Computer Graphics Lab 2

Name: James Davies
Student ID: 200682354

October 2023

1 Exercise A

1.1 A1: Pre-transforming

In order to rotate and scale our object, we make use of two transformation matrices, assigned to variables pre_rotate and pre_scale. Within the vertex shader, these are declared as uniform mat4 (4x4 matrices), and then multiplied as:

point = pre_rotate * pre_scale * point;

The 'point' variable is a vector consisting of vertex.x, vertex.y, 0.0 and 1.
We then change the actual implementation of the two matrices within the JS file; they are currently in identity form so multiplication results in the same matrix. For the rotation matrix, out implementation uses the Math module, using the Pi functionality via Math.PI:

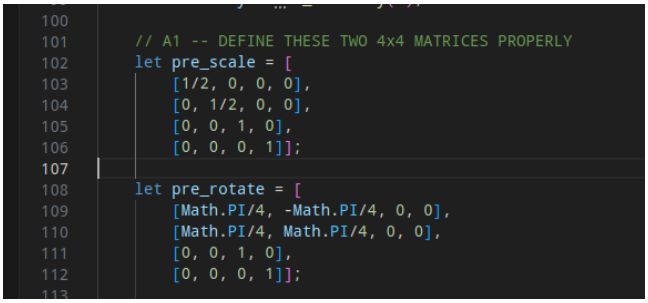

Figure 1: A1 Transformation Matrices

1.2 A2: Adding translation

To translate our object, we use another uniform mat4 (4x4 format defined in vertex shader) matrix called 'translate'. Our gl_Position variable within the

shader, used for writing homogeneous vertex position, is now set via:

gl_Position = translate * rotate * point;

We also have to set the location of the shader matrix; this is set via two function calls from the API: .getUniformLocation() and .uniformMatrix4fv(). The first gets the uniform location from the vertex shader and then the second allows us to declare/initialize the location of the shader matrix as a JS global variable. When we define the matrix in JS, we use the 'side' variable to translate the rotating square up and right by a factor of side/2:

1.3 A3: Rotation around a different point

With our object still top right of frame, we can rotate it around it's corner (top right specifically) instead of rotating around centre. We can do this by defining the inverse matrix of the translate matrix and then again setting gl_Position like so:

gl_Position = translate * rotate * translate_inv * point;

This allows for the object to be rendered in a way that the two matrices don't cancel each other out! An ordering of "translate * translate_inv * rotate * point" would mean no change for the object, as a matrix multiplied by its inverse is the identity matrix.

1.4 A4: Shear

Another transformation matrix is needed for shearing, and we can define one for horizontal shearing specifically using the code given in Fig.2. Once again we then use this matrix to update the gl_Position transformation expression:

gl_Position = shear * point;

```
// A2-5 DEFINE NEW MATRICES
let translate = [
    [1, 0, 0, side/2],
    [0, 1, 0, side/2],
    [0, 0, 1, 0],
    [0, 0, 0, 1]];

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

Figure 2: A2 and A4 - Translation and Shear Matrices

As we are declaring a new matrix, we need to follow the same steps as previous matrices in terms of calling .getUniformLocation() and .uniformMatrix4fv() and saving the location for the shear matrix in a global JS variable called shear_loc. After we have implemented this, the transformation has the effect of a sheared square (to essentially a parallelogram) to the horizontal axis and then back again. The process is then repeated.

1.5 A5: Projective Transformation

The last transformation of the square involves using a projective transformation, via the following matrix:

```
let projective = [
    [4/(2+side), 0, 0, 0],
    [0, 1, 0, ((side-2)*side)/(2*(side+2))],
    [0, 0, 1, 0],
    [0, (2*(side-2))/(side*(side+2)), 0, 1]];
```

Figure 3: Projective Matrix

We can then apply it to gl_Position via initialising the variable as "projective * point". This transforms then original square to a trapezium, which is a 2D equivalent of a frustum. We can actually reverse this process to mimic the mapping of a frustum to a cube, or Normalised Device Coordinates; the process in A5 is the 2D analogy of this. To reverse this process we can find the inverse matrix of "projective" and then use the following line for gl_Position:
gl_Position = projective_inv * projective * point;

2 Exercise B

2.1 B1: Copying the current view

Within the render_control() function there are two calls to the render() function. This, alongside the utilisation of a primary and secondary 'canvas' and a copying of the initial rendering using .drawImage(), allows for there to be multiple perspectives for each rendering within exercise B. The first render (and camera perspective) is rendered without the frustum at all, and only see the primitives. In B1, we just see a direct copy of this rendering across both canvasses.

The actual implementation of the copying uses:

// copy the webgl rendered image to other canvas
ctx.drawImage(gl.canvas, 0, 0);

The 'ctx' variable which .drawImage() is called via is the drawing context, which was further set by an API call called .getContext().

2.2 B2: Back view of the frustum

In the B2 rendering, we change the perspective of the second canvas by drawing the outline of the frustum in 3D. The original canvas still sees nothing, as it is on the boundary of the field of view.

We also change the camera position to a "backed-up" perspective by altering the z coordinate to +50, which pushes the viewpoint further behind camera. This, alongside setting of specific flags, allow us to now see instances of 'clipping' where primitives have either intersected either each other or the near/far planes of the frustum. The flags that are set (via changes to multiple boolean variables) within the changes in B2 are rendering the near, far and side edges of the frustum. So in the second canvas we now see the old viewpoint and near/far clipping planes.

2.3 B3: Back view of the near and far clipping planes

While in B2 we were concerned with changing the boolean flags to render the edges of the frustum, in B3 we now look at changing the flags to render the near, far and side planes of the frustum. This again allows us to see the differences between the two canvasses cause by clipping, as the rendered planes are translucent and allow us to see things rendered in the second canvas that will have been clipped in the first.

Specific to B3i and B3ii - B3i includes the rendering of the far plane only, and B3ii includes a rendering of the near, side and far planes. This again is set via boolean flags being changed from 'false' to 'true'. The full set of flags are:

render_triangles = true;
render_near_plane = true;
render_far_plane = true;
render_side_planes = true;
render_near_edges = true;
render_far_edges = true;
render_side_edges = true;

2.4 B4: External view of the whole frustum

Within this image, we change the perspective of the camera position (via the eye variable). Before this change, the coordinates (x, y, z) of the eye variable in a "backed-up" view are 0, 0, and 50.0 which puts the perspective directly behind the camera with a positive z coordinate.

To extend this into an overhead view, we can change the x/y coordinates to 110 and 140 respectively, which pushes the camera position up and to the right of where it was prior. The camera position is also pushed slightly back from 50 to 70.0 in B4. This allows us to get a macro perspective of the scene, seeing the full cluster of primitives relative to the size of the frustum externally.

2.5 B5: Adjusting the near plane

In adjusting the near plane of the viewing frustum (via changing the requisite camera parameter variable 'near' in line 18), we can shift the field of view and by extension the size of the primitives within the scene.

This camera parameter is implemented as a percentage; we can see in B5i that when the variable near = 10, the near plane makes up a fraction of the frustum. In contrast, when the camera parameter near = 90, the near plane makes up a large proportion of the frustum. This affects the perceived size of the primitives, as obviously the further away the near plane is from the camera the smaller they will be.

3 Exercise C

3.1 C1: API settings

In order to achieve rotation around of the centre of the cluster, we have to use a translation matrix and the inverse of the translation matrix, named translation and translation_inv respectively. The formal translating along z by max_depth/2 and the inverse occuring in the latter.

We then declare another matrix for the rotation, in which we use a transformation matrix for rotation along the y axis. The rotation and transformation are then disabled via setting of the identity matrices.

4 Images

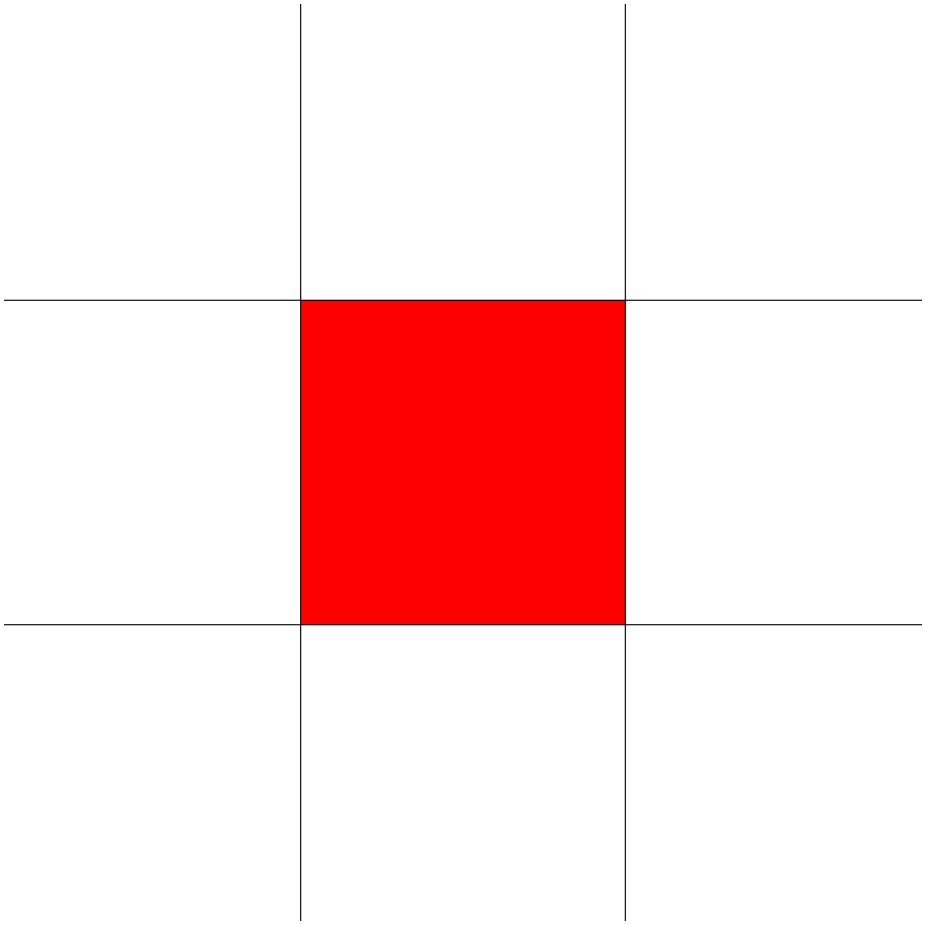

Figure 4: A1

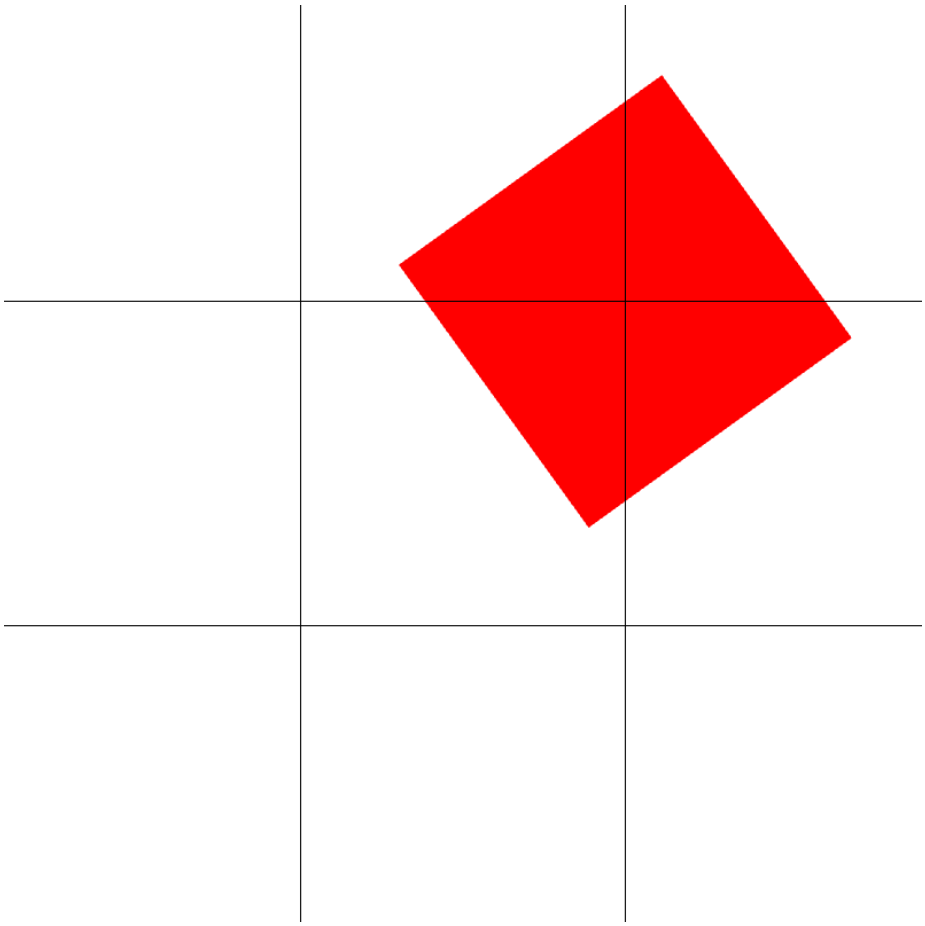

Figure 5: A2i

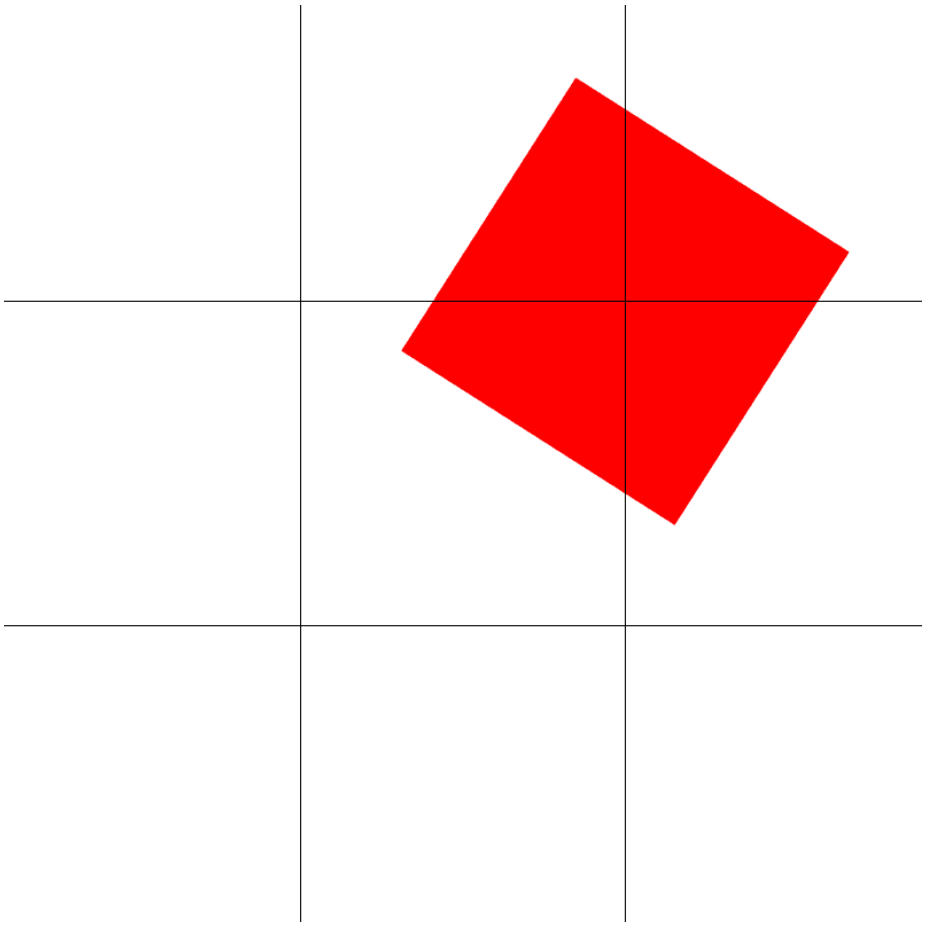

Figure 6: A2ii

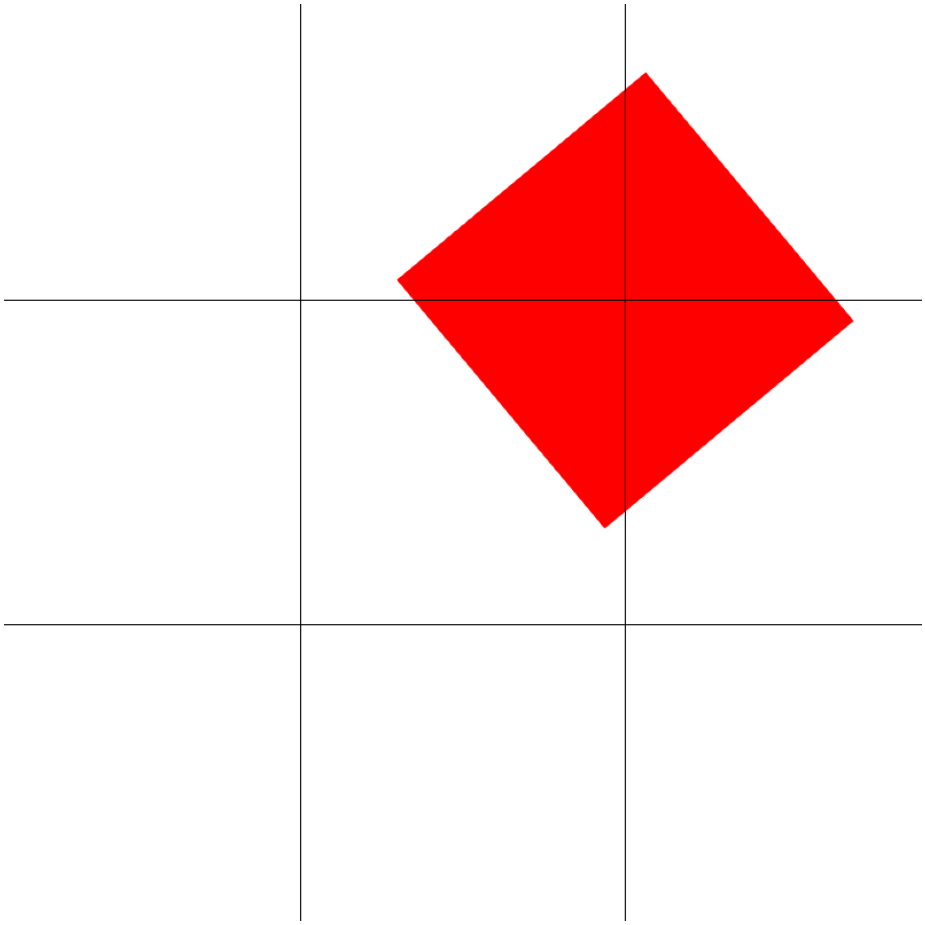

Figure 7: A2iii

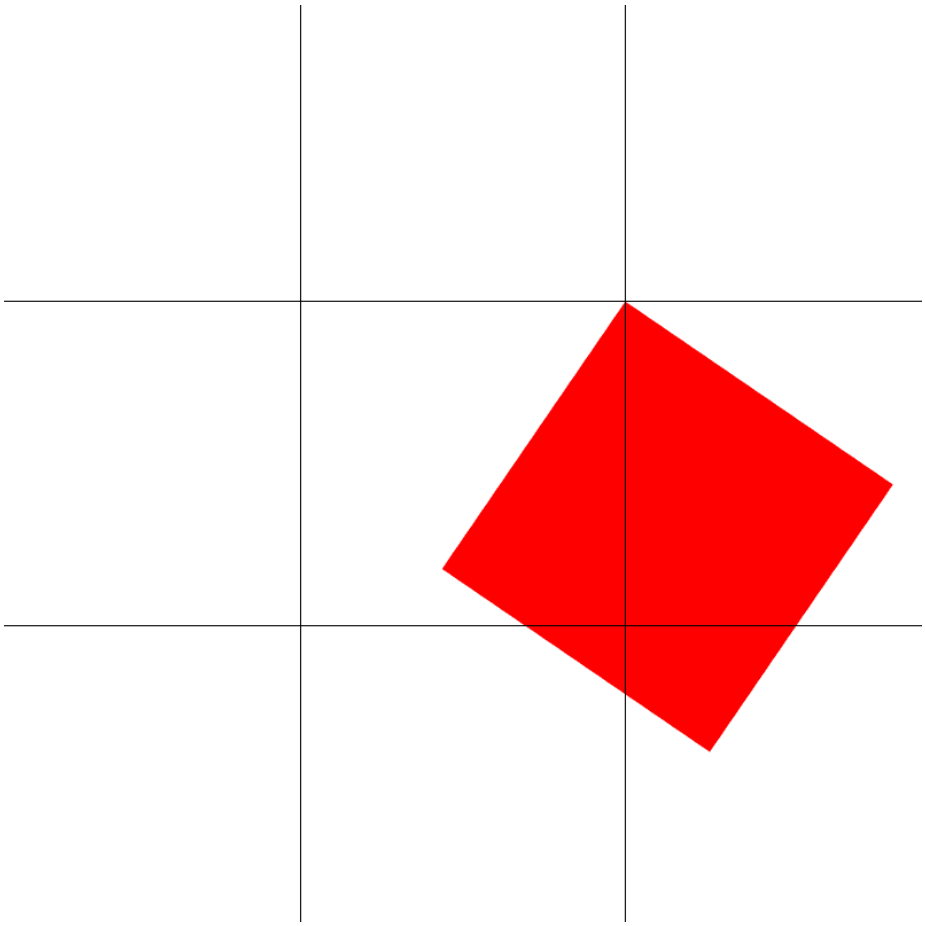

Figure 8: A3i

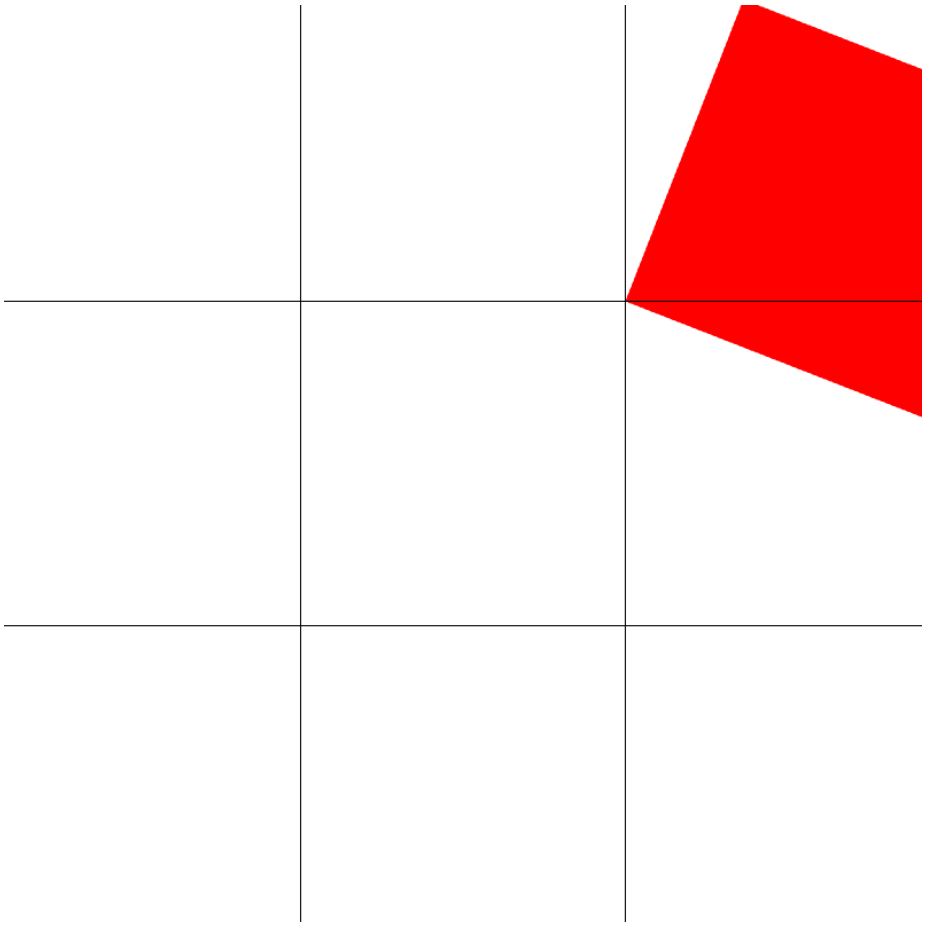

Figure 9: A3ii

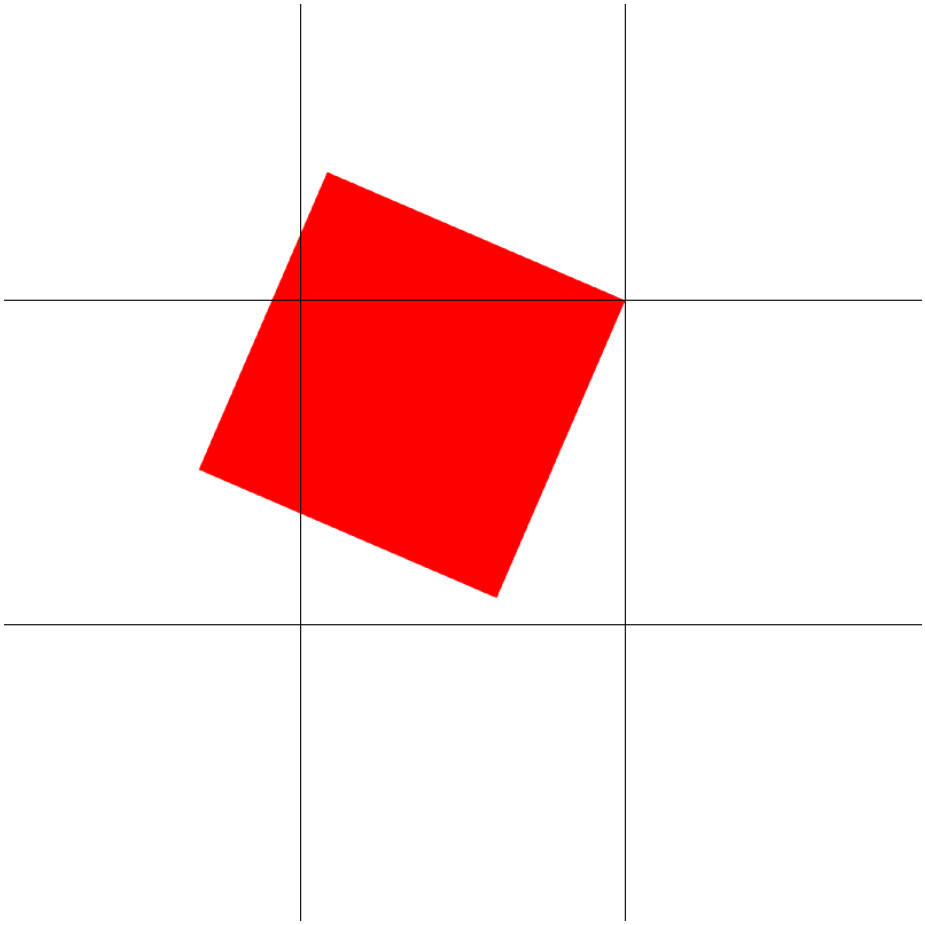

Figure 10: A3iii

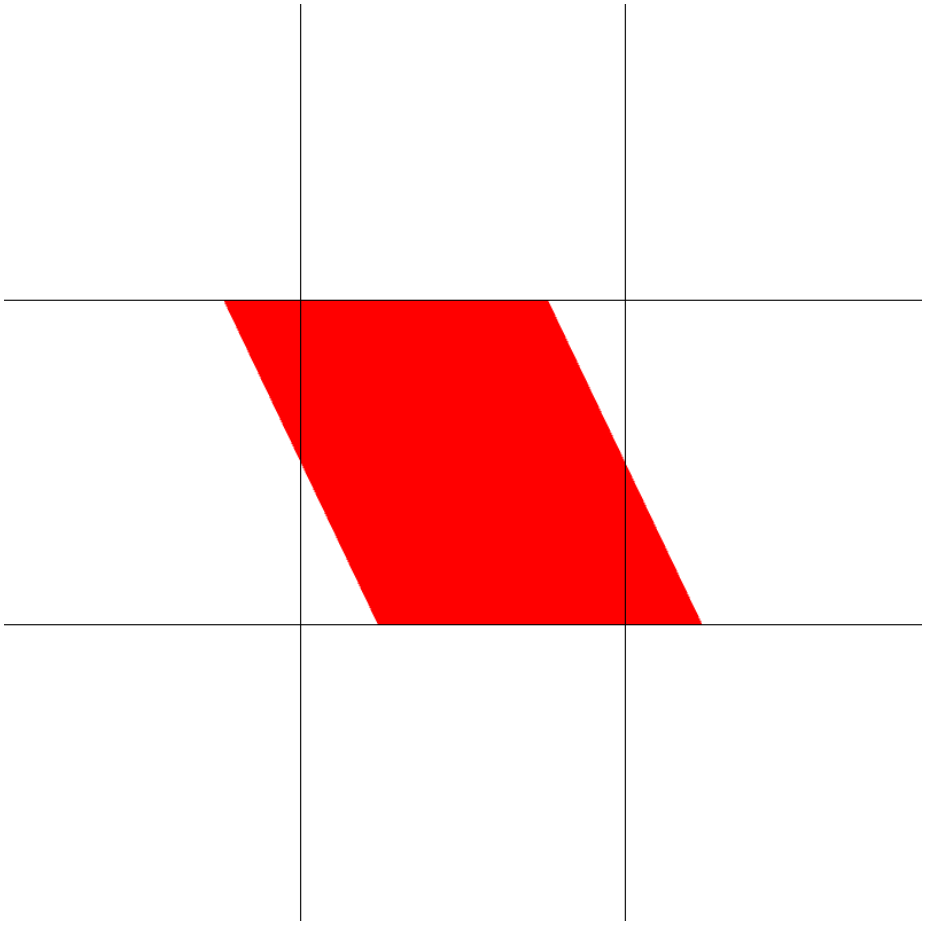

Figure 11: A4i

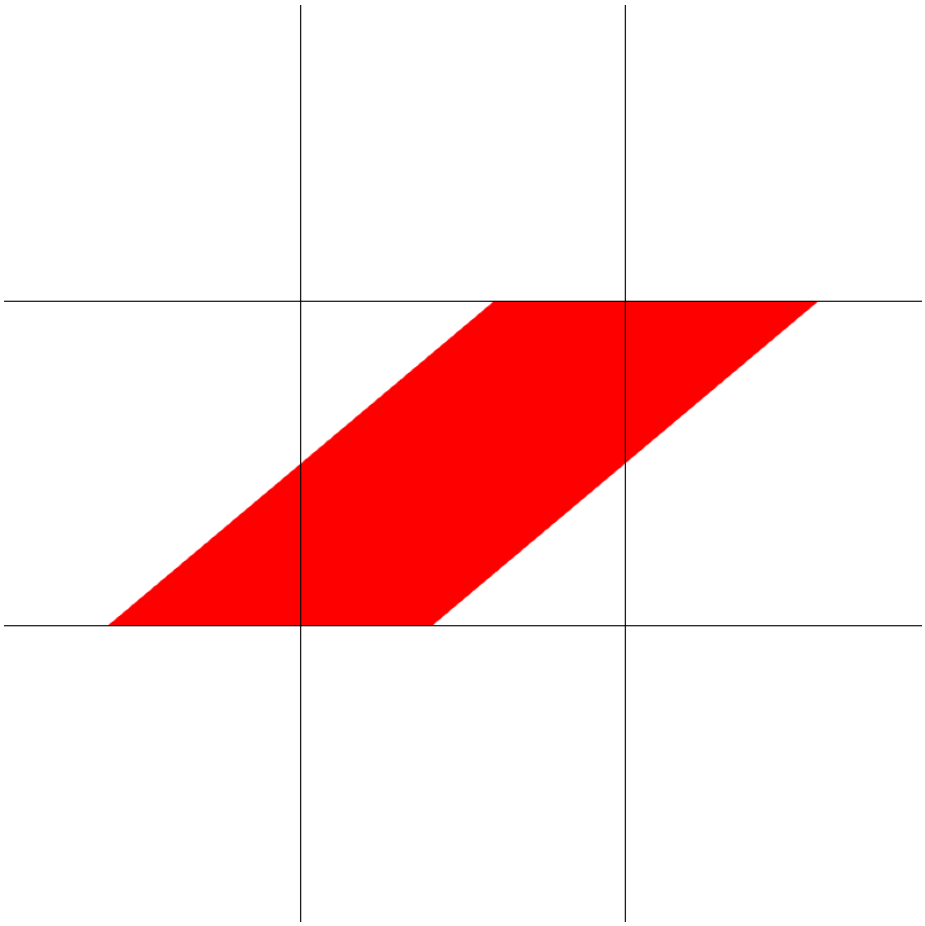

Figure 12: A4ii

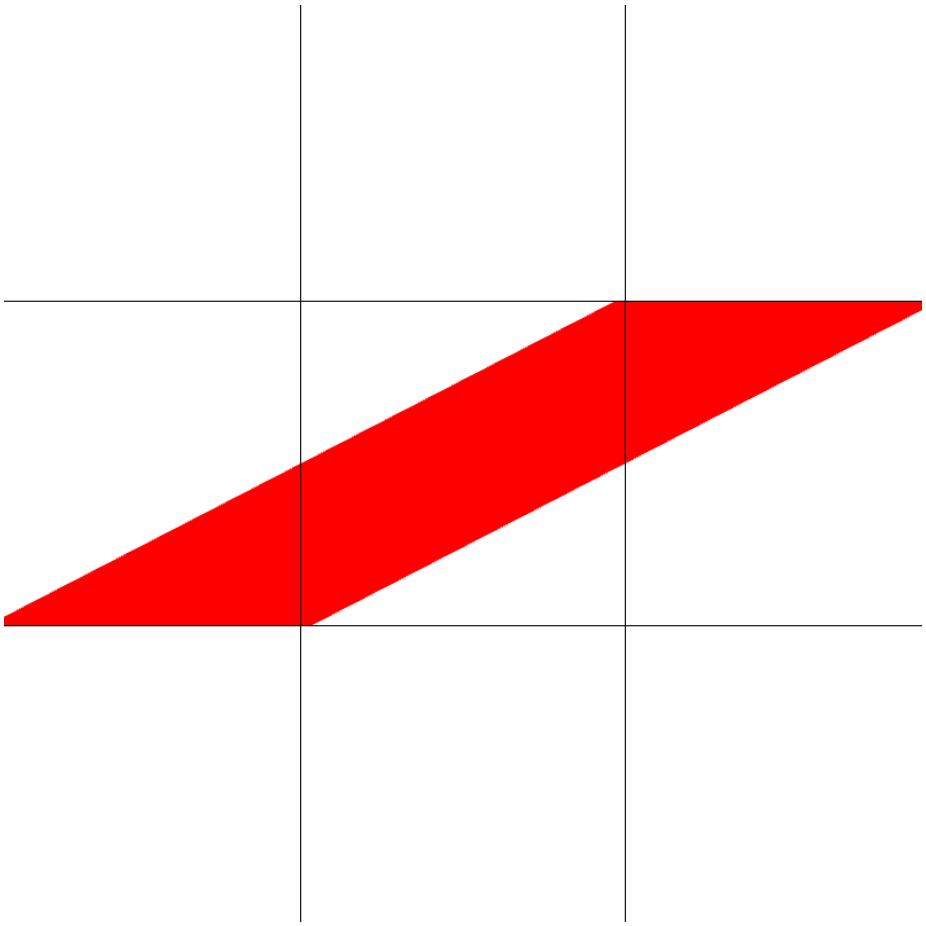

Figure 13: A4iii

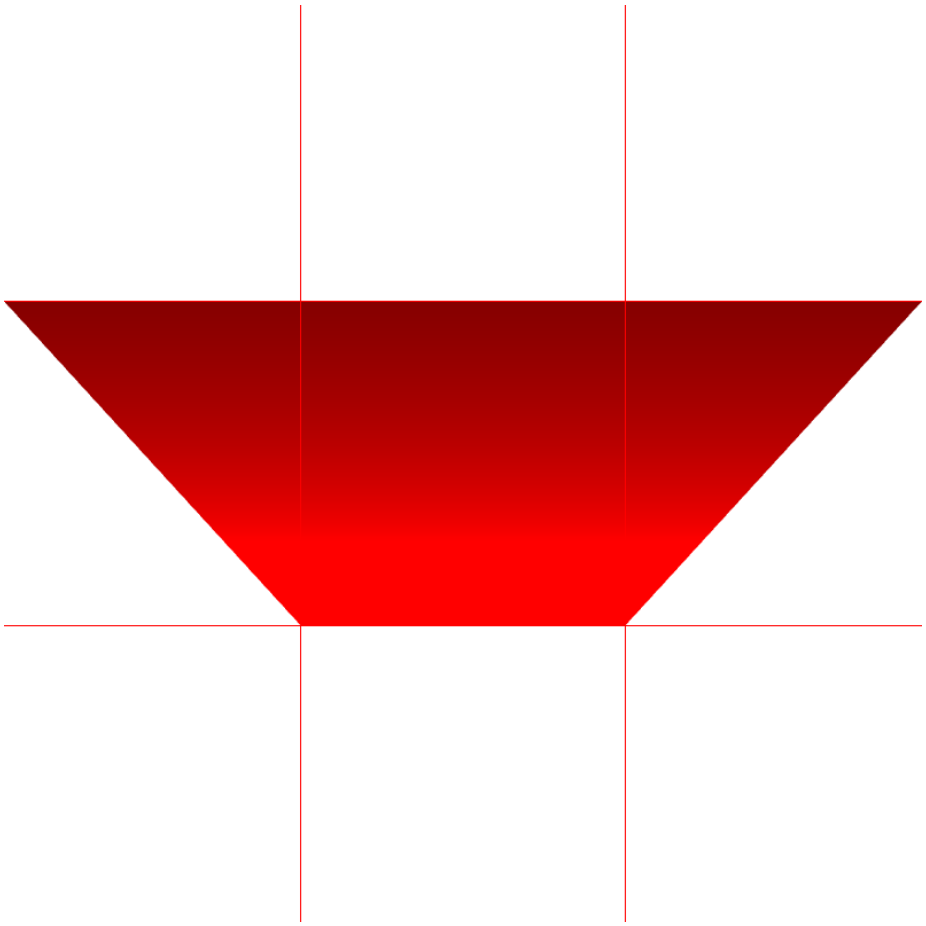

Figure 14: A5

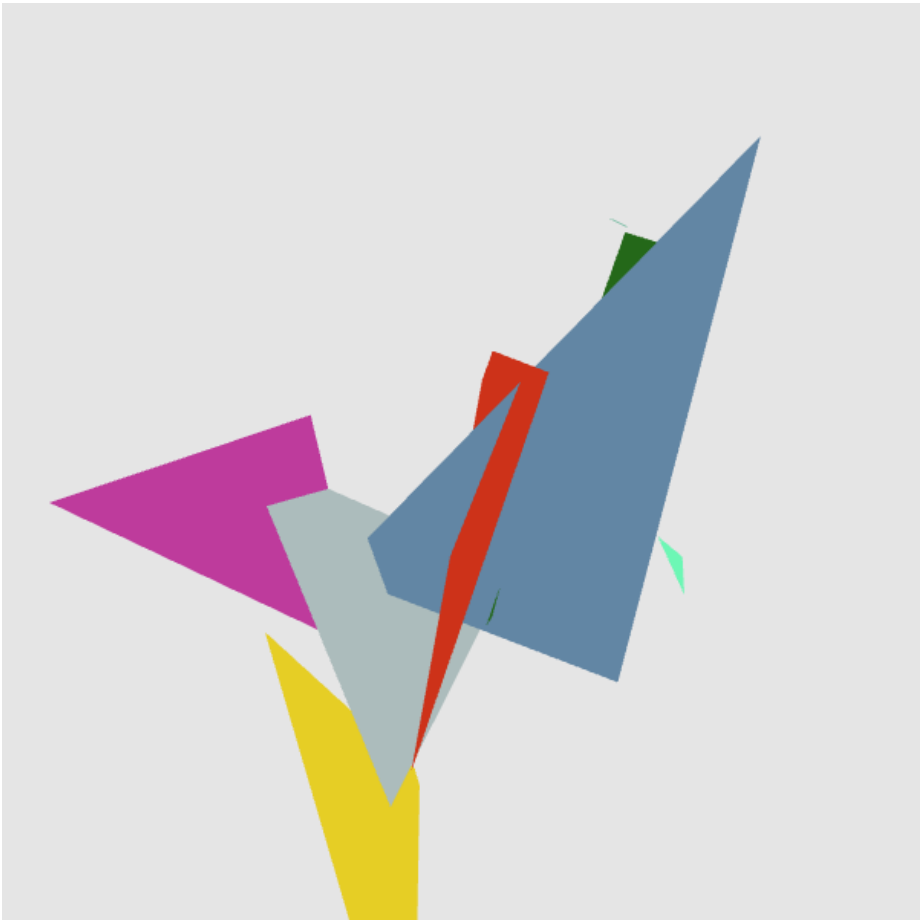

Figure 15: B1

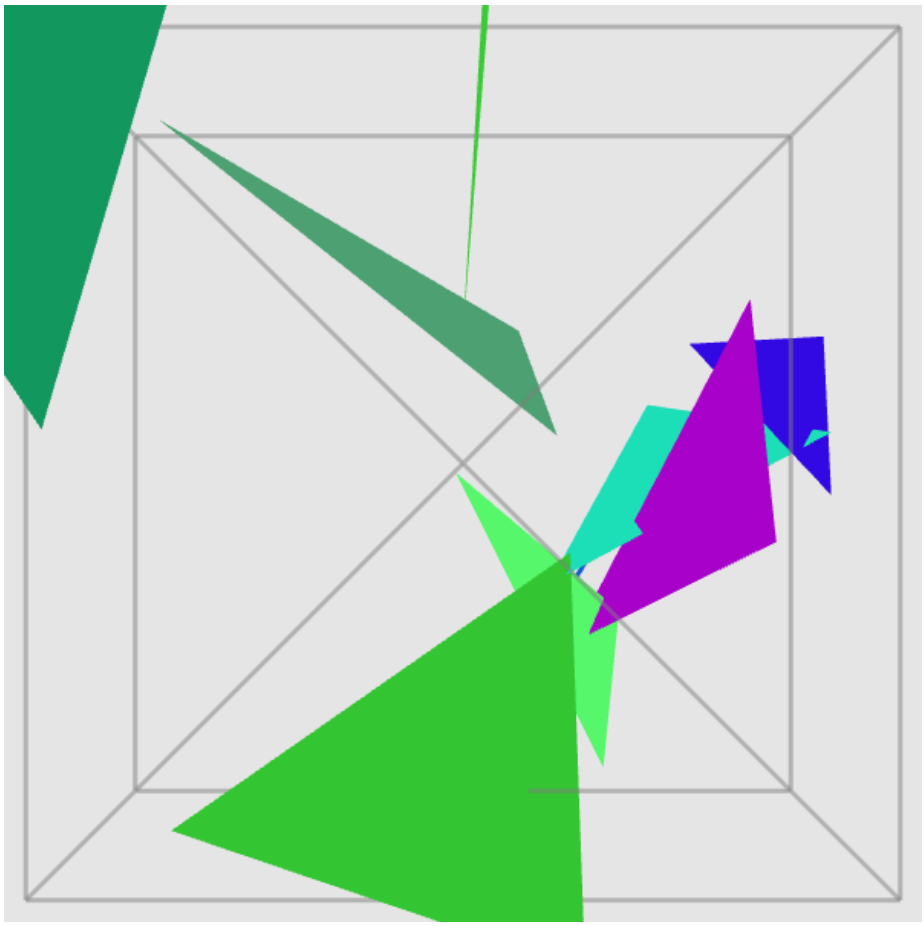

Figure 16: B2

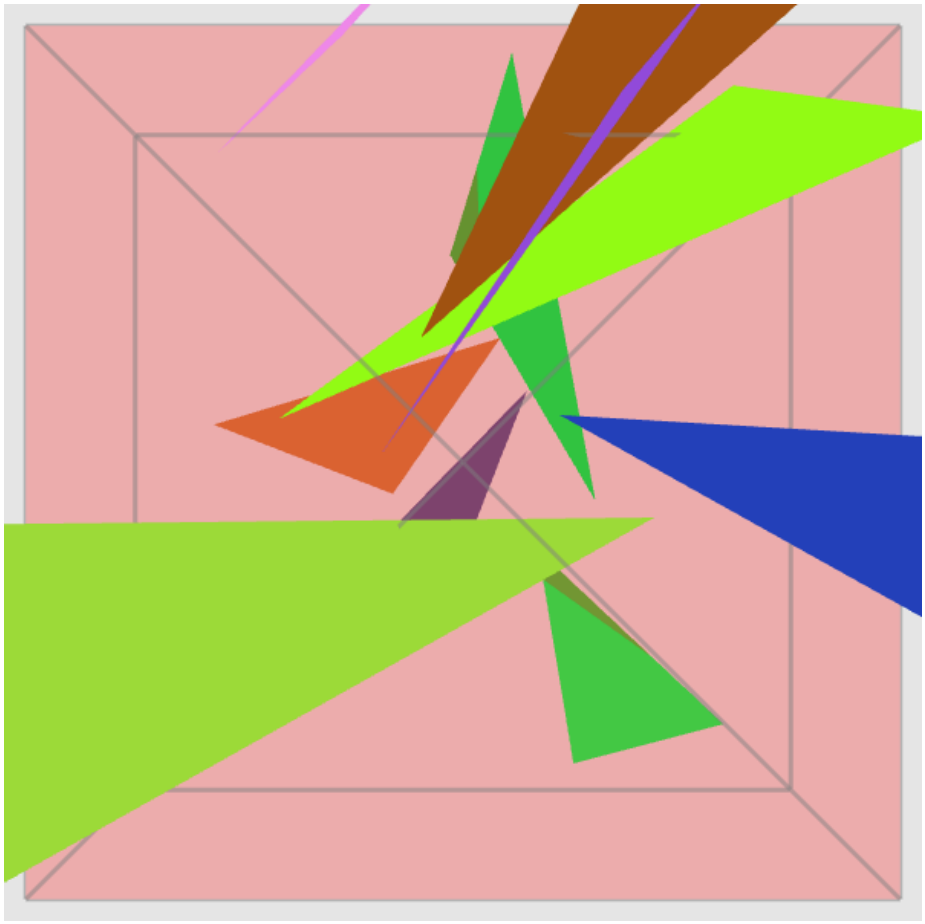

Figure 17: B3i

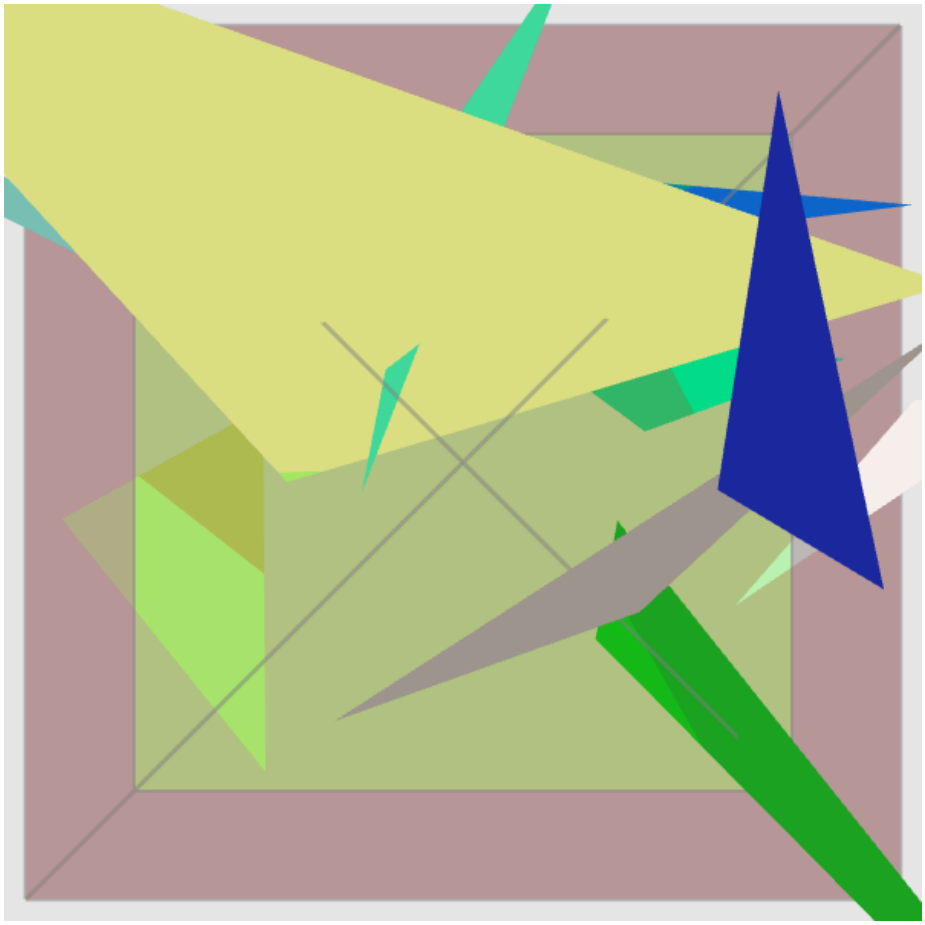

Figure 18: B3ii

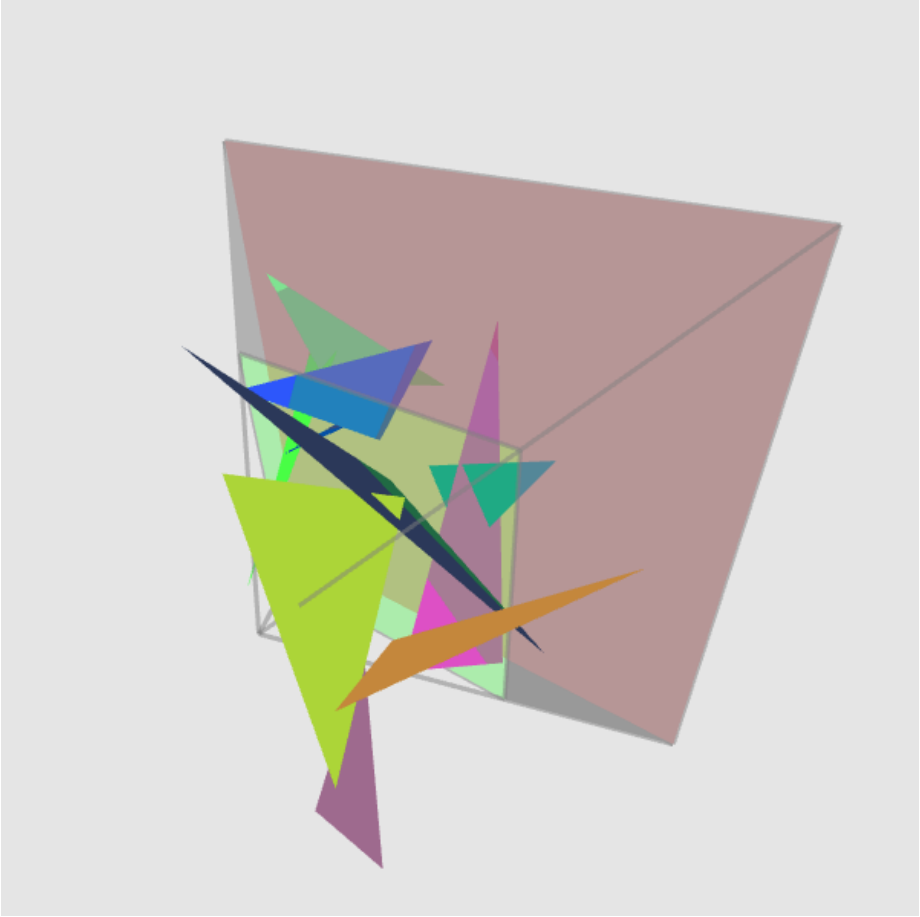

Figure 19: B4

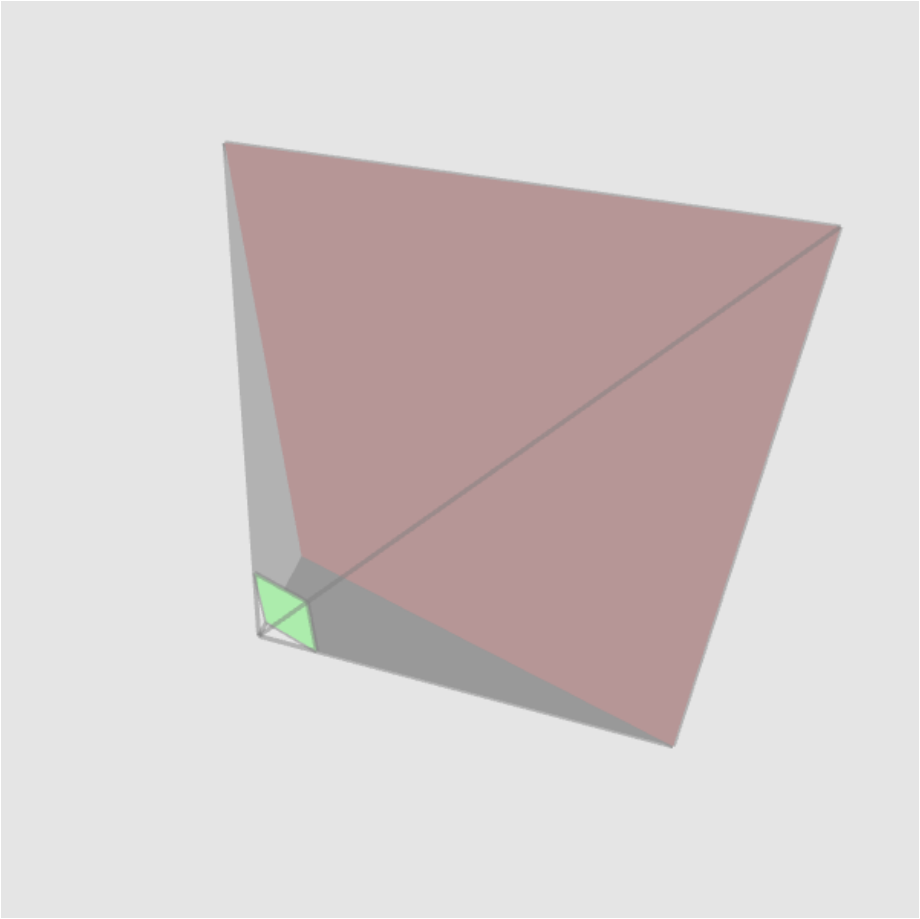

Figure 20: B5i

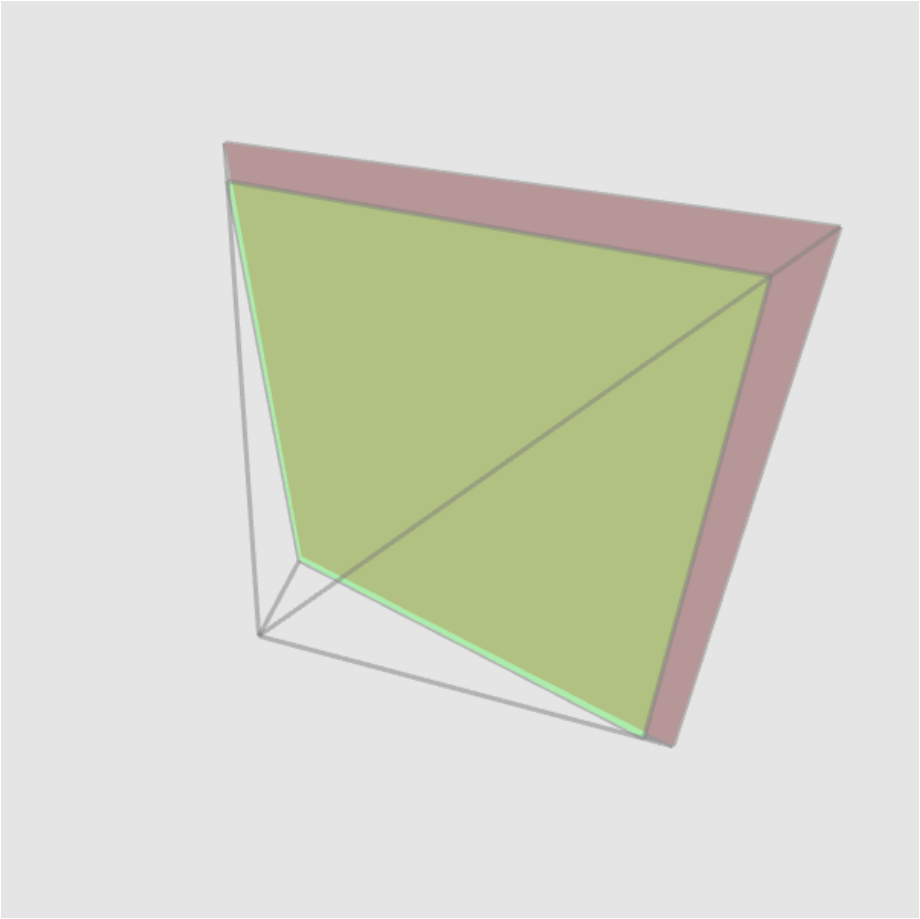

Figure 21: B5ii

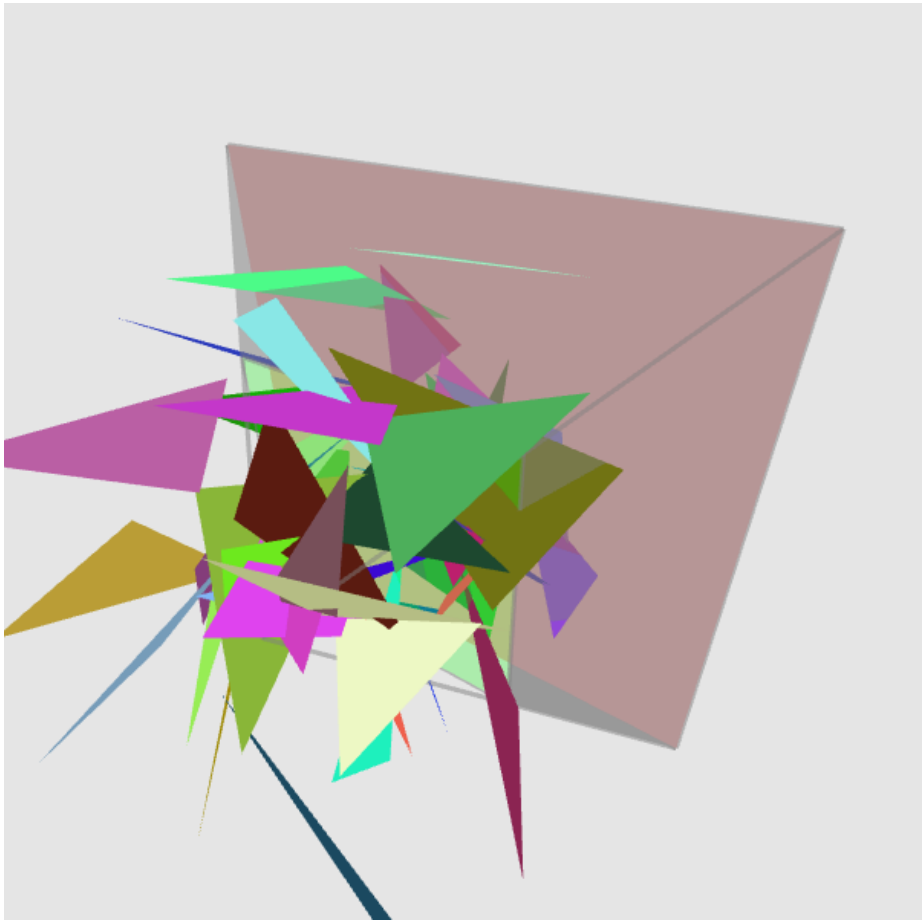

Figure 22: C1i

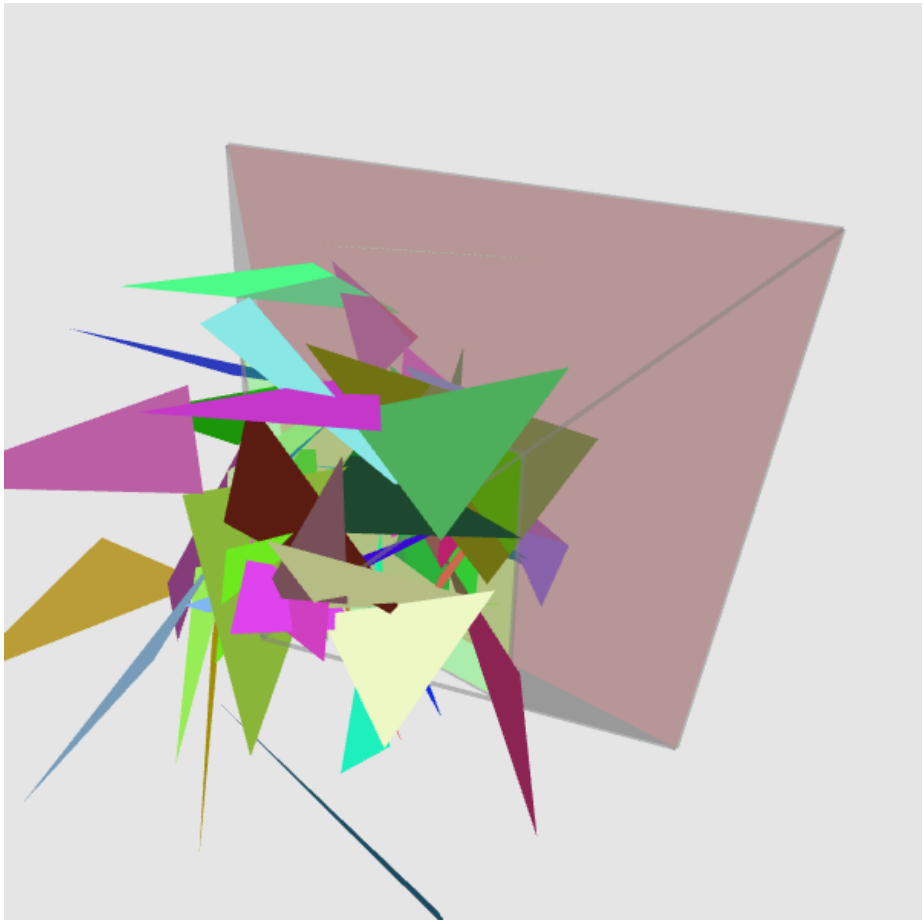

Figure 23: C1ii

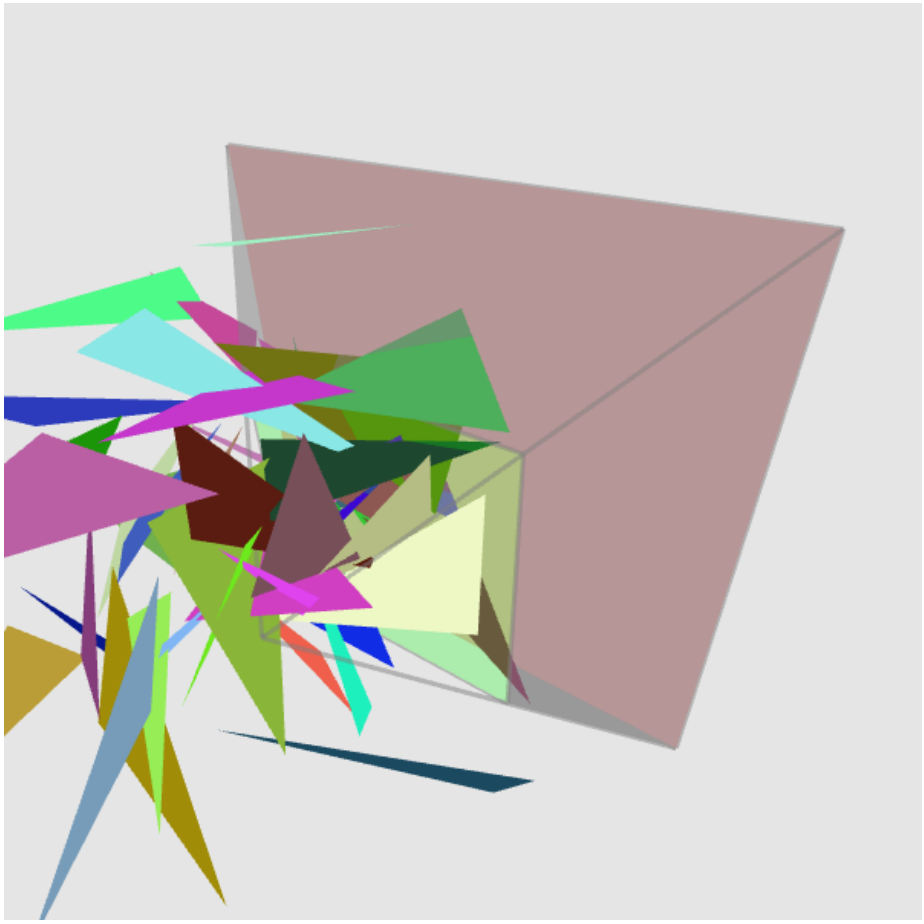

Figure 24: C1iii