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 `Rasterizer` 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::vec4`s, 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. On each frame, the clock should show the current time.

PART 2

I have posted the code framework for this assignment at https://git.iitd.ac.in/col781-2202/a1. I suggest you clone it using Git instead of simply downloading it, as I am likely to update it in the near future.

This code defines, in the file `src/api.hpp`, a simple rasterization API for drawing objects made of triangles and displaying them in a window on the screen. You can look at `examples/e1.cpp` for an example of how this API can be used, and read the comments in `src/api.hpp` for further details.

The API is implemented by two different classes. One named `COL781::Hardware::Rasterizer` is defined in `src/hw.?pp`; it performs hardware rasterization on the GPU using OpenGL. If you compile the project right now, the executable `e1` produced by `examples/e1.cpp` will use this class to draw the tick mark described above.

The other implementation of the API is named `COL781::Software::Rasterizer` and is declared in `src/sw.hpp`; it is intended to perform software rasterization on the CPU. However, its implementation in `src/sw.cpp` is incomplete, so it cannot be used right now!

1. Your first task is to complete the implementation of the software rasterizer by adding your own code to `src/sw.cpp`. Once your implementation is complete, you should be

able to comment/uncomment two lines in `examples/e1.cpp` to make it use your software rasterizer instead of the hardware one, and still get (approximately) the same result. Note that the software rasterizer class uses different types than the hardware rasterizer, but the function calls are the same, so the same example code can use either implementation.

2. You should also port your clock program to this API. Create a new `.cpp` file under `examples/` which draws the clock using our rasterization API, and add it to `CMakeLists.txt`.

We have included implementations of some helper classes to represent vertex attributes and uniform variables (see lecture 10).

- You are permitted to modify these classes (`Attribs` and `Uniforms`) to add more methods if you need. However, you should not change their purpose: the `Attribs` class should represent the attributes of *one* vertex or *one* fragment, not the entire array of attributes for all vertices.
- Do not modify the types of the vertex shader and fragment shader; these inputs and outputs are what real shaders on the GPU have access to. Do, however, refer to the implementations of the provided built-in shaders (`vsIdentity`, `vsTransform`, and `fsConstant`) to understand how they work.
- Do not modify the example application `examples/e1.cpp` (nor any other examples we provide in the future).

Part 3

To be added: 3D rendering.