midterm_util.js

```
const canvas = document.getElementById('glCanvas');
const gl = canvas.getContext('webgl2');

// util.js
// resizeAspectRatio
function resizeAspectRatio(gl, canvas) {
    window.addEventListener('resize', () => {
        const originalWidth = canvas.width;
        const originalHeight = canvas.height;
        const aspectRatio = originalWidth / originalHeight;

        let newWidth = window.innerWidth;
        let newHeight = window.innerHeight;

        if (newWidth / newHeight > aspectRatio) {
            newWidth = newHeight * aspectRatio;
        }

        else {
            newHeight = newWidth * aspectRatio;
        }

        canvas.width = newWidth;
        canvas.height = newHeight;

        // gl.viewport(Lower-left-x, Lower-left-y, width, height)
        gl.viewport(0, 0, canvas.width, canvas.height);
    })
}

// shader.js
export function compileShader(gl, source, type) {
    const shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);
    if(!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
        console.error('Error compiling shader:', gl.getShaderInfoLog(shader));
        gl.deleteShader(shader);
        return null;
    }
    return shader;
}

export function createProgram(gl, vertexShaderSource, fragmentShaderSource) {
    const vertexShader = compileShader(gl, vertexShaderSource, gl.VERTEX_SHADER);
    const fragmentShader = compileShader(gl, fragmentShaderSource, gl.FRAGMENT_SHADER);

    const program = gl.createProgram();
    gl.attatchShader(program, vertexShader);
    gl.attatchShader(program, fragmentShader);

    gl.linkProgram(program);

    if (!gl.getProgramParameter(program, gl.LINK_STATUS)) {
        console.error('Error linking program:', gl.getProgramInfoLog(program));
        gl.deleteProgram(program);
        return null;
    }
    return program;
}

export class Shader {
    constructor(gl, vertexSource, fragmentSource) {
        this.gl = gl;
        this.program = createProgram(gl, vertexSource, fragmentSource);
        if(!this.program) {
            throw new Error('Failed to initialize shader program');
        }
    }

    initShader(vertexSource, fragmentSource) {
        // vertex shader compile
        const vertexShader = this.gl.createShader(this.gl.VERTEX_SHADER);
        this.gl.shaderSource(vertexShader, vertexSource);
        this.gl.compileShader(vertexShader); // 위에 만든 export function과 다름름

        // 컴파일 결과 확인
        if (!this.gl.getShaderParameter(vertexShader, this.gl.COMPILE_STATUS)) {
            console.error('Error compiling vertex shader:',
this.gl.getShaderInfoLog(vertexShader));
            this.gl.deleteShader(vertexShader);
            return null;
        }

        // fragment shader compile
        const fragmentShader = this.gl.createShader(this.gl.FRAGMENT_SHADER);
        this.gl.shaderSource(fragmentShader, fragmentSource);
        this.gl.compileShader(fragmentShader);

        // 컴파일 결과 확인
        if (!this.gl.getShaderParameter(fragmentShader, this.gl.COMPILE_STATUS)) {
            console.error('Error compiling fragment shader:',
this.gl.getShaderInfoLog(fragmentShader));
            this.gl.deleteShader(fragmentShader);
            return null;
        }

        // program create & link
        const program = this.gl.createProgram(); // webGL의 createProgram(); 위에 만든 export
function과 다름름
        this.gl.attatchShader(program, vertexShader);
        this.gl.attatchShader(program, fragmentShader);

        this.gl.linkProgram(program);
    }

    //= useProgram(program)
    use() {
        if (!this.program) return;
        this.gl.useProgram(this.program);
```

```
    }

    // attrib setter
    setAttribPointer(name, size, type, normalized, stride, offset) {
        if(!this.program) return;
        const location = this.gl.getAttribLocation(this.program, name);
        if (location === -1) {
            console.warn('Attribute ${name} not found in shader program');
            return;
        }
        this.gl.enableVertexAttribArray(location);
        this.gl.vertexAttribPointer(location, size, type, normalized, stride, offset);
    }

    // uniform setter
    // 'fv': input [x, y, ...] 형태, 'f': input x, y, ... 형태태
    // setBool: gl.uniform1i, setInt: gl.uniform1i, setFloat: gl.uniform1f, setVec2: gl.uniform2fv & unifrom2f,
    // setVec3: gl.uniform3fv & unifrom3f, setVec4: gl.uniform4fv & unifrom4f
    // setMat2~setMat4: gl.uniformMatrix[2~4]fv
}

/* shader.js 실제 사용 pipeline */
let shader;
// ...
function setupBuffers() {
    const cubeVertices = new Float32Array([
        -0.15,  0.15,  // 좌상단
        -0.15, -0.15,  // 좌하단
         0.15, -0.15,  // 우하단
         0.15,  0.15   // 우상단
    ]);

    const indices = new Uint16Array([
        0, 1, 2,    // 첫 번째 삼각형
        0, 2, 3     // 두 번째 삼각형
    ]);

    const cubeColors = new Float32Array([
        1.0, 0.0, 0.0, 1.0,  // 빨간색
        1.0, 0.0, 0.0, 1.0,
        1.0, 0.0, 0.0, 1.0,
        1.0, 0.0, 0.0, 1.0
    ]);

    // VAO
    vao = gl.createVertexArray();
    gl.bindVertexArray(vao);

    // VBO
    const positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, cubeVertices, gl.STATIC_DRAW);
    shader.setAttribPointer("a_position", 2, gl.FLOAT, false, 0, 0); // <- 여기

    const colorBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, cubeColors, gl.STATIC_DRAW);
    shader. setAttribPointer("a_color", 4, gl.FLOAT, false, 0, 0); // <- 여기

    // EBO
    const indexBuffer = gl.createBuffer();
    gl.bindBUffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);

    gl.bindVertexArray(null); // initialize again
}

function render() {
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Here
    shader.use();
    shader.setMat4("u_transform", finalTransform); // vertex shader의 u_transform 유니폼 value에 finalTransform 행렬 삽입
    gl.bindVertexArray(vao);

    gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
}

async function initShader() {
    const vertexShaderSource = await readShaderFile('shVert.glsl');
    const fragmentShaderSource = await readShaderFile('shFrag.glsl');

    shader = new Shader(gl, vertexShaderSource, fragmentShaderSource); // <- 요기
}

async function main() {
    try {
        if(!initWebGL()) {
            throw new Error('WebGL 초기화 실패');
        }

        finalTransform = mat4.create();

        await initShader(); // <- 요기

        setupBuffers(); // 위에서 정의한 함수
        //..

        return true;
    } catch (error) {
        console.error('어쩌구', error);
        alert('저쩌구') // pop-up
        return false;
    }

}
```