CS184/284A Spring 2025
Homework 1 Write-Up

Names: Isabella Hu

Link to webpage: cal-cs184.github.io/hw-webpages-su25-IHU3025/hw1/index.html
Link to GitHub repository: github.com/cal-cs184/hw-rasterizer-team

Overview

In this assignment, I implemented a 2D rasterization pipeline capable of rendering SVG images. The project involved building fundamental computer graphics techniques from scratch, starting with basic triangle rasterization and gradually adding more sophisticated features to improve rendering quality.

Key components implemented:

- Triangle rasterization with point-in-triangle testing
- Antialiasing through supersampling
- 2D transformation operations (translation, scaling, rotation)
- Barycentric color interpolation
- Texture mapping with nearest and bilinear sampling method
- Mipmapping for texture anti-aliasing

Task 1: Drawing Single-Color Triangles

1.1 Triangle Rasterization Algorithm

To rasterize triangle, there are the implementaion we used:

1. **Determine Bounding Box:**
    - Calculate min/max x and y coordinates from the three vertices of the triangle
    - This gave us the smallest rectangle containing the triangle
    - Instead of iterate all the pixels on the screen, we only iterate through the pixels in the bounding box
2. **Point-in-Triangle Test:**
    - For each pixel in the bounding box:
    - Compute cross products between edge vectors and point vectors
    - Formula for cross product (with A, B being the vertices and P being the point we are checking):

      $$(B - A) \times (P - A) = (Bx - Ax) * (Py - Ay) - (By - Ay) * (Px - Ax)$$

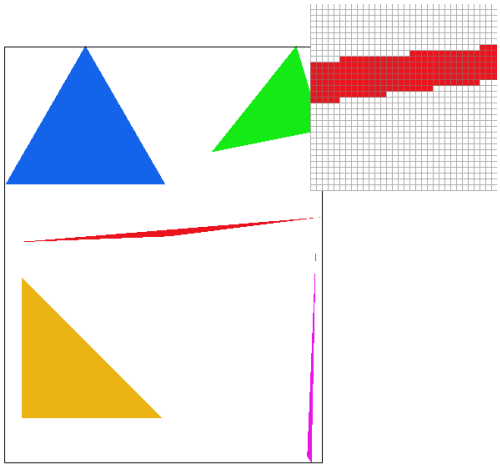
    - If all three cross products have the same sign, the point is inside:
        - Since the 2D cross product is proportional to the sine of the angle between vectors, so its sign indicates whether the directiion is clockwise or

counterclockwise. By calcualting the cross product of (P - A) with each triangle edge (B - A), (C - B), and (A - C). We check P's orientation relative to each edge. If all three cross products share the same sign, P lies on the same side of every edge (either all angle are between 0°–180° or 180°–360°). In either case, P is inside the triangle.

1.2 Algorithm Efficiency

Why this approach is no worse than checking each sample in the bounding box :

- **Bounded complexity:** Only samples within the triangle's bounding box are considered
- **Constant time edge tests:** Each point-in-triangle check is O(1) complexity
- Hence the algorithm runs in linear time with respect to the number of points in the bounding box

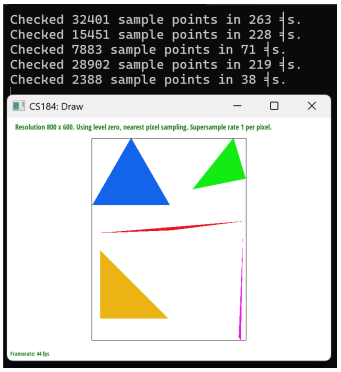


Basic/test4.svg rendered with default parameters

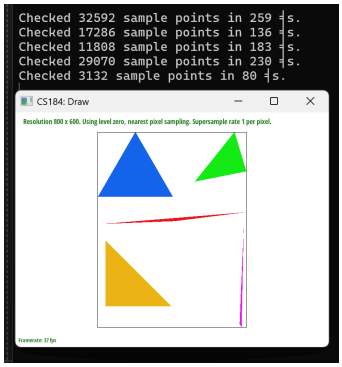
1.3 Extra Credit: Optimization with Early Rejection

Implemented additional optimization:

- **Row-wise Early Termination:**
  - Added a booleon variable tracks whether previous pixel in current row was inside triangle
  - If we find a pixel outside after being inside, that means we just exit the triangle, hence we stop checking that row
- **Performance:**
  - Measured using terminal output tracking the number of processed pixels
  - Optimized version processes fewer pixels than the original method



Rasterization with early exit optimization



Rasterization without early exit (normal case)

Task 2: Antialiasing by Supersampling

2.1 Supersampling Algorithm

To addresses aliasing (jaggies) through supersampling: we rasterize triangles at a high resolution and resolving them to the framebuffer by averaging the sample value for each pixel.

- **Core Concept:**
  - Binary pixel coverage tests create jagged edges (aliasing)
  - Supersampling evaluates multiple sample points per pixel
  - Final pixel color is the average of these sub-samples
  - Result smoother edge
- **Sampling Pattern:**
  - Instead sampling each pixel once, we creates a uniform grid of sample points within each pixel
  - Grid dimensions: √sample_rate × √sample_rate
  - Example: sample_rate=4 → 2×2 grid per pixel

2.2 Data Structure Modifications

We adapt from the original data structure from the basic triangle rasterization, using the sample_buffer to store the color vector of our sampling point.

Key changes to support supersampling:

Original	Supersampled
sample_buffer stores per-pixel colors	sample_buffer stores per-sample colors
Dimensions: width × height	Dimensions: width × height × sample_rate

Instead of each element in the sample_buffer representing a pixel (e.g., [A, B, C, D]), each element now represents a subsample within a pixel (e.g., [A0, A1, A2, A3, B0, B1, B2, B3], sample_rate = 4).

Index calculation for sub sample points:

```
sample_index = (y * width + x) * sample_rate + (i * √sample_rate + j)
```

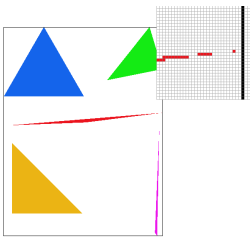
2.3 Pipeline Modifications

1. **Rasterization changes:**
    - Nested loops iterate through sub-samples within each pixel
    - Edge function evaluated at sub-sample locations
2. **Resolve Stage:**
    - Averages all sub-samples for each pixel
    - Stores result in final framebuffer

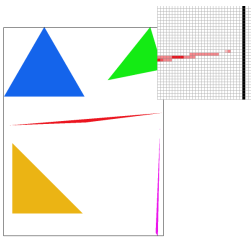
2.4 How Supersampling Reduces Aliasing

It capture more information about the high frequency edges before downsampling to pixel resolution, result smoother edges → less jaggies.

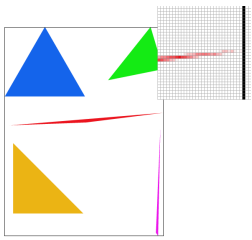
- **Visual Effect:**
    - Binary edges → gradient transitions
    - Jaggies replaced with smooth pixel transitions



sample rates = 1



sample rates = 4

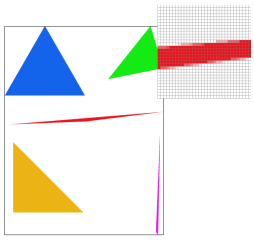


sample rates = 16

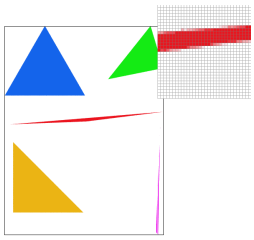
Extra Credit: Jittered Supersampling

As an alternative method to the uniform grid-based supersampling, I implemented **jittered sampling**. So instead of placing sub-samples uniformly in a regular grid, we apply a small random offset (jitter) to each sub-sample's position within its grid cell.

By adding this slight randomness, it helps avoid visible patterns and makes the image look smoother overall.



Without jittering, Sample rate = 9



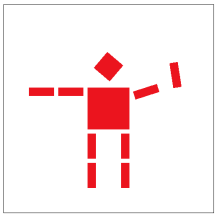
With jittering, Sample rate = 9

Task 3: Transforms

My goal was to pose the robot so it looks like it's waving its right arm.

Transformations Used

- **Translation:** Moved the right arm upward and outward.
- **Rotation:** Rotated arm segments



Cubeman waving

Extra Credit: Viewport Rotation Feature

I implemented viewport rotation using the `'['` and `']'` keys. Pressing `'['` rotates the view counterclockwise, and `']'` rotates it clockwise.

Implementation:

1. Added a rotation state in `drawrend.h`: created a new attribute to the `DrawRend` class to keep track of the rotation angle:

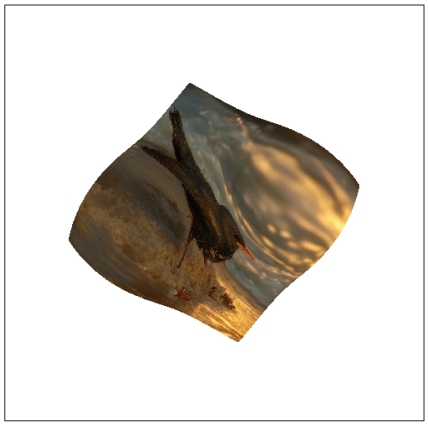
```
float rotation_angle = 0.0f;
```

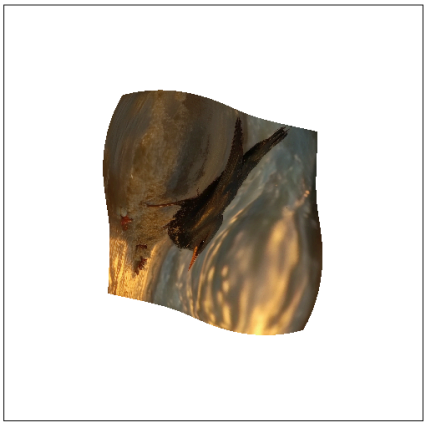
2. Updated transformation matrix in `DrawRend::redraw()`: inserted a rotation matrix between the `svg_to_ndc` and `ndc_to_screen` matrices. To rotate around the center of the screen rather than the origin, I surrounded the rotation with translation matrices (translate the object to the origin before applying rotation, then translate it back to its original position afterward)

3. Added keyboard controls in `DrawRend::keyboard_event()`, handled the `'['` and `']'` keys to adjust the rotation angle and trigger a redraw

Result

Below is an example image rotated using the new key controls:



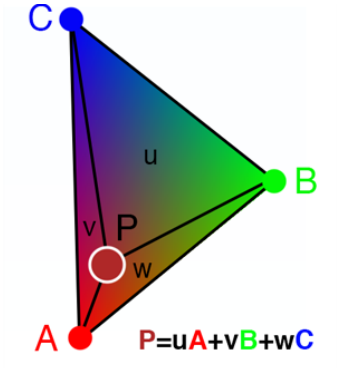


Task 4: Barycentric coordinates

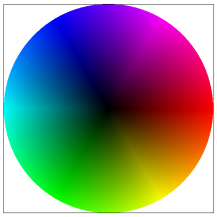
Barycentric coordinates are a way to express the position f a point within a triangle using a weighted combination of the triangle's three vertices.

Formula: $P = \alpha A + \beta B + \gamma C$, where $\alpha + \beta + \gamma = 1$

For example, if a point is exactly at one vertex, its weight for that vertex is 1 and 0 for the other two. If it's in the center of the triangle, since its effect by all 3 vertices equally, all three weights are equal (1/3, 1/3, 1/3). These weights always sum to 1.



Example on color interpolation using barycentric coordinate



test7 sample rate = 1

Task 5: "Pixel Sampling" for Texture Mapping

Pixel sampling is the technique in which we fetch color values from a texture and map them onto a 2D triangle. Since coordinates are continuous while texels are discrete, we need a sampling method to interpolate between texels.

5.1 Implementation

Texture Lookup:

Given a normalized `(u, v)` coordinate, we scale it to the texture's resolution to get the corresponding pixel location.

For example, if the texture is 512 × 512, a sample at `(0.5, 0.5)` maps to pixel `(256, 256)`.

Integration into Rasterization:

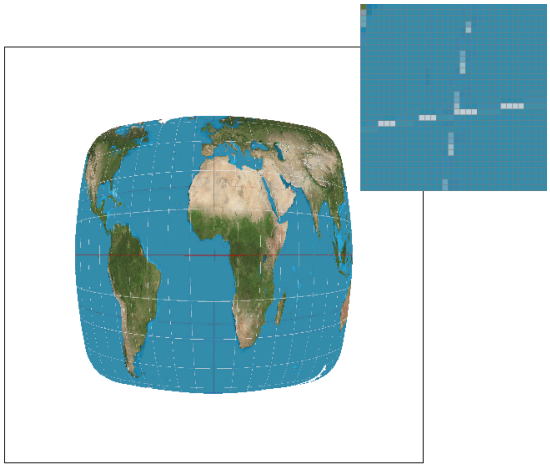
During triangle rasterization, for each sample point, we call either `sample_nearest()` or `sample_bilinear()` based on the `PixelSampleMethod` setting.

5.2 Nearest-Neighbor Sample (P_NEAREST)

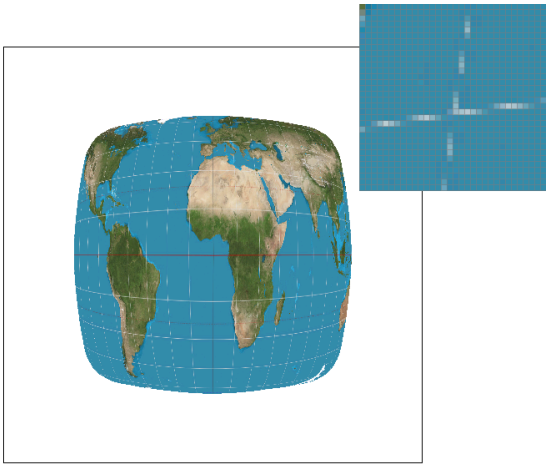
- Takes the color of the closest texel to the sample point by rounding.
- Less computation but produces a pixelated result since it does not take surrounding texels into account.

5.3 Bilinear Sampling (P_LINEAR)

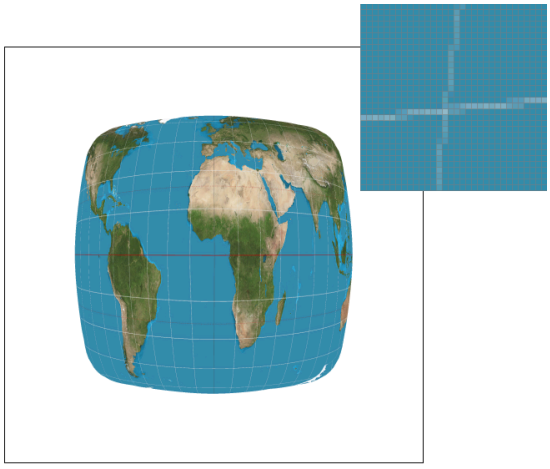
- Interpolates between the four nearest texels for a smoother result using the lerp formula: `(1 - t) * c00 + t * c01`, with `t` being the weight calculated from the distance between the point and `c00`.
- Perform linear interpolation horizontally first using `lerp(c00, c10, s)` and `lerp(c01, c11, s)`, then interpolates between the two results to get the final color.
- Reduces blockiness but requires more computation due to interpolation.



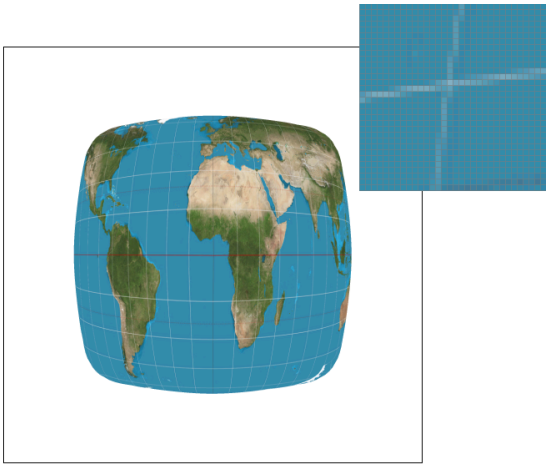
P_NEAREST, 1 sample/pixel



BILINEAR, 1 sample/pixel



P_NEAREST, 16 sample/pixel



BILINEAR, 16 sample/pixel

5.4 Difference Between the Two Sampling Methods

The difference is most noticeable when textures contain high-frequency detail, like the white longitude/latitude lines on a map. Bilinear filtering smooths transitions between texels, while nearest-neighbor sampling creates abrupt jumps.

Task 6: "Level Sampling" with mipmaps for texture mapping

6.1 Core Concept:

When a texture is minified or is viewed from far away or at a steep angle, each screen pixel corresponds to a large area of the texture. Sampling from the original high-resolution texture in this case can lead to two problems:

- **Blurring:** The texture looks smeared because each pixel averages too much detail from the texture.
- **Moiré Patterns:** High frequency texture details can create ripple-like patterns when undersampled.

Supersampling helps with geometric aliasing but not with texture aliasing, since it still samples from the same high-frequency texture. To address this problem, level sampling (mipmapping) uses precomputed lower-resolution textures and selects the appropriate one based on screen-space derivatives.

6.2 Implementation

6.2.1. Determining the Mipmap Level

In our `Texture::get_level()` function, we figure out which mipmap level to use by determine how extreme the texture coordinates are changing across the screen. Which helps us decide how "zoomed in" or "zoomed out" a texture is at a particular area.

We calculate two differences:

- `duv_dx = p_dx_uv - p_uv` → the rate of change of texture coordinates in the x-direction
- `duv_dy = p_dy_uv - p_uv` → the rate of change of texture coordinates in the y-direction

These differences show how much UV texture coordinates changes when screen space movement is change by 1 unit (1 pixel right or down). Higher this value is, it means the screen-space movement covers a larger area in texture space (texture is more zoomed out, need higher level of mipmap)

Formula to convert this into a mipmap level:

```
L = max(||duv_dx||, ||duv_dy||) × texture_width
level = log2(L)
```

6.2.2. Using the Mipmap Level

In `Texture::sample()`, we decide how to use the mipmap level based on the level and pixel sampling modes:

When using `L_LINEAR`, we calculate the two nearest level `level_base` and `level_next`

Then blend the results from both levels using lerp formula:

```
final_color = (1 - weight) * sample_from(level_base) + weight * sample_from(level_n
```

6.2.3. Setting Up Texture Derivatives in the Rasterizer

In `rasterize_textured_triangle()`, we need to calculate the derivative of uv. These values are passed to `get_level()` to choose the appropriate mipmap level.

We do this by transforming screen-space positions into barycentric coordinates, then using those to compute the corresponding UVs.

i.e. `u_dy = p_dy.x * u0 + p_dy.y * u1 + p_dy.z * u2;`

The results are stored in the `SampleParams` struct:

- `sp.p_uv`: UV at the current sub-pixel
- `sp.p_dx_uv`: UV one pixel to the right
- `sp.p_dy_uv`: UV one pixel down

6.3 Trade-offs Between Level Sampling Techniques

Technique	Speed	Memory Usage	Antialiasing Power
L_ZERO	Fast: always samples from base mip level (level 0)	Low: only uses the original texture	Poor: can result aliasing and moiré
L_NEAREST	Moderate: selects the closest mip level based on screen-space derivatives	Moderate: requires precomputed mipmaps	Good: reduces aliasing but may show level transitions
L_LINEAR	Slower: blends between two mip levels (trilinear filtering)	Moderate: uses mipmaps and performs extra interpolation	Better: smooth transitions and reduced aliasing

L_ZERO, P_NEAREST

L_ZERO, P_LINEAR

L_NEAREST, P_NEAREST

L_NEAREST, P_LINEAR

Extra Credit: Anisotropic Sampling

As an improvement to bilinear sampling, I implemented a basic form of anisotropic filtering. Which samples multiple points along the direction of greatest texture distortion (either `p_dx_uv` or `p_dy_uv`) to capture detail better in a stretched texture.

Implementation:

- 1. Chooses the dominant direction of UV change (horizontal or vertical) by caluclating the derivative
- 2. Samples `num_samples` points along that direction using bilinear sampling.
- 3. Averages the results to produce the final color.

Without Anisotropic Sampling

With Anisotropic Sampling

The version with anisotropic sampling is significantly smoother compared to standard bilinear sampling.