mkarsalan Update README.md 5a97123 11 minutes ago

1 contributor

262 lines (152 sloc) 13.1 KB

Assignment 1: Rasterizer

In this assignment you will implement a simple rasterizer, including features like supersampling, hierarchical transforms, and texture mapping with antialiasing. At the end, you'll have a functional vector graphics renderer that can take in modified SVG (Scalable Vector Graphics) files, which are widely used on the internet.

Logistics

Deadline

- First 3 parts are due on Tuesday, September 18th at 11:55pm.
- Entire Assignment 1 is due on Tuesday, September 23rd at 11:55pm.
- Assignments which are turned in after 11:55pm are a full day late -- there are no late minutes or late hours.

Getting started

You can either download the zipped assignment straight to your computer or clone it from GitHub using the command

```
git clone git@github.com:mkarsalan/PA1-Rasterizer.git
```

Building the assignment

We will be using CMake to build the assignments. If you don't have CMake (version >= 2.8) on your personal computer, you can install it using apt-get on Linux or Macports/Homebrew on OS X. Alternatively, you can download it directly from the CMake website.

To build the code, start in the folder that GitHub made or that was created when you unzipped the download. Run

```
mkdir build
```

to create a build directory, followed by

```
cd build
```

to enter the build directory. Then

```
cmake ..
```

to have CMake generate the appropriate Makefiles for your system, then

```
make
```

to make the executable, which will be deposited in the build directory.

Don't forget to run `make` every time you make changes to your files.

What you will turn in

You will need to *zip* your **src** folder (located in your main directory) and submit it on LMS.

Using the GUI

You can run the executable with the command

```
./draw ../svg/basic/test1.svg
```

After finishing Part 3, you will be able to change the viewpoint by dragging your mouse to pan around or scrolling to zoom in and out. Here are all the keyboard shortcuts available (some depend on you implementing various parts of the assignment):

Key	Action
`' '`	return to original viewpoint
`'-'`	decrease sample rate
`'='`	increase sample rate
`'Z'`	toggle the pixel inspector
`'P'`	switch between texture filtering methods on pixels
`'L'`	toggle scanLine
`'S'`	save a *png* screenshot in the current directory
`'1'-'9'`	switch between svg files in the loaded directory

The argument passed to `draw` can either be a single file or a directory containing multiple *svg* files, as in

```
./draw ../svg/basic/
```

If you load a directory with up to 9 files, you can switch between them using the number keys 1-9 on your keyboard.

Project structure

Section I: Rasterization (suggested completion checkpoint: Sunday 1/29)

- Part 1: Rasterizing single-color triangles
- Part 2: Antialiasing triangles
- Part 3: Transforms

Section II: Sampling

- Part 4: Barycentric coordinates
- Part 5: "Pixel sampling" for texture mapping

Section III: Scan Line

- Part 6: Scan Line

There is a fair amount of code in the CGL library, which we will be using for future assignments. The relevant header files for this assignment are *vector2D.h*, *matrix3x3.h*, *color.h*, and *renderer.h*.

Here is a very brief sketch of what happens when you launch `draw`: An `SVGParser` (in *svgparser.**) reads in the input *svg* file(s), launches a OpenGL `Viewer` containing a `DrawRend` renderer, which enters an infinite loop and waits for input from the mouse and keyboard. DrawRend (*drawrend.**) contains various callback functions hooked up to these events, but its main job happens inside the `DrawRend::redraw()` function. The high-level drawing work is done by the various `SVGElement` child classes (*svg.**), which then pass their low-level point, line, and triangle rasterization data back to the three `DrawRend` rasterization functions.

Here are the files you will be modifying throughout the project:

1. *drawrend.cpp*, *drawrend.h*
2. *texture.cpp*
3. *transforms.cpp*
4. *svg.cpp*

In addition to modifying these, you will need to reference some of the other source and header files as you work through the project.

Section I: Rasterization

Part 1: Rasterizing single-color triangles

Triangle rasterization is a core function in the graphics pipeline to convert input triangles into framebuffer pixel values. In Part 1, you will implement triangle rasterization using the methods discussed in lecture 2 to fill in the `DrawRend::rasterize_triangle(...)` function in *drawrend.cpp*.

Notes:

- It is recommended that you implement `SampleBuffer::fill_color()` function first, so that you can see rasterized points and lines for some SVGs.
- Remember to do point-in-triangle tests with the point exactly at the center of the pixel, not the corner. Your coordinates should be equal to some integer point plus (.5,.5).
- For now, ignore the `Triangle *tri` input argument to the function. We will come back to this in Part 4.
- You are encouraged but not required to implement the edge rules for samples lying exactly on an edge.
- Make sure the performance of your algorithm is no worse than one that checks each sample within the bounding box of the triangle.

When finished, you should be able to render many more test files, including those with rectangles and polygons, since we have provided the code to break these up into triangles for you. In particular, *basic/test3.svg*, *basic/test4.svg*, and *basic/test5.svg* should all render correctly.

For convenience, here is a list of functions you will need to modify:

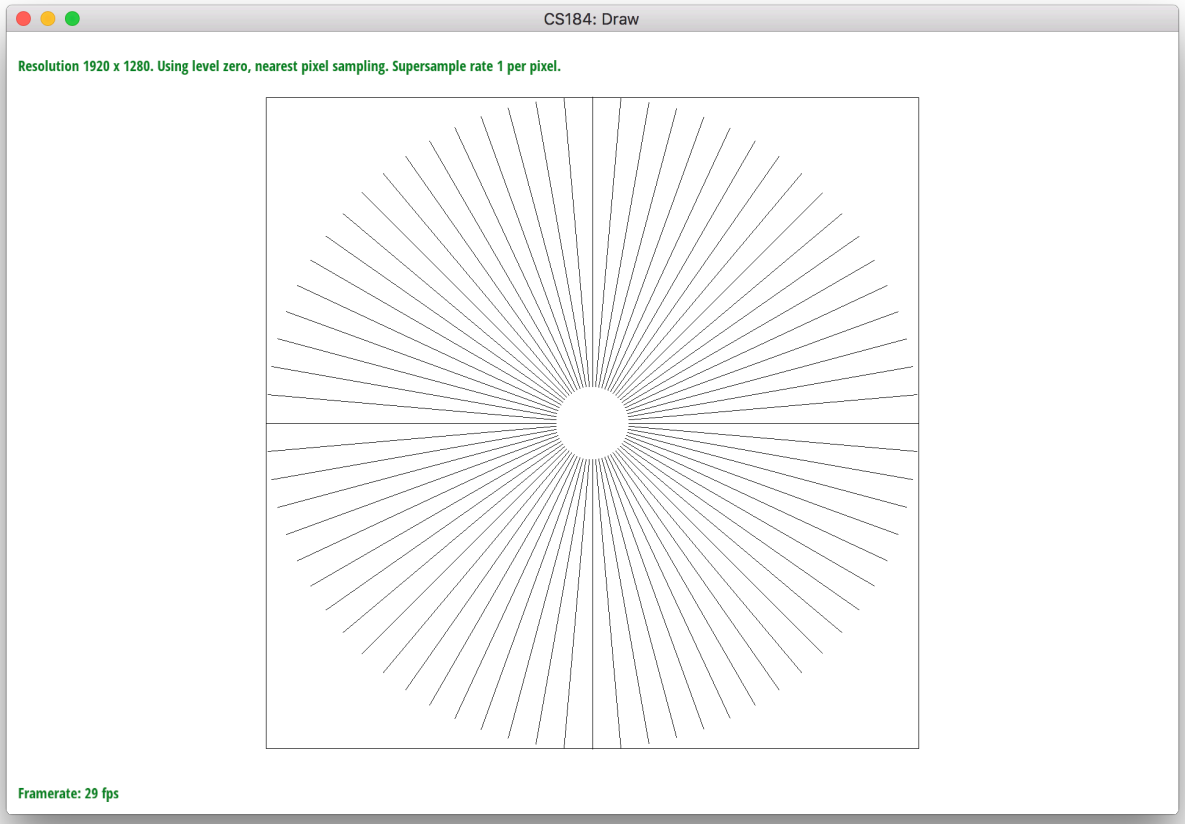
1. `SampleBuffer::fill_color` *located in src/drawrend.h*
2. `DrawRend::rasterize_triangle` *located in src/drawrend.cpp*

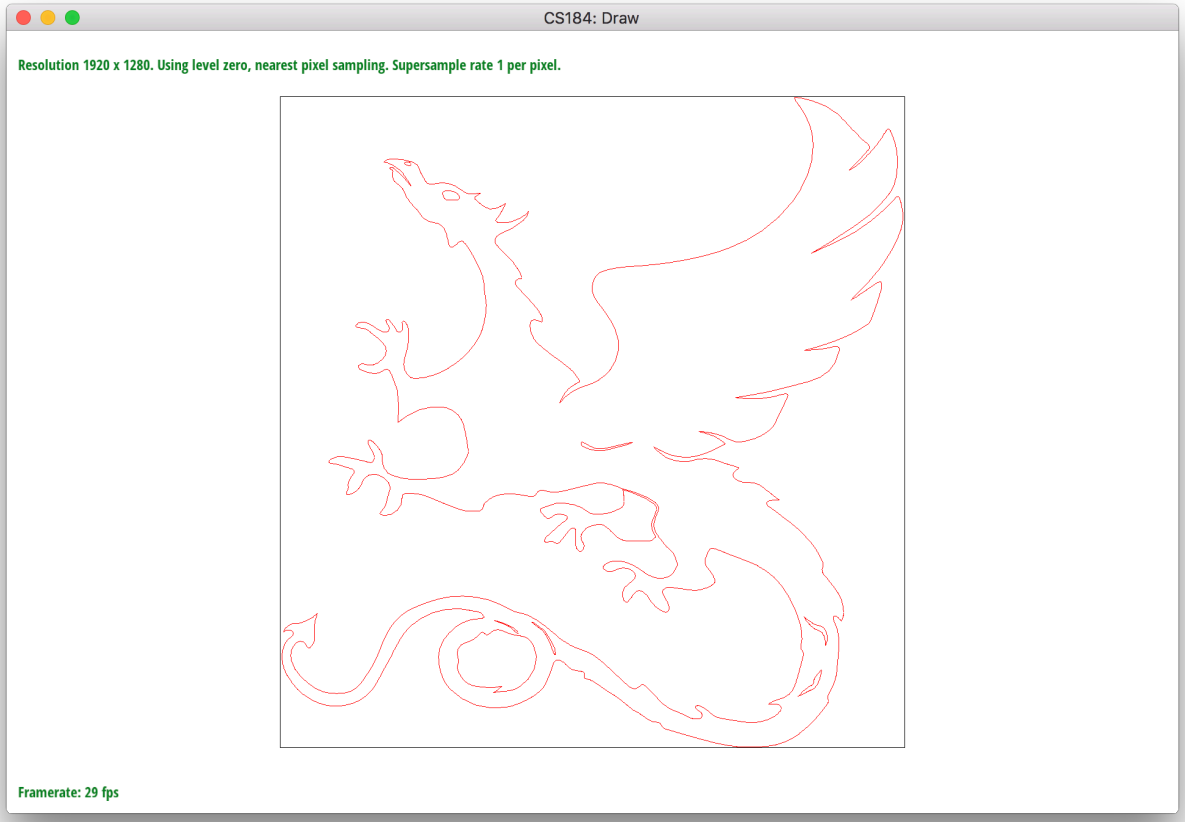
Extra Credit: Make your triangle rasterizer super fast (e.g., by factoring redundant arithmetic operations out of loops, minimizing memory access, and not checking every sample in the bounding box). Write about the optimizations you used.

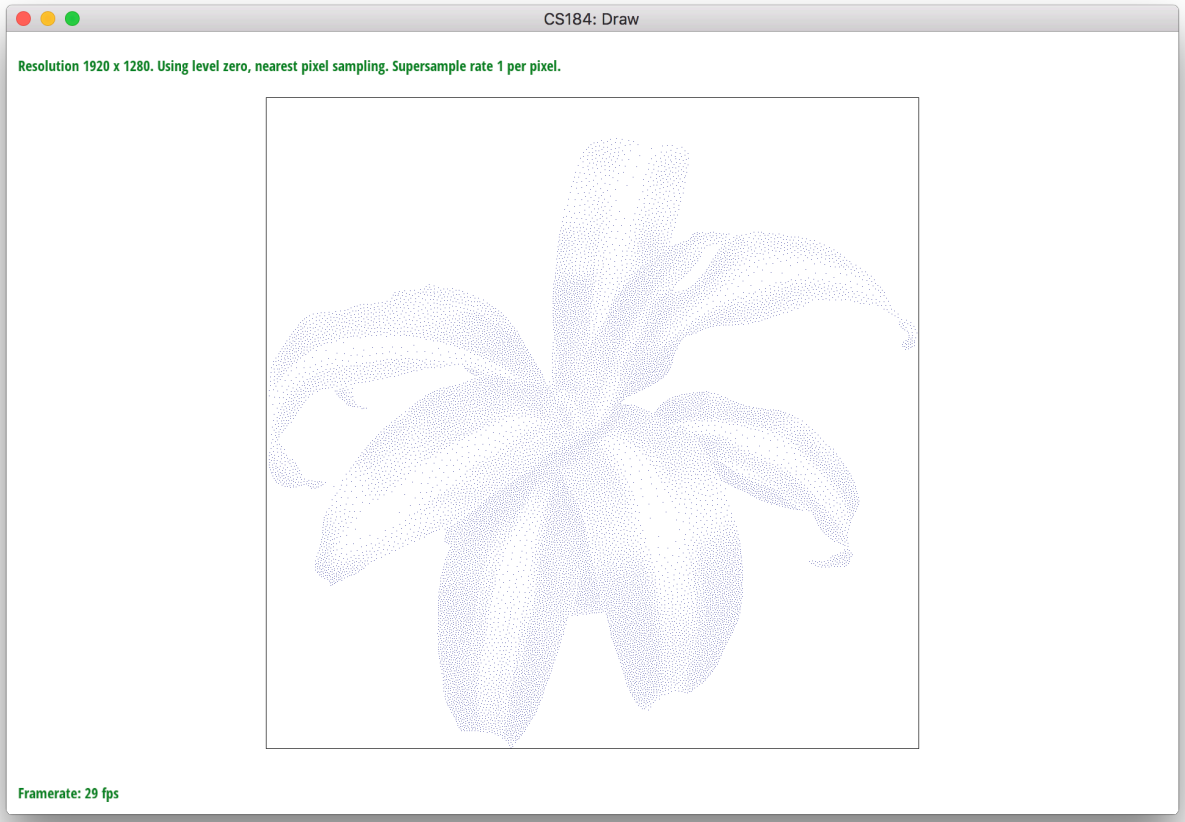
Execute following command in build directory. Cycle through different SVG files by pressing 1-9.

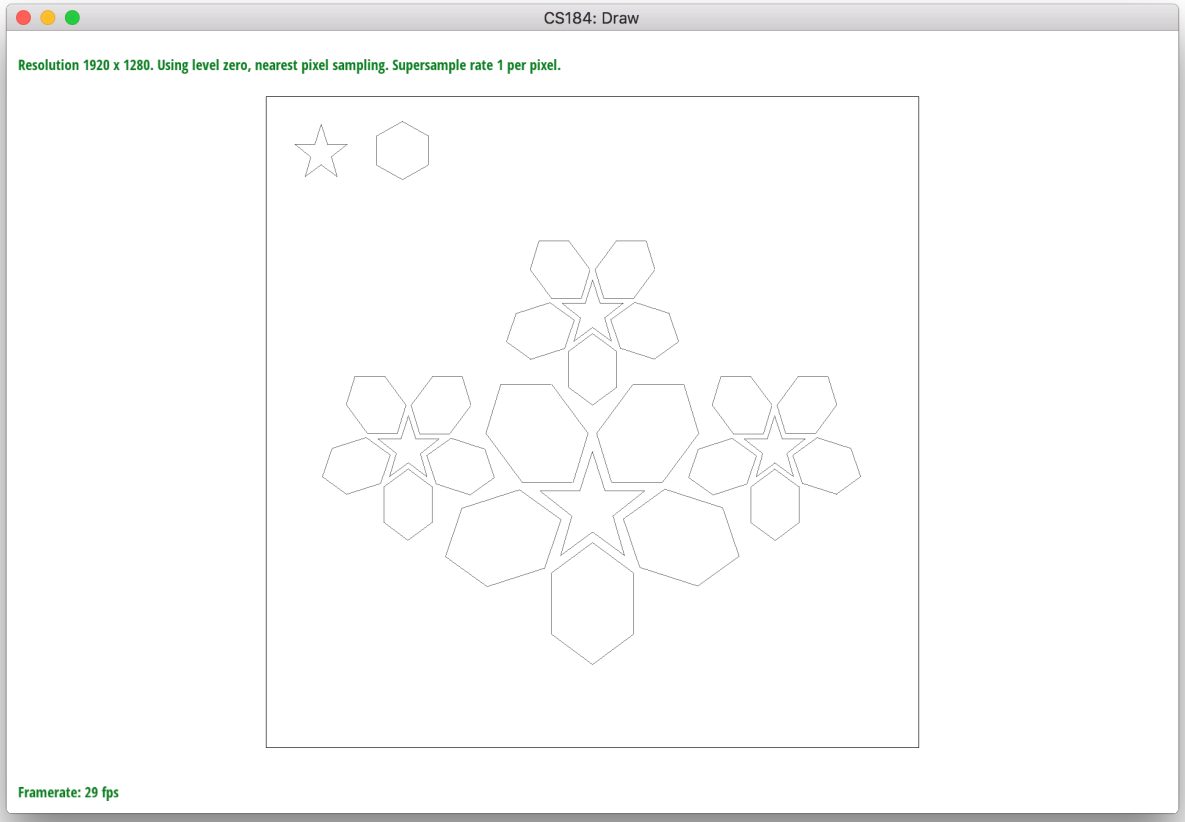
```
./draw ../svg/basic/
```

Once you have implemented **SampleBuffer::fill_color()**, you will be able to render the following SVGs:

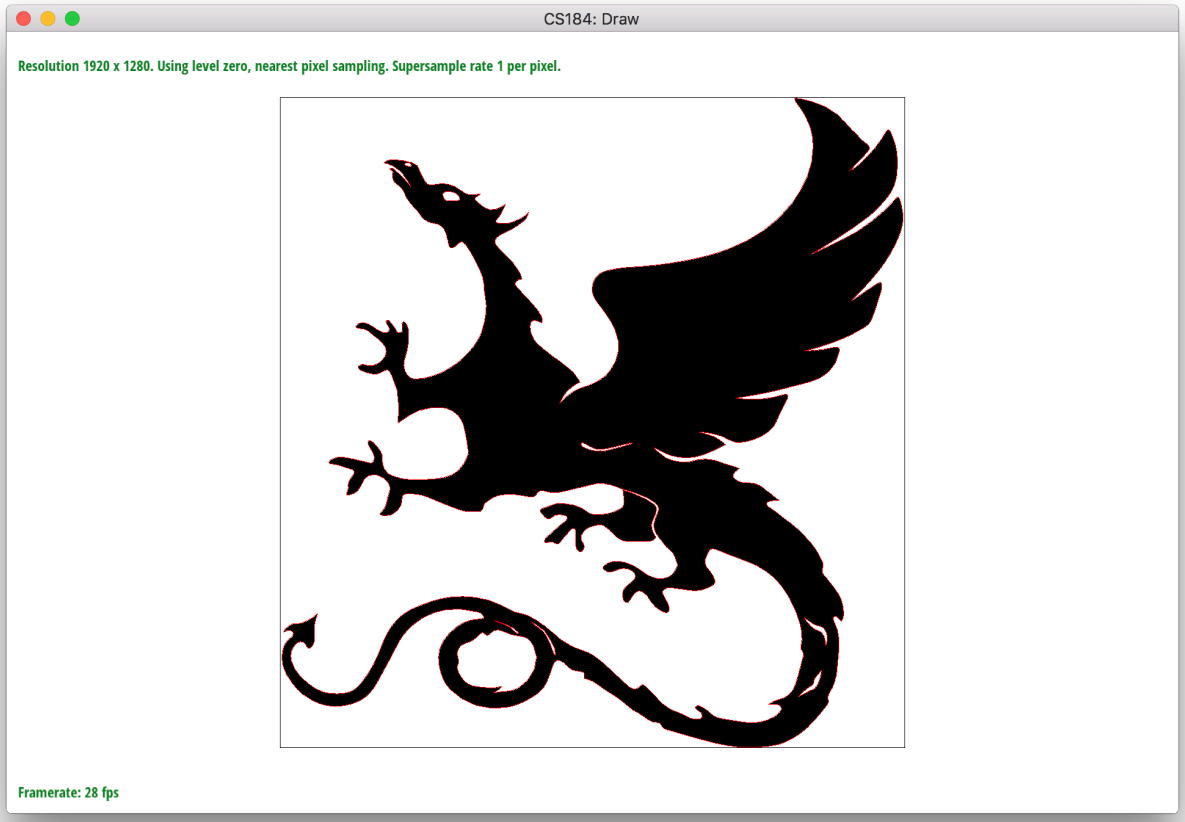


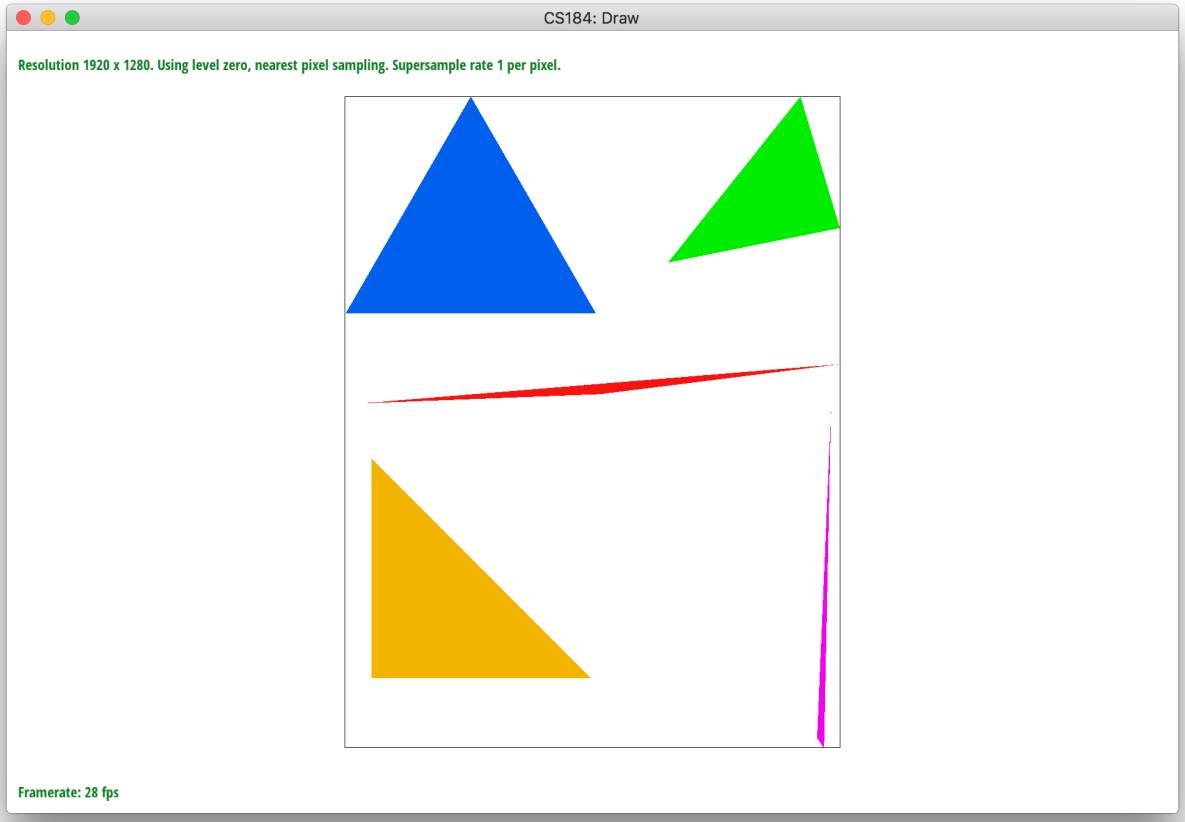

CS184: Draw
Resolution 1920 x 1280. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.
Framerate: 29 fps

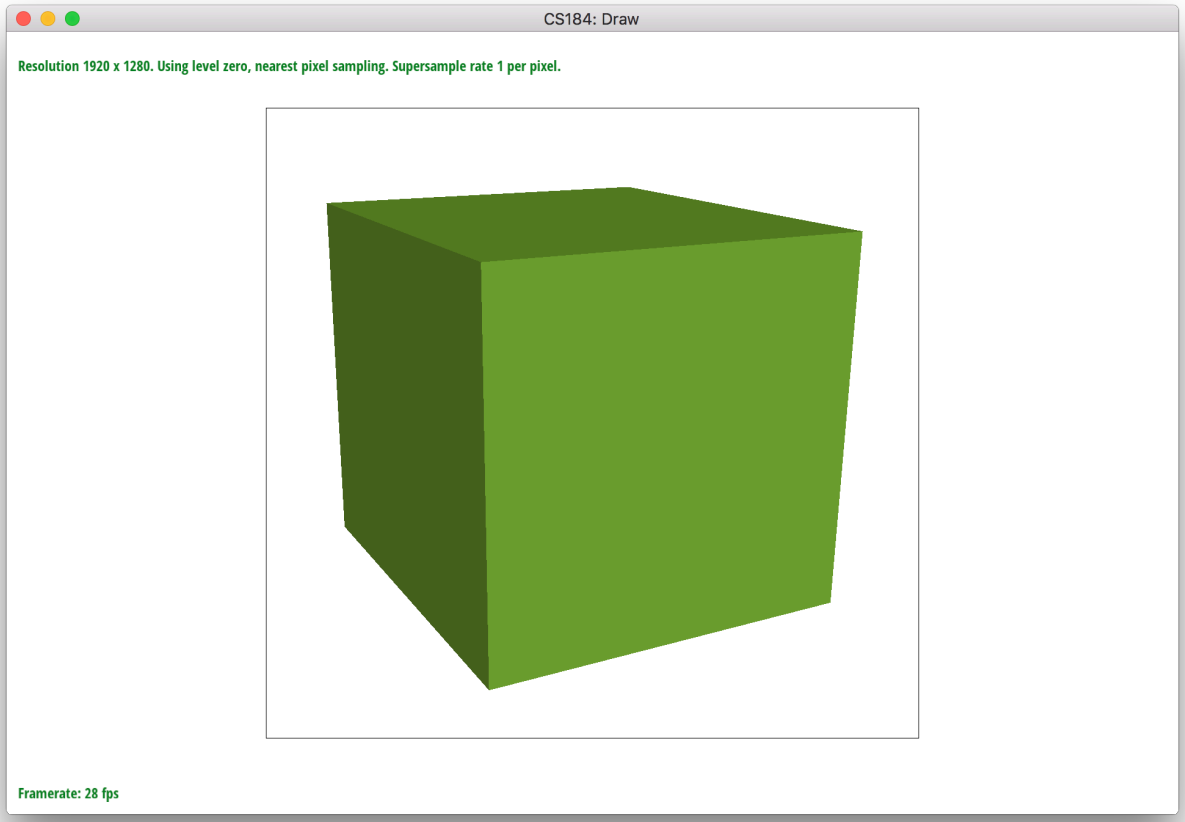

CS184: Draw
Resolution 1920 x 1280. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.
Framerate: 29 fps

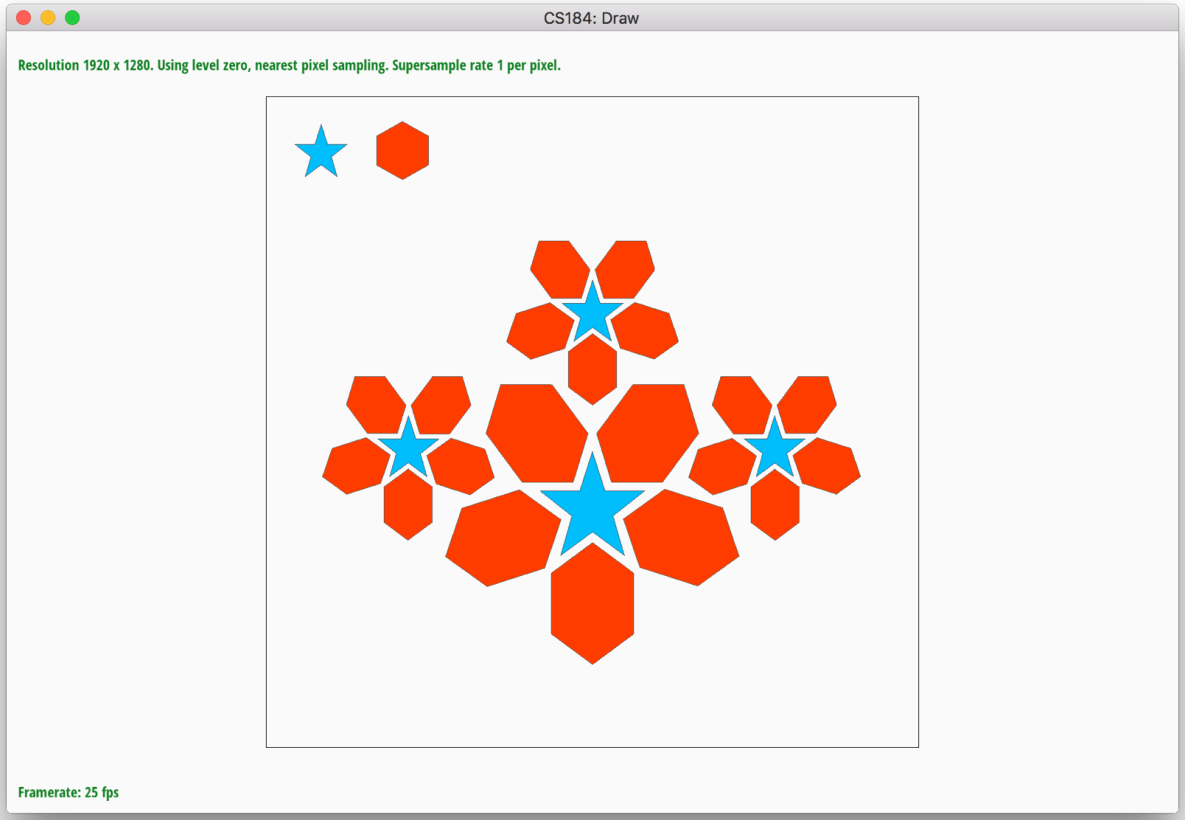


After implementing **DrawRend::rasterize_triangle**:

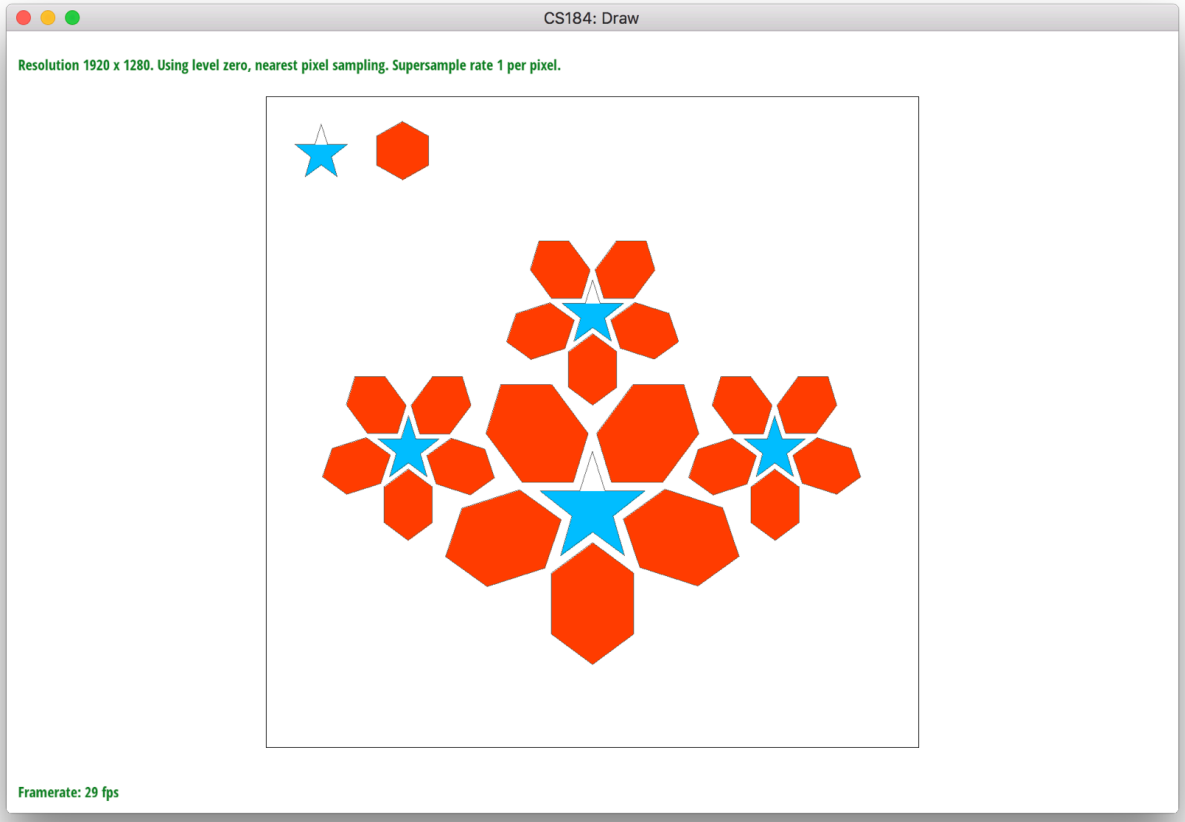


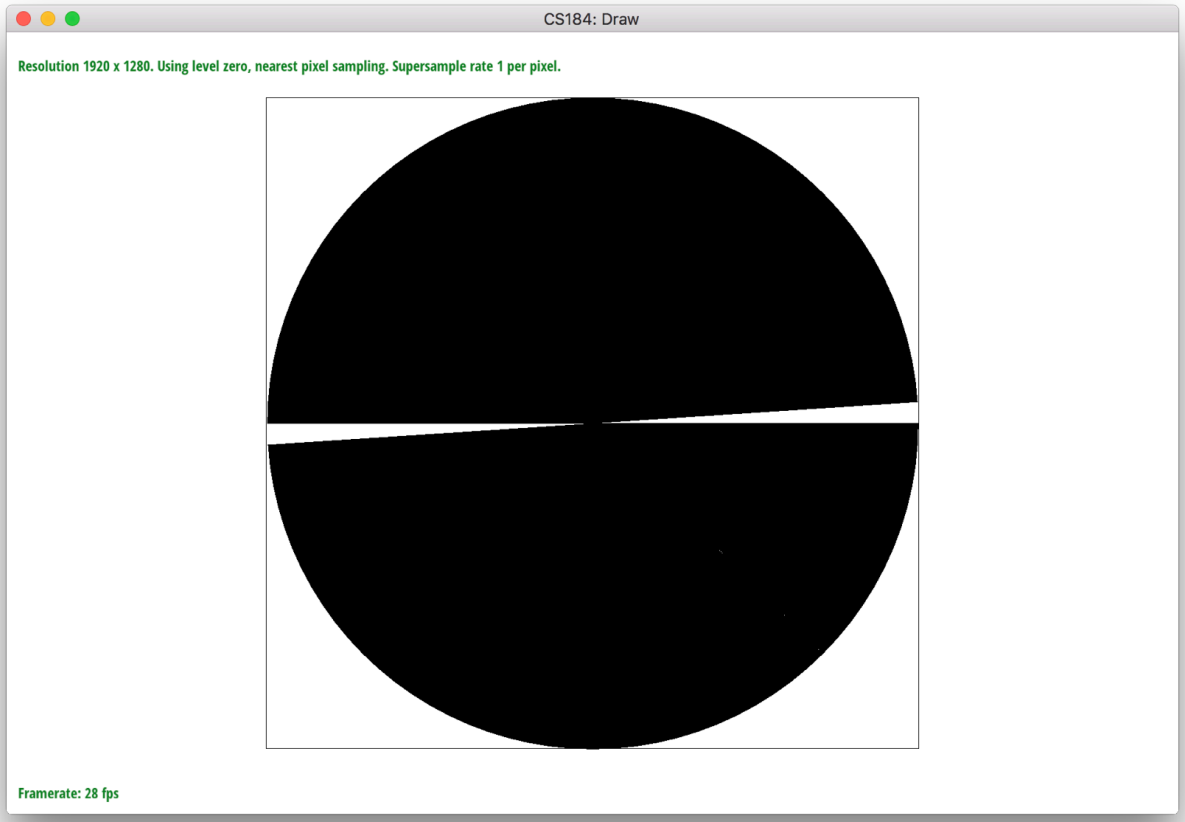

CS184: Draw
Resolution 1920 x 1280. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.
Framerate: 28 fps


CS184: Draw
Resolution 1920 x 1280. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.
Framerate: 28 fps



If you encounter some partially filled stars or a disconnected circle, then there was something wrong with your implementation, consider fixing it before submission.

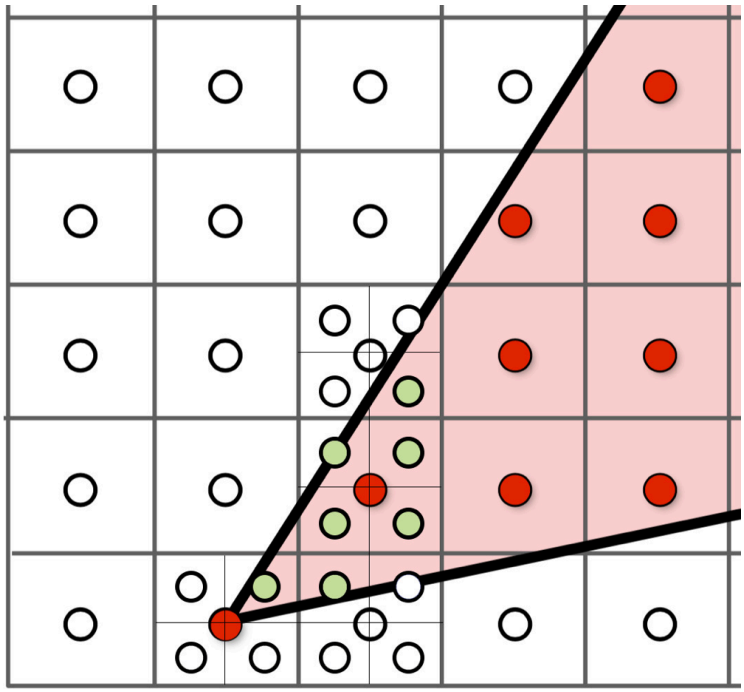




Part 2: Antialiasing triangles

Use supersampling to antialias your triangles. The `sample_rate` parameter in `DrawRend` (adjusted using the `-` and `=` keys) tells you how many samples to use per pixel.

The image below shows how sampling four times per pixel produces a better result than just sampling once, since some of the supersampled pixels are partially covered and will yield a smoother edge.



To do supersampling, each pixel is now divided into `sqrt(sample_rate) * sqrt(sample_rate)` sub-pixels. In other words, you still need to keep track of `height * width` pixels, but now each pixel has `sqrt(sample_rate) * sqrt(sample_rate)` sampled colors. You will need to do point-in-triangle tests at the center of each of these *sub-pixel* squares.

We provide a `SampleBuffer` class to store the sub-pixels. Each samplebuffer instance stores one pixel. Your task is to fill every sub-pixel with its correctly sampled color for every samplebuffer, and average all sub-pixels' colors within a samplebuffer to get a pixel's color. Since you've finished Part 1, you can use `SampleBuffer::fill_color()` function to write color to a sub-pixel.

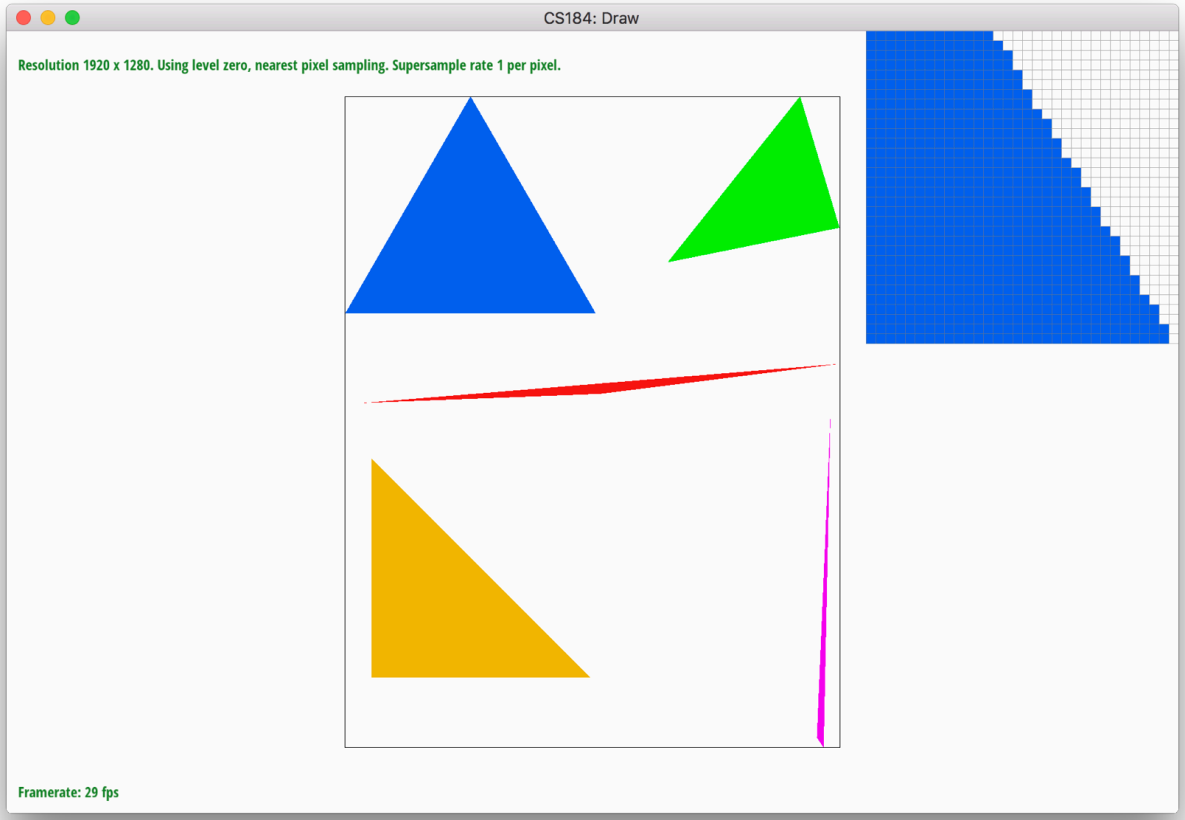
Your triangle edges should be noticeably smoother when using > 1 sample per pixel! You can examine the differences closely using the pixel inspector. Also note that, it may take several seconds to switch to a higher sampling rate.

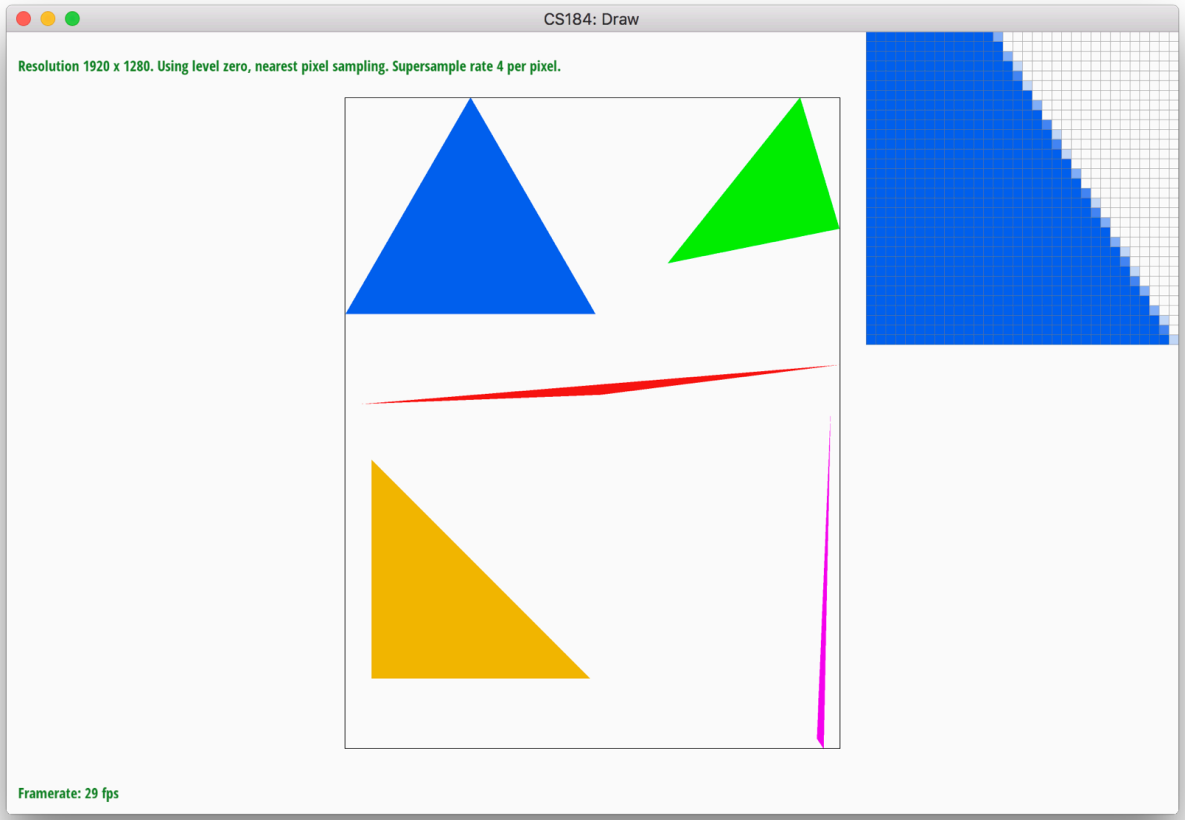
For convenience, here is a list of functions you will need to modify:

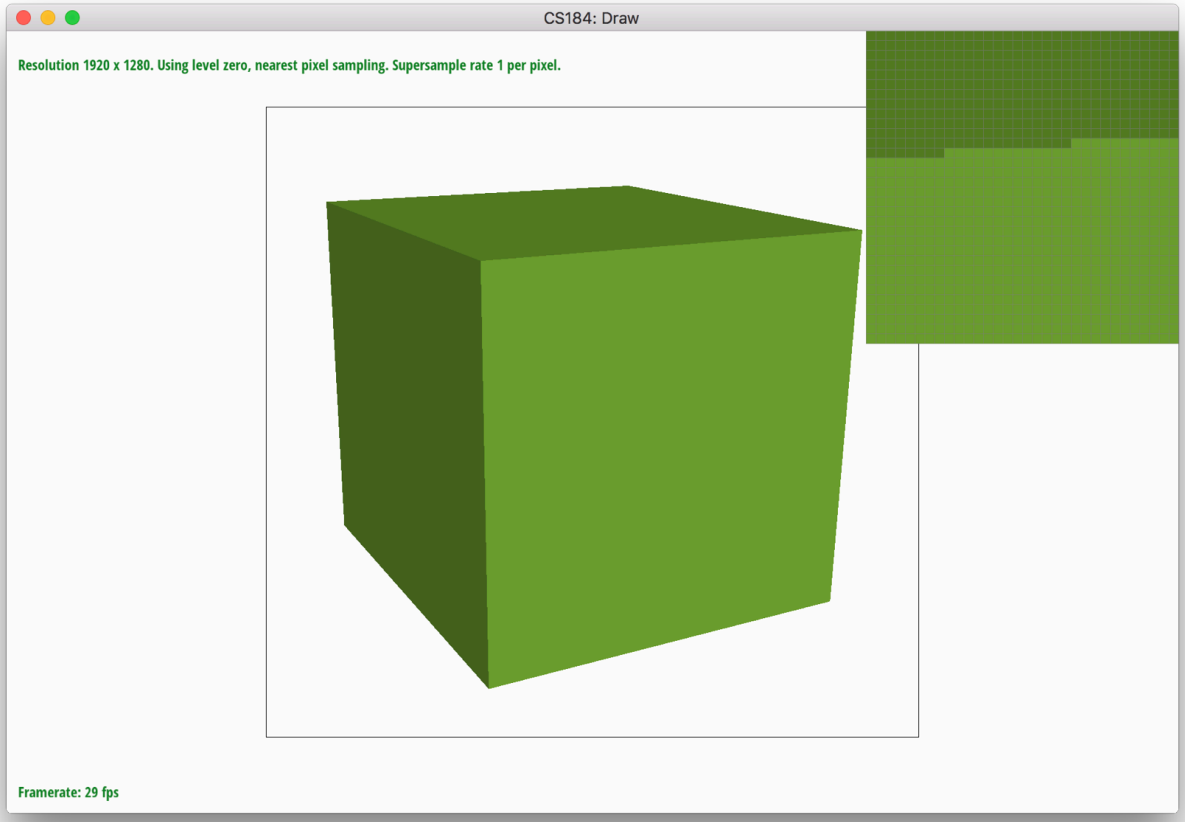
1. `DrawRend::rasterize_triangle`
2. `SampleBuffer::get_pixel_color`

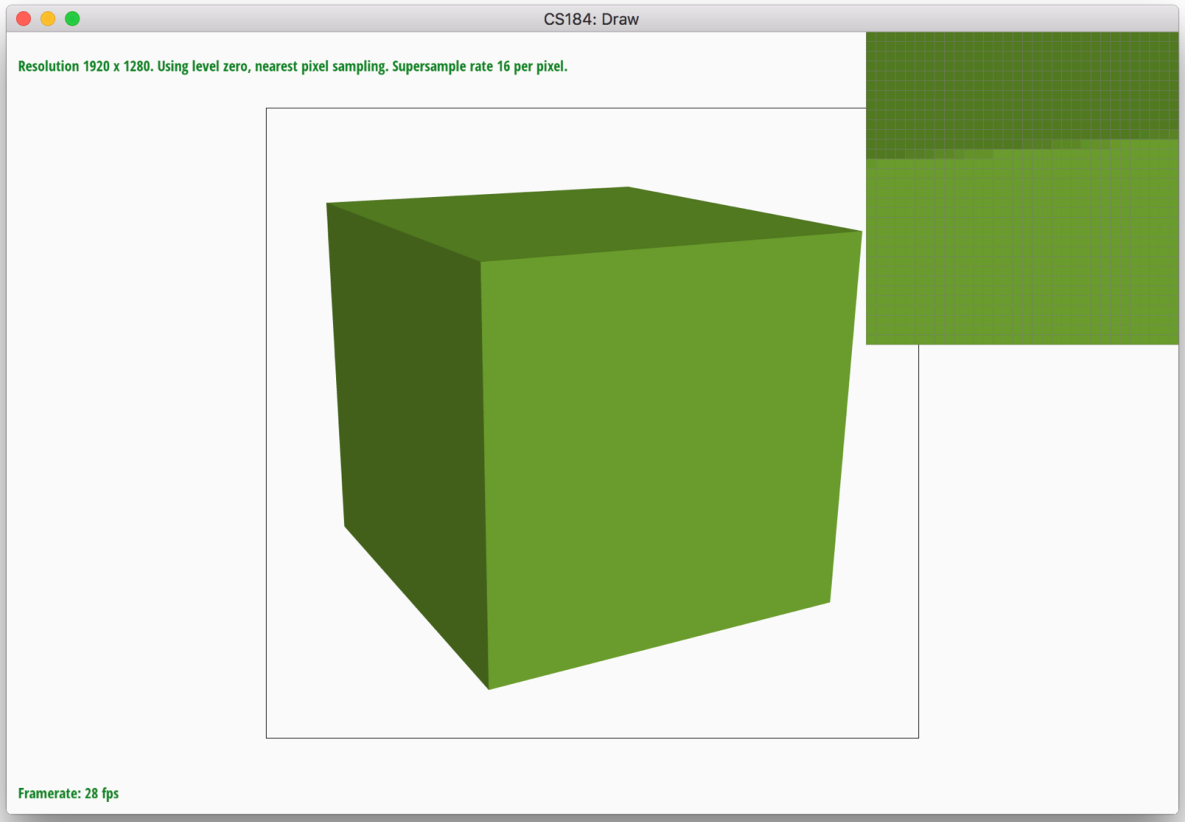
Extra Credit: Implement an alternative sampling pattern, such as jittered or low-discrepancy sampling.

Adjust your samples per pixel by using the `-` and `=` keys.


CS184: Draw
Resolution 1920 x 1280. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.
Framerate: 29 fps


CS184: Draw
Resolution 1920 x 1280. Using level zero, nearest pixel sampling. Supersample rate 4 per pixel.
Framerate: 29 fps





Part 3: Transforms

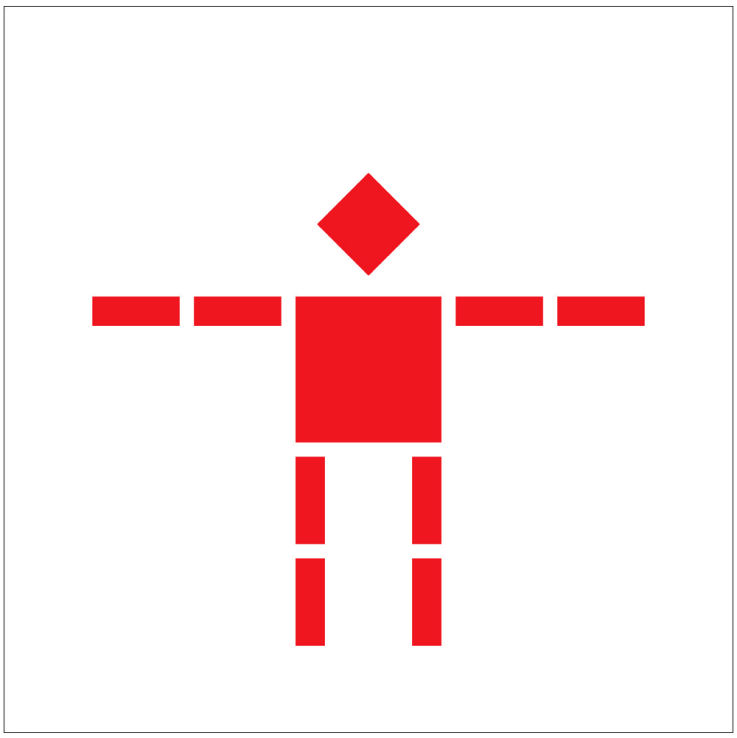
Implement the three transforms in the *transforms.cpp* file according to the SVG spec. The matrices are 3x3 because they operate in homogeneous coordinates -- you can see how they will be used on instances of `Vector2D` by looking at the way the `*` operator is overloaded in the same file.

For convenience, here is a list of functions you will need to modify:

1. `translate`
2. `scale`
3. `rotate`

located in src/transforms.cpp

Once you've implemented these transforms, *svg/transforms/robot.svg* should render correctly, as follows:



Section II: Sampling

Part 4: Barycentric coordinates

Familiarize yourself with the `ColorTri` struct in *svg.h*. Modify your implementation of `DrawRend::rasterize_triangle(...)` so that if a non-NULL `Triangle *tri` pointer is passed in, it computes barycentric coordinates of each sample hit and passes them to `tri->color(...)` to request the appropriate color.

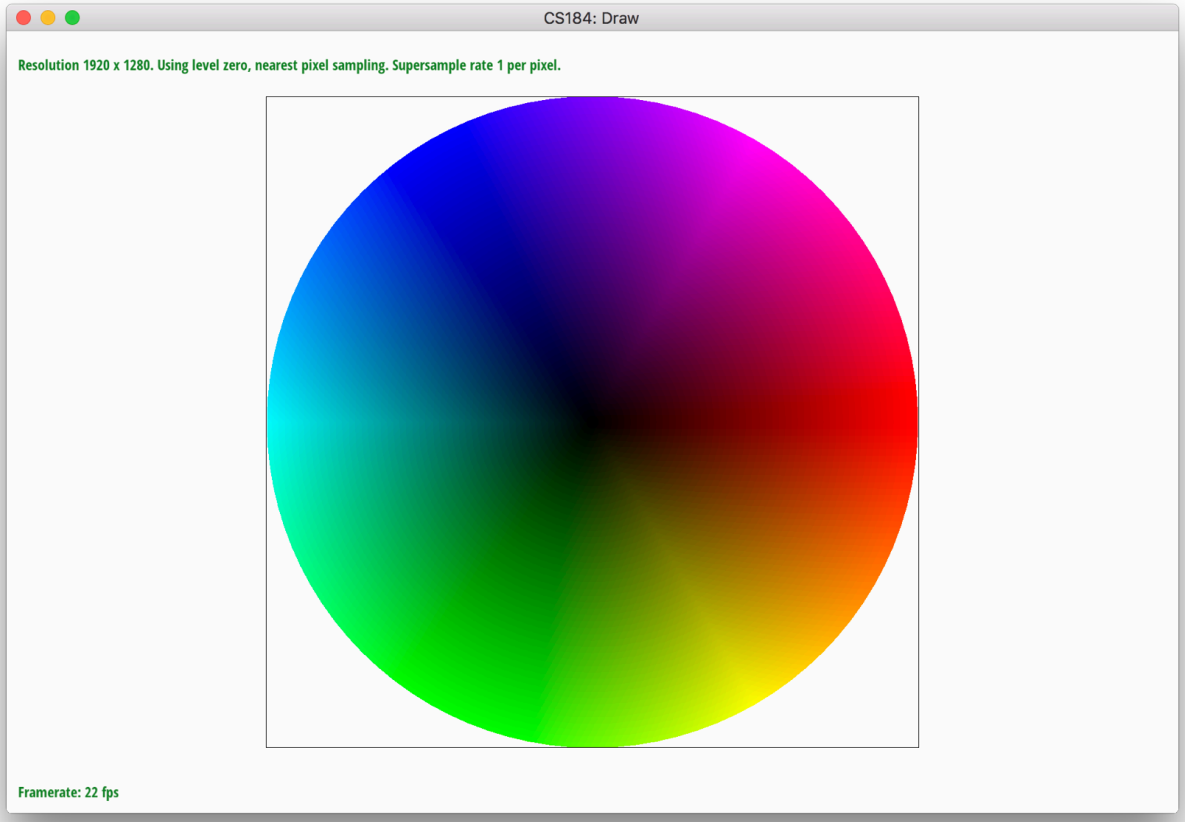
Implement the `ColorTri::color(...)` function in *svg.cpp* so that it interpolates the color at the point `p_bary`. `p0_col`, `p1_col`, `p2_col` represent colors at points `p0`, `p1`, `p2`, respectively, use them to calculate color at point `p0` using `p_bary` and return it.

You may use `Vector2D` and `vector3D` datastructures to make calculations easier, a few have been declared for you.

For convenience, here is a list of functions you will need to modify:

1. `DrawRend::rasterize_triangle`
2. `ColorTri::color`

After implementing this part, instead of a black circle, you will be able to view the colour wheel:



Part 5: "Pixel sampling" for texture mapping

Familiarize yourself with the `TexTri` struct in *svg.h*. This is the primitive that implements texture mapping. For each vertex, you are given corresponding *uv* coordinates that index into the `Texture` pointed to by `*tex`.

To implement texture mapping, `DrawRend::rasterize_triangle`, you should fill in the `psm` member of a `SampleParams` struct and pass it to `tri->color(...)`. Then `TexTri::color(...)` should fill in the correct *uv* coordinates in the `SampleParams` struct, and pass it on to `tex->sample(...)`. Then `Texture::sample(...)` should examine the `SampleParams` to determine the correct sampling scheme.

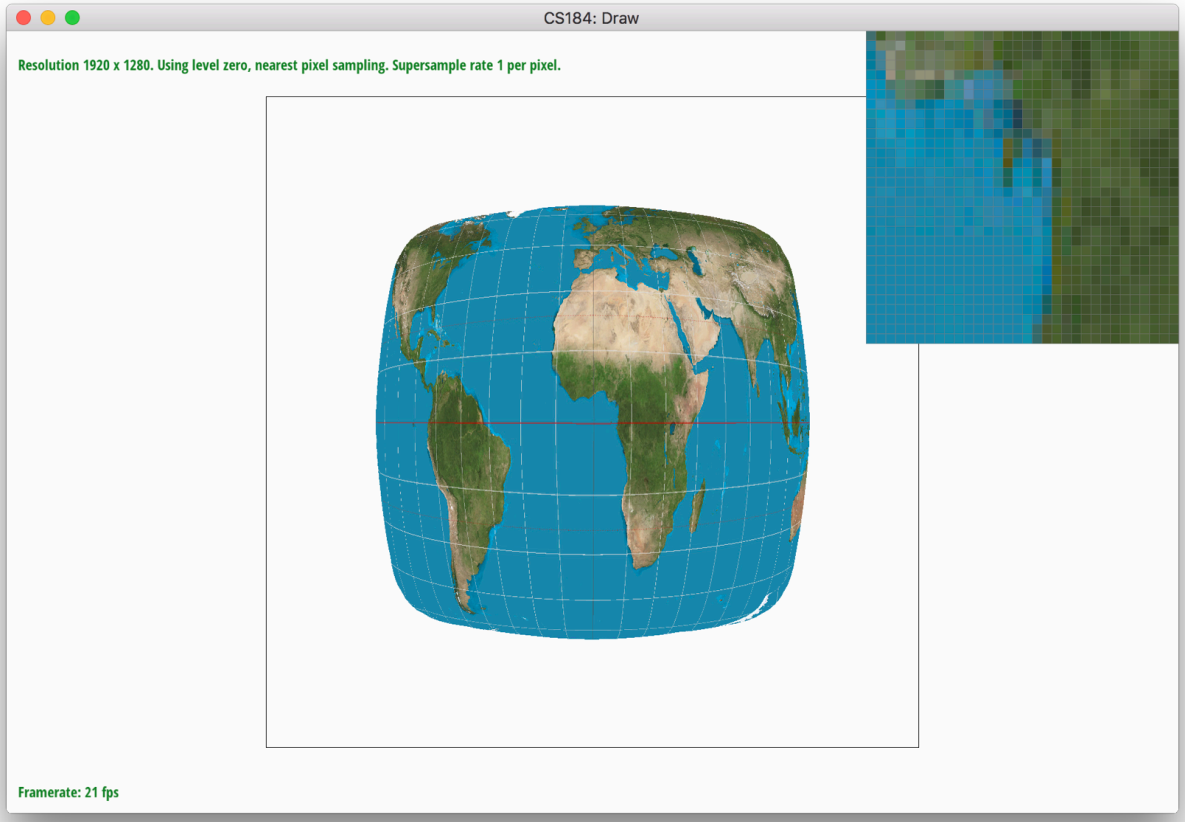
The GUI toggles `DrawRend`'s `PixelSampleMethod` variable `psm` using the `'P'` key, Use it to alternate between `sample_nearest` and `sample_bilinear`

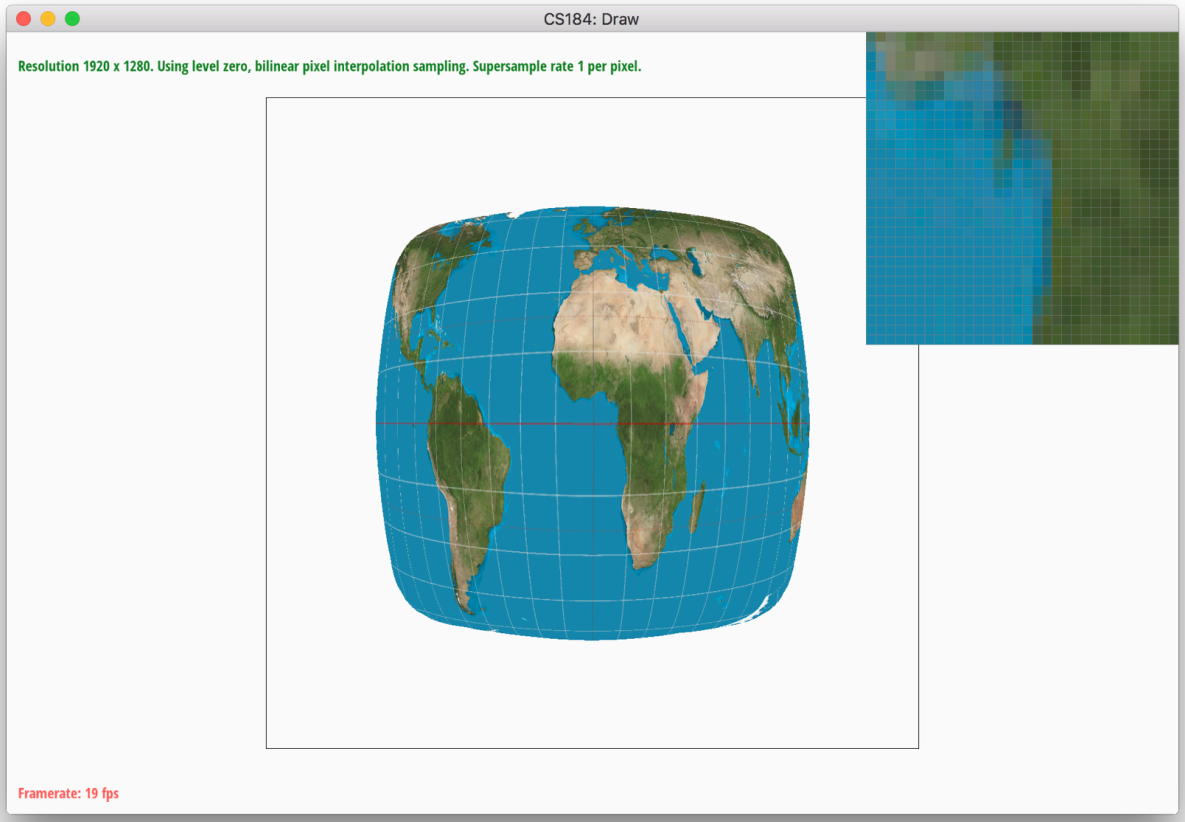
For convenience, here is a list of functions you will need to modify:

1. `DrawRend::rasterize_triangle`
2. `TexTri::color`
3. `Texture::sample`
4. `Texture::sample_nearest`
5. `Texture::sample_bilinear`

Execute following command in build directory. Cycle through different texmap files by pressing 1-6.

```
./draw ../svg/texmap/
```





Section III: Scan Line

Part 6: Implement Scan Line

Scan Line is an effecient algorithm to determine if a point lies within a polygon. Hence, in comparison to our implementation in part 1, it should be less sluggish.

The imput methods for `rasterize_scanLine` are the same as the simple `rasterize_triangle`. As mentioned before, the only change you need to make is that how you determine whether a point is in a polygon.

For convenience, here is the function you will need to modify:

```
DrawRend::rasterize_scanLine
```

Use `L` to toggle scanline. Note your time difference.

This assignment is adapted from UC Berkeley CS184 (https://cs184.eecs.berkeley.edu/article/3).