Name: **SID:**

CS130 - LAB 5 - Getting Started with Project 2

Project 2 consists of implementing a simplified rendering pipeline.
The source code can be found at https://www.cs.ucr.edu/~shinar/courses/cs130/proj-gl.html.

For this lab, you'll need to get the first test working (test 00.txt), which include (1) allocating memory for the image to be rendered, (2) assigning values to (3) vertices and rendering triangles. Let's get started by checking in which file you will need to implement the first steps and implementing (1), (2) and (3) on each step of this tutorial.

There are 5 cpp files in the project, you will need to implement your code only in *driver_state.cpp*. In *driver_state.cpp*, there are four empty functions with TODOs in them. What are these functions and what they do?

Function Name	Brief Description

PART (1): We will only work in 3 of these functions in this lab. Let's start with *initialize_render*. We need to allocate the memory space for the *color_image* and for the *depth_image*. Note that *color_image* is a pointer. Look at *driver_state.h* and in *common.h*, you will notice that color_image is a typedef for another type.

What is the typedef name of *color_image?*

What is the actual type of *color_image*?

Look at *make_pixel* in *common.h*. In which order is the RGB color information stored in a single pixel in *color_image*?

How many bytes each channel (red, green, blue) are used in a single pixel?

How can we set a pixel with the color white?

Implement *initialize_render* in *driver_state.cpp* by allocating the memory for *color_image*. Initialize all the pixels in *color_image* to **black**. We will not be using *depth_image* until we implement z-buffer, so you can ignore it for now. Make sure your code compiles and run without issues on **valgrind**. You can compile the code using **scons** and you can run on test 00.txt using **./driver -i tests/00.txt**.

PART (2): Next step we need to implement a few things in render. There are two parameters here, what these two parameters do?

Parameter	Brief description
driver_state &state	
render_type type	

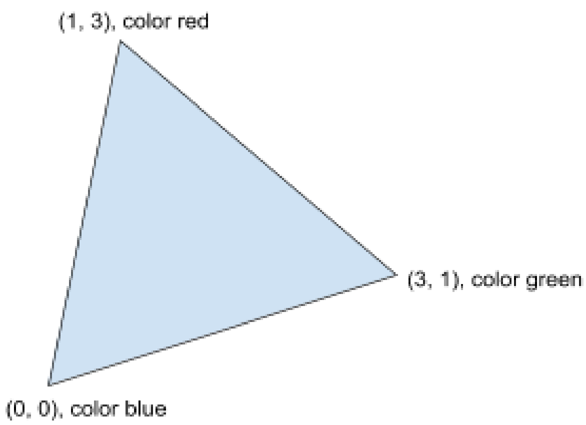
Start by creating a switch for the render type. There should be four types, let's focus only on triangle for now and leave the other 3 cases empty. In the triangle case, we need to prepare a *data_geometry* array of size 3 (one for each vertex of the triangle) and call rasterize triangle using this array. Let's first take a look at *data_geometry* defined in *common.h*. What are the two fields that we need to set on each *data_geometry* object?

data_geometry	Brief description

We will not set the position of *data_geometry*. This will be the responsibility of the vertex shader. Instead, we will copy the data we have from each vertex of the triangle into the *data_geometry* and then call the vertex shader. Ok, let's see how we get this data in the *render* triangle case. Look at *driver_state.h* and try to find where the data for each vertex is stored. There should be 3 relevant variables. What are they?

__________________________, __________________________ and __________________________

Consider we have the following triangle. Write on the right side of the triangle what would be the values in the three variables for this triangle.



Write the code in *render* for the triangle case to read every 3 vertices in *driver_state* into a the *data_geometry* array and call *rasterize_triangle*.

PART (3): Final step is to rasterize the triangle from part (2) in the *rasterize_triangle* function.

- ❏ Start by passing each vertex in *data_geometry* to the vertex shader (see function in driver_state.h).
- ❏ Recall we are using homogeneous coordinates, so you will need to divide the position in the *data_geometry* by w.
- ❏ Calculate the pixel coordinates of the resulting *data_geomtry* position. In particular, the *data_geomtry* x and y positions should be in Normalized Device Coordinates (NDC) with each dimension going from -1 to +1. You will need to transform x from 0 to width and y from 0 to height. You will also need to account for the fact that the NDC (-1, -1) corresponds to the bottom left corner of the screen but not the center of the bottom left pixel. Given (x, y) in NDC, what equation gives you (i, j) in pixel space (use w to denote width and h to denote height).

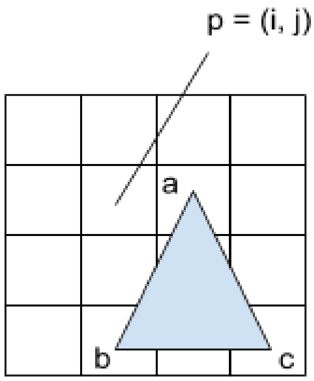
  i = ____

  j = ____

- ❏ Draw the vertices in the image (recall you can access the *image_color* in *driver_state*). Make sure they fall on the vertices of the 00.png image. You will have the (i, j) position of the pixel but you need to set a specific pixel in the width*height *color_image* of *driver_state*. How do you calculate the corresponding index in color_image using (i, j)?

  image_index =

- ❏ To rasterize the triangle, you can iterate over all pixels of the image. Say you are in the pixel with indices (i, j). You can use the barycentric coordinates of this pixel (i, j) to know if this pixel falls inside the triangle or not.



Barycentric coordinates can be calculated using triangle areas. Fill out the equations for the barycentric coordinates below.

α = AREA(_______) / AREA(abc)

β = AREA(_______) / AREA(abc)

γ = AREA(_______) / AREA(abc)

You can calculate the area of the triangle using the formula:

$$AREA(abc) = 0.5 * ((b_x c_y - c_x b_y) - (a_x c_y - c_x a_y) + (a_x b_y - b_x a_y))$$

- ❏ If all barycentric coordinates are >= 0, then make the pixel color white. You should be passing test 00.txt now. Make sure you don't have any errors on valgrind.

Other things to do

- ❏ Rather than visiting all the pixels of the image for every triangle, visit only the pixels in the square that contains the triangle. For this, calculate the maximum and minimum x and y coordinates and make your for loops iterate over the [minimum, maximum] range.
- ❏ Use the fragment shader to calculate the pixel color rather than setting to white. See *data_output* in *common.h* and the *fragment_shader* function in *driver_state.h.*
- ❏ Implement color interpolation by checking *interp_rules* in *driver_state* before sending the color to the fragment_shader. You have one interp_rule for each float in the *data_geometry.data*. If the rule type is *noperspective* (see interpolation types in *common.h*), then interpolate the float from the 3 vertices using the barycentric coordinates.