CS 184: Computer Graphics and Imaging, Spring 2019

Project 1: Rasterizer

Jireh Wei En Chew, CS184-agu

Overview

We've basically created all the functions to render a simple scene, both from the geometric perspective of manipulation objects with transformations and also using data, by applying textures to triangles, and some combination of them. Furthermore, we implement some techniques used to combat some of the artifacting that occurs due to sampling.

In essence, we've created the foundation and basis for everything else we will be doing throughout the course. One particular thing that I learnt is visual debugging, where the image appearance needs to be interpreted and used to determine where the mistake would be in code. For example, if the image appeared to be darker or brighter, then I would suspect that the math behind how I assigned colours to pixels was slightly off. If parts of the image seemed to be in the wrong position, then I would look at my texture indexing or my width/height/x/y values to see if I had messed them up.

Section I: Rasterization

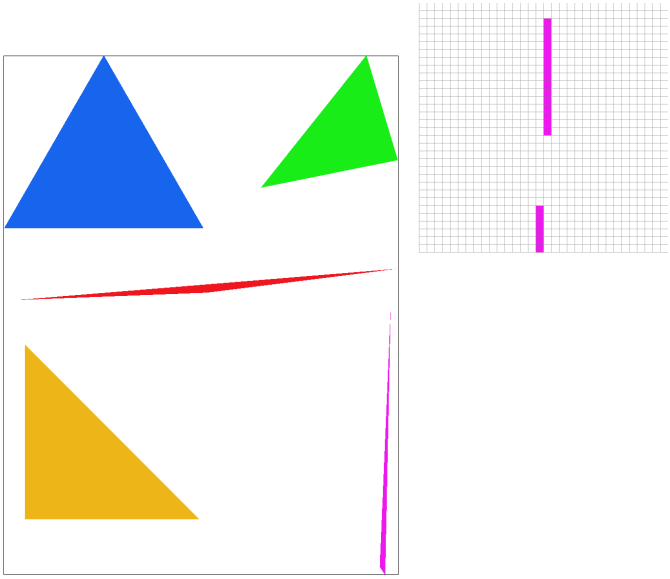
Part 1: Rasterizing single-color triangles

In DrawRend::rasterize_triangle, we are given a triangle's 3 vertices and the color of the triangle. We use this information to check on screen space, where we should be colouring our pixels. We use the centre of the pixel as the location to check whether a pixel shoul be coloured or not. For this, we use the equation of the 3 lines making up the triangle. The following lists the steps I took to rasterize the triangle.

1. Find the bounding box of the triangle by finding the minimum and maximum x and y coordinates from the 3 vertices.
2. Set up a double for loop to iterate over the pixels in the bounding box.
3. Using a created function DrawRend::PointInTriangleTest, test if the point is in the triangle.
   More specifically:
   - Calculate the point for which we want to test. For no supersampling, this is the middle of the pixel, so we add 0.5 to both the row and column of the current pixel we are checking.
   - Clculate the normal of each of the three lines. We assume that the points are given in clockwise fashion.
   - Calculate the dot product of the normal vector and the given point. If the result is negative, this means that the angle between the normal and the point is more than 90 degrees and thus outside of the line.
   - If all of the dot products are positive or 0, this means that the point is in the triangle or on one of the lines.
   - There is a another case to consider. If the dot products are all negative, this also means that we have assumed wrong and the points are wounded counterclockwise instead.
   - Thus, we check for when the dot products are either all positive or all negative.
4. If the point is inside, we colour it with the given colour.

This method checks all pixels within the bounding box, since we use the vertices to find the maximum and minimum coordinates. These coordinates form a bounding box around the triangle and thus we only check all of these pixels.

Interesting Photo of Test 4



This triangle appears to be broken, where in fact part of the triangle is still in the white areas, but due to checking at the centre of th pixel, the pixel is not coloured.

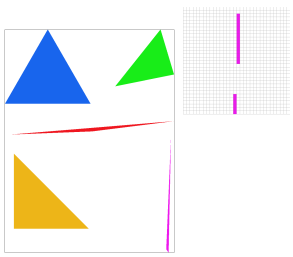
Part 2: Antialiasing triangles

In part 2, instead of adding 0.5 to each pixel row and column, we need to check every subpixel.

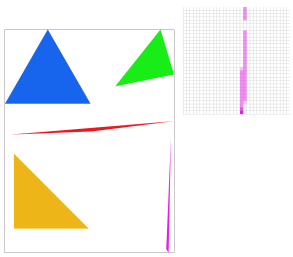
1. We create 2 more for loops, to loop over the subpixels row and column.
2. For each inner loop, we calculate the middle of the subpixel with this formula:
    - Subpixel X position centre = column + (subpixel x position)/(samples per side) + 1/(2 * samples per side)
    - Subpixel Y position centre = row + (subpixel y position)/(samples per side) + 1/(2 * samples per side)
3. This subpixel centre then gets tested with PointInTriangleTest.
4. If the subpixel is inside, we colour it with the given color using fill_color(subpixel y position, subpixel x position)
5. When the scene is drawn, the pixel will take all its subpixels and get an average colour of that pixel over all subpixels.

Supersampling is useful as it creates a smoother transition on the edges of the triangle. A triangle is not made up of small squares (pixels) and thus without supersampling, we get pixels with only 2 colours. If we use supersampling, this creates pixels with colours that are a blend between white and the triangle color. These intermediate pixels trick our eyes into thinking that the edge is a continuous line rather than made up of individual squares of pixels.

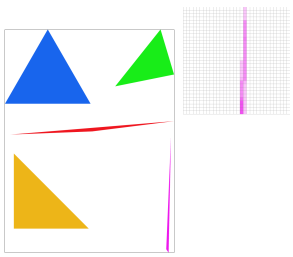
Comparison between 1x, 4x and 16x sampling



1x (no supersampling)



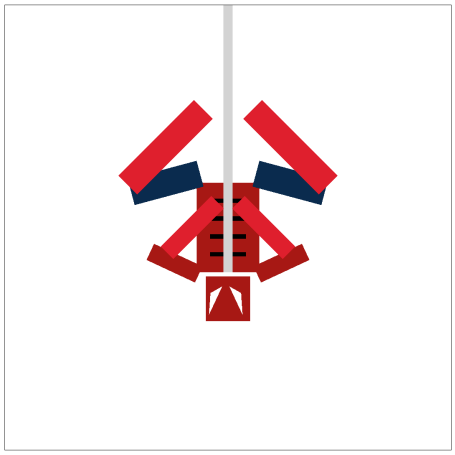
4x supersampling



16x supersampling

From 1x to 16x, we see get the "broken-ness" of the triangle decreases. When we hit 16x, the subpixels tested lie in that very skinny region of the triangle, and some of the subpixels are then coloured. However, for 1x and 4x, the subpixels tested may not have been lying inside the triangle, and the skinny triangle was lying between the samples, thus the pixels in that region were not coloured.

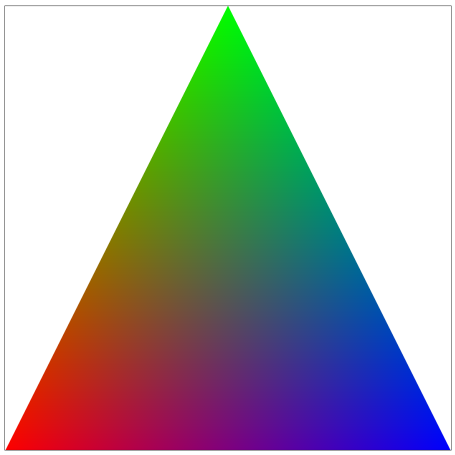
Part 3: Transforms



With great (transformative) power, comes great responsibility. - Prof Ren

Section II: Sampling

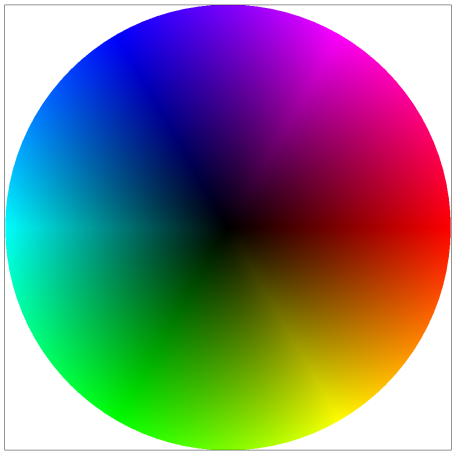
Part 4: Barycentric coordinates



A triangle created from an svg file demonstrating barycentric colouring.

Barycentric coordinates are basically another coordinate system that uses 3 values (alpha, beta and gamma) to define a point within the triangle. It is also a area-based coordinate system, where alpha, beta and gamma are the ratios of areas to the whole triangle for the 3 sub-triangles created at the given point. Since alpha, beta and gamma add up to 1, we can use them to interpolate any 3 values on the vertices for a given point on the triangle.

For example, the triangle above is drawn with very litte data. I have only given the three vertices the colour red, green and blue, and the point positions. The barycentric system is able to use this data and interpolate from red to green (giving yellow on the left side) and purple on the bottom of the triangle, as I am interpolating between the data points to provide a smooth transition from pixel to pixel.



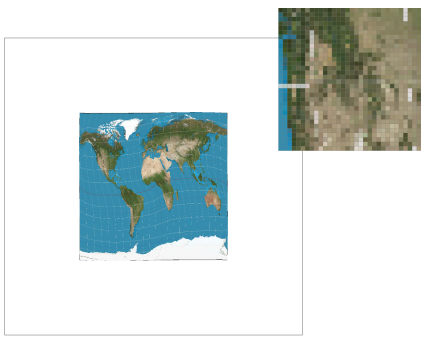
"Bary interesting..."

Part 5: "Pixel sampling" for texture mapping

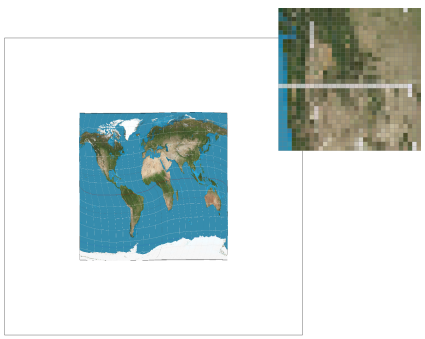
In Part 5, we had to take data from textures and use them to colour our pixels. Basically pixel sampling is instead of taking a colour and applying on the triangle, we map areas of the texture to triangles in our scene and by using barycentric interpolation, find the appropriate pixel of the texture to sample from.

The implementation of texture sampling was implemented as such:

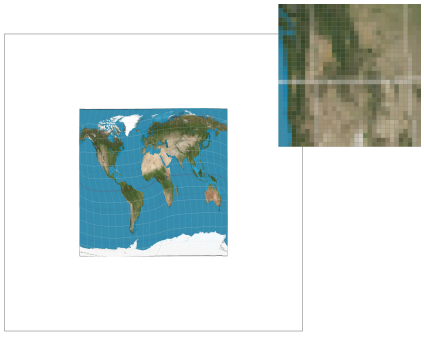
1. Calculate the barycentric coordinates of the point relative to the 3 vertices given.
2. Pass this data to the ColorTri:color function. Since this function knows the coordinates of the 3 points in uv space that map to t points in screen space, we use these three points and the barycentric coordinates to find the uv values of the given point.
3. We then use these uv values to find which pixel on the texture we should use for the colour of the pixel or subpixel we are colouring. This method can either use the nearest neighbour method or bilinear interpolation.
    1. Nearest neighbour method looks at the barycentric derived uv values, rounds them to the nearest pixel, and takes that pixel's value.
    2. Bilinear interpolation takes this a bit more seriously. Instead of taking the nearest pixel, it interpolates between the 4 neighbour pixel colour values using the value of the uv coordinates. If the uv values were 100.5 and 100.5, it would linearl interpolate between the colours of the (100,100), (101, 100), (100, 101) and (101, 101)th pixel, using the decimal value of 0 to interpolation between them.
4. Bilinear would definitely give more defintion to the image, as we are not treating the texture as discrete samples of data, but instead attempt to smooth out colours in between pixels using linear interpolation.



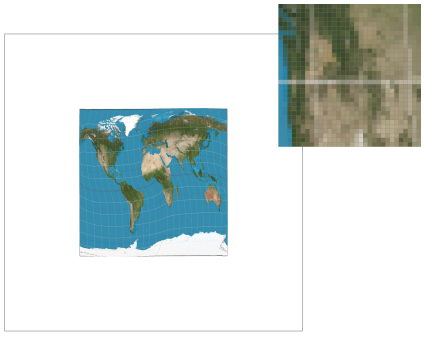
Nearest pixel, 1x sampling.



Bilinear interpolation, 1x sampling.



Nearest pixel, 16x sampling.



Bilinear interpolation, 16x sampling.

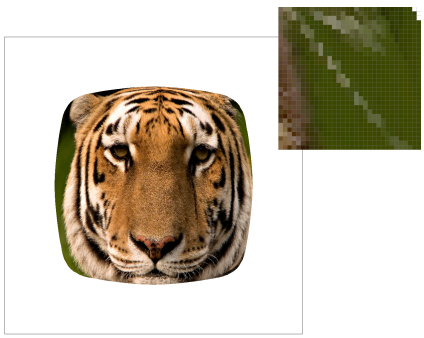
The largest difference is from nearest neighbour sampling to bilinear interpolation with no supersampling. It seems the pixels were lying nearer to the pixels of the map rather of the pixels of the gridline, and thus a large part of the gridline was ignored when using nearest neigbour. However, when we use bilinear interpolation, that takes into account the other neighbours pixels with white (representing the gridline colour) and thus the pixels have a partial amount of white that is rendered. Moving towards 16x supersampling just reduces the jaggies and gives the continents a smoothing transition of colour from pixel to pixel.

Part 6: "Level sampling" with mipmaps for texture mapping

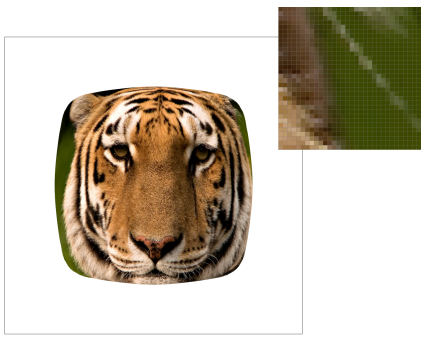
Level sampling is a method used to counteract the aliasing that happens when a scene includes both near and far objects that use textures. The further objects have a smaller screen footprint, and thus when sampling the full, high resolution texure, aliasing occurs In order to combat this, we precompute different versions of the textures from high to low resolution and select the best texture resolution for the pixel at the current point. Thus, we are able to avoid the aliasing for objects further away and yet still have high definition textures for objects closer to the camera.

Implementation of level sampling:

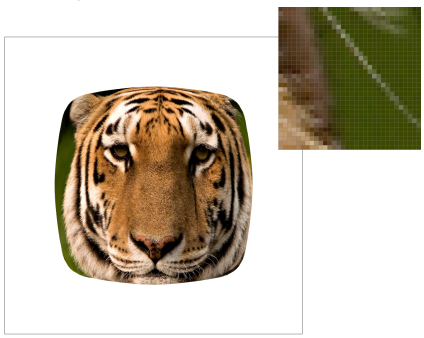
1. Besides calculating the barycentric coordinates of the current pixel, I also calculate the neighbouring pixel's barycentric coordinates.
2. With these coordinates, I can then calculate the relative shift from the corresponding texel to texel on the texture.
3. These distances tell me how far apart is each pixel from each other on the texture. A shorter distance requires a higher definitic texture, since the changes in texture distance are finer. Using this, we then calculate the depth of the map we need to access b taking log base 2 on the distances.
4. We then select the appropriate texture map to use for sampling. If both linear texture mapping and bilinear interpolation is selected, we use the decimal value of the log result to interpolate between the mipmap at that level and the adjacent mipmap.



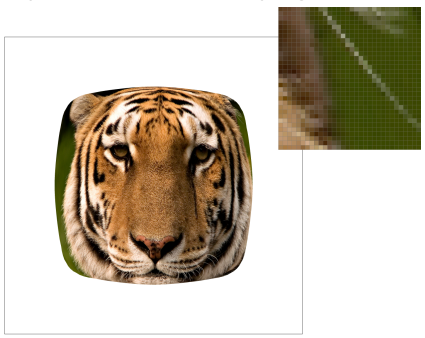
Nearest pixel, nearest level at 1x



Nearest pixel, nearest level sampling at 9x



Bilinear interpolation, nearest level sampling at 9x

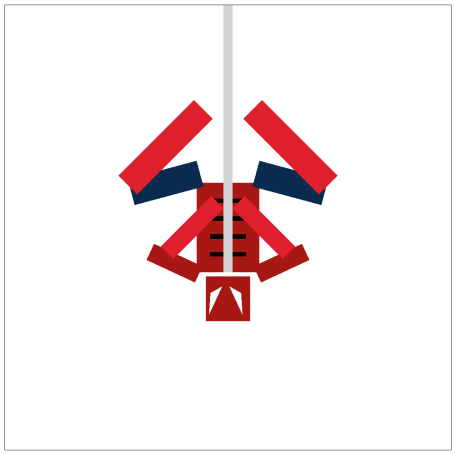


Bilinear interpolation, bilinear level sampling at 9x

It seems extra sampling takes the most time, but results in good images. However, a lot of the compute is done at runtime, and this would not be ideal for gaming or animation. As such, I feel that using mipmaps and precomputing all the mipmap levels (trading spa for time) is a good approach, so that any animation or games can run smoothly. However, nowadays, as our graphics cards become much faster and can parallelize better, I think a good middle ground can be reached using bilinear interpolation and nearest level sampling, while employing supersampling when time computation permits.

Section III: Art Competition

Part 7: Draw something interesting!



With great (transformative) power, comes great responsibility. - Prof Ren