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 have covered in class: triangle rasterization, supersampling, affine and perspective transformations, z-buffering, and perspective-correct interpolation.

Background

We will use SDL2 for window management, and GLM for vector and matrix datatypes. GLM provides convenient vector and matrix operations following GLSL syntax. Go through the linked page and learn the operations, so that you can use them instead of re-implementing everything like vector addition, dot products etc. yourself. Generally all the GLSL operations on the linked page are supported in GLM, except for precision qualifiers and "swizzling".

The code framework for this assignment is posted at https://git.iitd.ac.in/col781-2302/a1. I suggest you clone it using Git instead of simply downloading it, in case I update it in the future.

This code framework defines, in the file `src/api.inc`, 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.inc` 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 in class.

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!

Tasks

Your main job in this assignment is to add your own code to `src/sw.cpp` to make all the provided examples work with your software rasterizer. I suggest you do this incrementally, implementing enough methods to make `examples/e1.cpp` work, then do `examples/e2.cpp`, and so on. You can change the rasterizer used by each example by commenting/uncommenting the two `namespace R = ...` lines. 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.

As you modify the software rasterizer implementation, please keep a few things in mind:

- 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` , `fsConstant` , etc.) to understand how they work.
- Do not modify any of the example applications `examples/e*.cpp` .

Here are some further details on how to proceed:

1. Examples `e1` and `e2` just require you to implement a `clear` method that clears the framebuffer by setting all pixels to a given colour, a triangle rasterization procedure that draws a single triangle onto the framebuffer, and a `drawObject` method that draws an arbitrary triangulated shape specified via vertex coordinates and triangle vertex indices. Since this example is 2D, you can ignore the *w* and *z* coordinates for now.
    - In our API, colour components (red, green, and blue) are specified with values lying between 0 and 1. However, SDL expects the components to be integers between 0 and 255, so you will have to rescale them.
2. At this point you should also add support for supersampling, specified through the last (optional) parameter of the `initialize` method. 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. Verify this by modifying the `initialize` calls in `e1` and `e2` and checking that the triangle edges look smoother.

From `e3` onwards, you will have to add support for rendering 3D triangles. The modeling, viewing, and projection transformations will already be provided by the user and applied by the vertex shader. Your job is to do perspective division, perspective-correct interpolation, and depth testing.

3. Include a perspective division stage after the vertex shader, which maps a `glm::vec4(x, y, z, w)` to a `glm::vec3(x/w, y/w, z/w)` . This should not change the results of any of the 2D examples since their *w* is always 1, but it should make example `e5` look like a rotating 3D square.
4. Add a z-buffer to your rasterizer so that you can perform depth testing. This can just be an array of floats that you maintain yourself. Z-buffer testing should only be enabled once `enableDepthTest()` is called, to avoid changing the 2D examples. Then examples `e3` and `e4` should show correct visibility like in lecture 7. When clearing the screen, remember to clear the z-buffer as well. Also verify that this continues to work when supersampling is enabled.
5. Make sure the attribute interpolation you perform in the rasterization step is perspective-correct, so that the gradient from blue to yellow in example `e5` is drawn correctly. It

should not look like the distorted gradient in lecture 8.

With this, you should be able to run all five of the provided examples with your software rasterizer, and get results comparable to those produced by the hardware.

A secondary goal in this assignment is to use this API to draw some interesting scenes. For each scene, create a new `.cpp` file under `examples/` and add it to `CMakeLists.txt`.

6. Write a program that uses affine transformations to draw an analogue clock showing the current time. For full marks, the only mesh object your code contains should be a unit square having 4 vertices and 2 triangles. By applying various scaling, rotation, and translation operations, you can stretch it into a rectangle representing the hour hand, and similarly the minute hand, the second hand, and markings at the twelve hour locations (something like this picture, ignoring the circles). On each frame, the clock hands should point according to the current time.
7. Design an interesting 3D scene on your own, by creating a new program similar to the given examples. Your program should work with both the software and hardware backends. Feel free to make the scene as complicated as you like; it should contain at least two *different* meshes (not just two transformed copies of the same mesh), a perspective camera, and animated motion of at least one mesh or the camera. You could design the mesh geometry by sketching it on graph paper, or use any formulas of geometrical shapes like spheres, cones, tori, etc. that you know of.
   - Consider giving the vertices varying colours so that the shape can be seen better instead of being completely flat.
   - Try to avoid having any geometry move behind the camera; this may create rendering bugs since you have not been asked to implement clipping.
   - There will be a small amount of marks for how aesthetically pleasing your scene is.

Submission

Submit your assignment on Moodle before midnight on the due date. Your submission should be a zip file that contains your entire source code for the assignment, not including any binary object files or executables. The zip file should also contain a PDF report that includes the following:

- screenshots of all five example programs as drawn by your software rasterizer,
- screenshots of the two programs you had to write yourself, i.e. the clock and the creative 3D scene, and
- explanation of any components of the assignment you could not fully implement and why.

Separately, each of you should *individually* submit a short response in a separate form regarding how much you and your groupmate(s) contributed to the work in this assignment.