Introduction to Computer Graphics CMPS 4213

In the previous assignment, you likely had to manually code all the points in GL's native coordinates, which range from -1 to 1 in all directions, to display your scene on the screen. At best, you might have created a loop to generate some of your geometry, but this can be a tedious and demanding task. In this assignment, you will use **transformations**, which allow you to be less dependent on the native coordinate system. This approach lets you work with geometry that is easier to manage and track, making the process more efficient and flexible.

Part 1: Create a Square *(30 points)*

First, to create a unit square (1x1) in 3D space, you set the z-coordinate to 0, which makes it a 2D square. If the square is centered at its origin, its corners will be:

- Top-left: (-0.5, 0.5, 0)
- Top-right: (0.5, 0.5, 0)
- Bottom-left: (-0.5, -0.5, 0)
- Bottom-right: (0.5, -0.5, 0)

When creating geometry for rendering, it's important to consider the final coordinate system used by WebGL, which operates in a space where the x, y, and z coordinates range from -1 to 1. However, this coordinate system can be decoupled from the design of your geometry. Instead of worrying about the final position of your geometry right away, you can apply transformations later.

Transformations allow you to manipulate geometry through matrices that enable scaling, rotation, and translation. For example, when designing a square, you might define the coordinates of the bottom-left corner as (-1, -1) and the upper-right corner as (1, 1). This will center the square around the origin (0, 0), which is known as **object space**.

At the time of rendering, you'll apply a series of 4x4[1] matrices (or a single composite matrix) that move the square from object space into world space, the space used by the GL-native system. These transformations allow you to position, scale, and rotate your objects as needed.

You can apply these transformations either in your application code or in the vertex shader. There are several libraries, like **glMatrix**, that assist in creating and manipulating

[1] A 4x4 matrix is used in the 3D homogeneous coordinate system. In the 2D case, the matrix works by fixing the z-coordinate and adding the w-coordinate.

transformation matrices. Each matrix corresponds to a specific transformation, such as scaling, rotating, or translating. However, you can combine multiple transformations into a single matrix, which will apply all of the desired effects at once. (Remember, the order of operations matters!)

Once you've created the transformation matrix, you have two main options:

1. **In the application program**: You can transform the points directly by multiplying them with the transformation matrix. Then, pass the transformed points to the GPU for rendering.
2. **In the vertex shader**: You can pass the original points as an attribute and the transformation matrix as a uniform. The vertex shader will then apply the transformation to the points.

Your task is to implement both methods: one that transforms the points in the application code and another that uses the vertex shader to perform the transformation.

Extensions

You can add interactivity to your program. For instance, adding buttons for moving, scaling, and rotating the square.

Changing the color of the square in each render.

Part 2: Create More Squares *(30 points)*

Transformations provide a lot of flexibility. Now, you can create multiple squares by rendering variations of your original square with different positions, orientations, scales, and colors.

To display multiple squares on the screen, you can call drawArrays multiple times. Before each call, update the transformation matrix and color to create a unique variation of the square. These values can be passed to the vertex shader using **uniform variables**—one for the transformation matrix and another for the color.

Part 3: Make your scene rain *(25 points)*

You now have many squares on the screen, and each one can be manipulated individually using a transformation. Imagine these squares as raindrops—once they appear on the canvas, they should start falling downward due to gravity. To simulate this effect, each square should move slightly downward over time. This means you'll need to track the **transformation matrix** for each individual square and update it in every frame to reflect its new position. You can achieve this animation using the requestAnimationFrame function, which allows smooth, frame-by-frame updates. Alternatively, you can trigger the movement using a button to demonstrate the falling effect. Make the squares appear as if they are falling down like rain.

Extensions:

- Add triangles, circles, or other shapes.
- Create snowflakes.
- Combine this scene with your previous campus scene to simulate rain or snow falling on it.
- Add rotation to the square as it moves.
- Add wind.
- Extend the squares to cubes.

Report *(15 points)*

The report should be done in a word processor. Either a .doc, .docx, or .pdf file will be accepted.

1. Don't forget your name!

2. Provide an estimate of the total number of hours for the entire assignment

3. Results from parts 1 to 3.

4. Write up your experience, including any problems you ran into

6. Mentions added extensions.

5. List any sources you used

6. Provide snapshots for parts 1 to 3. Name the images FirstLastLab2a.jpg, FirstLastLab2b.jpg, where First and Last are Your names. The snapshots should also be included in the report.

Submission All files (all source, your report, and your images) should be put into a single zip file and uploaded. I can handle, .zip, A zip file with your final source code for parts II and III, the report, and the images. The images need to be embedded in the report and also as separate files in the zip.