< 문서 >

프로젝트 명: WebGPU-based 3D Rendering Framework 개발

요 약

WebGPU는 2021년에 출시된 차세대 웹 그래픽 API로, 기존의 WebGL을 대체하는 기술이다. WebGPU는 GPU를 이용해 웹 환경에서 그래픽 렌더링과 GPU 컴퓨팅을 구현할 수 있다. 기존의 WebGL과 비교했을 때, WebGPU로 더 나은 성능과 최적화를 구현할 수 있다. WebGPU를 이용한 프레임워크 개발은 웹 환경에서 3D 그래픽을 렌더링할 때 도움이 될 것이며 게임, VR, AR 등 다양한 웹 분야에서 사용될 수 있을 것이다. 프레임워크 사용자는 웹과 그래픽스 API에 깊게 접근하지 않아도 쉽게 웹 페이지에서 그래픽을 그릴 수 있을 것이다. 이렇게 개발된 WebGPU 기반 3D 렌더링 라이브러리는 많은 양의 3D 오브젝트를 그릴 수 있었으며 Desktop, Mobile 환경 모두에서 동작하는 것을 확인할 수 있었다.

1. 서론

1.1. 연구배경

WebGPU는 차세대 웹 그래픽 API로, 기존의 WebGL을 대체하는 기술이다. WebGL도 웹 브라우저에서 그래픽 요소를 렌더링하기 위해 출시된 API지만 Direct3D 12, Vulkan 등 하드웨어와 직접 통신하는 API보다 성능이 낮다는 단점이 있다. 이 단점을 해결하기 위해 웹에서 GPU에 접근할 수 있는 기술이 개발되었고 WebGPU가 등장하였다.

WebGL과 WebGPU는 웹 브라우저에서 그래픽을 렌더링을 지원하는 API다. 따라서 그래픽스 API와 유사한 파이프라인으로 동작한다. 버퍼에 있는 데이터를 순회하는 것으로 점, 선, 삼각형을 그려 그래픽을 표현한다. WebGPU, WebGL 모두 HTML 파일에서 <canvas> 태그로 지정된 공간에 그래픽 요소를 그릴 수 있다.

WebGPU는 그래픽 렌더링뿐만 아니라 웹 환경에서 그래픽과 GPU 컴퓨팅에도 사용할 수 있다. WebGPU는 하드웨어 접근으로 WebGL보다 더 높은 성능을 보여주며 멀티 스레딩과 크로스 플랫폼 동작을 지원한다.

1.2. 연구 목표

WebGPU를 이용한 프레임워크 개발을 목표로 연구를 진행한다. 프레임워크는 생산성과 확장성을 고려해 개발할 것이며 개발된 프레임워크를 이용한 DemoScene을 통해 프레임워크 사용 방법과 성능을 소개할 것이다. 다음으로 WebGPU가 크로스 플랫폼 상에서 그래픽을 렌더링을 할 수 있는지 알아볼 것이다. 마지막으로 많은양의 3D 모델을 렌더링 하는 것으로 스트레스 테스트를 진행할 것이다.

개발하는 프레임워크는 게임 개발이 가능한 생산성과 기능을 추가할 수 있는 확장성, 추후 성능 테스트를 목표로 하고 있는 만큼 프레임워크를 구조적으로 만들 것이다. 기능으로는 텍스처 매핑, 셰이딩을 이용한 3D 모델 렌더링을 기본으로 육면체, 구, 사면체의 도형과 2D Sprite 등의 요소를 렌더링할 수 있게 구현할 것이다. 또한 렌더링 프레임워크에 필요한 카메라, 라이팅과 같은 부가

기능을 구현할 것이다. 마지막으로 구현한 기능들을 모듈화 하여 프레임워크로 기능을 할 수 있도록 상호작용과 확장이 가능하게 구현할 것이다.

1.3. 기대 효과

개발한 그래픽 렌더링 라이브러리는 웹 그래픽의 바탕이 될 것이다. 웹 게임 개발과 3D 웹 구현에 큰 도움을 줄 것이다. WebGPU를 사용하면서 현재 개발하고 있는 웹 게임의 성능을 향상시켜줄 것이며 추후 개발에 있어서 생산성을 높일 수 있을 것이다. 추후 고성능 게임, VR, AR 데이터 시각화 등 다양한 분야에서 사용될 수 있을 것이다.

프레임워크를 이용한 연구 진행에도 도움이 될 것이다. API간의 렌더링 성능 비교, 플랫폼간 성능 비교가 가능할 것이며 연구를 위한 배경 지식 습득에도 사용될 수 있을 것이다.

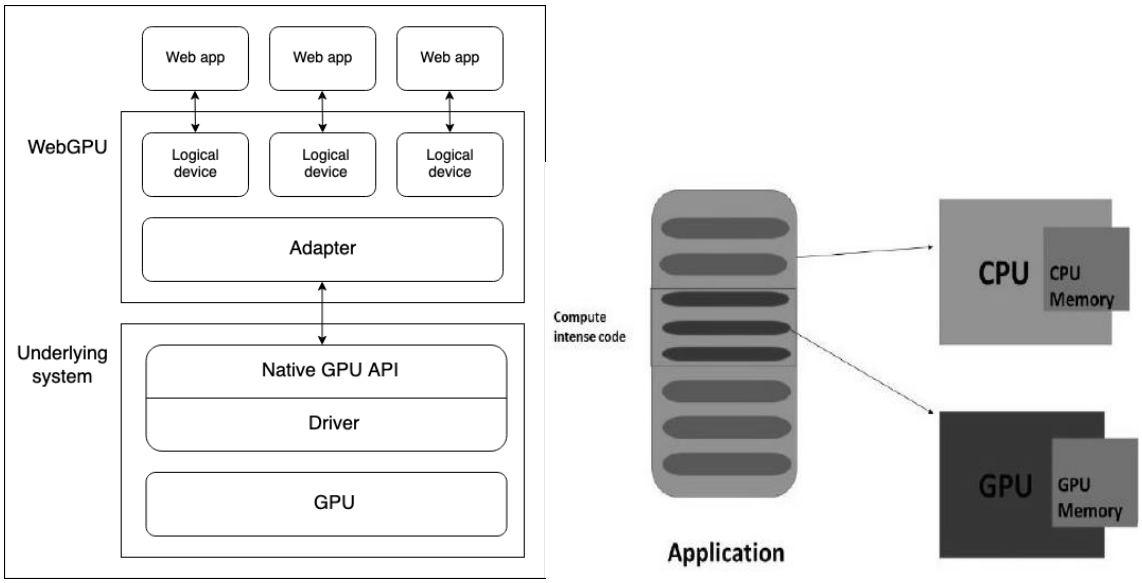
개발된 프레임워크는 Github를 통해 Public Repository로 배포될 것이다. 이는 적은 양의 WebGPU 예제를 보충할 수 있을 것이며 추후 WebGPU 연구를 시도할 때 참고자료로서 도움될 것이다.

2. 배경 지식 및 관련 연구

WebGPU는 Web 기반 API이므로 HTML/JavaScript/CSS에 대한 기초지식을 요구한다. 또한 배열 매핑, 구조 분해 할당, Typed 배열, async/await, ES6 Module에 대한 이해를 요구한다. 마지막으로 WebGPU의 셰이딩 언어인 WGSL은 Rust와 비슷한 문법 체계를 가지고 있으며 셰이딩 언어에 대한 이해도 필요하다.

2.1.1. WebGPU

WebGPU는 2021년 출시된 API이다. 그래픽 렌더링, GPGPU, 컴퓨팅 연산 등의 작업을 수행할 수 있는 API다. WebGPU는 Direct3D 12, Vulkan 등 Native GPU API를 호출하여 웹 상에서 GPU 연산이 가능하도록 한다.



브라우저는 WebGPU API를 통해 하드웨어와 통신한다. Adapter 객체는 Native GPU API에게 명령을 전달하며 그에 따른 결과를 받아오는 브릿지 역할을 수행한다.

2.1.2. WebGL

WebGL(Web Graphics Library)은 2011년 출시된 API이다. WebGL은 웹 브라우저 내에서 인터랙티브한 3D 그래픽을 구현할 수 있도록 하는 크로스 플랫폼이며 HTML5의 Canvas를 통해 3D 그래픽과 애니메이션을 렌더링할 수 있다.

2.1.3. Canvas

Canvas는 동적인 그래픽 콘텐츠를 그리기 위해 제공되는 HTML 요소로, 2D 그래픽과 이미지 조작을 가능하게 한다. Canvas는 JavaScript와 함께 사용되어 웹 페이지 내에서 그림, 게임, 데이터 시각화 등의 다양한 그래픽을 그릴 수 있는 공간을 제공한다. 픽셀 단위로 그림을 그리는 데 사용되며, 선, 사각형, 원, 텍스트 등을 그릴 수 있는 메서드를 제공한다. 이를 통해 이미지, 애니메이션과 인터랙티브 콘텐츠를 만들 수 있다. Canvas는 플러그인 없이도 다양한 브라우저에서 지원되며, 웹 애플리케이션의 그래픽 기능을 확장하는 데 중요한 역할을 한다.

2.1.4. WGSL

WGSL(WebGPU Shading Language)은 WebGPU를 위한 전용 셰이딩 언어로, GPU에서 실행되는 셰이더 프로그램을 작성하는 데 사용된다. WGSL은 웹 환경에 최적화된 설계로, 기존 셰이딩 언어인 GLSL(OpenGL Shading Language) 및 HLSL(High-Level Shading Language)과 유사하지만, 웹의 보안과 성능 요구 사항을 충족하기 위해 새로 개발되었다. Rust의 코드 스타일과 유사한 부분이 있으며 이를 통해 정점 셰이더와 프래그먼트 셰이더를 작성하여 그래픽 렌더링 파이프라인을 제어하고 3D 그래픽 및 연산 작업을 수행할 수 있다. WGSL은 WebGPU의 주요 구성 요소로, 브라우저에서 그래픽 렌더링과 GPU 컴퓨팅 작업을 안전하고 실행할 수 있게 한다.

2.1.5. Typed Array

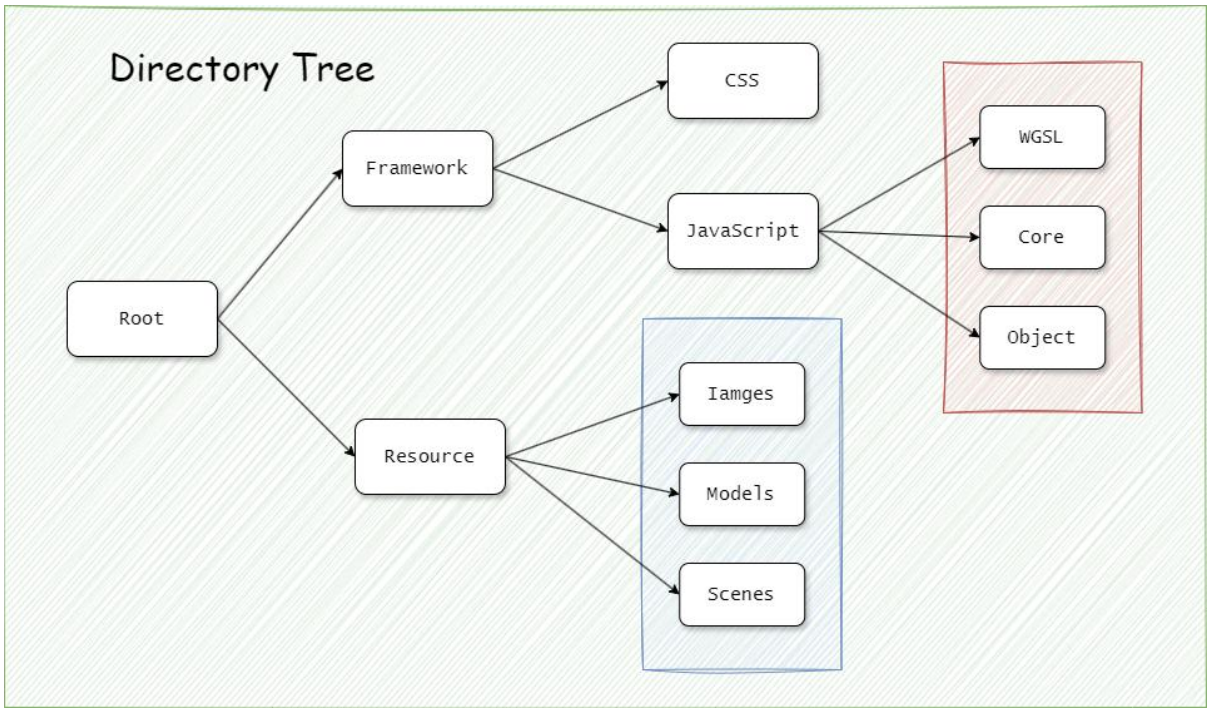
Typed Array는 JavaScript에서 배열 형태로 저장된 데이터에 대해 고정된 타입과 크기를 지정하여 효율적으로 다룰 수 있게 해주는 객체 유형이다. 숫자, 정수, 부동 소수점 등 특정 데이터 타입의 배열을 직접 다루며, 이진 데이터를 다루는 데 적합하다. 따라서 WebGPU의 버퍼로 사용되는 객체이다. Typed Array는 기본적으로 버퍼(ArrayBuffer)를 기반으로 하며, 다양한 타입(Int8Array, Uint8Array, Float32Array 등)으로 데이터에 접근할 수 있다. 그래픽 프로그래밍, 파일 I/O, 네트워크 데이터 처리 등에서 성능 향상과 메모리 관리가 기존 JavaScript의 Array보다 우수하다.

3. 추진 내용

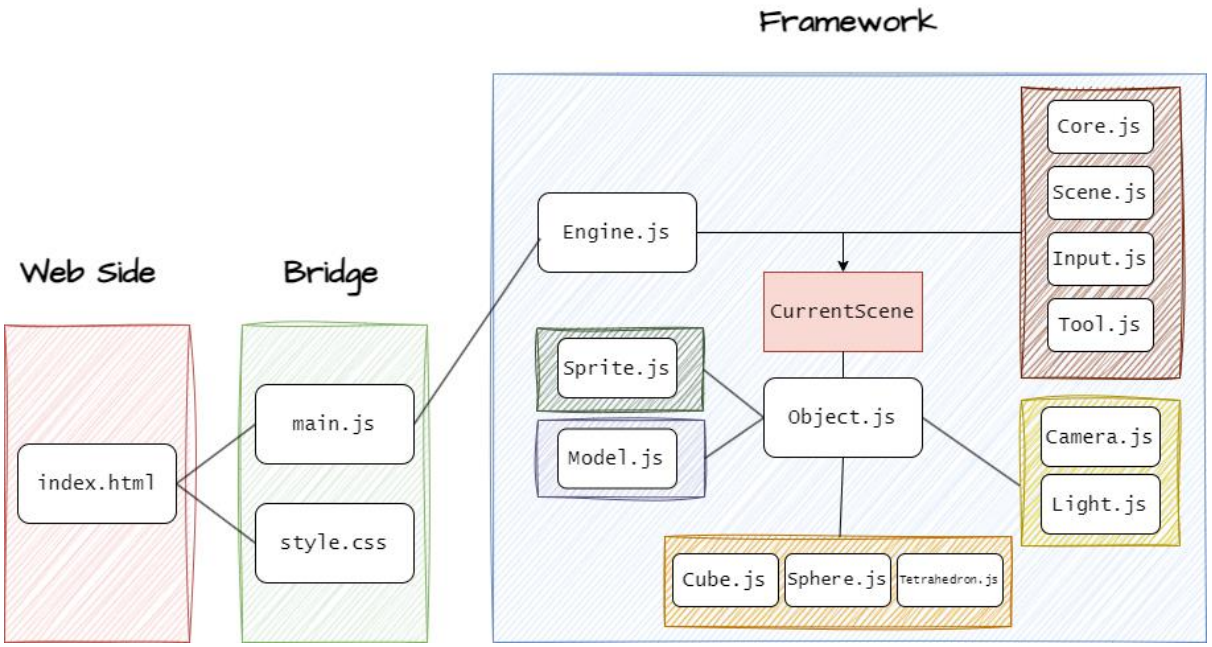
3.1. 팀 구성 및 역할

이름	역할	수행업무
DaLae37	팀장	프레임워크 개발 및 성능 비교

3.2. 전체 시스템 구성



프레임워크는 다음과 같은 Directory Tree구조로 구성된다. Root 폴더를 기준으로 Framework와 Resource 폴더로 나뉜다. Framework 폴더는 WebGPU 기반 렌더링 코드가 작성되어 있는 곳으로 그래픽 렌더링을 위한 중심 파일이 들어 있는 곳이다. Resource 폴더는 로컬에 이미지, 모델 파일을 저장한 곳이다. Scenes 폴더에 사용할 Scene 파일을 생성하고 Framework 코드를 호출함으로써 프레임워크를 사용할 수 있다.



프레임워크는 다음과 같은 동작 구조를 가진다.

index.html	프레임워크의 시작점, 웹 페이지 파일인 index.html에서 다른 파일을 호출
main.js	프레임워크의 진입점, 이곳에서 사용할 Scene을 선언하고 Engine을 초기화하는 것으로 프레임워크가 동작

style.css	canvas 제외 DOM 요소의 디자인을 관리
Engine.js	프레임워크의 동작을 통합한 파일, Core 객체 초기화 및 선언, Main Loop, Scene Update 동작을 수행
Core.js	canvas 생성, device 초기화, shader 관리
Scene.js	Scene의 템플릿 클래스와 Scene을 관리하는 SceneManager 관리
Input.js	사용자의 마우스, 키보드 입력을 매 프레임 확인
Tool.js	Matrix, Vector등의 연산과 Color, Transform 객체가 있는 파일
Camera.js	Camera의 기능을 정의한 파일, 모든 Scene은 하나의 Camera를 보유
Light.js	Lighting 기능을 정의한 파일, Object는 Light의 영향을 받음
Cube.js	정육면체 Object 파일, vertex, index 데이터를 미리 입력해둔 객체 Transform에 따라 위치가 지정되며 Color 값으로 Material 보유
Sphere.js	구 Object 파일, vertex, index 데이터를 미리 입력해둔 객체 Transform에 따라 위치가 지정되며 Color 값으로 Material 보유
Tetrahedron.js	정사면체 Object 파일, vertex, index 데이터를 미리 입력해둔 객체 Transform에 따라 위치가 지정되며 Color 값으로 Material 보유
Sprite.js	로컬 폴더에 있는 bitmap 파일을 렌더링하는 파일, 객체 생성 시 입력한 경로에 존재하는 파일을 가져오며 다른 3D Object 동일하게 World Space에 위치가 지정됨
Model.js	로컬 폴더에 있는 3D Model을 렌더링하는 파일, 현재 .obj 파일의 로딩만 구현되어 있으며 World Space에 위치가 지정됨
CURRENT_SCENE	현재 브라우저에서 보여지는 Scene, Engine.js에서 호출하여 매 프레임 업데이트

3.3. 사용자 시나리오

- 프레임워크 사용

3.2의 프레임워크 동작구조 참고

```html
<!DOCTYPE html>
<html lang="ko">

<head>
    <meta charset="UTF-8">
    <meta name="author" content="DaLae37">
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta property="og:type" content="">
    <meta property="og:url" content="">
    <meta property="og:title" content="">
    <meta property="og:description" content="">
    <meta property="og:image" content="">
    <meta property="og:image:width" content="">
    <meta property="og:image:height" content="">
    <meta property="og:locale" content="ko_KR">

    <title>DL-Engine-WebGPU</title>

    <link rel="stylesheet" type="text/css" href="Framework/CSS/style.css">

    <script defer type="module" src="Framework/JavaScript/main.js"></script>
</head>

<body>

</body>

</html>
```

프레임워크의 시작점인 index.html, 사용자가 웹 페이지를 띄우기 위해 서버에서 호스팅해야 하는 파일이다. 필요시 다른 라이브러리를 추가할 수 있지만 항상 main.js를 호출하는 스크립트 블록은 맨 아래에 존재해야 한다. 메타데이터의 변경, 웹 페이지 제목 변경을 할 수 있다.

```
//main.js
import { Engine } from "./Core/Engine.js";
import { MainScene as scene } from "../../Resource/Scenes/MainScene.js" //Input First Scene File and Class Names

const engine = new Engine();
await engine.Init();
engine.DoMainLoop(new scene(scene.name)); //Create First Scene
```

프레임워크의 진입점인 main.js, 이곳에서 프레임워크에 사용할 Scene을 불러오면 된다. 기본 Scene은 MainScene으로 선언되어 있으며 사용자는 시작 Scene을 변경할 때 Scene 클래스 명과 파일 이름만 변경하면 된다.

```
export class MainScene extends Scene {
    constructor(sceneName) {…
    }

    Update(deltaTime) {…
    }

    Render(deltaTime, pass) {…
    }
}
```

Scene의 구조는 생성자, Update, Render 함수로 이루어진다. 생성자에서는 사용자가 사용할 객체를 선언한다.

```
this.model1 = new OBJModel("Resource/Models/cat.obj", "Resource/Models/cat.png");
this.model1.SetPosition(new Vector3(-150 + x * 30, 30, -150 + y * 30));
this.AddObject(this.model1);
```

선언한 객체는 AddObject 함수를 통해 Scene의 objectList에 삽입할 수 있다. objectList에 삽입한 객체는 매 프레임마다 자동으로 Update와 Render가 이루어진다.

```
export class Engine {
    constructor() {
        this.FPS = 165;

        this.currentTime = 0;
        this.previousTime = 0;

        this.observer = null;
    }

    async Init() {
        canvas.InitCanvas();
        await device.InitDevice();
        device.InitContext();
        shaderModule.SetShader();

        inputManager.Init();

        this.InitObserver();
        this.InitDeltaTime();
    }
```

프레임워크의 동작을 통합한 Engine.js, 이곳에서 canvas 생성, device 초기화, context 초기화 shader 로딩이 이루어지며 싱글턴 패턴으로 존재하는 intputManager와 sceneManager가 초기화된다.

```
DoMainLoop(scene) {
    sceneManager.ChangeScene(scene, scene.sceneName);

    const mainLoop = setInterval(() => {
        inputManager.UpdateKeyState();
        if (sceneManager.currentScene.isLoaded) {
            sceneManager.UpdateScene(this.getDeltaTime());
            sceneManager.RenderScene(this.getDeltaTime());
        }
        let quitMessage = sceneManager.CheckQuitMessage();
        if (quitMessage) {
            clearInterval(mainLoop);
        }
    }, 1000 / this.FPS);
}
```

렌더링에 필요한 객체들이 생성되었을 때 실행되는 MainLoop다. Scene에서 quiteMessage를 호출하는 것으로 프로그램을 종료시킬 수 있다. 지정된 FPS에 맞게 함수가 실행되며 기본 FPS는 60으로 설정되어 있다. Scene의 변경이 있을 때, Scene의 생성자에서 선언하고 objectList에 삽입한 객체들이 초기화 되어야 Update와 Render 함수를 실행한다.

```
getDeltaTime() {
    this.currentTime = performance.now();
    let deltaTime = this.currentTime - this.previousTime;
    this.previousTime = this.currentTime;

    return deltaTime / 1000;
}
```

각 Update와 Render 함수 호출 사이를 Interpolation 하기 위한 DeltaTime을 계산하는 코드이다.

-프레임워크 확장

```
export class Object {
    constructor(name) {…
    }

    async LoadResource() {…
    }

    SetRenderTarget() {…
    }

    Update(deltaTime, cameraMatrix = Matrix4.identity()) {…
    }

    Render(deltaTime, pass = null) {…
    }
```

```
    GetPosition(){…
    }

    SetTransform(){…
    }

    SetPosition(position) {…
    }

    SetScale(scale) {…
    }

    SetRotation(rotation) {…
    }

    Translate(offset) {…
    }

    Scaling(offset) {…
    }

    Rotate(offset) {…
    }
```

해당 프레임워크는 확장이 가능하도록 객체지향 형식에 맞게 작성되었다. 렌더 타깃인 모든 대상은 Object 클래스를 상속받으며 LoadResource, SetRenderTarget, Update, Render 함수를 자식 클래스에서 구현해야 한다. JavaScript에는 순수가상함수가 존재하지 않아서 자식 클래스에서

구현하지 않아도 에러 없이 프레임워크가 동작한다.

```
class Model extends Object {···
}

export class OBJModel extends Model {
    constructor(modelSrc, textureSrc = "null", modelName = "null") {
        super(modelSrc, textureSrc, modelName);
    }

    async LoadResource() {
        await super.LoadResource();

        this.modelData = await objectManager.LoadModelToSrc(this.modelSrc);
        this.parseOBJ(this.modelData);

        super.SetRenderTarget();
        this.SetRenderTarget();
        this.isLoaded = true;
    }
```

3D Model 파일은 형식에 맞게 분리해 두었다. OBJ 파일은 OBJModel 클래스를 구현하여 사용하고 있다. Object 클래스를 상속받는 자식 클래스는 부모 생성자를 호출할 때 objectName 인자를 넘기면 된다.

```
async LoadResource() {
    0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, -1, 0,
  ]);

  this.vertexArray = new Float32Array(this.oneVertexLength * this.vertexLength);
  for (let i = 0; i < this.vertexLength; i++) {
    this.vertexArray.set(this.positionArray.subarray(i * this.positionArrayLength,
    this.vertexArray.set(this.colorArray.subarray(i * this.colorArrayLength, i * t
    this.vertexArray.set(this.uvArray.subarray(i * this.uvArrayLength, i * this.uv
    this.vertexArray.set(this.normalArray.subarray(i * this.normalArrayLength, i *
  }

  this.indexArray = new Uint16Array([
    //Bottom face
    0, 1, 2, 0, 2, 3,
    //Right face
    7, 8, 9, 7, 9, 10,
    //Top face
    12, 13, 14, 12, 14, 15,
    //Left face
    19, 20, 21, 19, 21, 22,
    //Front face
    24, 25, 26, 24, 27, 28,
    //Back face
    30, 31, 32, 32, 33, 34,
  ]);

  this.SetRenderTarget();
  this.isLoaded = true;
```

Object 클래스 구현을 위해 필수적으로 작성 해야 하는 LoadResource 함수이다. 이곳에서 3D 모델, 2D 이미지 같은 경우는 로컬에서 파일 로딩을 하며 육면체, 구체 같은 경우는 vertex data와 index data를 작성한다. 객체의 리스소에 대한 정의와 호출이 마무리 되었으면 동일 클래스의 SetRenderTarget 함수를 호출하여 WebGPU 파이프라인 작성을 진행한다.

```
SetRenderTarget() {
    depthStencil: {
      depthWriteEnabled: true,
      depthCompare: "less",
      format: "depth24plus",
    },
  });

  this.vertexBuffer = device.getDevice().createBuffer({
    size: this.vertexArray.byteLength,
    usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
  });
  device.getDevice().queue.writeBuffer(this.vertexBuffer, 0, this.vertexArray);

  this.indexBuffer = device.getDevice().createBuffer({
    size: this.indexArray.byteLength,
    usage: GPUBufferUsage.INDEX | GPUBufferUsage.COPY_DST,
  });
  device.getDevice().queue.writeBuffer(this.indexBuffer, 0, this.indexArray);
```

SetRenderTarget 함수는 WebGPU 파이프라인 작성을 진행하는 함수이다. Object 구현을 위해 반드시 필요한 함수이다. 렌더링에 필요한 파이프라인, 버퍼, 셰이딩 파일을 정의하며 그래픽 설정이 마무리 되었으면 객체의 isLoaded 변수가 true로 바뀌고 Scene에 렌더링할 준비가 완료된다.

```
Update(deltaTime, cameraMatrix) {
  super.Update(deltaTime);

  let worldMatrix = Matrix4.multiply(this.worldMatrix, cameraMatrix);
  let rotationMatrix = Matrix4.transpose(Matrix4.inverse(this.rotationMatrix));

  device.getDevice().queue.writeBuffer(this.uniformBuffer, 0, Matrix4.Mat4toFloat32Array(worldMatrix));
  device.getDevice().queue.writeBuffer(this.uniformBuffer, 4 * 4 * 4, Matrix4.Mat4toFloat32Array(rotationMatrix));
}
```

Update 함수는 객체의 위치를 계산하는 함수이다. Object 구현을 위해 반드시 필요한 함수이다. Camera의 Transform Matrix를 기준으로 Object의 World Matrix를 계산해준 뒤 그것을 이용해 position 정보를 기록한다. rotationMatrix는 Lighting 기능을 위한 행렬이다.

```
Render(deltaTime, pass) {
  super.Render(deltaTime);
  pass.setPipeline(this.pipeline);
  pass.setVertexBuffer(0, this.vertexBuffer);
  pass.setIndexBuffer(this.indexBuffer, "uint16");

  pass.setBindGroup(0, this.bindGroup);
  pass.drawIndexed(this.indexArray.length);
}
```

Render 함수는 객체를 렌더링하는 함수이다. Object 구현을 위해 반드시 필요한 함수이다. SetRenderTarget에서 선언한 파이프라인을 사용하여 데이터를 입력한다. Pass encoder는 Scene의 Render함수 시작 부분에서 선언한 encoder를 사용하여 모든 Object를 동시에 Render할 수 있도록 구현하였다.

```
class ObjectManager {
    constructor() {
        this.imageDictionary = {}; //Dictionary
        this.modelDictionary = {}; //Dictionary
    }

    async LoadImageToSrc(src) {···
    }

    async LoadModelToSrc(src){···
    }
}

export const objectManager = new ObjectManager();
```

이미지, 모델 파일 등 로컬에서 파일을 불러올 때에는 Object.js의 ObjectManger 객체를 이용해 LoadImageToSrc, LoadModelToSrc 함수를 호출함으로써 파일을 불러올 수 있다. Textured Model, Mipmap 등 추가 기능을 구현할 때 해당 함수를 사용할 수 있다.

-쉐이더 코드 추가

3.2의 프레임워크 Directory Tree 참고

WebGPU의 그래픽 렌더링을 위한 쉐이더 코드는 JavaScript/Shader 폴더에 .js파일로 정리되어 있다.

```
export const modelShader = `
struct ModelUniform {
    worldMatrix : mat4x4<f32>,
    rotationMatrix : mat4x4<f32>,
};

  struct DirectionalLightUniforms {
    lightDirection: vec4<f32>,
  };

  @group(0) @binding(0) var textureSampler: sampler;
  @group(0) @binding(1) var texture: texture_2d<f32>;
  @group(0) @binding(2) var<uniform> modelUniform : ModelUniform;
  @group(0) @binding(3) var<uniform> directionalLight : DirectionalLightUniforms;

  struct VSInput {
    @location(0) position : vec4<f32>,
    @location(1) uv : vec2<f32>,
    @location(2) normal : vec4<f32>,
  };
```

JavaScript 변수로 생성한 뒤 export 예약어를 통해 모듈화 한다.

```
import { spriteShader } from "../Shader/sprite.js"
import { modelShader } from "../Shader/model.js";
import { cubeShader } from "../Shader/cube.js";
import { sphereShader } from "../Shader/sphere.js";
import { tetrahedronShader } from "../Shader/tetrahedron.js";

class ShaderModule {
    constructor() {
        this.shaderDictionary = {} //Dictionary
        this.moduleDictionary = {} //Dictionary
    }

    SetShader() {
        if (device.getDevice()) { //Add Using Shaders
            this.shaderDictionary["sprite"] = spriteShader;
            this.shaderDictionary["model"] = modelShader;
            this.shaderDictionary["cube"] = cubeShader;
            this.shaderDictionary["sphere"] = sphereShader;
            this.shaderDictionary["tetrahedron"] = tetrahedronShader;
            //this.shaderDictionary["label"] = shader;
        }
        else {
            console.log("device not initialized");
        }
    }
```

그 후 Core.js의 ShaderModule 클래스 위에서 해당 파일과 변수를 import 한 뒤 SetShader 함수에

해당 쉐이더 변수 이름을 추가하면 된다.

```
SetRenderTarget() {
    this.pipeline = device.getDevice().createRenderPipeline({
        layout: "auto",
        vertex: {
            module: shaderModule.UseModule("model"),
            buffers: [
```

그 후 Shader Module이 필요한 곳에 다음과 같이 사용하면 된다.

-Canvas 사이즈 조정

Canvas의 사이즈는 브라우저의 크기에 따라 변하도록 Observer로 관리되고 있다.

```
InitObserver() {
    this.observer = new ResizeObserver(entries => {
        for (const entry of entries) {
            const canvas = entry.target;
            const width = entry.contentBoxSize[0].inlineSize;
            const height = entry.contentBoxSize[0].blockSize;
            canvas.width = Math.max(1, Math.min(width, device.getDevice().limits.maxTextureDimension2D));
            canvas.height = Math.max(1, Math.min(height, device.getDevice().limits.maxTextureDimension2D));
        }
    });
    this.observer.observe(canvas.getCanvas());
}
```

해당 프레임워크를 이용한 웹 페이지의 사이즈를 줄여도 화면비가 유지되는 것을 볼 수 있다. 하지만 최초 Canvas 크기와 화면비율을 변경할 수 있다.

```
class Canvas {
    constructor() {
        this.WIDTH = 1920;
        this.HEIGHT = 1080;

        this.canvas = null;
    }
```

Canvas 클래스의 생성자에서 WIDTH와 HEIGHT 변수 값을 변경하면 최초 생성되는 Canvas의 크기와 화면 비율이 변경된다.

-FPS 조정

```
export class Engine {
    constructor() {
        this.FPS = 165;

        this.currentTime = 0;
        this.previousTime = 0;

        this.observer = null;
    }
```

프레임워크의 최대 FPS는 Engine 클래스의 FPS 변수를 변경하는 것으로 변경할 수 있다.

-Scene 생성 및 구현

```
export class DemoScene3 extends Scene {
    constructor(sceneName) {
        for (let i = 0; i < treeNum; i++) {
            x = Math.floor(Math.random() * 9);
            y = Math.floor(Math.random() * 9);
            if (positions[y][x] == -1) {
                i -= 1;
                continue;
            }
            else {
                let tree = new OBJModel("Resource/Models/tree.obj", "Resource/Models/tree.png");
                tree.SetPosition(new Vector3(-150 + x * 30, 30, -150 + y * 30));
                this.AddObject(tree);
            }
        }
```

Scene의 생성은 Scene 클래스를 상속받는 임의의 이름을 가진 클래스를 선언하면 된다. 그 후 Scene의 생성자에서 Scene에 사용할 Object를 선언하면 된다. 이때 항상 Update와 Render시킬 객체는 AddObject 함수를 통해 Scene의 objectList에 삽입하면 되고 경우에 따라 사용할 Object는 할당만 시킨 뒤 조건에 맞게 사용하면 된다.

```
Update(deltaTime) {
    super.Update(deltaTime);
    this.stats.update();

    if (inputManager.GetKeyDown("KeyA")) {
        this.camera.Translate(new Vector3(-50 * deltaTime));
    }
    if (inputManager.GetKeyDown("KeyD")) {
        this.camera.Translate(new Vector3(50 * deltaTime));
    }
```

Scene의 동작은 Update 함수에서 이루어진다. 카메라와 Object 이동 등 Transform 기능을 수행하는 함수이다. 프레임워크에서 Render 함수보다 먼저 호출되기 때문에 렌더링 후 이동 기능 구현을 하려면 프레임워크 구조의 변경이 필요하다.

```
Render(deltaTime, pass) {
    super.Render(deltaTime, pass);

    if(inputManager.GetKeyUp("ArrowUp")){
        this.specialIcon.Render(deltaTime, pass);
    }
}
```

Scene의 그래픽 요소를 그리는 것은 Render 함수에서 이루어진다. 특정 조건에 Object를 렌더링하려면 이곳에서 Object의 Render 함수를 호출하는 것으로 구현할 수 있다.

```
this.renderPassDescriptor.depthStencilAttachment.view = this.depthTexture.createView();

let encoder = device.getDevice().createCommandEncoder(); //command buffer start
let pass = encoder.beginRenderPass(this.renderPassDescriptor);

this.currentScene.Render(deltaTime, pass);

pass.end(); //command buffer end
let commandBuffer = encoder.finish();
device.getDevice().queue.submit([commandBuffer]);
```

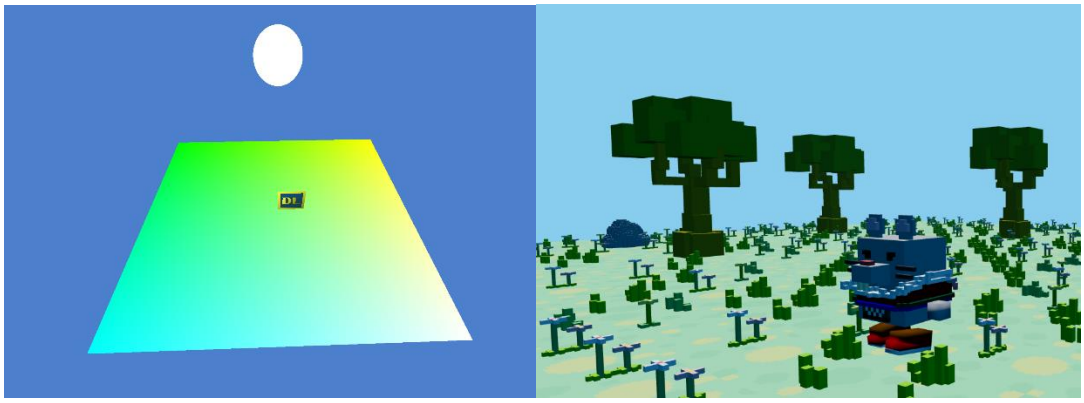
Scene의 command buffer는 SceneManger의 RenderScene 함수에서 선언되고 한 Scene의 모든

Object의 buffer를 불러와 GPU에 명령을 내린다.

3.4. 핵심 기능 개발

① 2D Image, 3D Model Rendering

2D Image 3D Model을 로컬로부터 불러오고 World Space에 렌더링 하는 기능을 구현했다.



3D Model은 .obj 파일만 불러올 수 있으며 Image는 확장자의 제한이 없다.

② Transform 기능 개발

```
export class Transform {
    constructor() {
        this.position = new Vector3(0, 0, 0);
        this.scale = new Vector3(1, 1, 1);
        this.rotation = new Vector3(0, 0, 0);
    }

    static degreeToRadian(degree) {
        let radian = degree * (Math.PI / 180);
        return radian;
    }

    static translate(matrix, offset) { //offset : Vector3
        return Matrix4.multiply(matrix, Matrix4.translation(offset));
    }

    static scale(matrix, scale) { //scale : Vector3
        return Matrix4.multiply(matrix, Matrix4.scaling(scale));
    }

    static rotate(matrix, radians) { //radians : Vector3
        let x = Matrix4.multiply(matrix, Matrix4.rotationX(radians.x));
        let y = Matrix4.multiply(x, Matrix4.rotationY(radians.y));
        let z = Matrix4.multiply(y, Matrix4.rotationZ(radians.z));

        return z;
    }
}
```

```
    static translation(offset) {···
    }

    static scaling(scale) {···
    }

    static rotationX(radian) {···
    }

    static rotationY(radian) {···
    }

    static rotationZ(radian) {···
    }
}
```

Object의 Position, Rotation, Scale 기능을 사용하기 위해 Object Matrix, Camera Matrix, World Matrix를 계산하는 기능을 구현하였다. position, rotation, scale은 Vector3 기반으로 동작하며 인자로 Vector3를 사용하면 된다.

③ Camera 기능 개발

```
look(eye, target, up) {···
}

projection(width, height, depth) {···
}

perspective(radianFOV, aspect, zNear, zFar) {···
}

orthographic(right, left, top, bottom, far, near) {···
}
```

```
Update(deltaTime) {
    this.rotationMatrix = Transform.rotate(this.rotationMatrix, new Vector3(this.pitch, this.yaw, this.roll));
    this.projectionMatrix = this.perspective(this.fieldOfView, canvas.getCanvas().clientWidth / canvas.getCanvas().clientHeight,
     this.near, this.far);

    this.matrix = this.look(this.eye, this.at, this.up);
    this.viewMatrix = Matrix4.inverse(this.matrix);

    this.cameraMatrix = Matrix4.multiply(this.viewMatrix, this.projectionMatrix);
}
```

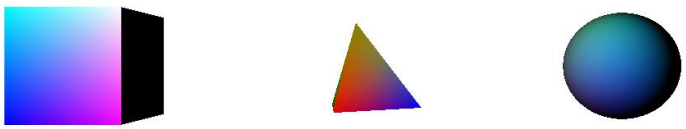
Camera 기능을 구현하기 위해 look, projection perspective, orthographic 함수를 통해 각각의 Matrix를 계산하는 기능을 추가하였다. 함수의 결과로 나온 Matrix를 이용해 최종 Camera Matrix를 계산하며 Object Matrix와 곱을 통해 World Space를 계산하였다.

④ Light 기능 개발

Light가 있을 때

Light가 없을 때

3D Model의 현실성을 위해 Lighting 기능을 추가하였다. Lighting은 현실감과 분위기를 창출하는 데 필수적인 요소이다. 3D Object에 깊이와 입체감을 부여하고 시각적 몰입감을 느낄 수 있다.



정육면체, 정사면체, 구 또한 Normal을 지정하여 Lighting 기능을 확인할 수 있게 구현하였다.

⑤ Resource Management (Flyweight Pattern)

로컬에서 불러오는 리소스를 관리하기 위해 디자인 패턴 중 Flyweight Pattern을 사용하였다.

```
async LoadImageToSrc(src) {
    if (src in this.imageDictionary) {
        return this.imageDictionary[src];
    }
    else {
        let response = await fetch(src);

        if (response.ok) {
            let image = await createImageBitmap(await response.blob());
            this.imageDictionary[src] = image;
            return image;
        }
    }
}
```

```
SetModule(label) {
    if (!(label in this.moduleDictionary) && label in this.shaderDictionary) {
        this.moduleDictionary[label] = device.getDevice().createShaderModule({
            label: label,
            code: this.shaderDictionary[label],
        });
    }
    else {
        console.log("wrong shader");
    }
}
```

이미지, 모델 파일을 불러오는 것과 셰이딩 파일을 모듈화 시키기 위한 부분에 사용이 되었다. 각 경로에 객체의 데이터가 있으면 해당 데이터를 재사용하고 경로가 Dictionary에 없으면 Dictionary에 해당 경로를 추가한다.

3.5. 이슈 및 대응

JavaScript의 Vector, Matrix 관련된 라이브러리가 없는 것은 그래픽 렌더링에 걸림돌이 되었다. 대표적으로 Direct3D 12 API 같은 경우는 XMVECTOR, XMMATRIX와 같은 타입의 Vector와 Matrix가 존재해 연산이 쉬운 반면 JavaScript는 해당 기능과 유사한 기능이 존재하지 않았다.

```
export class Vector3 {
    constructor(x = 0, y = 0, z = 0) { ···
    }

    static inner(a, b) { ···
    }

    static cross(a, b) { ···
    }

    static add(a, b) { ···
    }

    static subtract(a, b) { ···
    }

    static normalize(vector3) { ···
    }

    static Vec4toVec3(a) { ···
    }

    static ArraytoVec3(a) { ···
    }
}
```

```
export class Matrix4 {
    constructor() {
        this.matrix = Array.from(Array(4), () => new Float32Array(4).fill(0));
    }

    static identity() { ···
    }

    static Mat3toMat4(a) { ···
    }

    static Mat4toFloat32Array(a) { ···
    }

    static transpose(a) { ···
    }

    static multiply(a, b) { ···
    }

    static inverse(matrix) { ···
    }
```

해당 이슈를 해결하기 위해 Vector와 Matrix 관련 클래스를 생성하였고 그래픽 렌더링에 필요한 기능을 추가하였다.

```
static transpose(a) {
    let matrix = Array.from(Array(4), () => new Float32Array(4).fill(0));
    matrix[0][0] = a[0][0];
    matrix[0][1] = a[1][0];
    matrix[0][2] = a[2][0];
    matrix[0][3] = a[3][0];

    matrix[1][0] = a[0][1];
    matrix[1][1] = a[1][1];
    matrix[1][2] = a[2][1];
    matrix[1][3] = a[3][1];

    matrix[2][0] = a[0][2];
    matrix[2][1] = a[1][2];
    matrix[2][2] = a[2][2];
    matrix[2][3] = a[3][2];

    matrix[3][0] = a[0][3];
    matrix[3][1] = a[1][3];
    matrix[3][2] = a[2][3];
    matrix[3][3] = a[3][3];
```

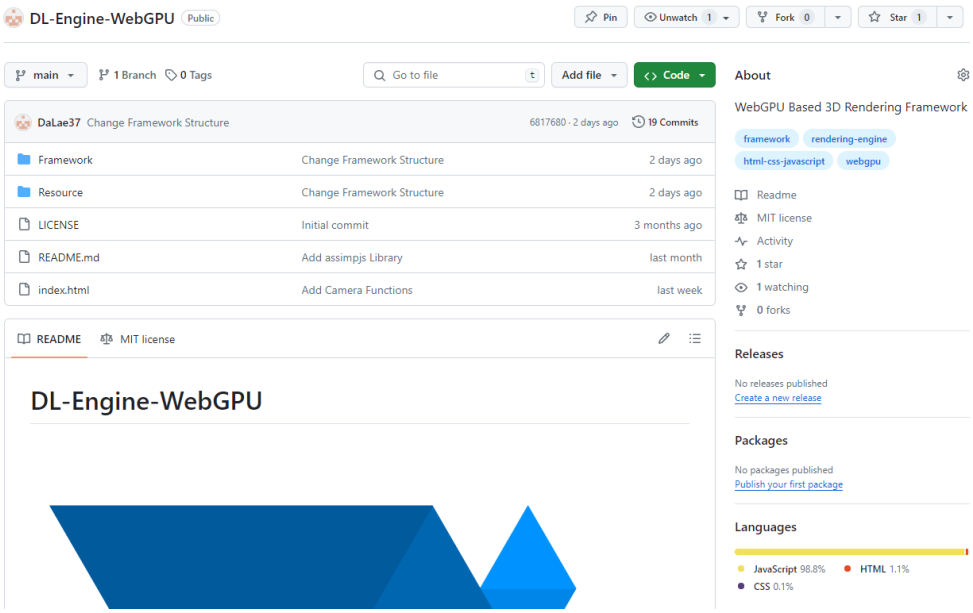
```
static inverse(matrix) {
    let inverseMatrix = Array.from(Array(4), () => new Float32Array(4).fill(0));

    inverseMatrix[0][0] = matrix[1][1] * matrix[2][2] * matrix[3][3] -
        matrix[1][1] * matrix[2][3] * matrix[3][2] -
        matrix[2][1] * matrix[1][2] * matrix[3][3] +
        matrix[2][1] * matrix[1][3] * matrix[3][2] +
        matrix[3][1] * matrix[1][2] * matrix[2][3] -
        matrix[3][1] * matrix[1][3] * matrix[2][2];
```

하지만 연산 자체는 최적화된 알고리즘을 사용하지 않고 하드코딩 방식으로 구현하였다. 따라서 다른 그래픽스 API 대비 연산 속도 면에서 손실이 있을 것이다. 더 큰 Performance를 얻기 위해서라면 추후 Vector, Matrix를 통한 연산 기능 추가가 필요해보인다.

4. 결과

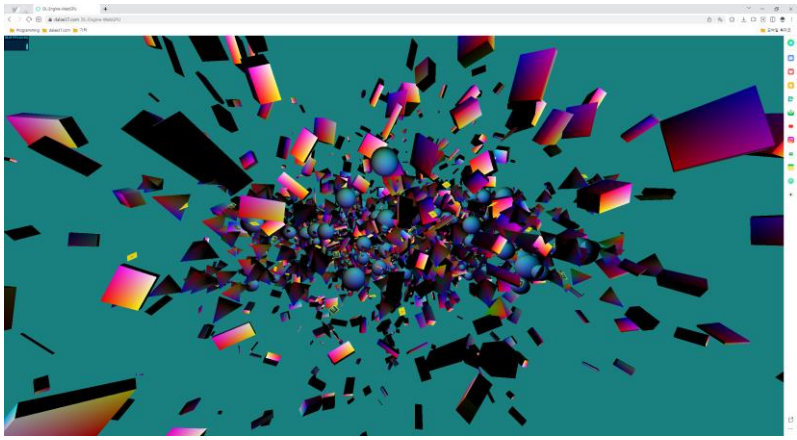
4.1. WebGPU Base Framework

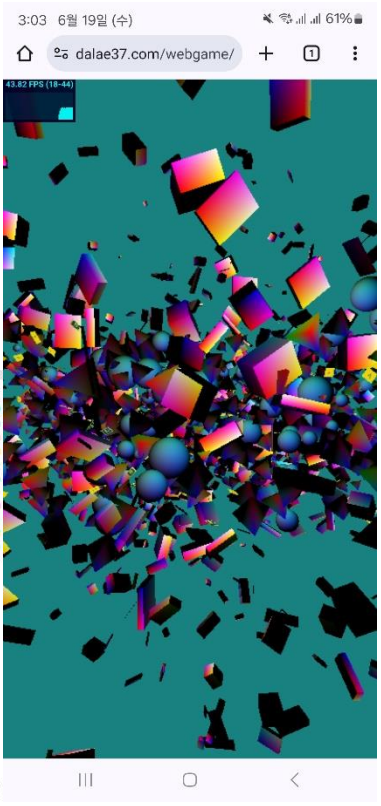


https://github.com/DaLae37/DL-Engine-WebGPU

연구를 통해 개발한 프레임워크를 Github의 Public Repository로 공개했다. MIT License로 배포되고 있으며 자유로운 사용이 가능하다.

4.2. Multi Platform





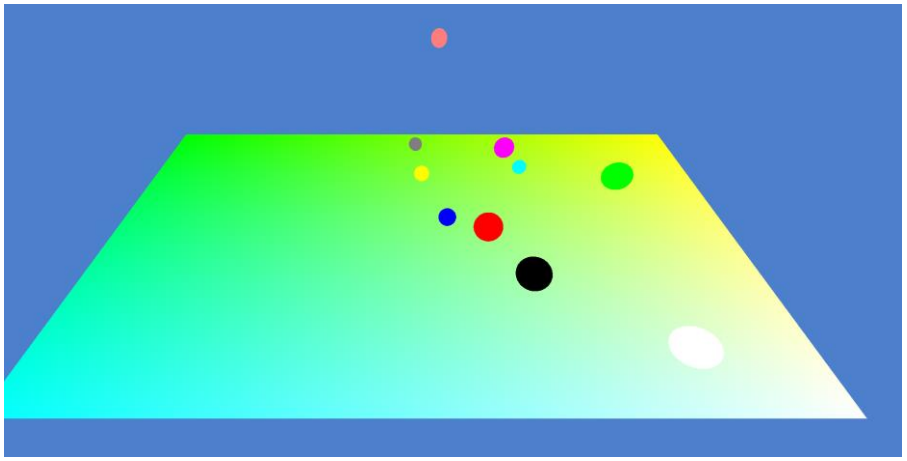
Desktop 환경 & Whale Browser (좌) Mobile 환경 & Chrome Browser (우)

WebGPU를 사용한 코드는 Desktop 환경과 Mobile 환경 모두 지원하는 것을 확인할 수 있었다. 또한 Desktop환경(RTX 3090) 대비 Mobile 환경에서는 낮은 렌더링 성능을 확인할 수 있었다.

4.3. WebGPU Demo Scene

프레임워크를 이용해 사용 방법 예제와 성능을 보여줄 수 있는 Demo Scene을 개발했다.

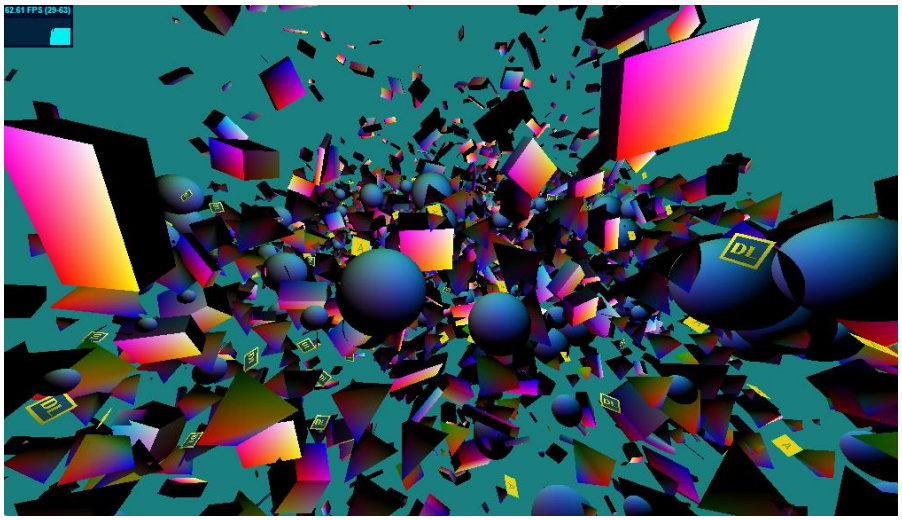
모든 DemoScene은 WASD, 키보드 방향 버튼을 통해 카메라를 이동시킬 수 있다.



DemoScene1. Bouce Ball

https://www.dalae37.com/webgame/bounce_ball_webgpu/index.html

해당 DemoScene에서는 Sphere에 중력 가속도를 부여한 모습을 볼 수 있다. 지정된 Force에 맞게 앞으로 밀리면서 Sphere가 Cube에 Bound 되는 모습을 볼 수 있다.



DemoScene2. Performance Test

https://www.dalae37.com/webgame/performance_test_webgpu/index.html

해당 DemoScene에서는 GPU의 렌더링 성능을 테스트 할 수 있다. 기본적으로 1000개의 육면체와 사면체, 100개의 구체, 200개의 이미지를 렌더링하며 GPU의 성능에 비례해 다른 FPS를 확인할 수 있다.

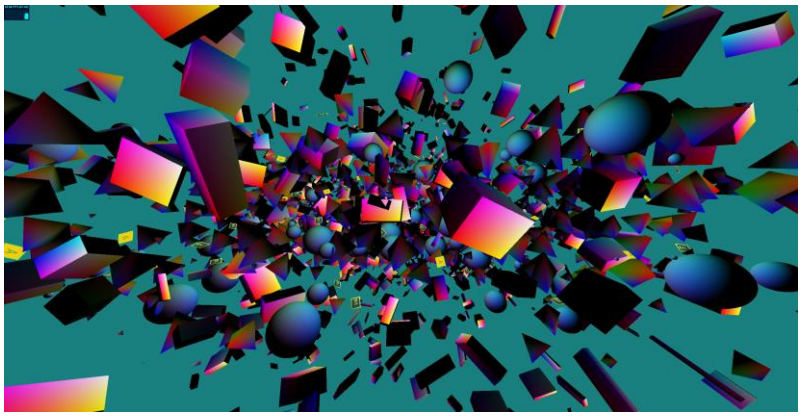
DemoScene3. 3D Models

https://www.dalae37.com/webgame/3d_models_webgpu/index.html

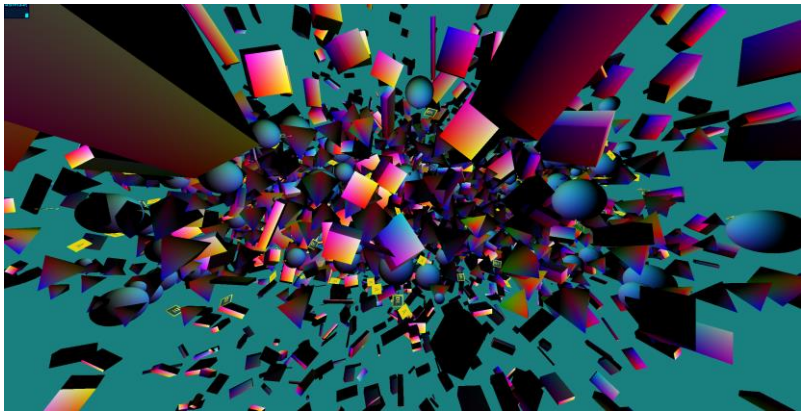
해당 DemoScene은 .obj파일과 texture 파일을 로딩 해 웹페이지에서 렌더링한 예제다. 정면에서 비추는 Directional Lighting을 적용시켜 음영 효과를 확인할 수 있다.

4.4. WebGPU Performance Test

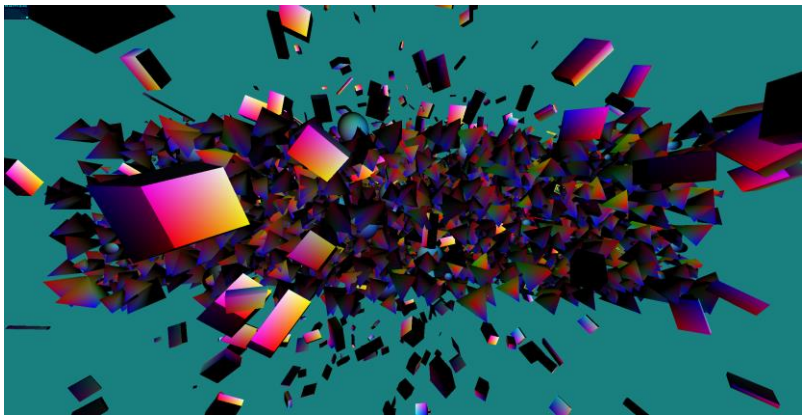
DemoScene2. Performance Test를 이용해 WebGPU의 렌더링 성능을 테스트했다.



Test1 육면체 1000개, 사면체 1000개, 구체 100개, 이미지 100개



Test2 육면체 2000개, 사면체 2000개, 구체 200개, 이미지 200개



Test2 육면체 10000개, 사면체 1000개, 구체 100개, 이미지 100개

FPS 측정은 Stats(https://github.com/mrdoob/stats.js)로 진행했으며 결과는 다음과 같다

	TEST1	TEST2	TEST3
FPS	104	42	21
Tetrahedron	1000	2000	1000

Cube	1000	2000	10000
Sphere	100	200	100
Sprite	100	200	100

5. 결론

WebGPU를 이용한 렌더링 프레임워크를 개발할 수 있었다. 개발된 결과물을 Github로 공유할 수 있었으며 Demo Scene을 만들어 웹 사이트로 호스팅할 수 있었다. 또한 호스팅 된 결과물이 Desktop과 Mobile 환경 모두에서 동작하는 것을 확인할 수 있었다. WebGPU를 사용한다면 많은 양의 Object를 렌더링할 수 있으며 게임 개발에서 사용 가능한 수준의 결과를 얻을 수 있었다. 해당 프레임워크를 이용해 API간의 렌더링 성능 비교, 플랫폼간 성능 비교가 가능할 것이며 이를 이용해 발전된 성능을 이끌어낼 수 있을 것이다.

다만 많은 한계점도 존재하였다. Javascript에 Matrix, Vector 연산 관련된 라이브러리가 존재하지 않았다. 이번 프레임워크 개발에서는 직접 Matrix와 Vector의 클래스와 함수를 정의해서 진행했지만 더 좋은 성능을 얻기 위해서는 추후 해당 부분의 연구가 필요해 보인다. 또한 JavaScript의 Array 탐색에 큰 Over Head가 필요했다. Array 말고 다른 자료구조를 사용할 필요성을 느꼈다.

참고문헌

[1] Benjamin Kenwright. 2023. Web Programming Using the WebGPU API. In ACM SIGGRAPH 2023 Courses (SIGGRAPH '23). Association for Computing Machinery, New York, NY, USA, Article 21, 1–184. https://doi.org/10.1145/3587423.3595543

[2] Benjamin Kenwright. 2023. Introduction to Computer Graphics and Ray-Tracing Using the WebGPU API. In SIGGRAPH Asia 2022 Courses (SA '22). Association for Computing Machinery, New York, NY, USA, Article 1, 1–139. https://doi.org/10.1145/3550495.3558218