Overview:
In all tasks of this assignment I worked towards implementing a full rasterization pipeline. Specifically, I worked on improving rasterizing triangles, color interpolation and texture interpolation. The algorithm iterates over the bounding box and tests whether each pixel center lies inside the triangle using implicit line equations. I then added in supersampling to antialias edges by sampling multiple points per pixel in an nxn grid and averaging the results into a higher-resolution buffer before resolving to the final image. For color interpolation across triangles, I used barycentric coordinates to blend vertex colors. The weights come from the proportional distance from a vertex to the opposite edge. I also implemented texture mapping by interpolating UVs at each sample with barycentrics and sampling the texture at the resulting (u, v). Pixel sampling was implemented with nearest-neighbor and bilinear methods, and sampling levels at zero, nearest, and linear to choose mipmap levels from UV derivatives.

The most interesting part was the math itself. I felt as if the concepts were fairly intuitive, and going into deriving the math behind it felt super enlightening. I feel like I have a very strong base for the rest of the class.

Task 1:

In rasterizing a triangle we are essentially finding all pixels whose center is within the bounds of a triangle defined by three coordinate points. These coordinate points are represented as (x0, y0), (x1, y1), and (x2, y2). If that pixel satisfies the condition, then we color it accordingly. Essentially, we are sampling all pixels whose center is inside the triangle and setting their color to the desired value. The workflow in my code is as follows:

1. Receive the required inputs:
    a. Points defining the triangle → (x0, y0), (x1, y1), (x2, y2)
    b. The color of the triangle → *color*
2. Get the limits of the triangles bounding box **(explained below)**
3. Iterate through all integer pairs of the bounding box → (x, y):
    a. **If** the integer pair is out of bounds **then** continue to the next loop
    b. Get the center of the pixel → (x + 0.5, y + 0.5):
    c. Check if the center is within the triangle **(explained below)**
    d. **If** the center is within the triangle **then** fill the pixel and the (x, y) position with *color*

To check if a point is within a triangle we first recognize that each pair of points (further referred to as p0, p1, and p2) produces an implicit line equation L(x, y) = Ax + By + C. With this line, we can recognize that for some point (x, y):

- On the line: L(x, y) = 0
- Above the line: L(x, y) > 0
- Below the line: L(x, y) < 0

Therefore, assuming the points are provided in clockwise or counterclockwise order, we can derive the lines p0 → p1, p1 → p2, and p2 → p0. If the point (x, y) lies above or on the line for all lines and the coordinate pairs are in counterclockwise order, then the point lies inside the triangle; if the point (x, y) lies below or on the line for all lines and the coordinate pairs are in clockwise order, then the point is inside the triangle. Otherwise, the point lies outside of the triangle.

The implemented algorithm looks only at those pixels within the bounding box of the triangle. We find those bounds by taking the minimum and maximum of the x-coordinates and y-coordinates and finding the nearest pixels by taking the floor and ceiling of the minimums and maximums, respectively. The following code snippet is used to do just that, and by using these values in our subsequent for loops, the algorithm passes the requirement of being “no worse than one that checks each sample within the bounding box of the triangle”

```
int min_x = (int)std::floor(min({x0, x1, x2}));
int max_x = (int)std::ceil(max({x0, x1, x2}));
int min_y = (int)std::floor(min({y0, y1, y2}));
int max_y = (int)std::ceil(max({y0, y1, y2}));
// Other Code
for (int x = min_x; x < max_x; x++) {
	for (int y = min_y; y < max_y; y++) {
		// Other code
	}
}
```

[Include the photo at Graphics/assignment1/Task1.png]
Screenshot of *basic/test4.svg*

Task 2:

Supersampling is the process where we take several samples per pixel instead of just the center; we sample multiple points equidistant from one another inside the pixel to get a representative subsample. With one sample per pixel, color is a binary choice and edges are either fully covered or fully uncovered. Even if a non-significant amount of the pixel is within the bounds of the triangle, a point is either colored or not colored, resulting in stairstepping. In sampling multiple points per pixel and applying the fractional average to the color value, we approximate proportional coverage along the edges and produce smoother transitions, minimizing aliasing. My implementation rasterizes into a higher-resolution buffer and then downsamples to the final image by averaging. The workflow is as follows:

1. **Data Structure Changes:** Alter sample_buffer to be sized as ***width * height * sample_rate*** to accommodate for the amount of samples per pixel that will be taken.
    a. For a pixel (x, y), the based index is **(y * width + x) * sample_rate**.
    b. Subsamples are indexed **base + 0** through **base + (sample_rate - 1)**.
    c. **set_sample_rate**, **set_framebuffer_target** now both resize **sample_buffer** accordingly and **fill_pixel** writes the given color to all subsamples for the pixel (is used for lines and points, not triangles).
2. **Changes to rasterize_triangle(...)**: For each pixel in the triangle's bounding box:
    a. Form an nxn grid where **n = sqrt(sample_rate)**.
    b. For each grid cell (i, j) which represents a subpoint in the pixel, take a sample at **(x + (i + 0.5) / n, y + (j + 0.5) / n)**.
    c. Test each sample if it is a point in the triangle (same logic as explained in Task 1)
    d. If a sample is inside, write the triangle color to **sample_buffer[base + (j * n + i)]**. Otherwise, no alteration.
3. **Changes to resolve_to_framebuffer()**: For each pixel, average all of its subsamples and write the result to the frame buffer.

The following images demonstrate the impact of higher rates of sampling. When the sampling rate is 1, we can see that pixels whose centers are not in the triangle are set to the background color, even if a substantial portion of the pixel is inside the triangle. With a higher sampling rate, those samples whose positions fall inside the triangle are better represented. The proportion of subsamples that are within the triangle proportionally influence the color of the pixel, smoothing the edges and including the skinnier parts of the triangle that would've been otherwise excluded.

[Include the photo at Graphics/assignment1/Task2-1.png]
Screenshot of *basic/test4.svg* with **sampling rate = 1**

[Include the photo at Graphics/assignment1/Task2-2.png]
Screenshot of *basic/test4.svg* with **sampling rate = 4**

[Include the photo at Graphics/assignment1/Task2-3.png]
Screenshot of *basic/test4.svg* with **sampling rate = 16**

Task 3

The next part was implementing the three possible transformations: translate, scale, and rotate. After altering the functions with the appropriate matrices, the following changes were made to cubeman. Instead of the bland, static, red figure that you see on the left, he is now the headless horseman. He is pictured with his right arm resting by his side and his left hand extended, presenting his pumpkin head. He is also wearing a black outfit with gray gloves.

Screenshot of the Original Cubeman
[Include the photo at Graphics/assignment1/Task3-1.png]

Screenshot of the Headless Horseman Cubeman
[Include the photo at Graphics/assignment1/Task3-2.png]

Task 4

Barycentric coordinates describe a point inside a triangle weighted by the three vertices. Each vertex has an associated weight (alpha, beta, gamma) where alpha + beta + gamma = 1 and alpha, beta, gamma >= 0. Given the vertices p0, p1, and p2, a point (x, y) inside the triangle can be represented as (x, y) = alpha * p0 + beta * p1 + gamma * p2. The larger the weight of the vertex, the closer that point is to the vertex. When the weight is 0, the point lies on the opposite edge; when the weight is 1, the point is the same as that vertex.

This helps us interpolate colors, as we apply the weights of the vertices at a given point to their respective color values. This is demonstrated in the following image where we can see a triangle with three vertices representing red, green, and blue. Points at the vertex are the solid color, the point equidistant from all three points is an even combination of all three, and all other points represent a gradient between the vertices' colors.

[Include the photo at Graphics/assignment1/Task41.png]
Triangle Gradient Demonstrating Barycentric Coordinates
Edge along green and red points creates a linear gradient between the two

[Include the photo at Graphics/assignment1/Task42.png]
Triangle Gradient Demonstrating Barycentric Coordinates
The mid section is an equal mix of the three - represents a gradient between the three

[Include the photo at Graphics/assignment1/Task43.png]
Triangle Gradient Demonstrating Barycentric Coordinates
At the blue corner, the pixels are blue with little influence from the other corners

In my implementation, I once again rasterize the triangle with supersampling, then interpolate vertex colors using the barycentric coordinates. For each subsample inside the triangle, I compute alpha, beta, and gamma as the following:

```
    float w0 = line_eq_result(x, y, x1, y1, x2, y2);
    float denom_alpha = line_eq_result(x0, y0, x1, y1, x2, y2);
    float alpha = w0 / denom_alpha;

    float w1 = line_eq_result(x, y, x2, y2, x0, y0);
    float denom_beta = line_eq_result(x1, y1, x2, y2, x0, y0);
    float beta = w1 / denom_beta;

    float gamma = 1.0f - alpha - beta;
```

This gets the proportional distance from one vertex to the opposite edge for the points weighted by alpha and beta. (This can also be represented by proportional area ratios but I chose the distance implementation). Therefore, gamma is just 1 - (alpha + beta) as gamma + alpha + beta

= 1. These distances represent the weight of each vertex applied on the point. The interpolated color is thus c = alpha * c0 + beta * c1 + gamma * c2, which is written into the sample buffer. I reuse the “in triangle” logic by evaluating the same implicit line equations for the ratios.

[Include the photo at Graphics/assignment1/Task4.png]
Screenshot of *svg/basic/test7.svg*

Task 5

Pixel sampling chooses a color from the texture when given continuous texture coordinates represented by (u, v). The texture is stored as a discrete grid of texels, so we select one from the below options:

1. **Nearest-neighbor Sampling**; Pick the single texel closest to (u, v). Map UV to texel indices with floor and clamp the values. Then, return the texel's color. It is fast and sharp, but has a blocky appearance when zoomed in.
    a. Implemented in sample_nearest(...)
2. **Bilinear Sampling**: Use the four texels around (u, v) and interpolate horizontally between the two top levels, then between the two bottom levels, then vertically between the two results. This produces smoother images than nearest, especially under magnification.
    a. Implemented in sample_bilinear(...)

The workflow in my code is:

1. **Texture mapping setup**: For each subsample inside the triangle, compute the barycentric weights (alpha, beta, gamma) and interpolate UVs:
    a. u = α*u0 + β*u1 + γ*u2
    b. v = α*v0 + β*v1 + γ*v2
2. **Sample selection**: Choose a sampling method according to the PSM flag:
    a. PSM == P_NEAREST → call sample_nearest(Vector2D(u, v), 0).
    b. PSM == P_LINEAR, call sample_bilinear(Vector2D(u, v), 0).

In the following images, the effects are demonstrated. I placed the pixel inspector over a portion of the longitude and latitude lines in the Atlantic ocean. In doing so, we can see how each method handles a narrow, curved line that is likely to be improperly sampled. In the nearest pixel sampling with a rate of 1, we see large gaps in the line as the sampling method is unable to interpolate between the points on the line. In contrast, the bilinear sampling with a rate of 1 does noticeably better, filling in parts of the gap and providing a smoother transition between marker and ocean. Once we bump the rate up to 16 pixels per subsample, the nearest sampling result is able to get rid of most of the gaps, but the gradient depicted on the line is often random and inaccurate. The bilinear sampling result is much better, providing clearer, smoother, and more accurate transitions from the inside of the line to the ocean.

[Include the photo at Graphics/assignment1/nearest1.png]
Screenshot of *svg/texmap/test2.svg*
Nearest pixel sampling - Supersample rate 1 per pixel

[Include the photo at Graphics/assignment1/bilinear1.png]
Screenshot of *svg/texmap/test2.svg*
Bilinear pixel sampling - Supersample rate 1 per pixel

[Include the photo at Graphics/assignment1/nearest16.png]
Screenshot of *svg/texmap/test2.svg*
Nearest pixel sampling - Supersample rate 16 per pixel

[Include the photo at Graphics/assignment1/bilinear16.png]
Screenshot of *svg/texmap/test2.svg*
Bilinear pixel sampling - Supersample rate 15 per pixel

Task 6:

Leveling sampling chooses which mipmap level to use when sampling a texture. Mipmaps, as explained in lecture, store the same texture at lower resolutions so that when a screen covers many texels we can use a smaller level to sidestep aliasing. The level is chosen from the UV derivatives: how quickly (u, v) changes per screen pixel. If they change quickly, many texels cover one pixel and we use a coarser level; if they change slowly, conversely, we use a finer level. Therefore, level sampling reduces aliasing in the distance or under perspective. My workflow makes use of certain constants to determine the level:

- **L_ZERO**: Always use level 0 (full resolution). No mipmapping.
- **L_NEAREST**: Compute the ideal level (a float) and round to the nearest integer. Sample that level only. Avoids some aliasing with a single sample.
- **L_LINEAR**: Treat the level as continuous. Sample the two adjacent levels (floor and ceil), then blend by the fractional part. Smoother transitions between levels.

With those, my code workflow is as follows:

- **Barycentric differentials in rasterize_textured_triangle**: For each sample at (sample_x, sample_y), compute UVs at three points:
    - **p_uv** at (sample_x, sample_y)
    - **p_dx_uv** at (sample_x + 1, sample_y)
    - **p_dy_uv** at (sample_x, sample_y + 1)
- Each (u, v) is interpolated from the vertex UVs using barycentric coordinates. With this, we fill out a SampleParams struct “sp” and pass it to tex.sample().
- **get_level**: Approximate UV derivatives as du/dx = p_dx_uv.x - p_uv.x, and similarly for dv/dx, du/dy, dv/dy. Scale by texture width and height to get texel-space derivatives, compute magnitudes for the x and y directions, take their max, and use level = log2(max(p, 1)). Clamp the result to the valid mip range.
- **sample**: Call **get_level** to obtain the level, then:
    - LSM == L_ZERO: use level 0
    - LSM == L_NEAREST: round to the nearest integer level and sample that level with the chosen pixel sampling method
    - LSM == L_LINEAR: sample at floor(level) and ceil(level), then blend by the fractional part: (1 - frac) * sample(floor) + frac * sample(ceil)

In regards to tradeoffs, supersampling has the strongest antialiasing because it takes multiple samples per pixel and averages them, but it increases memory use and costs. A 16x sample rate needs 16 samples per pixel, so both sample_buffer size and per-pixel work grow with the rate. Pixel sampling (nearest vs bilinear) changes cost without changing memory: bilinear needs four texels and linear interpolation, while nearest only samples one texel. Bilinear reduces jaggedness when magnified. Nearest can look blocky but it is faster. Level sampling affects neither memory nor the number of samples per pixel, since the MIP chain is already stored. L_ZERO uses level 0 only and is cheapest; L_NEAREST adds a level computation and one

sample; L_LINEAR samples two levels and blends them, which is a bit more work but gives smoother transitions. Overall, supersampling targets edge antialiasing and coverage, pixel sampling affects texture smoothness under magnification, and level sampling reduces aliasing from minification and perspective.

The results described above can be viewed in the images below:

[Include the photo at Graphics/assignment1/zero_nearest.png]
Screenshot of *svg/texmap/Task6.svg*
L_ZERO and P_NEAREST

[Include the photo at Graphics/assignment1/zero_linear.png]
Screenshot of *svg/texmap/Task6.svg*
L_ZERO and P_LINEAR

[Include the photo at Graphics/assignment1/nearest_nearest.png]
Screenshot of *svg/texmap/Task6.svg*
L_NEAREST and P_NEAREST

[Include the photo at Graphics/assignment1/nearest_linear.png]
Screenshot of *svg/texmap/Task6.svg*
L_NEAREST and P_LINEAR