Homework 1: Rasterizer

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

Homework Structure

The homework has 6 tasks, worth a total of 100 possible points. Some require only a few lines of code, while others are more substantial.

- Task 1: Drawing Single-Color Triangles (20 pts)
- Task 2: Antialiasing by Supersampling (20 pts)
- Task 3: Transforms (10 pts)

- Task 4: Barycentric coordinates (10 pts)
- Task 5: "Pixel sampling" for texture mapping (15 pts)
- Task 6: "Level sampling" with mipmaps for texture mapping (25 pts)
- Part 7: Extra Credit - Draw Something Creative!

Logistics

Deadline

Homework 1 is due **Tuesday 2/13, 11:59PM**. Both your code and write-up need to be turned in for your submission to be complete; assignments which are turned in after 11:59pm will use one of your late days -- there are no late minutes or late hours.

Partners

You can work with a partner on this homework. If you work with a partner, both of you should join the same team on Github Classroom. You should also produce 1 writeup and 1 Gradescope submission with your partner added to the team on Gradescope.

Checkpoint Quiz

There is a Gradescope Checkpoint quiz for this homework, due Friday 2/2, 11:59PM. This short quiz is intended to help you get started on Homework 1. The content necessary for answering all questions can be found on the HW 1 spec or in lecture material.

This checkpoint can be used to gain "buffer points" on the assignment. As long as you get at least 4 out of the 7 content-questions (Q1-7) correct, AND submit a valid screenshot of your homework building (Q8), you'll get the full 5 buffer points. No buffer points will be given if you did not meet these requirements; no late submissions will be allowed for the checkpoint quiz. These 5 buffer points will be added to your HW 1 grade, capped at increasing your score to 100. For

example, if you earned 96 points on the base homework and 2 points of extra credit, the 5 buffer points will bring your score up to 100, and the extra credit will bring your total score to 102.

Homework Parties

We will have 5 homework parties:

Date/Time	Location
Fri 2/2 4-6pm	Soda 310
Wed 2/7 5-7pm	Soda 380
Fri 2/9 1-3pm	Berkeley Way West 1st Floor (Room TBD)
Mon 2/12 5-7pm	Soda 380
Tue 2/13 1-3pm	Berkeley Way West 1st Floor (Room TBD)

Academic honesty

Please do not post code to a public GitHub repository, even after the class is finished, since these assignments will be reused both here and at other universities in the future.

The homeworks are to be completed individually, unless you are working with a partner. You are welcome to discuss the various parts of the assignments with your classmates, but you must implement the algorithms yourself. You are free to share all code with your partner.

Getting started

First, accept the assignment from your CS184/284A website profile, following the instructions from GitHub Classroom. Then, clone the generated private repo. Make sure you clone *your private repo*.

```
$ git clone <YOUR_PRIVATE_REPO>
```

Please consult this how to build assignments for CS184 article for more information on how to setup and build the assignment.

As you go through the assignment, refer to the write-up guidelines and deliverables section below. **It is recommended that you complete each section's write-up as you finish that section.** It's generally not a good idea to wait until the end to start your writeup. You may also find it helpful to skim the rubric before beginning your work.

Important: To avoid compression artifacts in your images, please do not convert the PNG screenshot images saved by the GUI into JPG or other formats! PNG images are losslessly compressed.

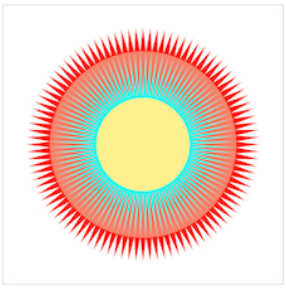
Finally, you may find the following resources helpful:

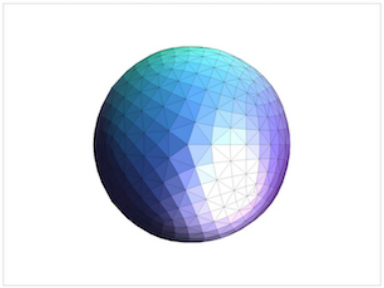
C++ Guide for some quick tips and tricks on getting started with C++. A slightly more detailed C++ guide can be found here.

Images as Data on how images and colors are often represented in code.

Vectors and Matrices in the CGL Library for a quick refresher on declaring and using vectors and matrices.

CGL Vectors API for the API listing of CGL Vectors library.





Using the GUI

You can run the executable with the command

```
./draw [path to svg file/folder to render]
```

For example, you could run this command:

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

> Note: For Visual Studio, the output folder is 3 layers deep. Therefore you should use `./draw ../../../svg/basic/test1.svg` For Linux / Unix / Mac commandline build, it should be 1 layer deep: `./draw ../svg/basic/test1.svg` All the IDEs has some form of debug / launch settings. You can use those to specify the SVG file and then you can use the debugger provided by the IDE. Anyways, the path should always be relative to the executable file!

You'll see a flower composed of blue dots, based on point and line rasterization provided in the starter code. **Most other SVG files won't render correctly until you work through the assignment.** Here are the keyboard shortcuts available (some depend on you implementing various parts of the assignment):

Key	Action
`space`	return to original viewpoint
`-`	decrease sample rate
`=`	increase sample rate
`Z`	toggle the pixel inspector
`P`	switch between texture filtering methods on pixels
`L`	switch between texture filtering methods on mipmap levels
`S`	save a *PNG* image 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.

Familiarize Yourself with the Starter Code

Most of your modifications will be constrained to implementing or modifying functions in `rasterizer.cpp`, `transforms.cpp` and `texture.cpp`.

In addition to modifying these, you will need to understand other source and header files as you work through the homework. As one example, the starter code for this and future assignments use the CGL library. For this assignment, you may want to familiarize yourself with classes defined in `vector2D.h`, `matrix3x3.h` and `color.h`.

Here is a brief sketch of what happens when you launch `draw`:

1. An `SVGParser` (in `svgparser.h/cpp`) reads in the input *svg* file(s)
2. It launches a OpenGL `Viewer` containing a `DrawRend` (in `drawrend.h/cpp`) renderer, which enters an infinite loop and waits for input from the mouse and keyboard.
3. In `DrawRend::redraw()` function, the high-level drawing work is done by the various `SVGElement` child classes (in `svg.h/cpp`), which then pass their low-level point, line, and triangle rasterization data to appropriate methods of a `Rasterizer` class.

A Simple Example: Drawing Points

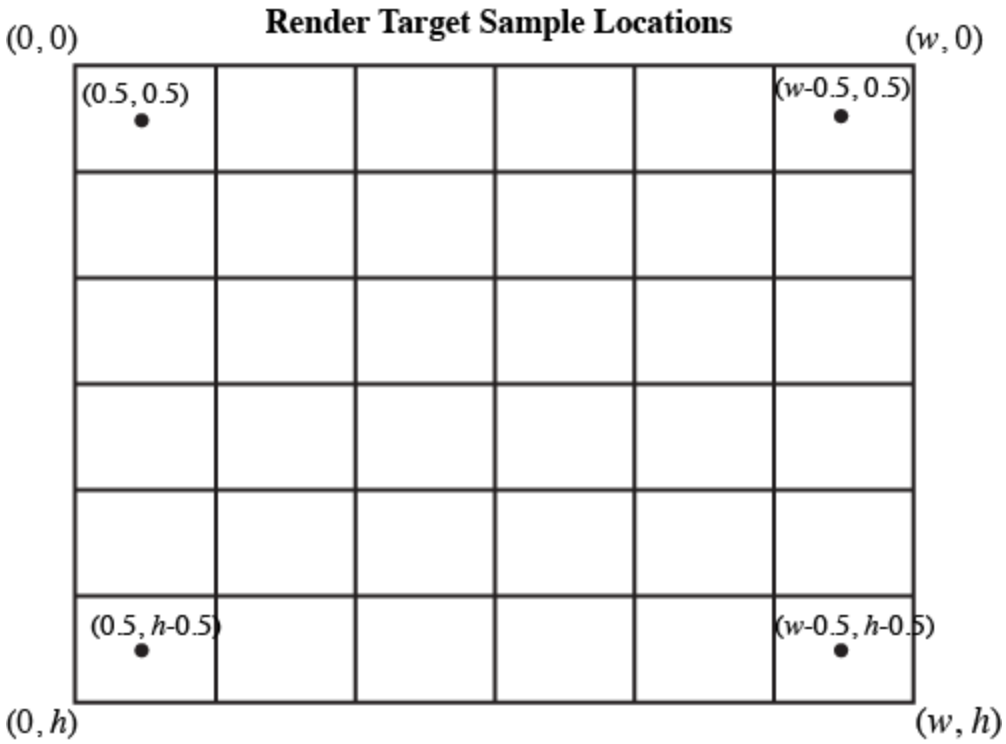
You are given starter code that already implements drawing of 2D points. To see how this works, begin by taking a look at `SVG::draw()` in `svg.h`.

1. The SVG object draws all elements in the SVG file via a sequence of calls to their `draw()` functions.
2. Each element type calls an appropriate draw function on a `Rasterizer` object.
    - In the case of the `Point` element type, `Point::draw()` eventually calls the concrete draw function implemented in `RasterizerImp::rasterize_point()` in `rasterizer.cpp`. The position of SVG elements in an SVG file is defined in a local coordinate frame, so `Point::draw()` transforms the point's

position into screen-space coordinates before passing it to `RasterizerImp::rasterize_point()`.

The function `RasterizerImp::rasterize_point()` is responsible for actually drawing the point. In this assignment we define screen space for an output image of size `(target_w, target_h)` as follows:

- `(0, 0)` corresponds to the top-left of the output image
- `(target_w, target_h)` corresponds to the bottom-right of the output image
- **Please assume that screen sample positions are located at half-integer coordinates in screen space. That is, the top-left sample point is at coordinate (0.5, 0.5), and the bottom-right sample point is at coordinate (target_w-0.5, target_h-0.5).**



You may also wish to read this resource for more detail on how colors and images are represented as data.

To rasterize points, we adopt the following rule: a point covers at most one screen sample: the closest sample to the point in screen space. This is implemented as follows, assuming (x, y) is the screen-space location of a point.

```
int sx = (int) floor(x);
int sy = (int) floor(y);
```

Of course, the code should not attempt to modify the render target buffer at invalid pixel locations.

```
if ( sx < 0 || sx >= target_w ) return;
if ( sy < 0 || sy >= target_h ) return;
```

If the points happen to be on screen, we fill in the pixel with the RGB color associated with the point.

```
  rgb_framebuffer_target[3 * (y * width + x)] = (unsigned char)(c
  rgb_framebuffer_target[3 * (y * width + x) + 1] = (unsigned cha
  rgb_framebuffer_target[3 * (y * width + x) + 2] = (unsigned cha
```

(Note: In this assignment, we do not support partial transparency or alpha blending, even though this is part of the SVG file format.)

Homework Tasks

Task 1: Drawing Single-Color Triangles (20 pts)

Relevant lecture: 2

In this task, you will implement the `rasterize_triangle` function in `rasterizer.cpp`. Your solution should:

- Rasterize the triangle by using the sampling methods described in class.
- For each pixel, please perform the point-in-triangle tests with a sample point in the center of the pixel, not the corner. The coordinates of your sample should be equal to an integer point plus (.5,.5).

- In Part 2 you will implement sub-pixel supersampling, but here you should just sample once per pixel and call the `fill_pixel()` helper function. Follow the example in the `rasterize_point` function in the starter code.
- To receive full credit, your implementation should assume that a sample on the boundary of the triangle is to be drawn. You are encouraged but not required to implement the OpenGL edge rules for samples lying exactly on an edge. Do make sure that none of your edges are left un-rasterized.
- Your implementation should be at least as efficient as sampling only within the bounding box of the triangle (not simply every pixel in the framebuffer).
- Your code should draw the triangle regardless of the winding order of the vertices (i.e. clockwise or counter-clockwise). Check `svg/basic/test6.svg`.

When finished, you should be able to render test SVG files with single-color polygons, which are triangulated into triangles elsewhere in the code before being passed to your function. Files `basic/test3.svg`, `basic/test4.svg`, `basic/test5.svg`, and `basic/test6.svg` should render correctly.

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

1. `RasterizerImp::rasterize_triangle()` function in `rasterizer.cpp`.

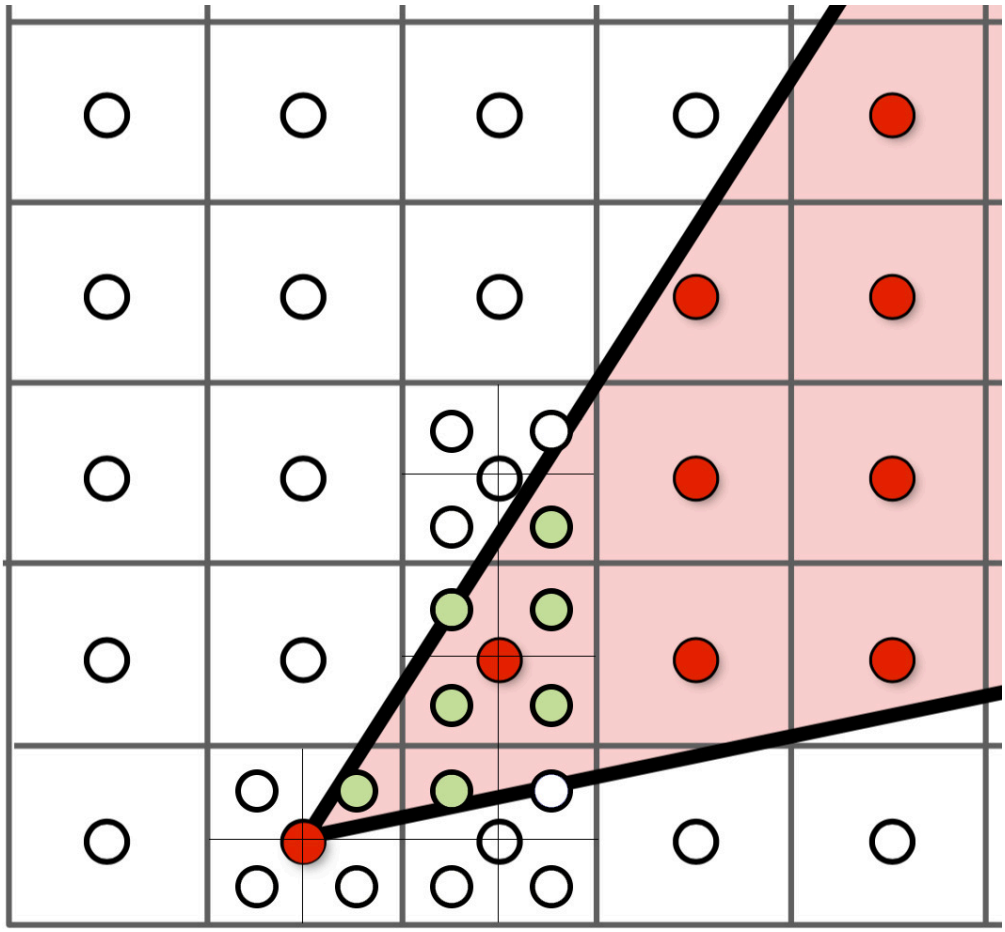
Extra Credit: Make your triangle rasterizer super fast (e.g., by factoring redundant arithmetic operations out of loops, minimizing memory access, and not checking every sample in the bounding box). Write about the optimizations you used. Use `clock()` or `std::chrono::high_resolution_clock` to get precise timing comparisons between your basic and optimized implementations.

Task 2: Antialiasing by Supersampling (20 pts)

Relevant lecture: 3

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. The fraction of the supersamples within the triangle yields a smoother edge.



To implement supersampling, please sample at `sqrt(sample_rate) * sqrt(sample_rate)` grid locations distributed over the pixel area. (sample_rate is a member variable of the RasterizerImp class)

One reasonable way to think about supersampling is simply rasterizing an image that is higher resolution, then downsampling the higher resolution image to the output resolution of the framebuffer.

The original fill_pixel function used in Task 1 directly draws onto the framebuffer, but for supersampling, you should draw into the sample_buffer first, filling all the subsamples corresponding to the output pixel.

To reiterate the overall pipeline of the rasterizer:

1. SVGParser parses the svg file into SVG class representation.
2. When rasterization starts, the renderer (`DrawRend::redraw`) calls SVG::draw.
3. `SVG::draw` calls the specific line / triangle / point rasterization functions to generate the image primitive by primitive.
4. `DrawRend::redraw` calls line rasterization to draw the square boundary.
5. `DrawRend::redraw` calls `RasterizerImp::resolve_to_framebuffer()` to translate the internal buffer of the rasterizer to the screenbuffer so the image can be displayed and written into a file.

Suggestions for this task:

- You will need to manage appropriate memory to store your supersample data. We recommend that you use the `RasterizerImp::sample_buffer` vector (see file `rasterizer.h`) for this purpose. It depends on your algorithm, but it is likely that the size of the sample buffer you need will depend on the framebuffer dimensions (which changes when the window is resized) and the supersampling rate (which changes with keystrokes as described above). You will need to update the size of the buffer dynamically. There are hints below and in the code for where you may want to manage the size of your buffer.
- Clear the values in your sample buffer memory and/or framebuffer appropriately at the beginning of redrawing the frame. This is erasing the frame before you start drawing.
- Update your `rasterize_triangle` function to perform supersampling into your supersample buffer memory. There are multiple ways to organize the data in your supersample buffer, and the choice is up to you.
    - Based on the above way to think about supersampling, your sample buffer is just a temporary, higher-resolution framebuffer. For example, 4x4 supersampling with a 1000x1000 pixel framebuffer means rasterizing a 4000x4000 (high-res) image of the scene into your sample buffer. After you rasterize the high-res image, you need to downsample to 1000x1000 final pixels by averaging down the 4x4 grid of

sample values that are related to each output pixel. In this way of thinking, you need to store more memory in order to perform the high-res supersampled rasterization. (Test your understanding: can you achieve the same results without needing more memory, and if so, what are the engineering tradeoffs?)

- At the end of rasterizing all the scene elements, you will need to populate the framebuffer from your supersamples. This is sometimes called resolving the samples into the framebuffer. Notice that the `RasterizerImp::resolve_to_framebuffer` function is called as the last step in rendering the frame **in** `drawrend.cpp`, so you may wish to implement this part of your algorithm here.
- Note that you will need to convert between different color datatypes. `RasterizerImp::rgb_framebuffer_target` stores a pointer to the framebuffer pixel data that is finally drawn to the display. `rgb_framebuffer_target` is an array of 8-bit values for each of the R, G and B components of each pixel's color -- this is the compact data format expected by most real graphics systems for drawing to the display. In contrast, the `RasterizerImp::sample_buffer` variable that we suggest you use for your supersample memory is an array of `Color` objects that store R, G and B internally as floating point values. You may wish to familiarize yourself with the `Color` class. You may need to convert between these datatypes. Watch out for floating point to integer conversion errors, such as rounding and overflow.
- In most of graphics field, color is just the same as any other vector data: They are 3 dimensional (4 if it has the alpha channel). This means any XYZ vector representing a point or a direction could also represent a color. We will cover color space and color theory later in this course.
- You will likely find that points and lines stop rendering correctly after your supersampling modifications. Lines and points are not supersampled, but they still need to be drawn into the supersample buffer. Modify `RasterizerImp::fill_pixel` if needed to restore functionality. One way to think about this is to fill all the supersamples corresponding to the point or line with the same color, so it comes out as a single sampled pixel in the framebuffer. You do NOT need to antialias points and lines.

Once your implementation is complete, your triangle edges should look noticeably smoother when using more than one sample per pixel! You can examine the differences closely using the pixel inspector (see controls listed above). 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 likely want to use or modify.

1. For managing supersample buffer memory:
   `RasterizerImp::sample_buffer`,
   `RasterizerImp::set_sample_rate()`,
   `RasterizerImp::set_framebuffer_target()`,
   `RasterizerImp::clear_buffers()` in `rasterizer.h/cpp`.
2. To implement triangle supersampling:
   `RasterizerImp::rasterize_triangle()`,
   `RasterizerImp::fill_pixel()`, in `rasterizer.cpp`.
3. For resolving supersamples to framebuffer:
   `RasterizerImp::resolve_to_framebuffer()`.

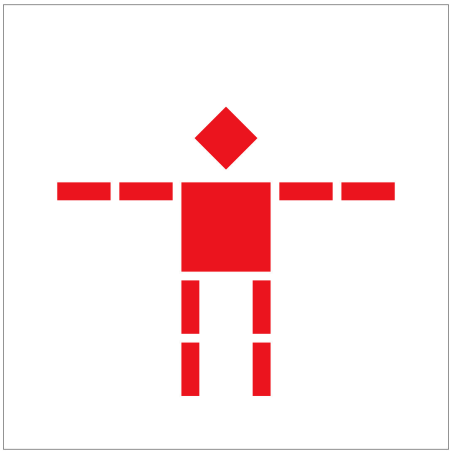
Extra Credit: Implement an alternative sampling pattern, such as jittered or low-discrepancy sampling. Create comparison images showing the differences between grid supersampling and your new pattern. Try making a scene that contains aliasing artifacts when rendered using grid supersampling but not when using your pattern.

Task 3: Transforms (10 pts)

Relevant lecture: 4

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.

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



For convenience, here is a list of functions in *transforms.cpp* you will need to modify:

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

Extra Credit: Add an extra feature to the GUI. For example, you could make two unused keys to rotate the viewport. Save an example image to demonstrate your feature, and write about how you modified the SVG to NDC and NDC to screen-space matrix stack to implement it.

Task 4: Barycentric coordinates (10 pts)

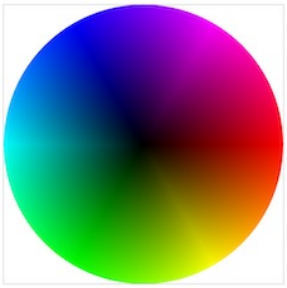
Relevant lecture: 5

Implement `RasterizerImp::rasterize_interpolated_color_triangle(...)` to draw a triangle with colors defined at the vertices and interpolated across the triangle area using barycentric interpolation.

Once Part 4 is done, you should be able to see a color wheel in *svg/basic/test7.svg* (below, right).

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

1. `RasterizerImp::rasterize_interpolated_color_triangle(...)`



Task 5: "Pixel sampling" for texture mapping (15 pts)

Relevant lecture: 5

Implement `RasterizerImp::rasterize_textured_triangle(...)` to draw a triangle with colors defined by texture mapping with the given 2D texture coordinates at each vertex and the given `Texture` image. Here in Task 5 you will implement texture sampling on the full-resolution texture image using nearest neighbor and bilinear interpolation, as described in lecture.

The GUI toggles `RasterizerImp`'s `PixelSampleMethod` variable `psm` using the `'P'` key. When `psm == P_NEAREST`, you should use nearest-pixel sampling, and when `psm == P_LINEAR`, you should use bilinear sampling. Please do so by implementing `Texture::sample_nearest` and `Texture::sample_bilinear` functions and calling them from `RasterizerImp::rasterize_textured_triangle(...)`. This approach will allow you to reuse these functions for trilinear texture filtering in Task 6.

Once Part 5 is done, you should be able to rasterize the *svg* files in *svg/texmap/*, which rely on texture maps.

Notes:

- The `Texture` struct in `texture.h` stores a mipmap, as described in lecture, of texture images in decreasing resolution, in the `mipmap` variable. Each texture image is stored as an object of type `MipLevel`.
- `MipLevel::texels` stores the texture image pixels in the typical RGB format described above for framebuffer pixels.
- `MipLevel::get_texel(...)` may be helpful.

- At this part of the homework, you haven't implemented level sampling (mip-mapping) yet, so the program should default to zero-th level (full resolution).

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

1. `RasterizerImp::rasterize_textured_triangle`
2. `Texture::sample_nearest`
3. `Texture::sample_bilinear`

Task 6: "Level sampling" with mipmaps for texture mapping (25 pts)

Relevant lecture: 5

Finally, update `RasterizerImp::rasterize_textured_triangle(...)` to support sampling different mipmap levels (`MipLevel` s). The GUI toggles `RasterizerImp` 's `LevelSampleMethod` variable `lsm` using the `L` key. **Please implement the following level sampling methods in the helper function `Texture::sample` .**

- When `lsm == L_ZERO` , you should sample from the zero-th `MipLevel` , as in Part 5.
- When `lsm == L_NEAREST` , you should compute the nearest appropriate mipmap level and pass that level as a parameter to the nearest or bilinear sample function.
- When `lsm == L_LINEAR` , you should compute the mipmap level as a continuous number. Then compute a weighted sum of using one sample from each of the adjacent mipmap levels as described in lecture.

In addition, implement `Texture::get_level` as a helper function. You will need $\left(\frac{du}{dx}, \frac{dv}{dx}\right)$ and $\left(\frac{du}{dy}, \frac{dv}{dy}\right)$ to calculate the correct mipmap level. In order to get these values corresponding to a point (x, y) inside a triangle, you must perform the following.

1. Calculate the *uv* barycentric coordinates of (x, y), $(x+1, y)$, and $(x, y+1)$ in `rasterize_textured_triangle(...)` as `sp.p_uv` , `sp.p_dx_uv` , and `sp.p_dy_uv` , assign them to a `SampleParams`

struct `sp`, along with other values required by the struct, and pass `sp` to `Texture::get_level`
2. Calculate the difference vectors `sp.p_dx_uv - sp.p_uv` and `sp.p_dy_uv - sp.p_uv` inside `Texture::get_level`, and finally
3. Scale up the difference vectors **accordingly** by the width and height of the full-resolution texture image.

With these, you can proceed with the calculation from the lecture slides.

Notes:

- The `lsm` and `psm` variables can be set independently and interacted independently. In other words, all combinations of `psm==[P_NEAREST, P_LINEAR] x lsm==[L_ZERO, L_NEAREST, L_LINEAR]` are valid.
- When `lsm == L_LINEAR` and `psm == P_LINEAR`, this is known as trilinear sampling, or trilinear texture filtering, as described in lecture.
- You may find it helpful to visualize what parts of the image use different levels of the mipmap. One way to do this is by normalizing the value returned by `Texture::get_level` by the maximum level (i.e. size of the mipmap) and have that value returned by `Texture::sample` as a color. Zoom in and out of the image to see how the levels change. This is a great way to both debug your implementation as well as gain intuition about level sampling! See below for two examples, where we zoom out/in to illustrate how the computed levels change.
- Please be careful do not make copies of an entire Miplevel. Make sure you always use a pointer or a reference to access the miplevel. Copying entire miplevels as arguments is extremely slow!

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

1. `RasterizerImp::rasterize_textured_triangle`
2. `Texture::sample`
3. `Texture::get_level`

Extra Credit: Implement anisotropic filtering or summed area tables. Show comparisons of your method to nearest, bilinear, and trilinear sampling. Use `clock()` to measure the relative performance of the methods.

Potential Extra Credit - Draw Something Creative!

Use your newfound powers to render something fun. You can look up the svg specifications online for matrix transforms and for Point, Line, Polyline, Rect, Polygon, and Group classes. The ColorTri and TexTri are our own inventions, so you can intuit their parameters by looking at the svgparser.cpp file. You can either try to draw something "by hand" or try to output an interesting pattern programmatically. For example, we wrote some simple programs to generate the texture mapped *svg* files in the *texmap* directory as well as the color wheel in *basic/test7.svg*.

Flex your right or left brain -- either show us your artistic side, or generate awesome procedural patterns with code. This could involve a lot of programming either inside or outside of the codebase! If you write a script to generate procedural *svg* files, include it in your submission and briefly explain how it works.

Tips and guidelines for your submission:

- Your resulting *png* screenshot should be 800x800 resolution. Keep this in mind when writing your *svg* file.
- Use the GUI's `'S'` functionality to save your screenshot as a *png*. Don't take your own screenshot of your rasterized result, or you'll ruin the quality of your hard work!
- Note: The rasterizer cannot display svg Path elements, so do not include any curves in the svg you wish to load.

Students will vote on their favorite submissions and the top voted submission(s) will receive extra credit! More details regarding the art competition will be announced next week. Stay in tune!

Submission

Please consult this article on how to submit the assignment.

You will submit your code as well as some deliverables (see below) in a webpage write-up.

Homework write-up guidelines and instructions

We have provided a simple HTML skeleton in *index.html* found within the *docs* directory to help you get started and structure your write-up.

You are also welcome to create your own webpage report from scratch using your own preferred frameworks or tools. However, **please follow the same overall structure as described in the deliverables section below**.

The goals of your write-up are for you to (a) think about and articulate what you've built and learned in your own words, (b) have a write-up of the homework to take away from the class. Your write-up should include:

- An overview of the homework, your approach to and implementation for each of the parts, and what problems you encountered and how you solved them. Strive for clarity and succinctness.
- On each part, make sure to include the results described in the corresponding Deliverables section in addition to your explanation. If you failed to generate any results correctly, provide a brief explanation of why.
- The final (optional) part for the art competition is where you have the opportunity to be creative and individual, so be sure to provide a good description of what you were going for and how you implemented it.
- Clearly indicate any extra credit items you completed, and provide a thorough explanation and illustration for each of them.

The write-up is one of our main methods of evaluating your work, so it is important to spend the time to do it correctly and thoroughly. Plan ahead to allocate time for the write-up well before the deadline.

Write-up Deliverables and Rubric

Note that this rubric is rather coarse. The content and quality of your write-up are extremely important, and you should make sure to at *least* address all the points listed below. The extra credit portions are intended for students who want to challenge themselves and explore methods beyond the fundamentals, and are not worth a large amount of points. In other words, don't necessarily expect to use the extra credit points on these homeworks to make up for lost points elsewhere.

Overview

Give a high-level overview of what you implemented in this homework. Think about what you've built as a whole. Share your thoughts on what interesting things you've learned from completing the homework.

Task 1 (20 pts)

- Walk through how you rasterize triangles in your own words.
- Explain how your algorithm is no worse than one that checks each sample within the bounding box of the triangle.
- Show a *png* screenshot of *basic/test4.svg* with the default viewing parameters and with the pixel inspector centered on an interesting part of the scene.
- *Extra credit:* Explain any special optimizations you did beyond simple bounding box triangle rasterization, with a timing comparison table (we suggest using the c++ `clock()` function around the `svg.draw()` command in `DrawRend::redraw()` to compare millisecond timings with your various optimizations off and on).

Task 2 (20 pts)

- Walk through your supersampling algorithm and data structures. Why is supersampling useful? What modifications did you make to

the rasterization pipeline in the process? Explain how you used supersampling to antialias your triangles.
- Show *png* screenshots of *basic/test4.svg* with the default viewing parameters and sample rates 1, 4, and 16 to compare them side-by-side. Position the pixel inspector over an area that showcases the effect dramatically; for example, a very skinny triangle corner. Explain why these results are observed.
- *Extra credit:* If you implemented alternative antialiasing methods, describe them and include comparison pictures demonstrating the difference between your method and grid-based supersampling.

Task 3 (10 pts)

- Create an updated version of *svg/transforms/robot.svg* with cubeman doing something more interesting, like waving or running. Feel free to change his colors or proportions to suit your creativity. Save your *svg* file as *my_robot.svg* in your *docs/* directory and show a *png* screenshot of your rendered drawing in your write-up. Explain what you were trying to do with cubeman in words.

Task 4 (10 pts)

- Explain barycentric coordinates in your own words and use an image to aid you in your explanation. One idea is to use a *svg* file that plots a single triangle with one red, one green, and one blue vertex, which should produce a smoothly blended color triangle.
- Show a *png* screenshot of *svg/basic/test7.svg* with default viewing parameters and sample rate 1. If you make any additional images with color gradients, include them.

Task 5 (15 pts)

- Explain pixel sampling in your own words and describe how you implemented it to perform texture mapping. Briefly discuss the two different pixel sampling methods, nearest and bilinear.
- Check out the *svg* files in the *svg/texmap/* directory. Use the pixel inspector to find a good example of where bilinear sampling clearly defeats nearest sampling. Show and compare four *png*

screenshots using nearest sampling at 1 sample per pixel, nearest sampling at 16 samples per pixel, bilinear sampling at 1 sample per pixel, and bilinear sampling at 16 samples per pixel.
- Comment on the relative differences. Discuss when there will be a large difference between the two methods and why.

Task 6 (25 pts)

- Explain level sampling in your own words and describe how you implemented it for texture mapping.
- You can now adjust your sampling technique by selecting pixel sampling, level sampling, or the number of samples per pixel. Describe the tradeoffs between speed, memory usage, and antialiasing power between the three various techniques.
- Using a *png* file you find yourself, show us four versions of the image, using the combinations of `L_ZERO` and `P_NEAREST`, `L_ZERO` and `P_LINEAR`, `L_NEAREST` and `P_NEAREST`, as well as `L_NEAREST` and `P_LINEAR`.
    - To use your own *png*, make a copy of one of the existing *svg* files in *svg/texmap/* (or create your own modelled after one of the provided *svg* files). Then, near the top of the file, change the texture filename to point to your own *png*. From there, you can run ./draw and pass in that svg file to render it and then save a screenshot of your results.
    - **Note**: Choose a *png* that showcases the different sampling effects well. You may also want to zoom in/out, use the pixel inspector, etc. to demonstrate the differences.
- *Extra credit:* If you implemented any extra filtering methods, describe them and show comparisons between your results with the other above methods.

(Optional) Potential Extra Credit

- Save your best *svg* file as *competition.svg* in your *docs/* directory, and show us a 800x800 *png* screenshot of it in your write-up!
- Explain how you did it. If you wrote a script to generate procedural *svg* files, include it in your submission in the *src/* directory and briefly explain how it works.

Website tips and advice

- Please include a link to your webpage at the top of your writeup
- Note that only *one* webpage is needed for both partners
- Be sure to include and turn in all of the other files (such as images) that are linked in your report!
- Use only *relative* paths to files, such as `"./images/image.jpg"`
- Do *NOT* use absolute paths, such as `"/Users/student/Desktop/image.jpg"`
- Pay close attention to your filename extensions. Remember that on UNIX systems (such as the instructional machines), capitalization matters. `.png != .jpeg != .jpg != .JPG`
- Be sure to adjust the permissions on your files so that they are world readable. For more information on this please see this tutorial.
- Start assembling your webpage early to make sure you have a handle on how to edit the HTML code to insert images and format sections. (Or you can use Markdown)

Feedback Form

Please fill out this feedback form if you would like to share any feedback on this homework, the homework parties, or the course in general.