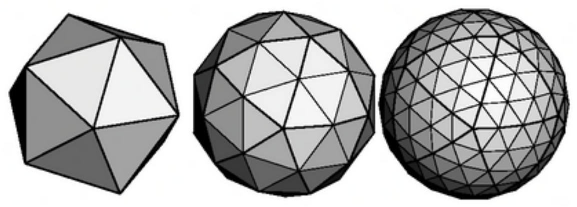
Javascript 3D engine

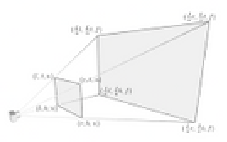
This is a simple rasterised JavaScript 3D engine that is entirely CPU based (because of a lack of dedicated GPU support in JavaScript). Due to the nature of the project it can be quite complicated for someone with no previous knowledge of a 3D engine so I will try my best to use this first slide to explain how the project works.

The first thing that you will need to understand is that all 3D graphics on a computer are made of triangles. This is because this is the simplest 2D shape and a triangle can make any other shape. This is explained below. To draw triangles to the screen in JavaScript I will be using an HTML 5 canvas and some simple methods that I made to make it easy to draw to the screen.



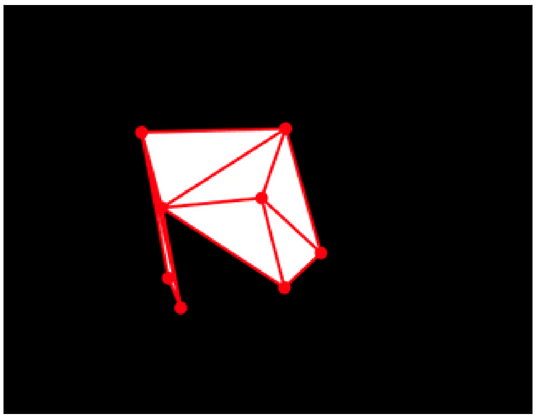
We will need a way to draw these 3D shapes onto a 2D screen. This is done using a method of multiplying matrices. We will use many different matrices, the main one being a projection matrix which allows us to perform the calculation from 3D shapes to 2D shapes. Below you can see the projection matrix and how it is used.

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



It is important to say, that when it comes to these matrices I treat then as "black boxes". I don't understand them because it is uni level maths, but I do know how to use them and that is what is important. There are also lots of other types of matrices used for different things - there are ones for rotation, scaling and many others.

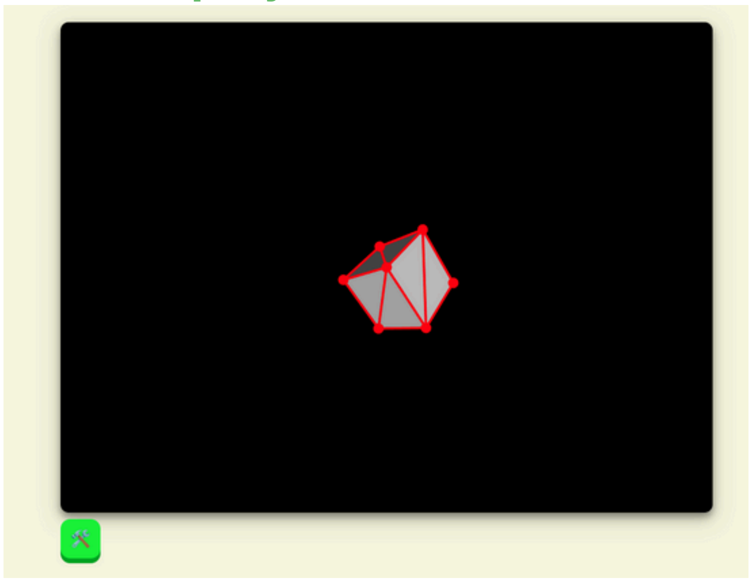
The next thing that we need to think about is how we know when to draw each triangle. We could draw them all but then you would be able to see the back of a cube at all times and that is not what we want. To fix this we will need to find the normal of each face and then see if it is 90 degrees or less from the camera. If it is then we can draw the face. Below I have attached an image of what happens if we only see above 90 degrees and you can tell that we can see the other side of the cube.



[You can view the source here]()

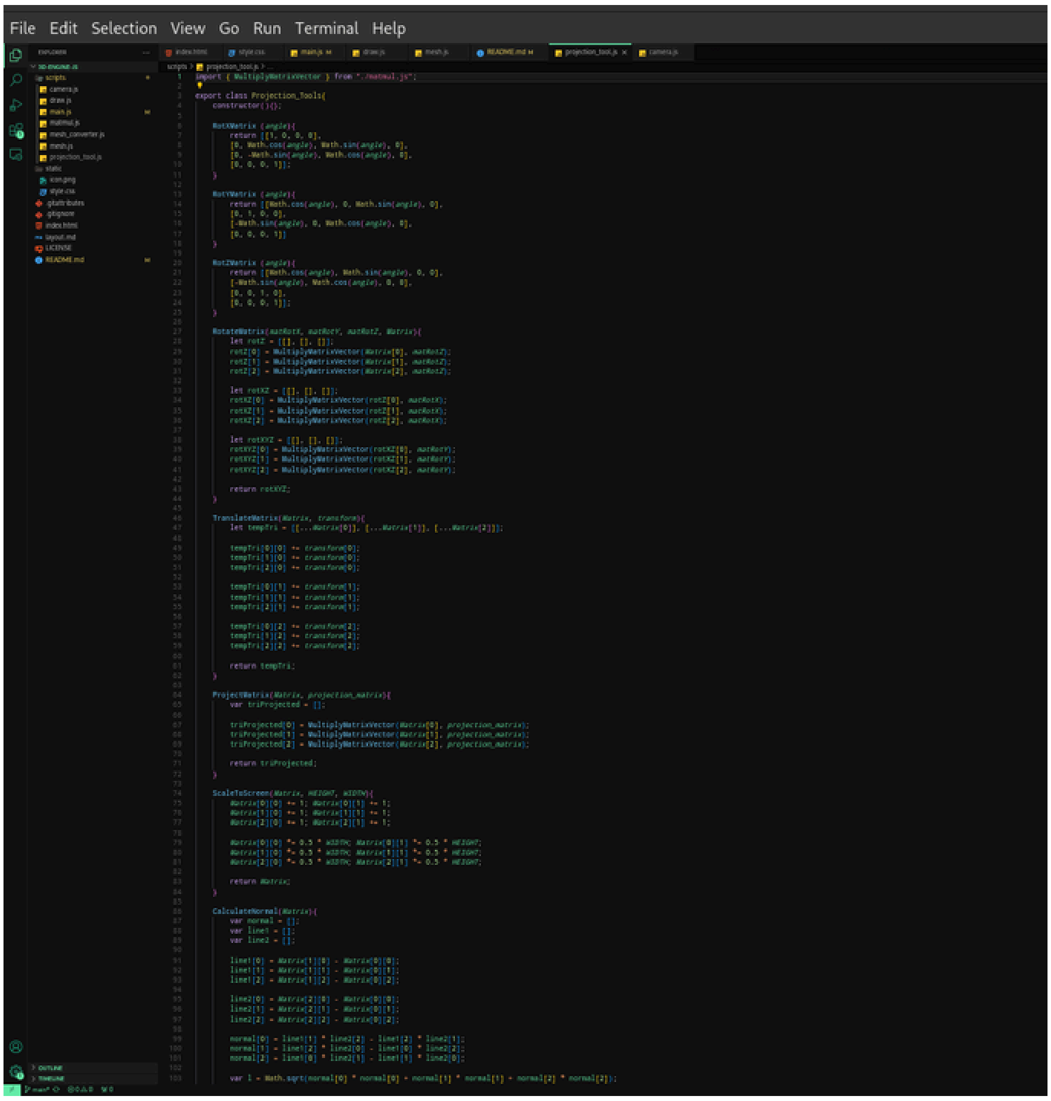
[You can view the test the project out here]()

Current project:



This is a screen shot of where the project is right now. It is not finished yet. This only has a cube but it can have any model that is converted to the vector 3 array and loaded into the project.

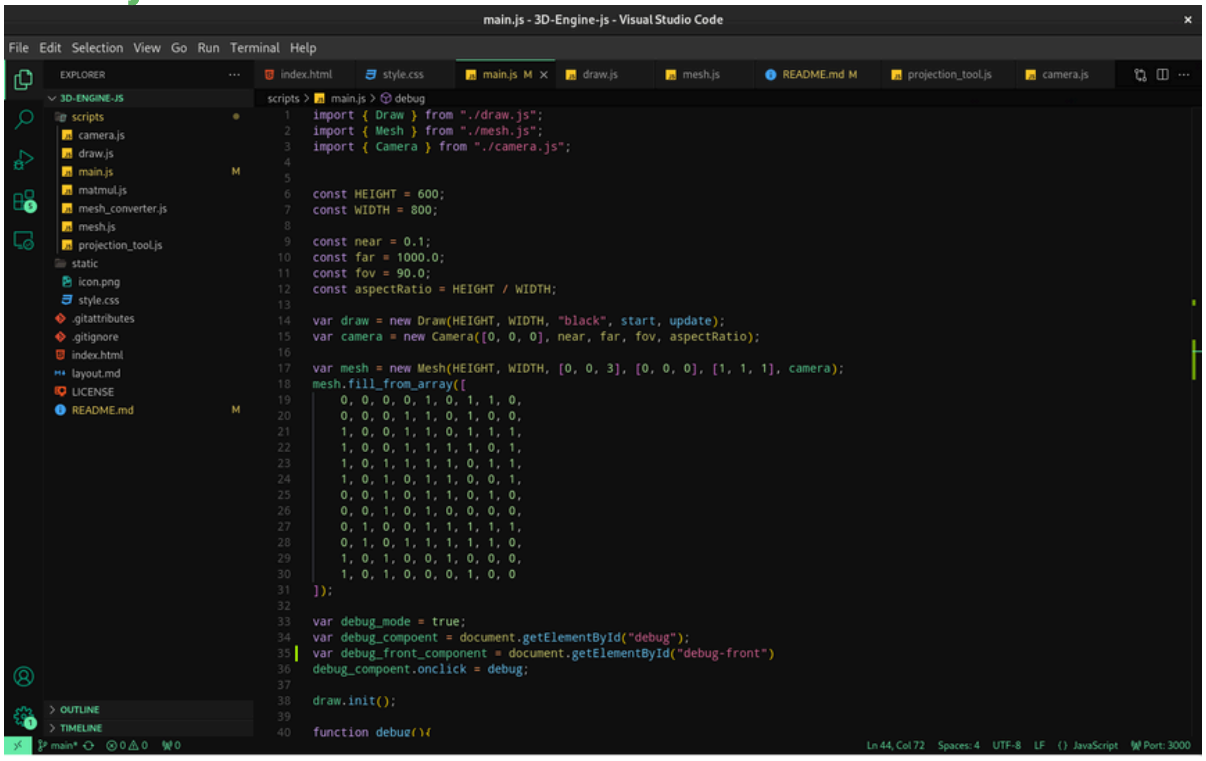
Projection Tools class:



```
import { MultiplyMatrixVector } from "./matmul.js";

export class Projection_Tools{
    constructor(){};

    RotXMatrix (angle){
        return [[1, 0, 0, 0],
        [0, Math.cos(angle), Math.sin(angle), 0],
        [0, -Math.sin(angle), Math.cos(angle), 0],
        [0, 0, 0, 1]];
    }

    RotYMatrix (angle){
        return [[Math.cos(angle), 0, Math.sin(angle), 0],
        [0, 1, 0, 0],
        [-Math.sin(angle), 0, Math.cos(angle), 0],
        [0, 0, 0, 1]]
    }

    RotZMatrix (angle){
        return [[Math.cos(angle), Math.sin(angle), 0, 0],
        [-Math.sin(angle), Math.cos(angle), 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]];
    }

    RotateMatrix(matRotX, matRotY, matRotZ, Matrix){
        let rotZ = [[], [], []];
        rotZ[0] = MultiplyMatrixVector(Matrix[0], matRotZ);
        rotZ[1] = MultiplyMatrixVector(Matrix[1], matRotZ);
        rotZ[2] = MultiplyMatrixVector(Matrix[2], matRotZ);

        let rotXZ = [[], [], []];
        rotXZ[0] = MultiplyMatrixVector(rotZ[0], matRotX);
        rotXZ[1] = MultiplyMatrixVector(rotZ[1], matRotX);
        rotXZ[2] = MultiplyMatrixVector(rotZ[2], matRotX);

        let rotXYZ = [[], [], []];
        rotXYZ[0] = MultiplyMatrixVector(rotXZ[0], matRotY);
        rotXYZ[1] = MultiplyMatrixVector(rotXZ[1], matRotY);
        rotXYZ[2] = MultiplyMatrixVector(rotXZ[2], matRotY);

        return rotXYZ;
    }

    TranslateMatrix(Matrix, transform){
        let tempTri = [[...Matrix[0]], [...Matrix[1]], [...Matrix[2]]];

        tempTri[0][0] += transform[0];
        tempTri[1][0] += transform[0];
        tempTri[2][0] += transform[0];

        tempTri[0][1] += transform[1];
        tempTri[1][1] += transform[1];
        tempTri[2][1] += transform[1];

        tempTri[0][2] += transform[2];
        tempTri[1][2] += transform[2];
        tempTri[2][2] += transform[2];

        return tempTri;
    }

    ProjectMatrix(Matrix, projection_matrix){
        var triProjected = [];

        triProjected[0] = MultiplyMatrixVector(Matrix[0], projection_matrix);
        triProjected[1] = MultiplyMatrixVector(Matrix[1], projection_matrix);
        triProjected[2] = MultiplyMatrixVector(Matrix[2], projection_matrix);

        return triProjected;
    }

    ScaleToScreen(Matrix, HEIGHT, WIDTH){
        Matrix[0][0] += 1; Matrix[0][1] += 1;
        Matrix[1][0] += 1; Matrix[1][1] += 1;
        Matrix[2][0] += 1; Matrix[2][1] += 1;

        Matrix[0][0] *= 0.5 * WIDTH; Matrix[0][1] *= 0.5 * HEIGHT;
        Matrix[1][0] *= 0.5 * WIDTH; Matrix[1][1] *= 0.5 * HEIGHT;
        Matrix[2][0] *= 0.5 * WIDTH; Matrix[2][1] *= 0.5 * HEIGHT;

        return Matrix;
    }

    CalculateNormal(Matrix){
        var normal = [];
        var line1 = [];
        var line2 = [];

        line1[0] = Matrix[1][0] - Matrix[0][0];
        line1[1] = Matrix[1][1] - Matrix[0][1];
        line1[2] = Matrix[1][2] - Matrix[0][2];

        line2[0] = Matrix[2][0] - Matrix[0][0];
        line2[1] = Matrix[2][1] - Matrix[0][1];
        line2[2] = Matrix[2][2] - Matrix[0][2];

        normal[0] = line1[1] * line2[2] - line1[2] * line2[1];
        normal[1] = line1[2] * line2[0] - line1[0] * line2[2];
        normal[2] = line1[0] * line2[1] - line1[1] * line2[0];

        var l = Math.sqrt(normal[0] * normal[0] + normal[1] * normal[1] + normal[2] * normal[2]);
```

This is the main logic class and has complex mathematical functions for calculating different things with in the program. It can rotate, transform and scale the matrix as well as calculating its normal relative to the camera.

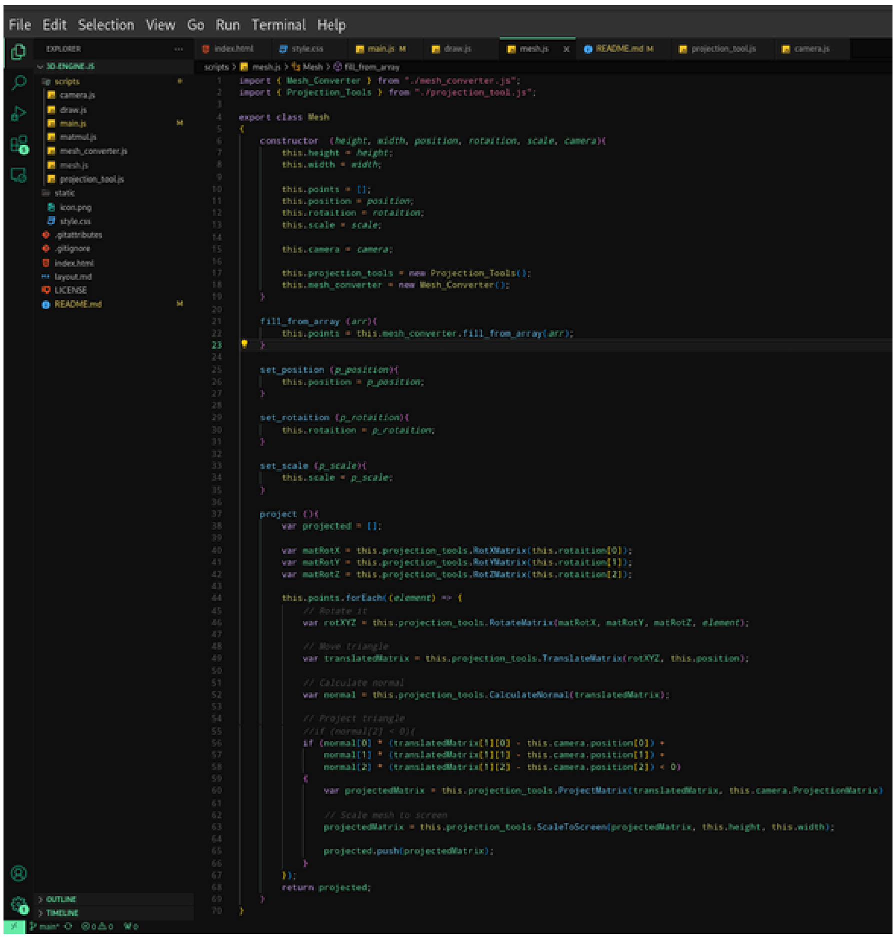
Main.js:



```js
import { Draw } from "./draw.js";
import { Mesh } from "./mesh.js";
import { Camera } from "./camera.js";


const HEIGHT = 600;
const WIDTH = 800;

const near = 0.1;
const far = 1000.0;
const fov = 90.0;
const aspectRatio = HEIGHT / WIDTH;

var draw = new Draw(HEIGHT, WIDTH, "black", start, update);
var camera = new Camera([0, 0, 0], near, far, fov, aspectRatio);

var mesh = new Mesh(HEIGHT, WIDTH, [0, 0, 3], [0, 0, 0], [1, 1, 1], camera);
mesh.fill_from_array([
    0, 0, 0, 0, 1, 0, 1, 1, 0,
    0, 0, 0, 1, 1, 0, 1, 0, 0,
    1, 0, 0, 1, 1, 0, 1, 1, 1,
    1, 0, 0, 1, 1, 1, 1, 0, 1,
    1, 0, 1, 1, 1, 1, 0, 1, 1,
    1, 0, 1, 0, 1, 1, 0, 0, 1,
    0, 0, 1, 0, 1, 1, 0, 1, 0,
    0, 0, 1, 0, 1, 0, 0, 0, 0,
    0, 1, 0, 0, 1, 1, 1, 1, 1,
    0, 1, 0, 1, 1, 1, 1, 1, 0,
    1, 0, 1, 0, 0, 1, 0, 0, 0,
    1, 0, 1, 0, 0, 0, 1, 0, 0
]);

var debug_mode = true;
var debug_compoent = document.getElementById("debug");
var debug_front_component = document.getElementById("debug-front")
debug_compoent.onclick = debug;

draw.init();

function debug(){
```

This is the main file that I used and is just there to wrap all the top level classes together in one file. It has a mesh that is defined as an array of floating point numbers. Every three numbers is a Vector 3 (A point in local 3D space) and every three Vector 3's creates a triangle which is the simplest 2D shape and is what is used to display complex 3D shapes. This is the same method that would be used in all 3D games that you see.

Mesh.js Class:



```js
import { Mesh_Converter } from "./mesh_converter.js";
import { Projection_Tools } from "./projection_tool.js";

export class Mesh
{
    constructor  (height, width, position, rotaition, scale, camera){
        this.height = height;
        this.width = width;

        this.points = [];
        this.position = position;
        this.rotaition = rotaition;
        this.scale = scale;

        this.camera = camera;

        this.projection_tools = new Projection_Tools();
        this.mesh_converter = new Mesh_Converter();
    }

    fill_from_array (arr){
        this.points = this.mesh_converter.fill_from_array(arr);
    }

    set_position (p_position){
        this.position = p_position;
    }

    set_rotaition (p_rotaition){
        this.rotaition = p_rotaition;
    }

    set_scale (p_scale){
        this.scale = p_scale;
    }

    project (){
        var projected = [];

        var matRotX = this.projection_tools.RotXMatrix(this.rotaition[0]);
        var matRotY = this.projection_tools.RotYMatrix(this.rotaition[1]);
        var matRotZ = this.projection_tools.RotZMatrix(this.rotaition[2]);

        this.points.forEach((element) => {
            // Rotate it
            var rotXYZ = this.projection_tools.RotateMatrix(matRotX, matRotY, matRotZ, element);

            // Move triangle
            var translatedMatrix = this.projection_tools.TranslateMatrix(rotXYZ, this.position);

            // Calculate normal
            var normal = this.projection_tools.CalculateNormal(translatedMatrix);

            // Project triangle
            //if (normal[2] < 0){
            if (normal[0] * (translatedMatrix[1][0] - this.camera.position[0]) +
                normal[1] * (translatedMatrix[1][1] - this.camera.position[1]) +
                normal[2] * (translatedMatrix[1][2] - this.camera.position[2]) < 0)
            {
                var projectedMatrix = this.projection_tools.ProjectMatrix(translatedMatrix, this.camera.ProjectionMatrix)

                // Scale mesh to screen
                projectedMatrix = this.projection_tools.ScaleToScreen(projectedMatrix, this.height, this.width);

                projected.push(projectedMatrix);
            }
        });
        return projected;
    }
}
```

This is one of the classes that I decided to call a "storage class". Its main purpose is to hold variables and basic logic that can be passed through to a logic class, in this case that logic class is the projection tools class seen above. The point of this class it to store the data for a mesh. It also loops over the mesh when it is time to project it and for every Vector 3 point it is passed to the projection tools where the point is projected into the 2D context of the screen.