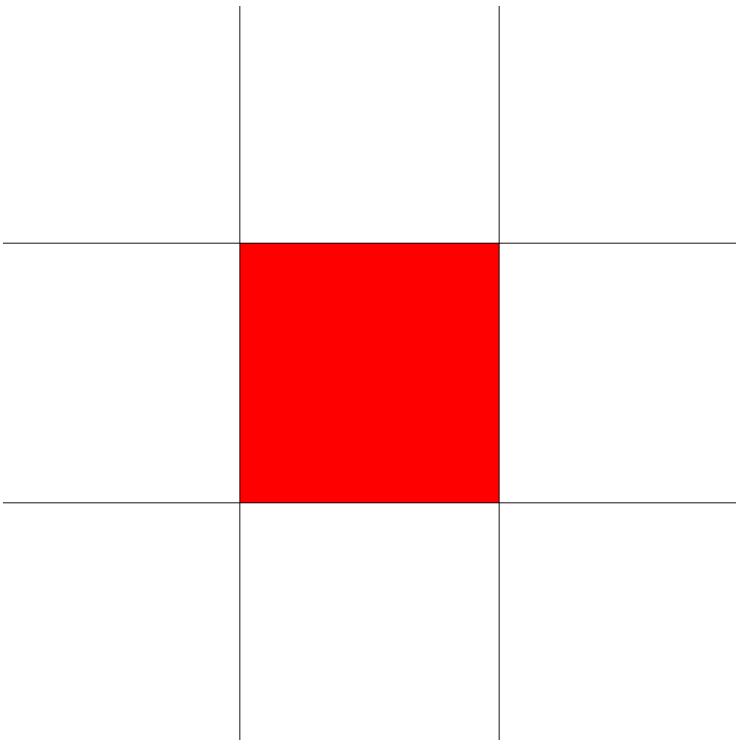
Computer Graphics Lab2

Full name (ID) : Erfan RafieiOskouei (240842587)

A1 — Pre-transforming



```
    // Pre-scale matrix: scaling by 1/2
let pre_scale = [
    [0.5, 0.0, 0.0, 0.0],
    [0.0, 0.5, 0.0, 0.0],
    [0.0, 0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0, 1.0]
];

// Pre-rotate matrix: rotating by pi/4 (45 degrees)
let pre_rotate = [
    [Math.cos(Math.PI / 4), -Math.sin(Math.PI / 4), 0.0, 0.0],
    [Math.sin(Math.PI / 4), Math.cos(Math.PI / 4), 0.0, 0.0],
    [0.0, 0.0, 1.0, 0.0],
    [0.0, 0.0, 0.0, 1.0]];
```

```
// A1 -- ADD CODE HERE

point =  pre_rotate * pre_scale * point;

// A1, A2, A3, A4, A5 -- MODIFY HERE
gl_Position = rotate * point;
```

So, firstly I implemented transformations using GLSL shaders in conjunction with JavaScript to change the appearance of an object and scaling it by a factor of 1/2 and rotating it by π/4 radians (45 degrees).
These transformations were applied using two matrices:
a scaling matrix (pre_scale) and a rotation matrix (pre_rotate).

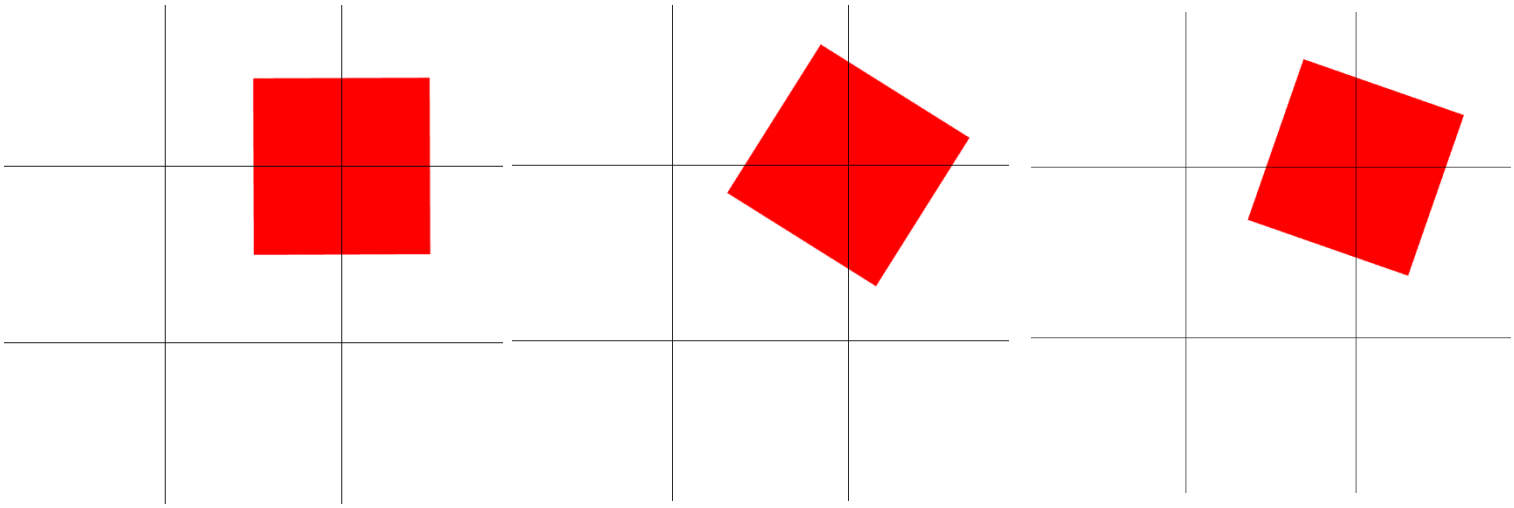
In GLSL, I applied both transformations needed by multiplying the pre_rotate and pre_scale matrices with the vertex coordinates as the following code :

*point = pre_rotate * pre_scale * point;*

This sequence of actions first scales the object down and then rotates it, ensuring that the transformations are applied correctly. The final result is an object that is both reduced in size by half and rotated by 45 degrees.

In the JavaScript code, I initialized the two matrices and passed them to the shader, so in this way the transformations are computed for each vertex.

A2 — Adding translation



```
// A1, A2, A3, A4, A5 -- MODIFY HERE

gl_Position = translate * rotate * point;
```

```
var translate_loc;
```

```
translate_loc = gl.getUniformLocation(program, 'translate');
```

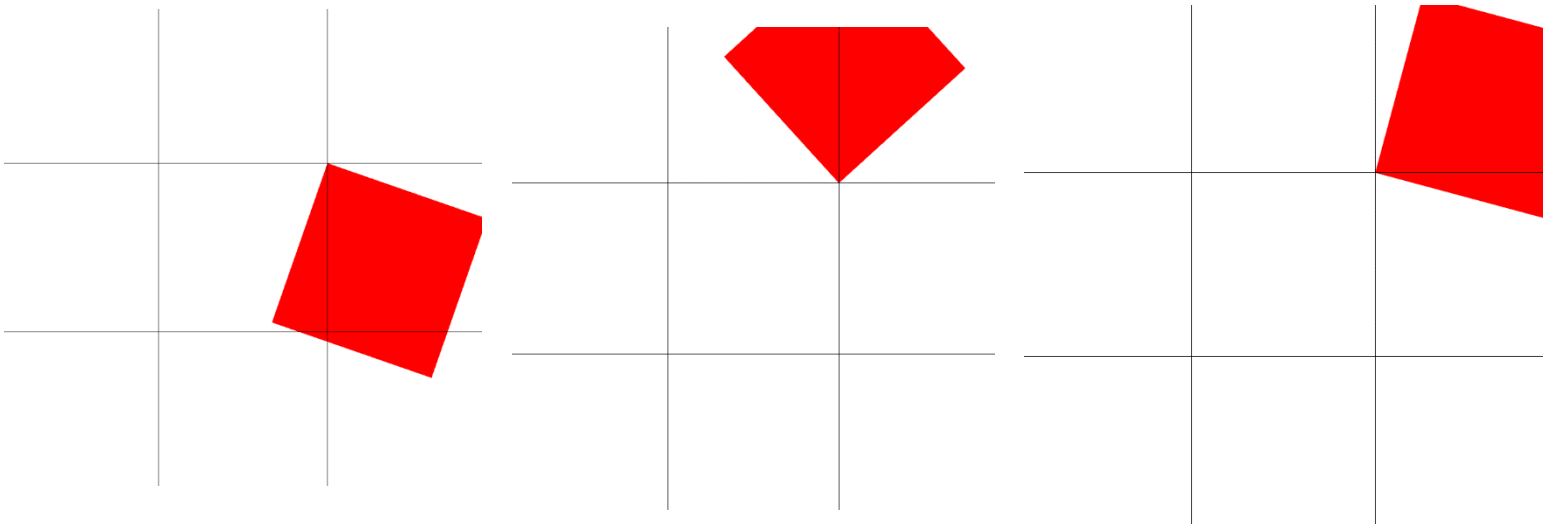
```
let translate = [[1, 0, 0, side/2],
                 [0, 1, 0, side/2],
                 [0, 0, 1, 0],
                 [0, 0, 0, 1]];
```

```
gl.uniformMatrix4fv(translate_loc, false, mat_float_flat_transpose(translate));
```

```
gl.uniformMatrix4fv(translate_loc, false, mat_float_flat_transpose(identity));
```

For this part, what I did was I did the steps mentioned in the exercise and I used the pre-defined codes as my guide to how to write these functions. The only tricky thing about this part was that the multiplications in the shader code must be in the correct order for it to work. So, the translate must be the first variable.

A3 — Rotation around a different point



I began by duplicating the existing translation matrix within the shader's main loop, naming this new matrix "translate_inv". The purpose of this matrix was to store the inverse of the original translation:

```
mat4 translate_inv = translate;
```

Next, I modified the x and y components of translate_inv to be the negatives of those in the original translate matrix. This step was crucial for creating the opposite translation effect. I accessed the matrix elements using the syntax translate[col][row] to make these adjustments.

The code for the inversed matrix was like this :

```
translate_inv[3][0] = -translate[3][0];
translate_inv[3][1] = -translate[3][1];
```

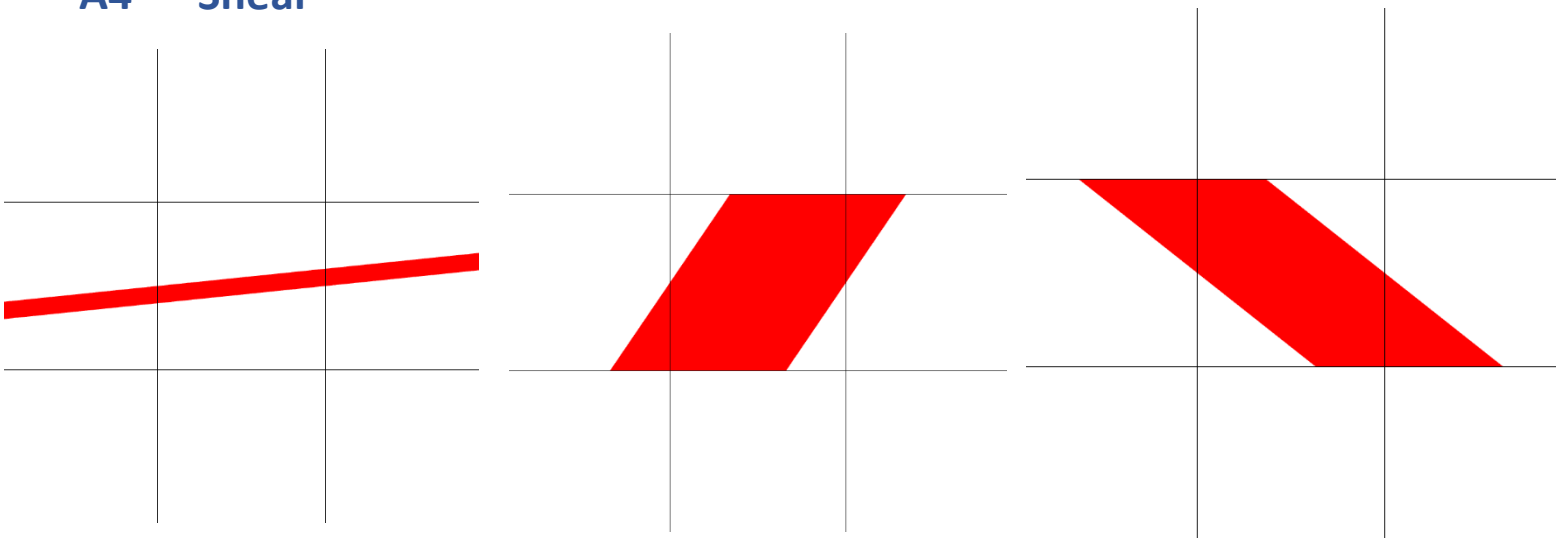
The next step was to incorporate the rotation matrix into this setup. I modified the transformation statement to:

```
gl_Position = translate * rotate * translate_inv * point;
```

After I refreshed the page, the output was the same as the one given in the exercise.

The order of these matrix operations is critical to understanding the result. First, the inverse translation moves the square so its top-right corner is at the origin. Then, the rotation is applied around this point. Finally, the original translation moves the square back to its starting position, but now rotated around the desired point.

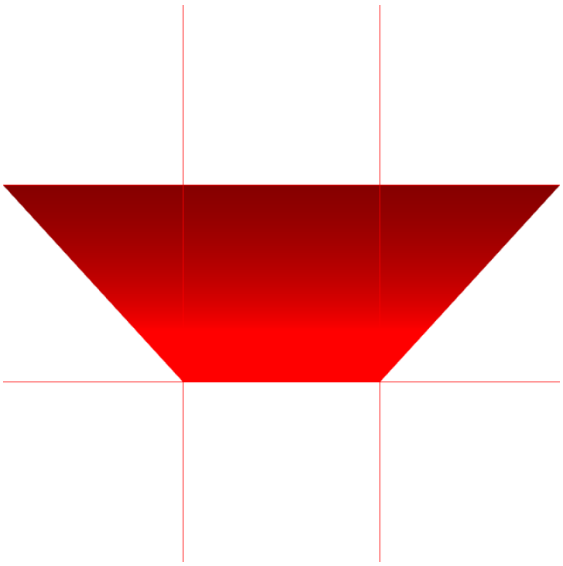
A4 — Shear



```
let shear = [[1, Math.tan(theta), 0, 0],
             [0, 1, 0, 0],
             [0, 0, 1, 0],
             [0, 0, 0, 1]];
```

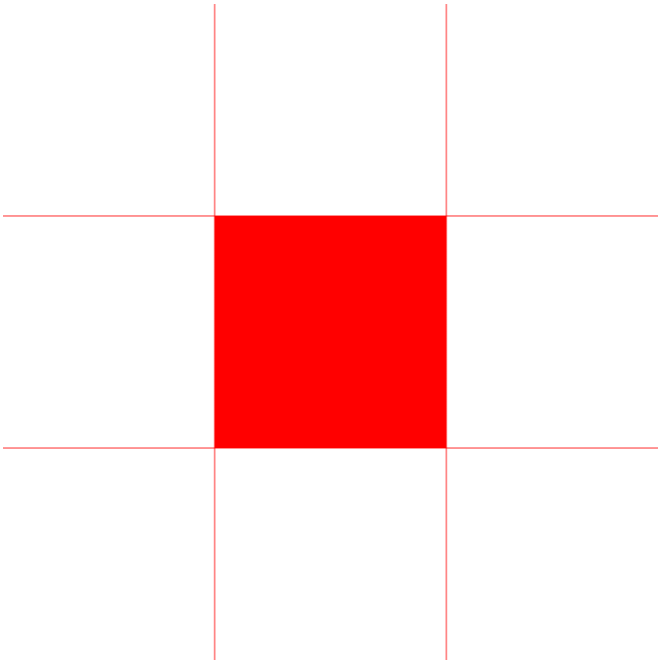
For this part, I've added the shear matrix as the exercise wanted and connected the location from shader code to JS code. The rest was as same as the previous steps. And in the end for the gl_position I multiplied with the rotate point which creates the outputs above.

A5 — Projective transformation

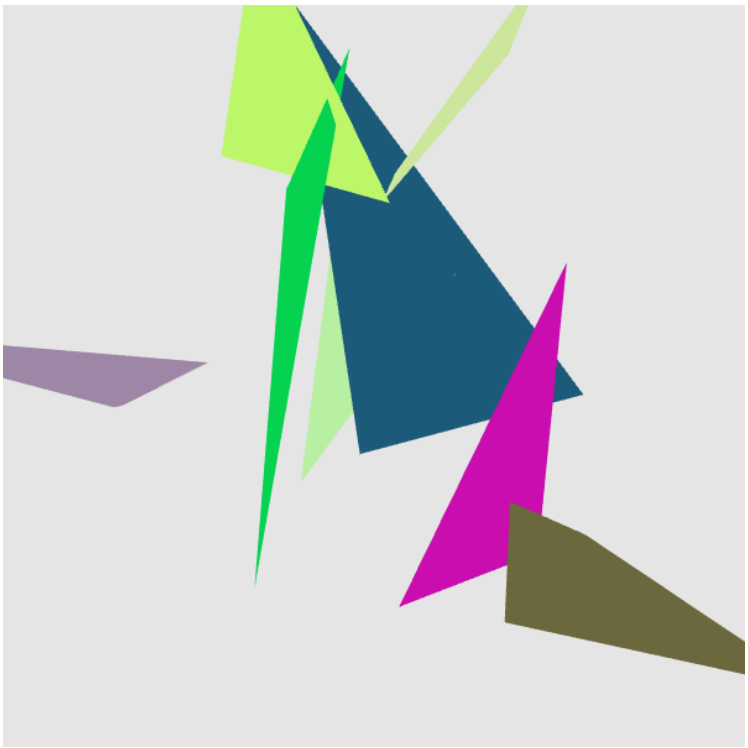


For this part I did exactly as the instructions told.

This is the last image after it turns back to the first square :



B1 — Copying the current view

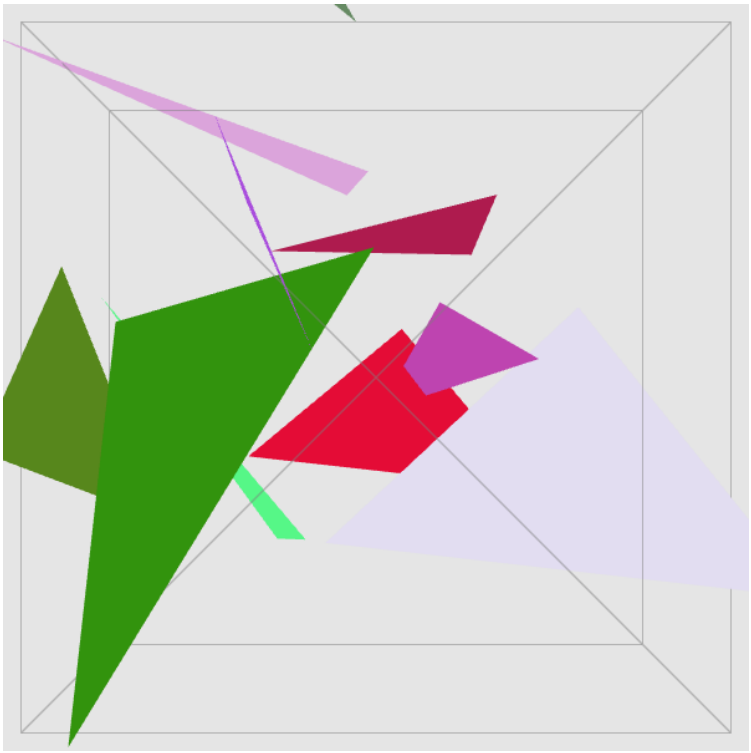


I did the steps told in the exercise.

```
ctx.drawImage(gl.canvas, 0, 0);
```

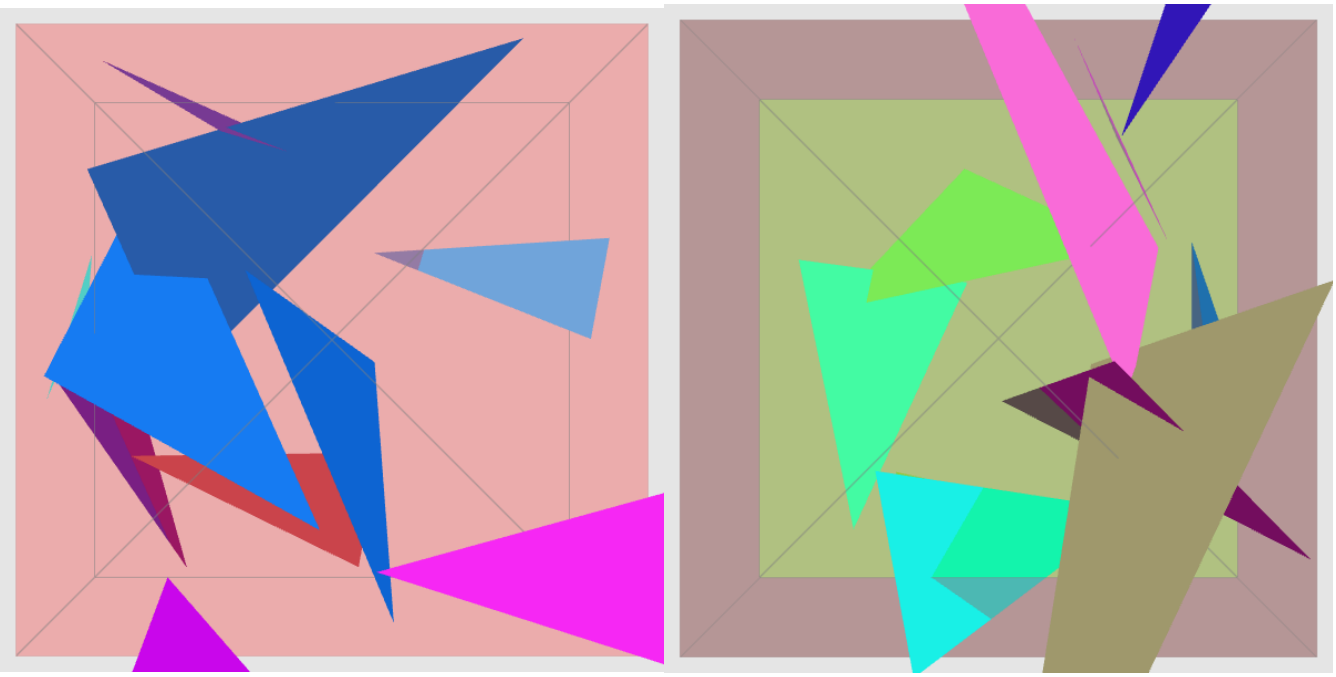
```
window.setTimeout(render_control, 1000/60);
```

B2 — Back view of the frustum



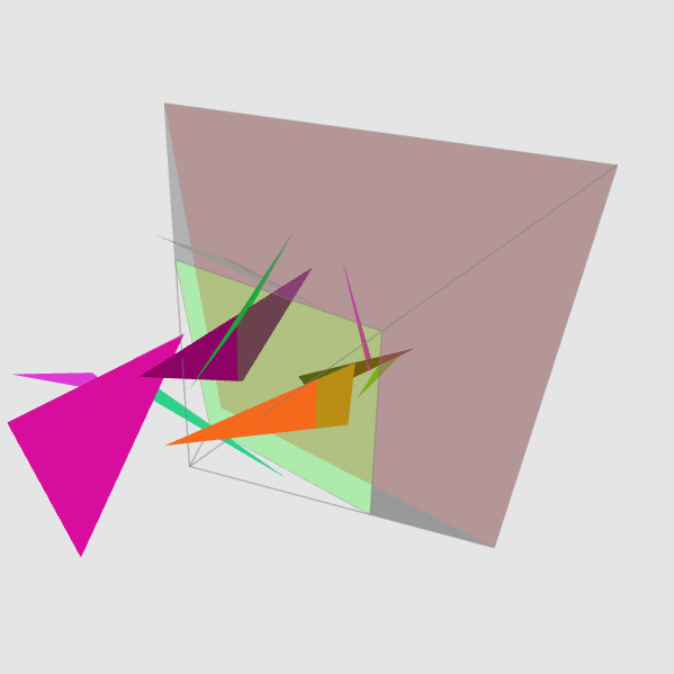
I’ve added the requested changes that was needed in the exercise.

B3 — Back view of the near and far clipping planes



The new changes added the rendering of the far plane and the rendering of the near plane and side planes.

B4 — External view of the whole frustum



So, the camera change made the view point different and it makes it more comprehensible.

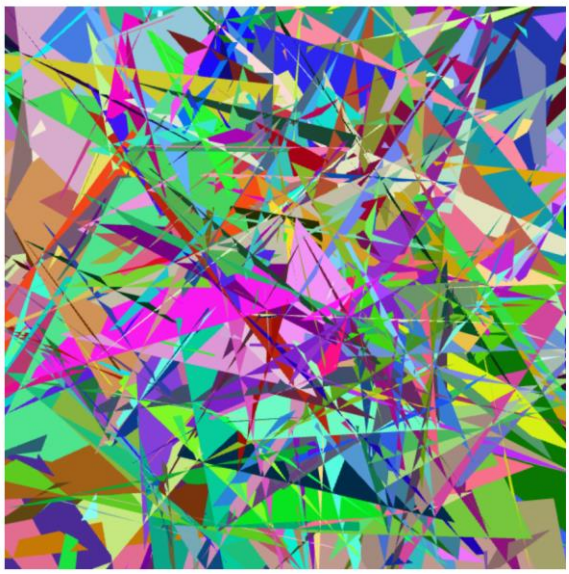
After changing the

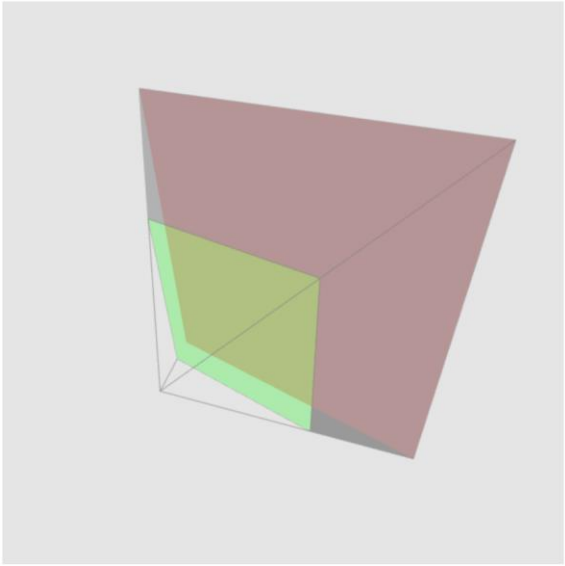
```
var num_triangles = 1000;
```

and changing the

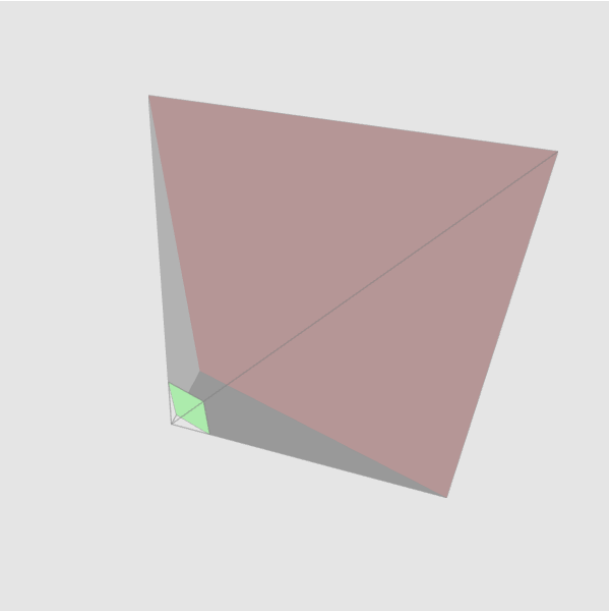
```
render_triangles = false;
```

we get this output :

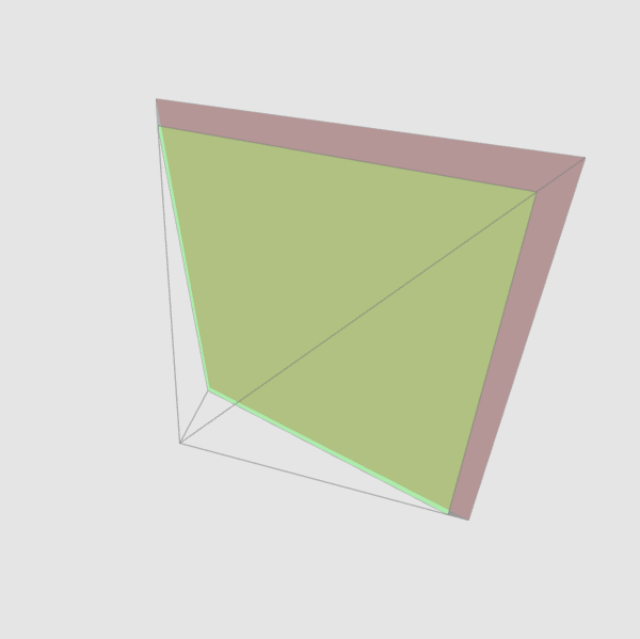




B5 — Adjusting the near plane

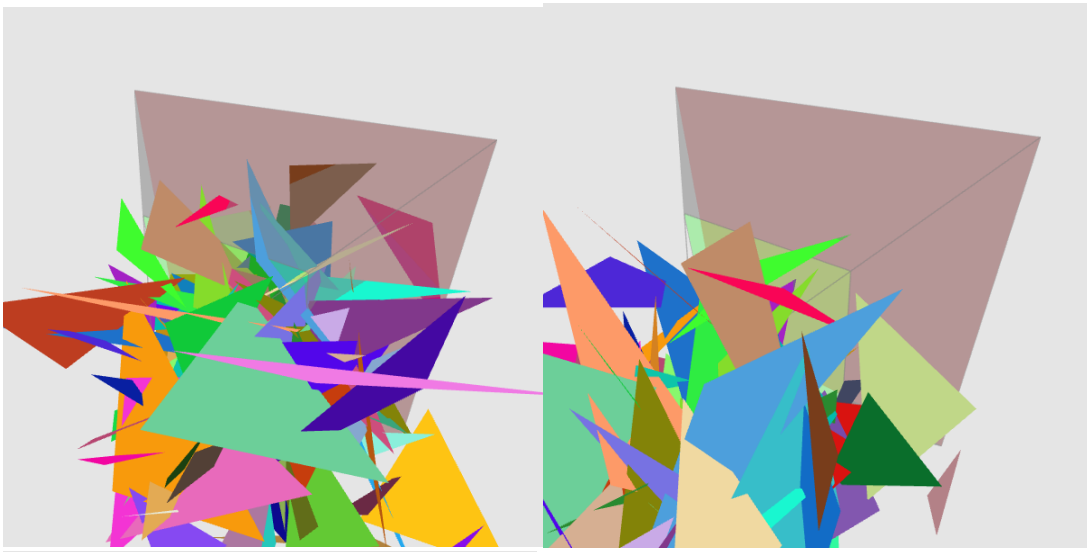


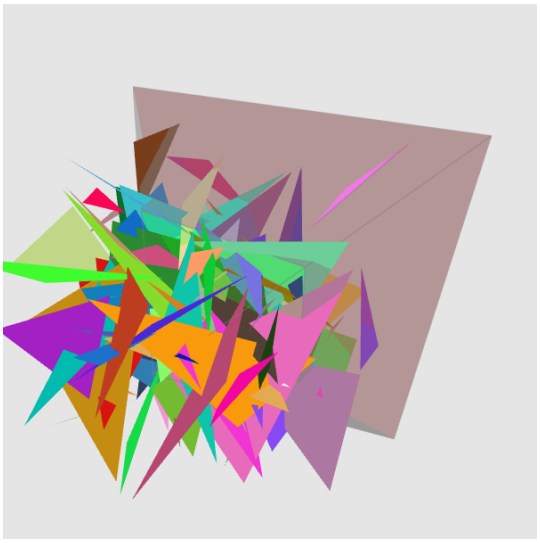
The near plane is adjusted to 10



The near plane is adjusted to 90

C1 — API settings & GUI control





So, for this change to happen I had to add some things to the JS code,

First I've added new declarations :

```
// buffers and attributes
var vertices, projection, modelview, rotation, translation;

// uniform locations
var vertex_loc, colour_loc, projection_loc, modelview_loc, rotation_loc, translation_loc;

// C1: DECLARE translation_inv and translation_inv_loc here

var translation_loc,translation_inv, translation_inv_loc, rotation_loc;
```

After that I added the locations :

```
  // C1: GET ROTATION AND TRANSLATION LOCATIONS HERE

  translation_loc = gl.getUniformLocation(program, 'translation');
  translation_inv_loc = gl.getUniformLocation(program, 'translation_inv');
  rotation_loc = gl.getUniformLocation(program, 'rotation');

  // --- rendering options ---
```

After that like A3 I declared the matrices for the rotation and translation and translation_inv :

```
// Define the translation matrices
translation = [
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, max_depth / 2],
    [0, 0, 0, 1]
];

translation_inv = [
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, -max_depth / 2],
    [0, 0, 0, 1]
];

// Define the rotation matrix around the y-axis
rotation = [
    [Math.cos(theta), 0, Math.sin(theta), 0],
    [0, 1, 0, 0],
    [-Math.sin(theta), 0, Math.cos(theta), 0],
    [0, 0, 0, 1]
];
```

And I updated the theta steps too :

```
// Update the rotation angle
    theta += theta_step;
```

And the rest of the steps were the same as A3.

In the end in the shader code I multiplied them for the last result :

```
// Apply the transformations
 vec4 transformed_point = translation * translation_inv * rotation * vertex;
 gl_Position = projection * modelview * transformed_point;
```

C2 — Camera matrix construction (advanced)

To create the alternative wrapper function mat_perspective_alt(right, top, near, far), I used the top-right corner coordinates to compute the fovy (vertical field of view) and the aspect ratio. Then I tried to call the original mat_perspective(fovy, aspect, near, far) function with these parameters.

My implementation is like this :

```
function mat_perspective_alt(right, top, near, far) {
   // Calculate the aspect ratio
   let aspect = right / top;

   // Calculate the vertical field of view (fovy) in radians
   let fovy_rad = 2 * Math.atan(top / near);

   // Convert fovy from radians to degrees
   let fovy_deg = fovy_rad * (180 / Math.PI);
```

```
    // Call the original mat_perspective function with fovy, aspect, near, and far
    return mat_perspective(fovy_deg, aspect, near, far);
}
```

Explanation:

1. **Aspect Ratio**: The aspect ratio is the ratio of the right coordinate to the top coordinate, defining the width-to-height relationship of the near-plane.
2. **Vertical Field of View**: The field of view is computed using the inverse tangent (Math.atan) of top/near, which gives the half-angle of the field of view in radians. I multiplied it by 2 to get the full vertical field of view, and then converted the radians to degrees for use in the mat_perspective() function.
3. **Function Call**: The wrapper function calls the original mat_perspective() using the calculated fovy and aspect.

For the second version I used the help of generative AI and it gave me instructions for this function and I did the following steps :

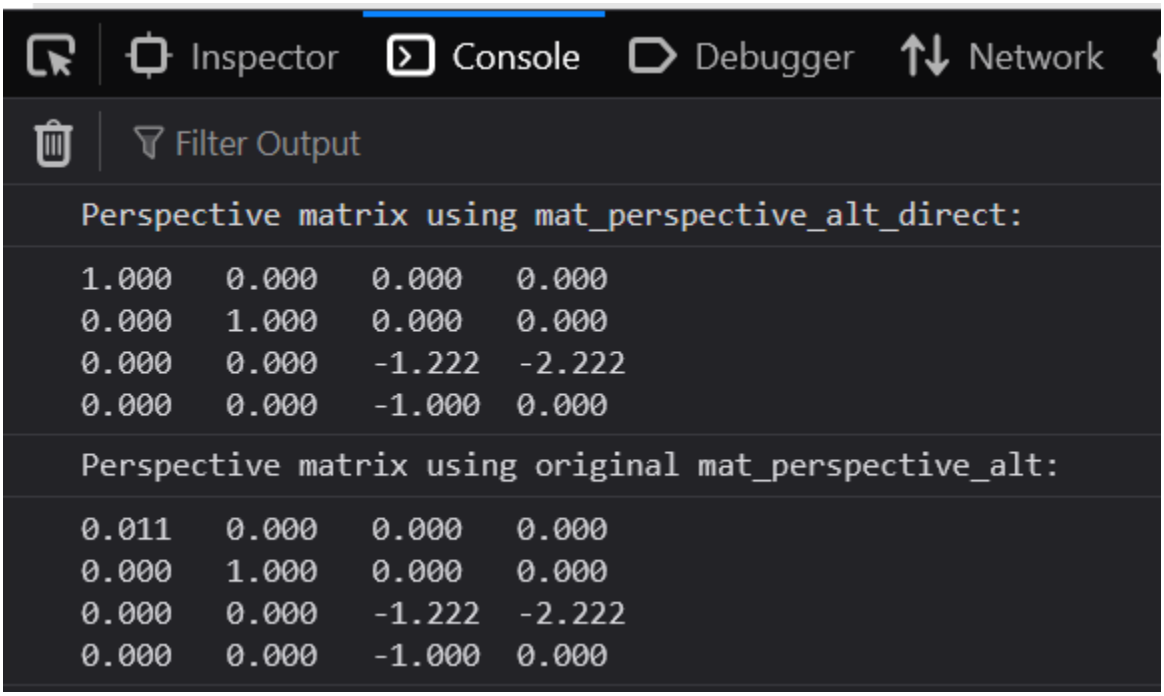
I Constructed matrix N according to the formula in slide 04-B/21, using the provided near and far values. The I Constructed matrix S using near, right, and top values as shown in the slide and in the end, I Multiplied N and S using the mat_prod() function to get the final perspective projection matrix.

I've added a testing function to compare them too. My code is as following :

```
function mat_perspective_alt_direct(right, top, near, far) {
    // Create matrix N
    let N = mat_zero(4, 4);
    N[0][0] = 1;
    N[1][1] = 1;
    N[2][2] = -(far + near) / (far - near);
    N[2][3] = -2 * far * near / (far - near);
    N[3][2] = -1;

    // Create matrix S
    let S = mat_zero(4, 4);
    S[0][0] = near / right;
    S[1][1] = near / top;
    S[2][2] = 1;
    S[3][3] = 1;
```

```
    // Multiply N and S to get the final perspective matrix
    return mat_prod(N, S);
}

// Test function to compare results
function testPerspectiveMatrixDirect() {
    // Test values
    let right = 1, top = 1, near = 1, far = 10;

    // Calculate using our new direct function
    let directMatrix = mat_perspective_alt_direct(right, top, near, far);

    console.log("Perspective matrix using mat_perspective_alt_direct:");
    mat_console_log(directMatrix);

    // Compare with the original function
    let fovy = 2 * Math.atan(top / near) * (180 / Math.PI);
    let aspect = right / top;
    let originalMatrix = mat_perspective_alt(fovy, aspect, near, far);

    console.log("Perspective matrix using original mat_perspective_alt:");
    mat_console_log(originalMatrix);
}
```

And the results for the console debug are as following :



As you can see the end results are pretty similar to each other.