Assignment 1: Software Rasterization

In this assignment, you will implement a rasterization library in software that emulates the way 3D scenes are rendered on the GPU. Your software rasterizer should support the following features that we (will) have covered in class: triangle rasterization, supersampling, affine and perspective transformations, z-buffering, perspective-correct interpolation, and texture mapping. You are not required to implement mipmapping.

We have already provided starter code that shows how to open a window, create a framebuffer i.e. an array of pixels onto which the scene is to be drawn, and display the framebuffer on the window. Your job is to implement the remaining functionality in the graphics pipeline in order to draw geometric primitives onto the framebuffer. We will use SDL2 for window management, and GLM for vector and matrix datatypes.

PART 1

1. Implement a SoftwareRasterizer class that encapsulates the necessary functionality, including window management etc. Add methods to do the following:

    - Clear the framebuffer by setting all pixels to a given colour. The colour should be specified as a single 4-component vector glm::vec4(r, g, b, 1.0) with red, green, and blue values lying between 0 and 1. Note that SDL expects colour components to be integers between 0 and 255, so you will have to rescale them.
    - Rasterize a single triangle onto the framebuffer, given the positions of its vertices (as glm::vec4s, ignoring the *w* and *z* coordinates for now) and the colour of the triangle (as a glm::vec4). The vertex positions should lie between -1 and 1 like in normalized device coordinates, so you will have to transform them to pixel coordinates on the framebuffer yourself.

2. Add a method that can draw an arbitrary triangulated shape specified as a list of vertex coordinates along with a list of triangle vertex indices. For example, passing in the following data

    ```
    glm::vec4 vertices[] = {
        glm::vec4(-0.8,  0.0, 0.0, 1.0),
        glm::vec4(-0.4, -0.8, 0.0, 1.0),
        glm::vec4( 0.8,  0.8, 0.0, 1.0),
        glm::vec4(-0.4, -0.4, 0.0, 1.0)
    };
    glm::ivec3 indices[] = {
        glm::ivec3(0, 1, 3),
    ```

```
    glm::ivec3(1, 2, 3)
};
```

should produce a tick mark by drawing a triangle connecting vertices 0, 1, 3, and another triangle connecting vertices 1, 2, 3.

3. Modify your class to support supersampling. The user should be able to call a single method to turn on supersampling with a given n samples per pixel. If you want, you may assume that n must be a perfect square. Once supersampling is on, the user should be able to draw the scene as usual and automatically obtain anti-aliased results.

4. Add a method by which the user can set the current transformation matrix as a glm::mat4. This matrix should be applied to all the vertex positions given in subsequent draw calls. Demonstrate this functionality by writing a program that contains a single array of 4 vertices and 2 triangles representing a unit square, and draws it multiple times with different transformations to display an analogue clock with an hour hand, a minute hand, a second hand, and markings at the twelve hour locations.

PART 2

To be added: In this part, you will reorganize your code from part 1 to form a graphics pipeline closer to how real graphics APIs like OpenGL work. You will also add support for shaders.

PART 3

To be added: 3D rendering.