering command chain. For instance:

```
frame (camera, objects){
command = // initialize commands encoder
for each object of objects
// instruct all the objects to update their transformation given
// a camera
object.updateTransformation(camera)
// Give a command to draw themselves
object.drawYouself(commands)
// draw frame to canvas
command.execute()
```

3.2.2 Using GPU resource

To access gpu adapter and gpu device, we use "**navigator.gpu.requestAdapter"**, one thing worth noticing is that WebGPU API is an asynchronous function. Use either **Vanilla JS** or install **webgpu types** package to use webgpu api with **TypeScript.**

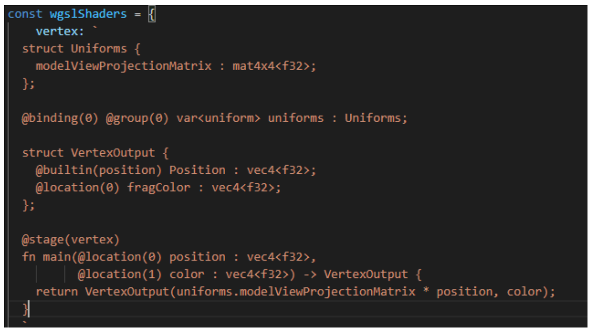
3.2.3 Rendering Object

Rendered objects are primitively triangles using a specified list of vertex positions. A complete pattern is called **Fragment**. The order of the vertices creates a winding direction that is used for the calling mechanism and causes the object to be rendered. The winding direction must be counterclockwise and not clockwise in order for the object to be successfully rendered. Some information about the object appearance is required for easier understanding, such as: **color, UVs - Unwrapped Version,** more like a net, to map texture of the object.

3.2.4 Shader

Shaders are computation structures to the GPU, which will execute the object rendering based on the compute units inside the GPU hardware. This can be done after all information of the 3D world has been uploaded to the GPU memory, like: **vertices, colors, transformation, textures, environment lights.**

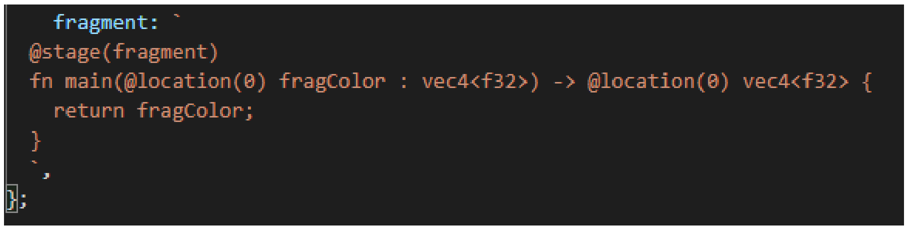
In this 3d rendering project, we create a shader using WebGPU Shading Language two type of shaders. In Figure 32, the first shader is the **Vertex** to calculate the position of the a vertex using the model provided model projection matrix.



```
const wgslShaders = {
    vertex: `
  struct Uniforms {
    modelViewProjectionMatrix : mat4x4<f32>;
  };

  @binding(0) @group(0) var<uniform> uniforms : Uniforms;

  struct VertexOutput {
    @builtin(position) Position : vec4<f32>;
    @location(0) fragColor : vec4<f32>;
  };

  @stage(vertex)
  fn main(@location(0) position : vec4<f32>,
          @location(1) color : vec4<f32>) -> VertexOutput {
    return VertexOutput(uniforms.modelViewProjectionMatrix * position, color);
  }
  `
```

Figure 32. Vertex structure

The other shader is the **Fragment** is used to determine the current pixel's color as shown in Figure 33.



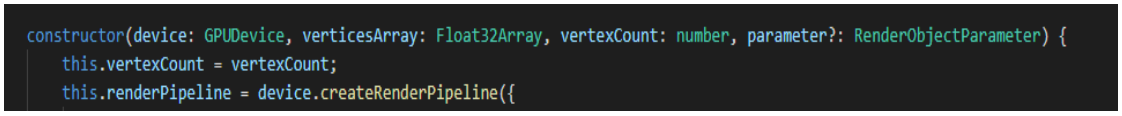
```
   fragment: `
 @stage(fragment)
 fn main(@location(0) fragColor : vec4<f32>) -> @location(0) vec4<f32> {
   return fragColor;
 }
 `,
};
```

Figure 33. Fragment structure

3.2.5 Object Initialization

A rendered object must also have a position and an orientation, this can be done via linear algebra, using transformation matrices. The idea is by multiplying all vertex positions with such matrix changes the position and the rotation of the whole object. For this to work, we will need linear algebra libraries such as **gl-matrix**. The transformation matrix is used together with the **camera** information.

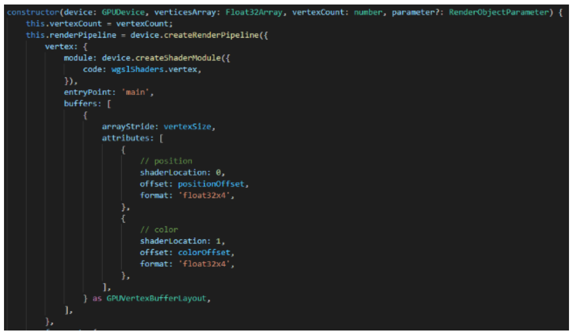
With this project, we first create some web GPU instructions, the first thing to do is to define a rendering pipeline, as described in Figure 34.



```
constructor(device: GPUDevice, verticesArray: Float32Array, vertexCount: number, parameter?: RenderObjectParameter) {
    this.vertexCount = vertexCount;
    this.renderPipeline = device.createRenderPipeline({
```

Figure 34. Create GPU instruction

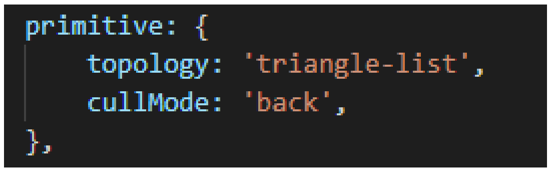
Within the rendering pipeline, we calculate the vertices and fragments that have been pre-defined by using WebGPU Shader Language in the shader.



```
constructor(device: GPUDevice, verticesArray: Float32Array, vertexCount: number, parameter?: RenderObjectParameter) {
    this.vertexCount = vertexCount;
    this.renderPipeline = device.createRenderPipeline({
        vertex: {
            module: device.createShaderModule({
                code: wgslShaders.vertex,
            }),
            entryPoint: 'main',
            buffers: [
                {
                    arrayStride: vertexSize,
                    attributes: [
                        {
                            // position
                            shaderLocation: 0,
                            offset: positionOffset,
                            format: 'float32x4',
                        },
                        {
                            // color
                            shaderLocation: 1,
                            offset: colorOffset,
                            format: 'float32x4',
                        },
                    ],
                } as GPUVertexBufferLayout,
            ],
        },
```

Figure 35. Vertices calculation

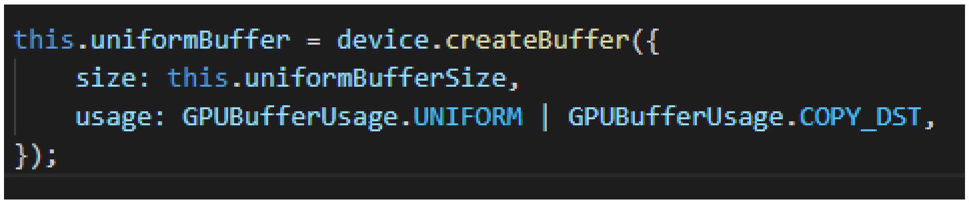
Figure 35 above shows the calculation of the vertices, where we also set a lot of low level values such as the changes of the position and color.



```
primitive: {
    topology: 'triangle-list',
    cullMode: 'back',
},
```

Figure 36. Object topology

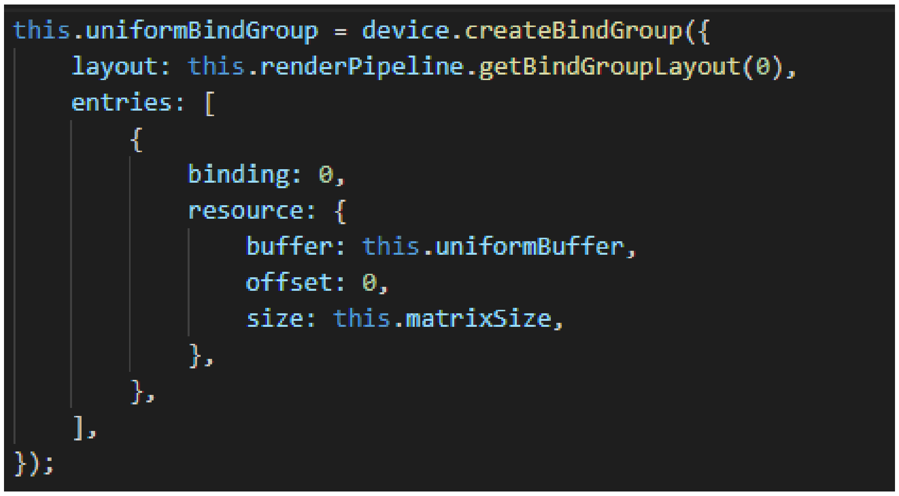
There are various types of topologies that can be use for drawing object, but the main and most fundamental one will always be the triangle shape, as shown in Figure 36, even thought in some occasion we might see some cube looking object, consider just one side of the cube, it is actually two half of an triangle combined that creates full cube.



```
this.uniformBuffer = device.createBuffer({
    size: this.uniformBufferSize,
    usage: GPUBufferUsage.UNIFORM | GPUBufferUsage.COPY_DST,
});
```

Figure 37. Buffer

In Figure 37 above, we then again create buffers, which will do a contiguous block of memory in the GPU that stores the rendering data for a model. First buffer is the uniform and the copy destination.

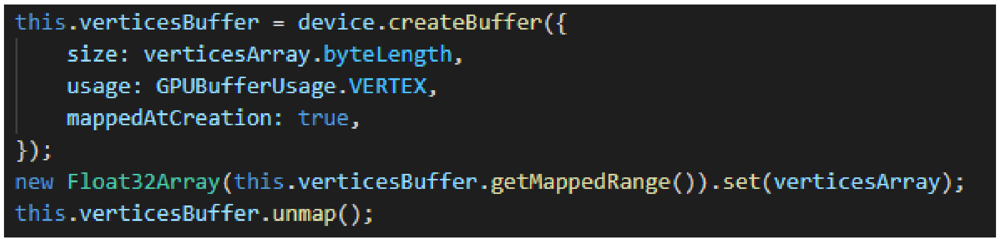


```
this.uniformBindGroup = device.createBindGroup({
    layout: this.renderPipeline.getBindGroupLayout(0),
    entries: [
        {
            binding: 0,
            resource: {
                buffer: this.uniformBuffer,
                offset: 0,
                size: this.matrixSize,
            },
        },
    ],
});
```

Figure 38. Bind group

Having created the buffer for the transformation matrix, we now combine the resources that may changes overtime with the bind group, this can significantly minimize the work done by the driver, as shown in Figure 38.

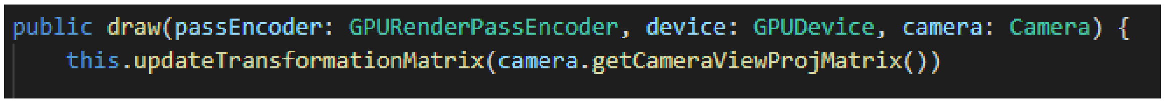
To capture the object changes overtime, in Figure 39 we must define the vertices array and upload this array to the GPU memory.



```
this.verticesBuffer = device.createBuffer({
    size: verticesArray.byteLength,
    usage: GPUBufferUsage.VERTEX,
    mappedAtCreation: true,
});
new Float32Array(this.verticesBuffer.getMappedRange()).set(verticesArray);
this.verticesBuffer.unmap();
```

Figure 39. Vertices array

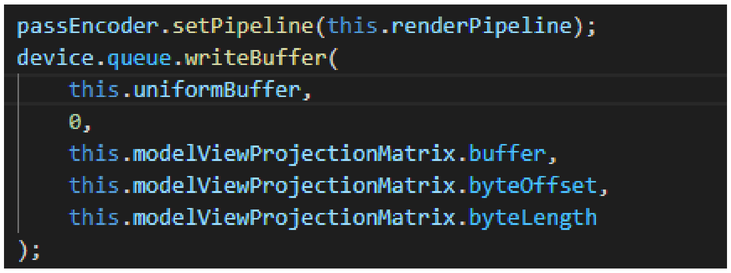
Now we create a draw function to update its object transformation given a perspective from the outside, see in Figure 40.



```
public draw(passEncoder: GPURenderPassEncoder, device: GPUDevice, camera: Camera) {
    this.updateTransformationMatrix(camera.getCameraViewProjMatrix())
```

Figure 40. Draw function

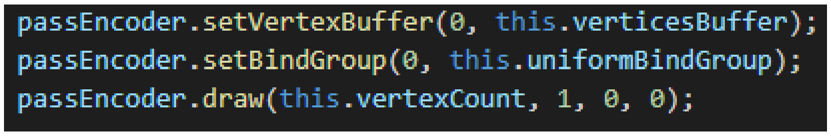
As shown in Figure 41, within the update transformation matrix, we setting up the current rendering pipeline, write the current transformation matrix to the proper buffer, and set the current vertex as the objects' own buffer.



```
passEncoder.setPipeline(this.renderPipeline);
device.queue.writeBuffer(
    this.uniformBuffer,
    0,
    this.modelViewProjectionMatrix.buffer,
    this.modelViewProjectionMatrix.byteOffset,
    this.modelViewProjectionMatrix.byteLength
);
```

Figure 41. Compute pipeline

After the pipeline, we need to set the binding group and call the draw functions, this is where all 3D objects are given the instructions how to draw themselves, following Figure 42.

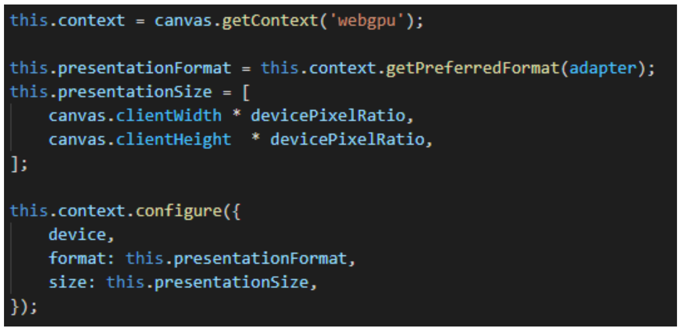


```
passEncoder.setVertexBuffer(0, this.verticesBuffer);
passEncoder.setBindGroup(0, this.uniformBindGroup);
passEncoder.draw(this.vertexCount, 1, 0, 0);
```

Figure 42. Binding group

3.2.6 Renderer

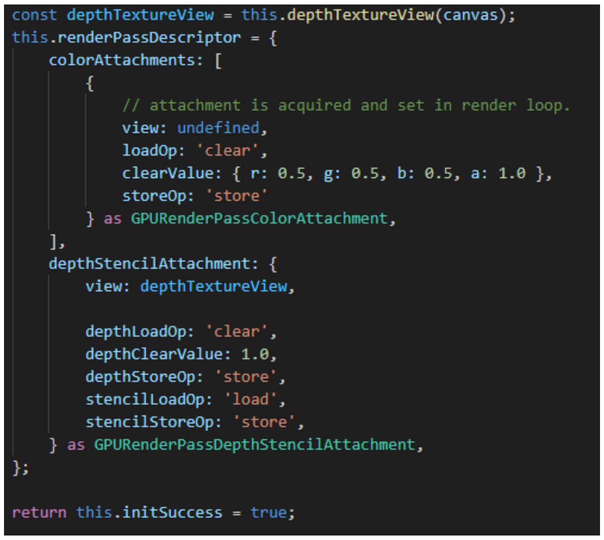
Initializing the renderer, one must provide a canvas, in canvas we create a swap chain that enables the rendered frames are transmitted from the GPU memory to the system memory and then to the canvas, as shown in Figure 43.



```
this.context = canvas.getContext('webgpu');

this.presentationFormat = this.context.getPreferredFormat(adapter);
this.presentationSize = [
    canvas.clientWidth * devicePixelRatio,
    canvas.clientHeight  * devicePixelRatio,
];

this.context.configure({
    device,
    format: this.presentationFormat,
    size: this.presentationSize,
});
```

Figure 43. Canvas for drawing

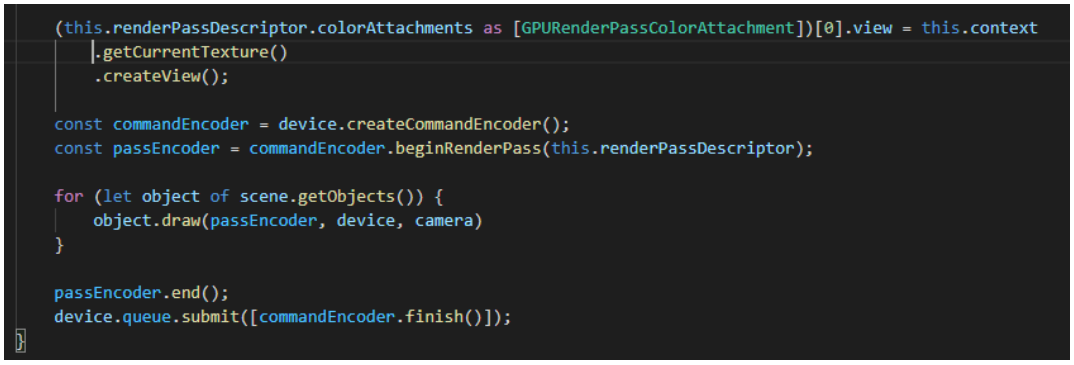
In Figure 44, we also need to define a render pass descriptor with serveral attachments.



```
const depthTextureView = this.depthTextureView(canvas);
this.renderPassDescriptor = {
    colorAttachments: [
        {
            // attachment is acquired and set in render loop.
            view: undefined,
            loadOp: 'clear',
            clearValue: { r: 0.5, g: 0.5, b: 0.5, a: 1.0 },
            storeOp: 'store'
        } as GPURenderPassColorAttachment,
    ],
    depthStencilAttachment: {
        view: depthTextureView,

        depthLoadOp: 'clear',
        depthClearValue: 1.0,
        depthStoreOp: 'store',
        stencilLoadOp: 'load',
        stencilStoreOp: 'store',
    } as GPURenderPassDepthStencilAttachment,
};

return this.initSuccess = true;
```

Figure 44. Pass descriptor

In frame function, will do some initialization tell all the objects of the scene to draw themselves and then submit the command chain and let the GPU do the work, see in Figure 45.

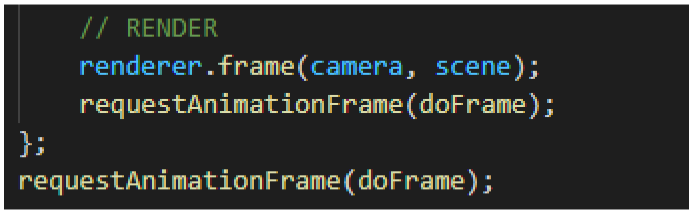


```
    (this.renderPassDescriptor.colorAttachments as [GPURenderPassColorAttachment])[0].view = this.context
        .getCurrentTexture()
        .createView();

    const commandEncoder = device.createCommandEncoder();
    const passEncoder = commandEncoder.beginRenderPass(this.renderPassDescriptor);

    for (let object of scene.getObjects()) {
        object.draw(passEncoder, device, camera)
    }

    passEncoder.end();
    device.queue.submit([commandEncoder.finish()]);
}
```

Figure 45. Frame function

3.2.7 Animation Loop

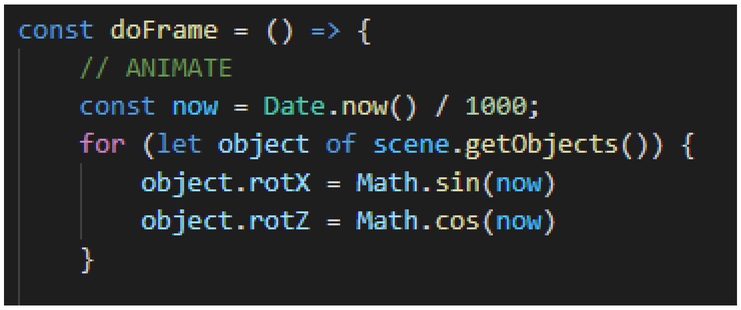
At this point we already have a scene with 3D objects, a camera, a renderer that instructs GPU to render a frame, a canvas where the frames are displayed. Now all we need is an animation loop to instruct the renderer to draw a frame, in Figure 46, here we can call the **requestAnimationFrame** function to request the browser to do an animation frame.



```
    // RENDER
    renderer.frame(camera, scene);
    requestAnimationFrame(doFrame);
};
requestAnimationFrame(doFrame);
```

Figure 46. Animation frame request function

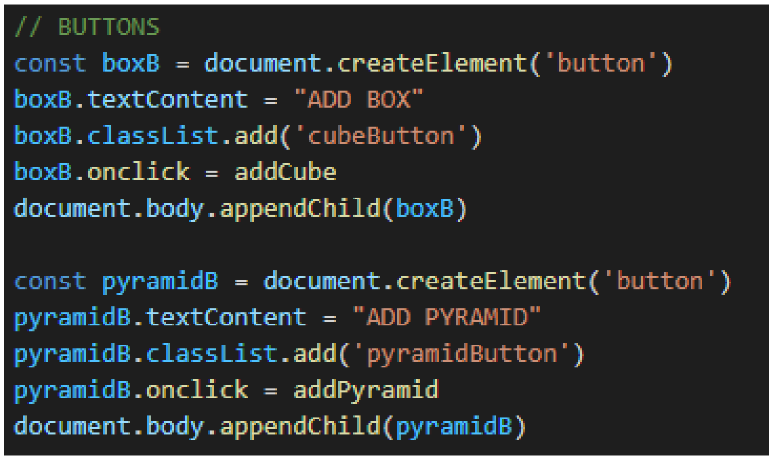
There are many types of animation we can do when creating the frame, in this particular project, we simply create an animation to rotate the objects, as shown in Figure 47.



```
const doFrame = () => {
    // ANIMATE
    const now = Date.now() / 1000;
    for (let object of scene.getObjects()) {
        object.rotX = Math.sin(now)
        object.rotZ = Math.cos(now)
    }
```

Figure 47. Animation type

In addition to the project, we create some buttons to add either a cube or a pyramid, see Figure 48.

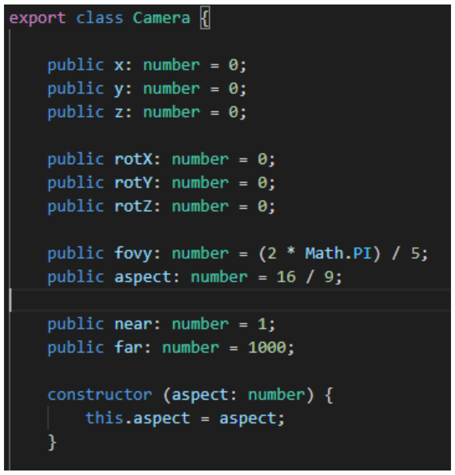


```
// BUTTONS
const boxB = document.createElement('button')
boxB.textContent = "ADD BOX"
boxB.classList.add('cubeButton')
boxB.onclick = addCube
document.body.appendChild(boxB)

const pyramidB = document.createElement('button')
pyramidB.textContent = "ADD PYRAMID"
pyramidB.classList.add('pyramidButton')
pyramidB.onclick = addPyramid
document.body.appendChild(pyramidB)
```

Figure 48. Interaction buttons

3.2.8 Camera

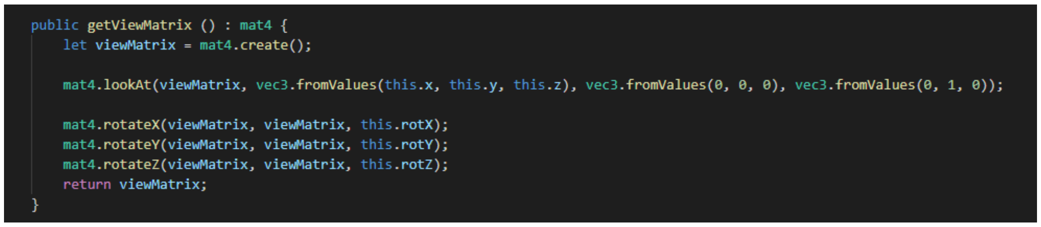
A camera has a looking position, a looking direction and can be rotated. For the camera, it uses a projection matrix which has the attributes aspect ratio, field of view, near and far values. Objects within the camera are projected on a canvas frame. In Figure 49, we first create a camera class which defines three dimensional, so we can rotate, zoom in and out of the object.



```
export class Camera {

    public x: number = 0;
    public y: number = 0;
    public z: number = 0;

    public rotX: number = 0;
    public rotY: number = 0;
    public rotZ: number = 0;

    public fovy: number = (2 * Math.PI) / 5;
    public aspect: number = 16 / 9;

    public near: number = 1;
    public far: number = 1000;

    constructor (aspect: number) {
        this.aspect = aspect;
    }
```

Figure 49. Camera class

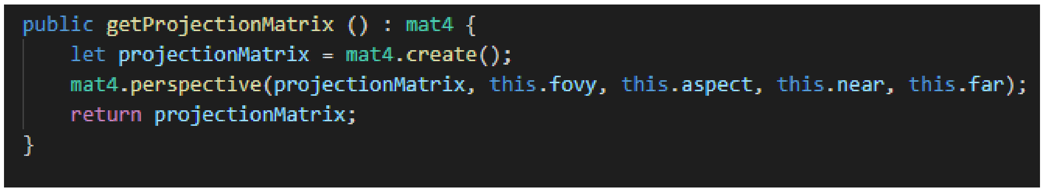
In Figure 50 shows, the creation of the rotational mechanism for the camera.



```
public getViewMatrix () : mat4 {
    let viewMatrix = mat4.create();

    mat4.lookAt(viewMatrix, vec3.fromValues(this.x, this.y, this.z), vec3.fromValues(0, 0, 0), vec3.fromValues(0, 1, 0));

    mat4.rotateX(viewMatrix, viewMatrix, this.rotX);
    mat4.rotateY(viewMatrix, viewMatrix, this.rotY);
    mat4.rotateZ(viewMatrix, viewMatrix, this.rotZ);
    return viewMatrix;
}
```

Figure 50. Camera rotation

In Figure 51 we create the depth mechanism for the camera, so we can zoom in and out of the object.

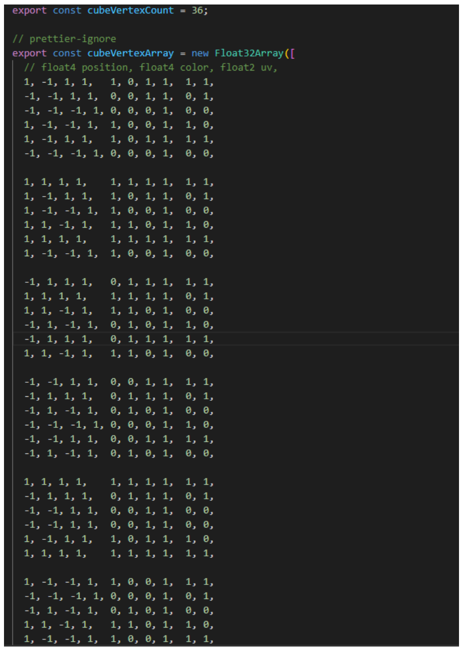


```
public getProjectionMatrix () : mat4 {
    let projectionMatrix = mat4.create();
    mat4.perspective(projectionMatrix, this.fovy, this.aspect, this.near, this.far);
    return projectionMatrix;
}
```

Figure 51. Depth mechanism

3.2.9 Vertices

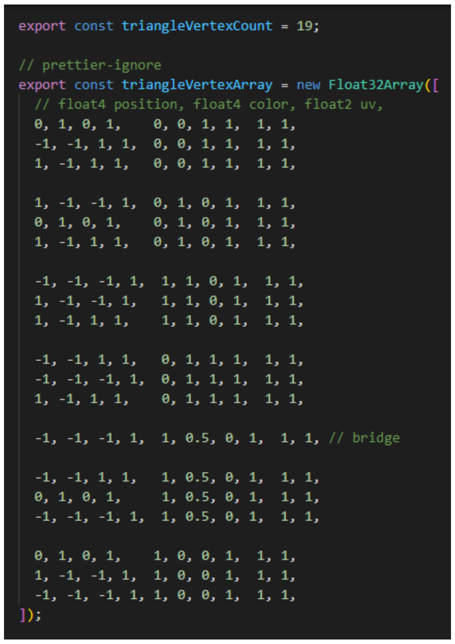
In the canvas we now have all the tools we need, the camera, the animation loop, renderer, and the shader for the object. At this point, we now create the cube vertices as shown in Figure 52.



```
export const cubeVertexCount = 36;

// prettier-ignore
export const cubeVertexArray = new Float32Array([
  // float4 position, float4 color, float2 uv,
  1, -1, 1, 1,   1, 0, 1, 1,  1, 1,
  -1, -1, 1, 1,  0, 0, 1, 1,  0, 1,
  -1, -1, -1, 1, 0, 0, 0, 1,  0, 0,
  1, -1, -1, 1,  1, 0, 0, 1,  1, 0,
  1, -1, 1, 1,   1, 0, 1, 1,  1, 1,
  -1, -1, -1, 1, 0, 0, 0, 1,  0, 0,

  1, 1, 1, 1,    1, 1, 1, 1,  1, 1,
  1, -1, 1, 1,   1, 0, 1, 1,  0, 1,
  1, -1, -1, 1,  1, 0, 0, 1,  0, 0,
  1, 1, -1, 1,   1, 1, 0, 1,  1, 0,
  1, 1, 1, 1,    1, 1, 1, 1,  1, 1,
  1, -1, -1, 1,  1, 0, 0, 1,  0, 0,

  -1, 1, 1, 1,   0, 1, 1, 1,  1, 1,
  1, 1, 1, 1,    1, 1, 1, 1,  0, 1,
  1, 1, -1, 1,   1, 1, 0, 1,  0, 0,
  -1, 1, -1, 1,  0, 1, 0, 1,  1, 0,
  -1, 1, 1, 1,   0, 1, 1, 1,  1, 1,
  1, 1, -1, 1,   1, 1, 0, 1,  0, 0,

  -1, -1, 1, 1,  0, 0, 1, 1,  1, 1,
  -1, 1, 1, 1,   0, 1, 1, 1,  0, 1,
  -1, 1, -1, 1,  0, 1, 0, 1,  0, 0,
  -1, -1, -1, 1, 0, 0, 0, 1,  1, 0,
  -1, -1, 1, 1,  0, 0, 1, 1,  1, 1,
  -1, 1, -1, 1,  0, 1, 0, 1,  0, 0,

  1, 1, 1, 1,    1, 1, 1, 1,  1, 1,
  -1, 1, 1, 1,   0, 1, 1, 1,  0, 1,
  -1, -1, 1, 1,  0, 0, 1, 1,  0, 0,
  -1, -1, 1, 1,  0, 0, 1, 1,  0, 0,
  1, -1, 1, 1,   1, 0, 1, 1,  1, 0,
  1, 1, 1, 1,    1, 1, 1, 1,  1, 1,

  1, -1, -1, 1,  1, 0, 0, 1,  1, 1,
  -1, -1, -1, 1, 0, 0, 0, 1,  0, 1,
  -1, 1, -1, 1,  0, 1, 0, 1,  0, 0,
  1, 1, -1, 1,   1, 1, 0, 1,  1, 0,
  1, -1, -1, 1,  1, 0, 0, 1,  1, 1,
```

Figure 52. Cube vertices

In Figure 53, we create the triangle vertices array.

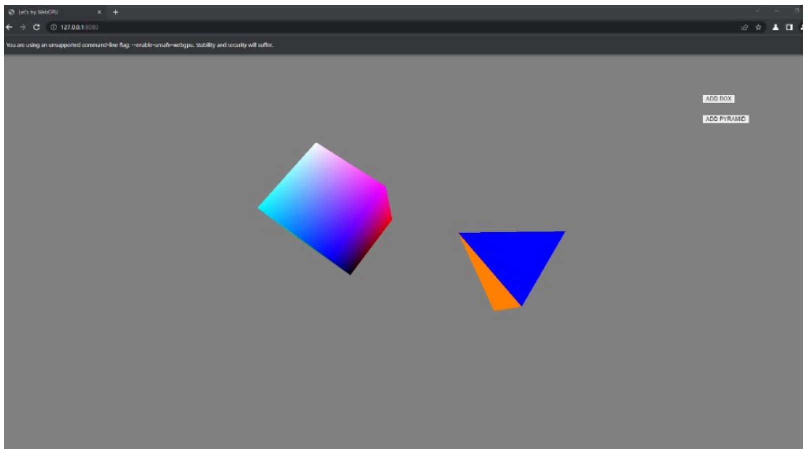


```
export const triangleVertexCount = 19;

// prettier-ignore
export const triangleVertexArray = new Float32Array([
  // float4 position, float4 color, float2 uv,
  0, 1, 0, 1,    0, 0, 1, 1,  1, 1,
  -1, -1, 1, 1,  0, 0, 1, 1,  1, 1,
  1, -1, 1, 1,   0, 0, 1, 1,  1, 1,

  1, -1, -1, 1,  0, 1, 0, 1,  1, 1,
  0, 1, 0, 1,    0, 1, 0, 1,  1, 1,
  1, -1, 1, 1,   0, 1, 0, 1,  1, 1,

  -1, -1, -1, 1,  1, 1, 0, 1,  1, 1,
  1, -1, -1, 1,   1, 1, 0, 1,  1, 1,
  1, -1, 1, 1,    1, 1, 0, 1,  1, 1,

  -1, -1, 1, 1,   0, 1, 1, 1,  1, 1,
  -1, -1, -1, 1,  0, 1, 1, 1,  1, 1,
  1, -1, 1, 1,    0, 1, 1, 1,  1, 1,

  -1, -1, -1, 1,  1, 0.5, 0, 1,  1, 1, // bridge

  -1, -1, 1, 1,   1, 0.5, 0, 1,  1, 1,
  0, 1, 0, 1,     1, 0.5, 0, 1,  1, 1,
  -1, -1, -1, 1,  1, 0.5, 0, 1,  1, 1,

  0, 1, 0, 1,    1, 0, 0, 1,  1, 1,
  1, -1, -1, 1,  1, 0, 0, 1,  1, 1,
  -1, -1, -1, 1, 1, 0, 0, 1,  1, 1,
]);
```

Figure 53. Triangle vertices

3.2.10 Result

For testing purposes, the application will be run on a local machine as can be seen in Figure 54, which shows the result of the complete application. First time, when starting the application with **npm** package, the default web browser is set by the system, normal it will use Microsoft Edge or set by ourselves during the usage. For that, again, there will be nothing shown on the browser, because normal browsers like Edge, Firefox or Google Chrome does not have the WebGPU API flag supported by the developers. Only by using Chrome Canary or Firefox Nightly, can we truly see the properly working application.



You are using an unsupported command-line flag: --enable-unsafe-webgpu. Stability and security will suffer.

Figure 54. Fully rendered 3D objects

Now, try adding in some boxes and pyramids. In Figure 55 the result is expected to have many more boxes and pyramids at this point.

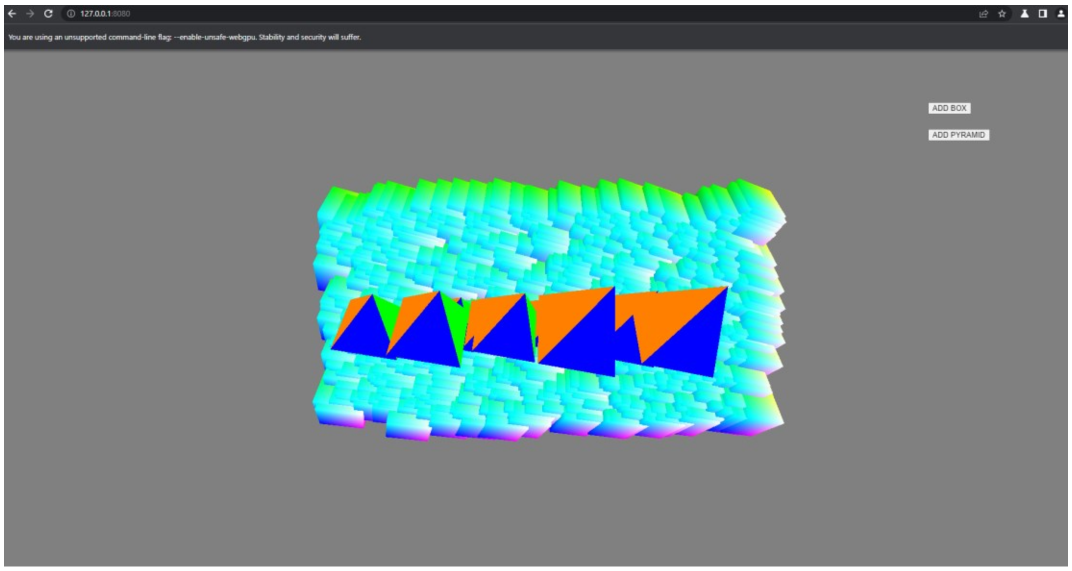
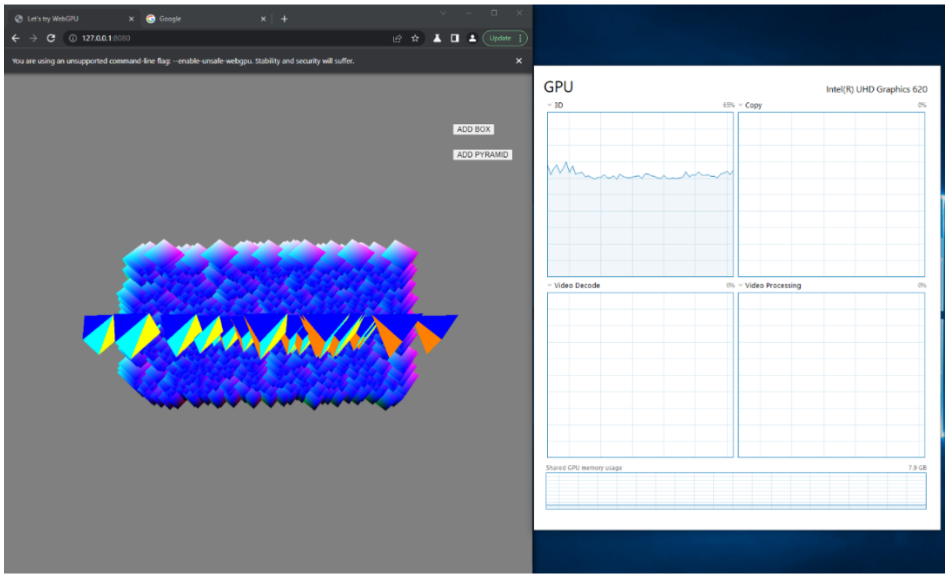

127.0.0.1:8080
You are using an unsupported command-line flag: --enable-unsafe-webgpu. Stability and security will suffer.
ADD BOX
ADD PYRAMID

Figure 55. Adding more objects into the canvas

3.2.11 Resource Consumption

Testing out the rendering consumption of WebGPU technology, we need two computers, and call it computer A and computer B, one of which will host the WebGPU and the other will access it. For this demonstration, we use computer A for WebGPU hosting and computer B try to render the object.

Normally, whenever a web application is being held by a server, that server's resources will be used for rendering and processing all the elements of that specific application. This might be worth the consideration, since WebGPU application is fairly new technology which allows the objects to be rendered faster by using a tremendous amount of GPU resources. Surprisingly, during the experiment, when we tried to use a computer B to access the web application, and try to render the object, it was not the computer A resources that was used for rendering, but computer B resources itself. Thanks to this new **"navigator.gpu.requestAdapter"**, which will navigate to the nearest GPU adapter or that specific device to render that object.



You are using an unsupported command-line flag: --enable-unsafe-webgpu. Stability and security will suffer.

Figure 56. Resource comsumption monitor

At this point, the result gives us the idea of how this WebGPU should be implemented and what benefits it will bring afterward. One thing can be assured, that if we are planning to host this new technology using cloud technology, it will not cause us any extra fees during high demand.

4 CONCLUSION

The purpose of this thesis was to create a simple web application, to demonstrate how convenient, how fast and cost effective this new Web API truly is. Though WebGL has been around for decades, making many fundamental improvements, but compared to WebGPU, the new WebGPU definitely a major technology upgrade for construct.

The API is cleaner, simpler, and easier to understand, while OpenGL's way of doing things was never exactly popular. Applications have much more control over exactly how rendering happens and can do more to optimize performance. It cuts out a huge amount of complexity and overhead in the graphics driver, bringing much simplier and smaller software driver that reduces bugs in near future.

Browsers do a lot of validation and security checks, which has a performance overhead. With WebGPU it can do most of that in advance instead of during rendering, reducing the browser overhead. This without a doubt will be the bleeding edge of the latest web technologies and will be amongst the first engines to benefit from the improvements that come with it.

REFERENCES

Aron Granberg. WebGPU Shading Language https://sotrh.github.io/learn-wgpu/beginner/tutorial3-pipeline/#what-s-a-pipeline [Accessed 7 April 2022]

Ashley. WebGPU compared to WebGL. WWW document. Available at: https://www.construct.net/en/blogs/ashleys-blog-2/webgl-webgpu-construct-1519 [Accessed 28 March 2022]

François Beaufort. WebGPU. WWW document. Available at: https://web.dev/gpu-compute/ [Accessed 15 March 2022]

JavaScript Language. WWW document. Available at:https://en.wikipedia.org/wiki/JavaScript#Misplaced_trust_in_the_client [Accessed 14 April 2022]

Stevewhims. Compute pipeline | Microsoft Docs. WWW document. Available at: https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/compute-pipeline [Accessed 20 April 2022]

TypeScript. WWW document. Available at: https://en.wikipedia.org/wiki/TypeScript#cite_note-7 [Accessed 14 April 2022]

Vertex, fragment. WWW document. Available at: https://sotrh.github.io/learn-wgpu/beginner/tutorial3-pipeline/#what-s-a-pipeline [Accessed 22 April 2022]

LIST OF FIGURES