CSC4140 Assignment III

Computer Graphics

February 24, 2023

MiddleTerm: Rasterization

This assignment is 18%(with 5.4 extra credit of the total mark.

The maximum you can obtain is still 18, grades will be clipped if you beyond 18.

Strict Due Date: 11:59PM, Mar 12^{th}, 2022

Student ID:

Student Name:

This assignment represents my own work in accordance with University regulations.

Signature:

Overview

In the last assignment, you have make a great achievement to move a model to screen. This time, you will dig deeply into the rasterization! Basically, this project has 6 tasks, worth a total 130 points (30 extra points can be added if you completed a fancy additional work). The tasks are as follows:

- Draw a single color triangles (20 points + 10 extra)
- Antialiasing (20 points + 10 extra)
- Transforms (10 points)
- Barycentric coordinate (10 points)
- Barycentric coordinate (10 points)
- Texture mapping (10 points)
- Mipmapping(30 points) (10 extra included if you complete well)

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 project to take away from the class. Your write-up should include:

- An overview of the project, 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.

Note, before you start to do assignment, install OpenGL using the following command:

```
$sudo apt update
$sudo apt install freeglut3-dev
$sudo apt install libfreetype6-dev
$sudo apt install xorg-dev libgl1-mesa-dev
```

After compiling, you will get a "*draw*" executable file. use it to run the given *.svg* files.

1 Draw a single color triangles (20 points)

Implement the function *rasterize_triangle* function in *rasterizer.cpp*.

You should:

- For each pixel, perform the point-in-triangle tests with a sample point in the **center** of the pixel. Sample coordinates = the integer point + (.5,.5).
- In task 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.
- Your implementation should assume that a sample on the boundary of the triangle is to be drawn. 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. Complete the given function in *rasterizer.cpp*.

```
RasterizerImp::rasterize_triangle()
```

2 Antialiasing (20 points)

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.

Figure 2 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.

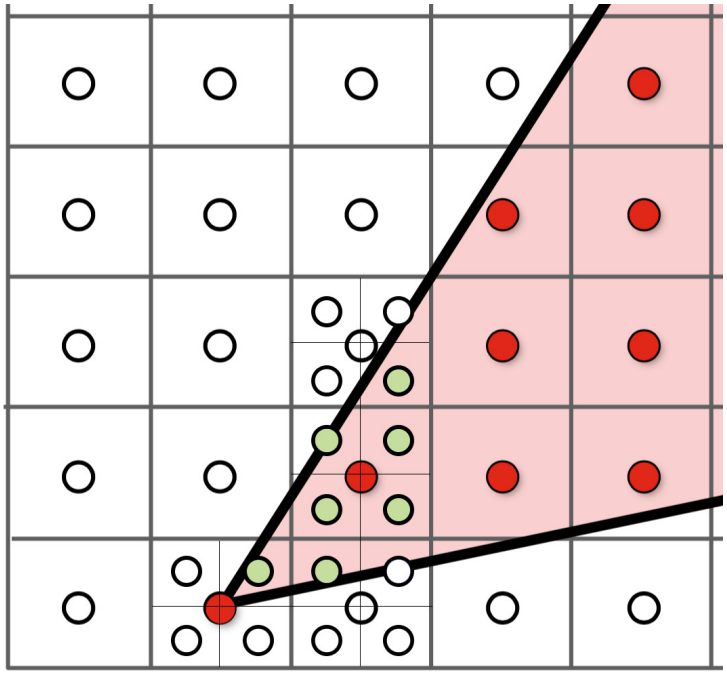

Figure 1: Supersampling

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:

- $SVGParser$ parses the svg file into SVG class representation.
- When rasterization starts, the renderer ($DrawRend :: redraw$) calls $SVG :: draw$.
- $SVG :: draw$ calls the specific line / triangle / point rasterization functions to generate the image primitive by primitive.
- $DrawRend :: redraw$ calls line rasterization to draw the square boundary.
- $DrawRend :: redraw$ 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.

To manage the memory for supersampled data, 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. (If you implement it with no extra memory, you will get 10 more points. Memory consumed no more than the size of the image. Explain what you did and what's the trade-off in your report).
- 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.
- 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.

Complete or use the given functions.

```
\\For managing supersample buffer memory
\\in  rasterizer.h  and  rasterizer.cpp
```

```
RasterizerImp::rasterize_triangle()
RasterizerImp::set_sample_rate()
RasterizerImp::set_framebuffer_target()
RasterizerImp::clear_buffers()

\\To implement triangle supersampling  rasterizer.cpp
RasterizerImp::rasterize_triangle()
RasterizerImp::fill_pixel()

\\For resolving supersamples to framebuffer
RasterizerImp::resolve_to_framebuffer()
```

3 Transforms (10 points)

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:

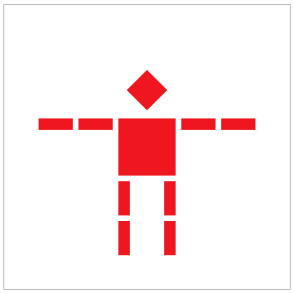

Figure 2: Texture map (left) and Transform(right)

Complete the given functions.

```
translate()
scale()
rotate()
```

4 Barycentric coordinates (10 points)

Implement *RasterizerImp* :: *rasterize*$_i$*nterpolated_color_triangle*(...) to draw a triangle with colors defined at the vertices and interpolated across the triangle area using barycentric interpolation.

Once done, you should be able to see a color wheel in *svg/basic/test*7.*svg*(*below*, *right*).

Complete the given functions.

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

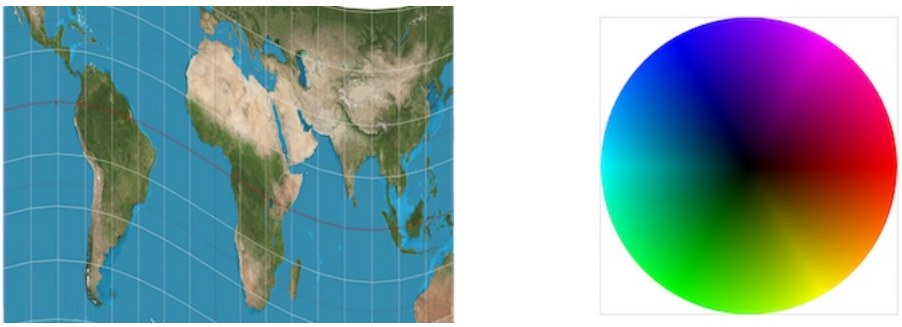

Figure 3: transform

5 Pixel sampling for texture mapping (10 points)

Implement *RasterizerImp* :: *rasterize_textured*$_t$*riangle*(...) 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 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*$_n$*earest* and *Texture* :: *sample*$_b$*ilinear* functions and calling them from *RasterizerImp* :: *rasterize_textured*$_t$*riangle*(...). This approach will allow you to reuse these functions for trilinear texture filtering in Task 6.

Now you should be able to rasterize the svg files in *svg/texmap/*, which rely on texture maps.

Hints:

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

Complete the given functions.

```
1 RasterizerImp::rasterize_textured_triangle()
2 Texture::sample_nearest()
3 Texture::sample_bilinear()
```

6 Mipmapping (20 Points)

Continue with the last task. 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. 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 $(\frac{du}{dx}, \frac{dv}{dx})$and $(\frac{du}{dy}, \frac{dv}{dy})$ to calculate the correct mipmap level. In order to get these values corresponding to a point (x,y)(x,y) inside a triangle, you must perform the following.

- Calculate the uv barycentric coordinates of (x,y)(x,y), (x+1,y)(x+1,y), and (x,y+1)(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*
- 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

- 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 $psm == [P\_NEAREST, P_LINEAR] \times lsm == [L_ZERO, L\_NEAREST, L\_LINE$ 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.
- 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!

Complete the given functions.

```
1 RasterizerImp::rasterize_textured_triangle()
2 Texture::sample()
3 Texture::get_level()
```

7 Your Submissions

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 projects to make up for lost points elsewhere.

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

Task 1 (20 pts + 10 extra) 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: Draw a triangle "A" over another triangle "B" (both with alpha channel) using z-buffer (10 pts)

Task 2 (20 pts + 10 extra) 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:
1. If you implemented alternative antialiasing methods, describe them and include comparison pictures demonstrating the difference between your method and grid-based supersampling. (5 pts)
2. When implement supersampling, reduce the memory consumption to the same level w/o supersampling. (5 pts)

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 (10 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 (30 pts, 10 extra included) 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.