CS148 Assignment 3

2D Rasterization and Transforms

Goals: *Implement basic 2D rasterization, with support for barycentric color interpolation and transformations.*

This week, you will create simple 2D line and triangle art from scratch. In Part 1, you will rasterize lines using Bresenham's algorithm. In Part 2, you will rasterize triangles. In Part 3, you will add support for varying color across triangles. Finally, in Part 4, you will implement basic transformations on your primitives.

Starter code

The starter code[1] contains the following helpful utilities:

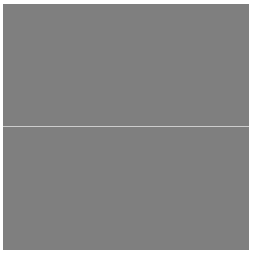
- `SimpleImage.h/.cpp` : A simple image utility. Use this to export your images in `.png` format.
- `io.cpp` : Code to read in `.pline` and `.tri` files.
- `raster2d.h/.cpp` : skeleton code for the main program.
- `util.h` : A few miscellaneous functions you may find useful.

The `SimpleImage` class requires the installation of `libPNG`[2]. The starter code contains a README with more information on installing `libPNG`. If you need additional platform-specific installation instructions or tips, please check Piazza.

The SimpleImage class is an easy-to-use utility for the import and export of PNG files[3]. For instance, the following lines of code initialize a SimpleImage, `I`, from an image `foo.png`. Then, the lower left corner pixel is changed to red, and the altered image is written out to `bar.png`:

```
SimpleImage I;
I.LoadPNG("foo.png");
I.setPixel(0,0,RGBColor(1,0,0));
I.WritePNG("bar.png");
```

The RGBColor of a pixel at location (x,y) can be read with parenthetical accessors, i.e. `I(x,y)`.



The starter version of `raster2d.h/cpp` contains a simple `draw` method that will produce a gray image with a white horizontal line (inset). To verify that you have everything working, compile and run the following command:

```
./raster2d input/rainbow_square.tri temp.png 400 400
```

For now, this command will ignore the input file (*.tri or *.pline, see below). It instead generates an output PNG image file `temp.png` which is of size 400x400 pixels and should match the inset figure.

[1] https://www.stanford.edu/class/cs148/assets/hws/hw3_starter.zip

[2] http://www.libpng.org/pub/png/libpng.html

[3] The SimpleImage class is for your convenience only. If for any reason you have trouble installing libpng, you may use your own image class. For instance, the PPM file format is particularly straightforward.

The general usage is:

```
./raster2d [input_file] [output_file] [x_res] [y_res]
```

Input File Format

The input file will be in one of two formats: polylines or colored triangles. A polyline (`.pline`) file is used to draw images comprised of line segments, and looks like this:

```
-1 1 -1 1
polyline
   -1.0 -1.0
    1.0 -1.0
    1.0 1.0
   -1.0 1.0
   -1.0 -1.0
polyline
    1.  0.
    -0.8 0.
    0.3 0.9
    0.3 -0.9
    -0.8 0.5
    1.  0.
```

The first line of the file contains values [`x-min x-max y-min y-max`] defining the boundary of the rectangular region being drawn to. Each polyline keyword starts a new line and each subsequent pair of real numbers indicates an (x,y) position. When drawing a polyline, the program should start at the first location and then draw lines connecting the dots through the rest. The program should not return to the first point: if a closed curve is desired, the first point will be repeated at the end.

A colored triangle (`.tri`) file is used to draw a collection of colored triangles. It looks like this:

```
-1 1 -1 1
p -0.500000 0.500000
p 0.500000 0.500000
p 0.500000 -0.500000
p -0.500000 -0.500000
c 0 .63 .19 .13
c 1 .3 .46 .74
c 2 .63 .19 .13
c 3 .96 .81 0
t 0 1 3
t 2 3 1
```

As above, the first line of the file indicates the bounding box. Lines beginning with `p` contain the (x,y) coordinates of triangle vertices. Vertex colors `c` are associated with vertices via an integer index followed by R,G,B values in [0,1]. Finally, triangles `t` are specified by triplets of vertex indices. For instance, in the above file, four vertices indexed 0...3 are specified, assigned colors, and finally used to form two triangles, one connecting vertices 0, 1, and 3 and the other connecting vertices 2, 3, and 1.

The code necessary to parse these files can be found in `io.cpp`. Sample input files can be found in the `input/` directory, or you are welcome to generate your own.

1 2D Line Rasterization

The starter code in the `main()` function of `raster2d.cpp` contains a `Raster2d` object named `raster_image`. After loading an input `.pline` file, this will contain a member variable `m_lines`, which is an STL vector of line segments. Use Bresenham's algorithm[4] to complete the line drawing method in `draw.cpp`, so that all segments of the polyline file are rendered correctly. You may use any color line you prefer.

Example usage (this command should produce the image shown in Figure 1(a)):

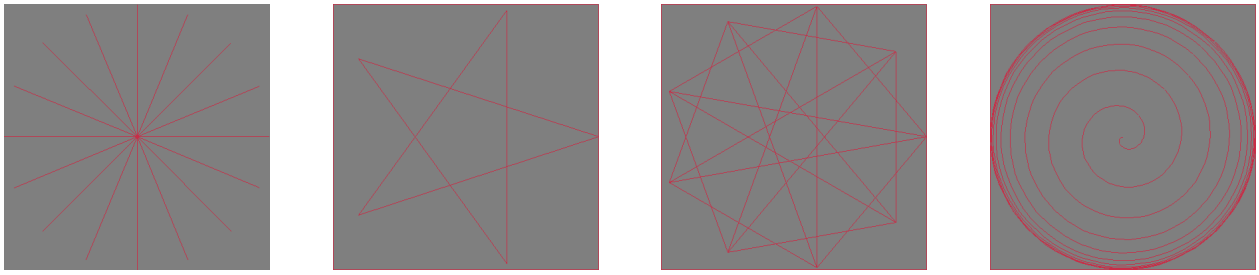
```
./raster2d input/binaryclock.pline binaryclock.png 400 400
```

Question 1: Please turn in your own rendering of each example `.pline` file shown in Figure 1.

2 2D Triangle Rasterization

As in the previous section, after loading an input `.tri` file, `raster_image` will contain a member variable `m_triangles` that is an STL vector of triangles. Ignoring vertex colors at first, rasterize the triangles by coloring pixels inside the bounding vertices any color you prefer.

[4] http://en.wikipedia.org/wiki/Bresenham's_line_algorithm



(a) (b) (c) (d)

Figure 1: *Reference .pline files: (a) binaryclock (b) starbox (c) nonagons (d) spiralbox*

Example usage:

```
./raster2d input/rainbow_square.tri rainbow_square.png 400 400
```

Hint: Using barycentric coordinates (Shirley Ch. 2.7) to check if a pixel is inside a triangle will make the next part trivial!

3 Barycentric Color Interpolation

Now, extend your triangle rasterization to smoothly blend the vertex colors across triangle faces using barycentric interpolation. Colors for triangle vertices (e.g., the vertex `a`) can be accessed for any triangle, `tri`, via `tri.a_color`.

Question 2: Please turn in your own rendering of each example `.tri` file shown in Figure 3.

Question 3: In Figure 2, you will see two squares, each comprised of two triangles, whose vertices are identically colored. However, color interpolation yields different results. Only two lines of the corresponding `.tri` files differ. What is different?

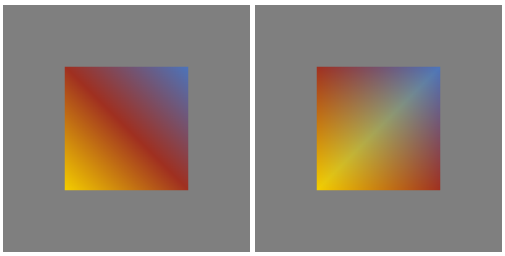
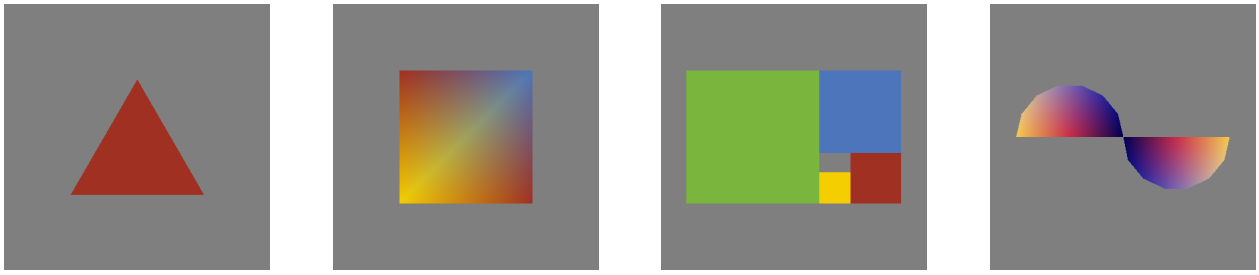


Figure 2: *The corresponding vertices in each square have the same colors assigned.*



(a) (b) (c) (d)

Figure 3: *Reference .tri files: (a) red triangle (b) rainbow square (c) golden ratio rectangles (d) triangle fan*

4 2D Transforms

The final feature you will be adding to your rasterizer is the ability to transform primitives by rotation, translation, and scaling. Please refer to the readings and Shirley Ch. 6 if you get stuck.

You may implement your transformations any way you choose; however, you should use homogeneous coordinates and support the following operations:

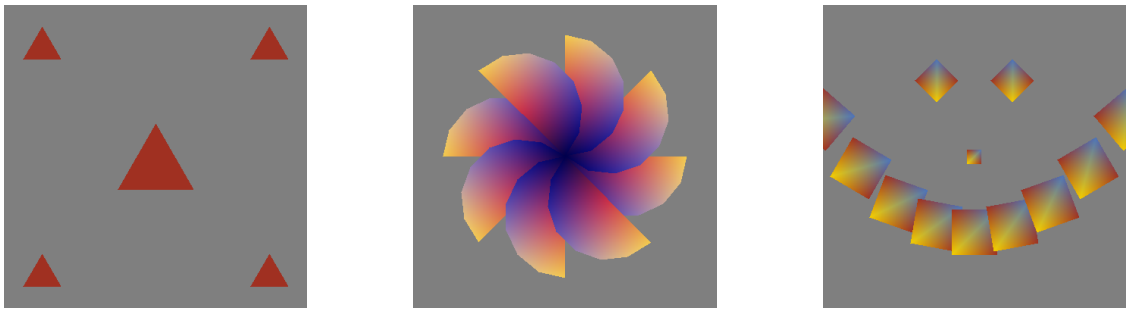
- Independently scaling x and y coordinates by given floating point factors
- Translating x and y coordinates by given offsets
- Rotating a point by a given number of degrees about the origin. Conventionally, positive θ is anticlockwise.

In the following questions, you will be asked to recreate target images to the best of your ability (your image need not match exactly). For each, please include pseudocode showing the order in which the transformations were applied, as well as your rendering.

Question 4: Use your scale and translate transformation matrices to recreate the image in Fig. 4(a).

Question 5: Use your rotation transformation matrix to recreate the image in Fig. 4(b).

Question 6: Use your transformation matrices to recreate the image in Fig. 4(c).
Bonus credit if you can make the face less creepy. Bonus credit also given to anyone able to make the face even more creepy.



(a) (b) (c)

Figure 4: *Use your 2D transformations and the primitives given to recreate these images.*

5 Make an Image!

Using your line and triangle rasterizations, generate a new `.pline` and/or `.tri` file of your creation. Submit this specification along with your rasterization results. Please include a sentence or two describing what you were trying to accomplish, and your process for achieving the final image.

6 What to turn in

To submit your assignment, please bundle the following into a `.zip` file:

1. A PDF file containing:
   - Part 1 : Your renderings of {`binaryclock, nonagons, spiralbox, starbox`}`.pline`
   - Part 2 and 3 : Your renderings of {`red_triangle, rainbow_square, golden, triangle_fan`}`.tri`, along with your answer to Question 3.
   - Part 4 : Your images and pseudocode for Questions 4-6.
   - Part 5 : Your original image and specification file.
2. All original code that we would need to recreate your images, and any special instructions needed to compile it.
3. Any "accidental art"– weird images, mistakes, &c.– demonstrating challenges you faced. Include a brief explanation of what went wrong and how you fixed it.
4. A text file README containing anything else we should know: names of classmates you worked with, help received, references, etc.

Your `.zip` file can be submitted using the form on the students' section of the course webpage. Happy rasterizing!