CS130 - LAB - Getting Started with Project 2

Name: _____ SID: _____

Project 2 consists of implementing a simplified rendering pipeline.

For this lab, you'll need to get the first test working (test `00.txt`), which includes (1) allocating memory for the image to be rendered, (2) assigning values to vertices and (3) 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`. (You should not modify any of the other cpp files.) In `driver_state.cpp`, there are four functions with TODOs in them. What are these functions and what should they do?

Name 1: _____
Description: ■

Name 2: _____
Description: ■

Name 3: _____
Description: ■

Name 4: _____
Description: ■

1 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 `image_color` and for the `image_depth`. Note that `image_color` is a pointer. Look at `driver_state.h` and in `common.h`, you will notice that `image_color` is a `typedef` for another type.

What is the typedef name of `image_color`? _____

What is the actual type of `image_color`? _____

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

How many bytes for 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 memory for `image_color`. Initialize all the pixels in `image_color` to black. We will not be using `image_depth` until we implement the z-buffer, so you can ignore it for now. Make sure your code compiles and runs without issues in valgrind. You can compile the code using `scons`, and you can run test `00.txt` using `./driver -i 00.txt`.

2 Part 2

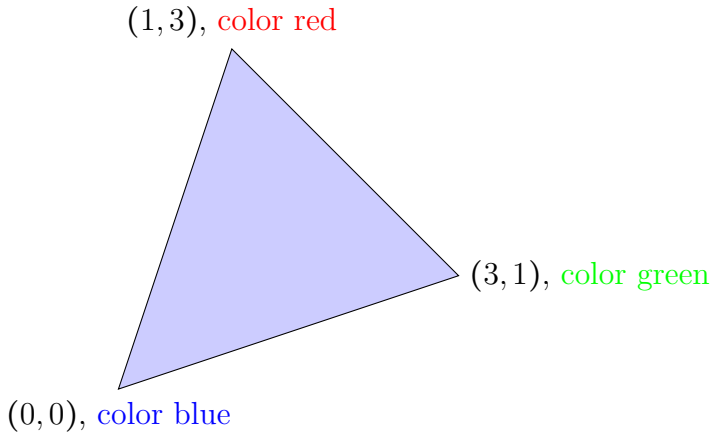
For the next step, we need to implement a few things in render. There are two parameters here, what these two parameters do?

`driver_state &state` description: ■

`render_type type` description: ■

We will need to locate the vertex data and understand how it is being stored in memory. 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:



What are the values in the three variables for this triangle? (The data layout is up to the user. For now, choose a reasonable layout. This is just to help us understand what the input looks like.)

Variable 1: _____
Value 1: _____

Variable 2:_____
Value 2: _____

Variable 3:_____
Value 3: _____

We will be using two new structures: `data_vertex` and `data_geometry`. These are defined in `common.h`. What are the two fields in `data_geometry` object?

Field 1: _____
Description: ■

Field 2: _____
Description: ■

What is the only field in `data_vertex`?

Field 1: _____
Description: ■

We will implement this function in two steps. In the first step, we will allocate an array of `data_geometry` objects (one for each vertex). Note that each of these objects has a `data` member, which you will also need to allocate. These pointers need to point to freshly allocated space. Leaving these pointers uninitialized is a common bug. Initializing these pointers to other arrays that already exist is also a common bug. *(General piece of programming advice. If you see a pointer and don't know what to do with it, the two worst things that you can do are (1) leave it uninitialized and (2) point it at something you see lying around. Option (1) makes your code segfault. Option (2) leads to perplexing bugs that take several hours of time and a trip to office hours to track down. Reading the comments helps a lot with this. The comments are there to help you. Spending some time at the beginning reading and digesting the comments can save you many hours of frustration later on.)*

Once all of the the space is fully allocated and the pointers set to valid data, we need to fill in the contents of the objects. *This is done by calling the vertex shader. Do not attempt to do this yourself.* You do not have enough information available to you do do this; the vertex shader must do it for you. Note that the input for the vertex shader is a `data_vertex` structure, which also contains a `data` pointer. This pointer should point at the vertex data for that vertex. It should not make a copy of that data. Once this has been done, we will have one `data_geometry` object for each vertex.

The next step is to group the vertices into triangles and pass the triangles to the next stage of the pipeline. Start by creating a switch for the render type. There four four render types. Let's focus only on `triangle` for now and leave the other 3 cases empty. Each set of three `data_geometry` objects corresponds to a triangle. Pass these on to the next stage of the pipeline, which for now will be rasterization.

If there are 9 vertices, how many triangles will be rendered? _____

If there are 8 vertices, how many triangles will be rendered? _____

3 Part 3

The final step (for today) is to rasterize the triangles from part (2) in the `rasterize_triangle` function.

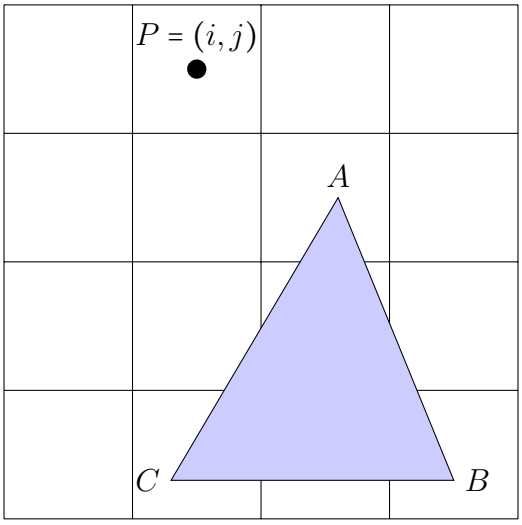
- Before you call the `rasterize_triangle` function, make sure that the `data_geometry` objects have been prepared using the vertex shader (see the function description in `driver_state.h`).
- In the `rasterize_triangle` function, recall that we are using homogeneous coordinates, so you will need to divide the position in the `data_geometry` by the w coordinate.
- Calculate the pixel coordinates of the resulting `data_geometry` position. The x and y positions in `data_geometry` are in Normalized Device Coordinates (NDC) with each dimension going from -1 to +1. Pixels on the other hand are indexed from `0` to `image_width-1` for x or from `0` to `image_height-1` for y. You will also need to account for the fact that the NDC $(-1, -1)$ corresponds to the bottom left corner of the screen but pixel index $(0, 0)$ corresponds to the center of the bottom left pixel. Given (x, y) in NDC, what equation gives you (i, j) in pixel space (use `image_width` and `image_height`).

  i = _____

  j = _____

- Draw the vertices in the image (recall you can access the `image_color` in `driver_state`). Make sure they fall very close to 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 array `image_color` of size `image_width*image_height` in `driver_state`. How do you calculate the corresponding index in `image_color` using (i, j)? `index` = _____. Note that we are drawing the vertices only for debugging to test earlier pipeline stages as well as your transformations. Once these things have been fixed, this should be removed.

- 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 test 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(_____)
  β = AREA(_____)/AREA(_____)
  γ = AREA(_____)/AREA(_____)
  You can calculate the area of the triangle using the formula:
  $\text{AREA}(ABC) = 0.5((B_xC_y - C_xB_y) + (C_xA_y - A_xC_y) + (A_xB_y - B_xA_y))$

- If all barycentric coordinates are non-negative, then make the pixel color white (for now). You should be passing test `00.txt` now. *Make sure you don't have any errors on valgrind. This simple step can save you many hours of frustration later.*

Other things to do later:

- Rather than visiting all the pixels of the image for every triangle, visit only the pixels in the square that contains the triangle.

- 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 `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.