CS184/284A Spring 2025 Homework 1 Write-Up

Name: Dmytro Krukovskyi

Link to webpage: https://dmkruk.github.io/cs184hws/hw1/index.html
Link to GitHub repository: https://github.com/cal-cs184-student/sp25-hw1-solitary-disguise

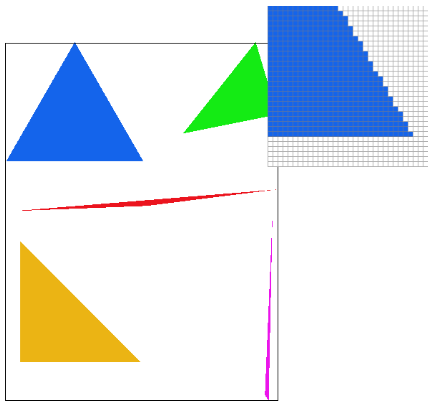
Overview

In this homework I built a simple rendering pipeline. I started by drawing single-color triangles and then improved the visual quality using supersampling for antialiasing. I implemented geometric transforms and used barycentric coordinates to interpolate values across triangles, which was crucial for accurate texture mapping. Additionally, I added support for texture mapping and mipmap support.

Task 1: Drawing Single-Color Triangles

Here is an example 2×2 gridlike structure using an HTML table. Each **tr** is a row and each **td** is a column in that row. You might find this useful for framing and showing your result images in an organized fashion.

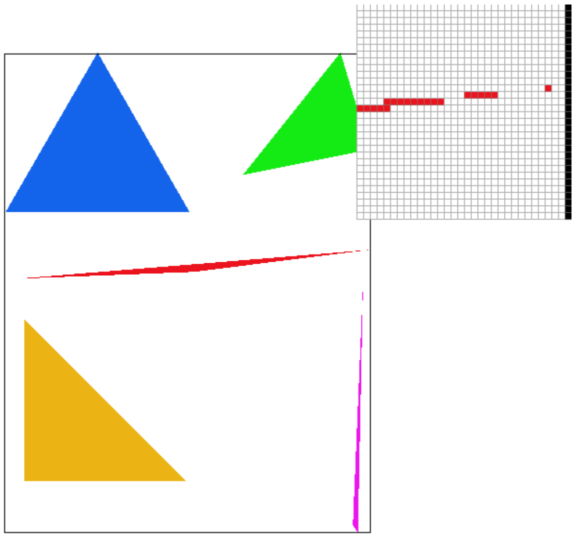
I begin by computing the triangle's bounding box—finding the minimum and maximum x and y values among the three vertices. Then, I loop over all the pixels within this box and use the sample point at each pixel's center ((x + 0.5, y + 0.5)) for the three lines test. Using edge functions, I check if the center point lies inside the triangle. If it does (including on the boundary), I call the helper function to fill the pixel. This method is no worse than simply testing every pixel in the bounding box because it restricts its attention to the bounding box, so no extra work is done.

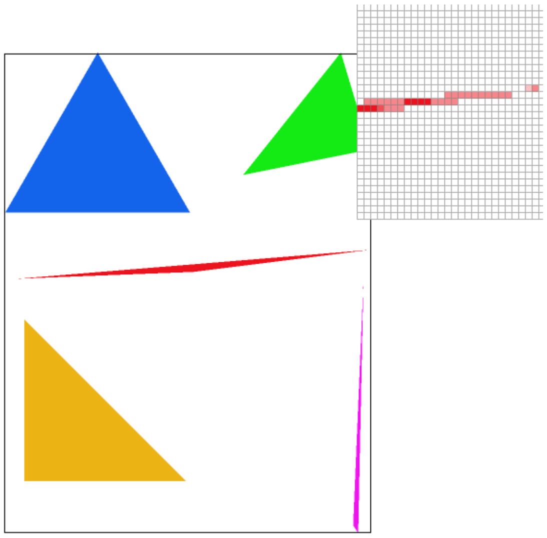


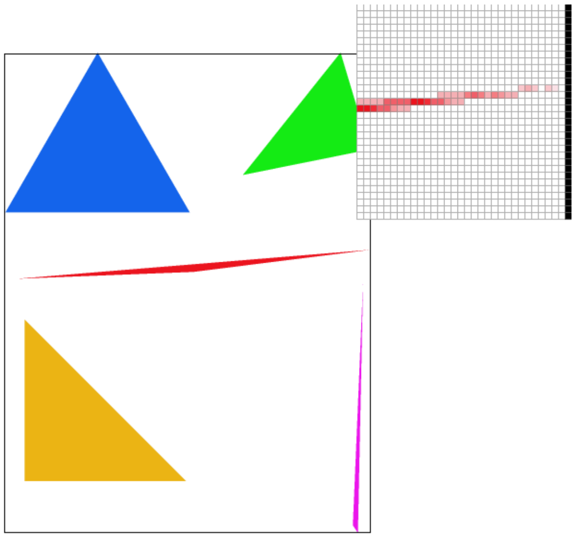
Extra credit: To speed up the process I precommputed ABC coefficients at the beginning, rather then calculate them every loop. Also, I reduced the number of multiplications needed. When we move from pixel (x,y) to (x+1,y) (one pixel to the right): The new edge value will be: A∗(x+1) + B*y + C, so we just need to add A to the previous value which will reduce number of operations needed. We those optimizations in place, for the thrid image (the dragon) I got from 45 ms to 33 ms which is 25% improvment.

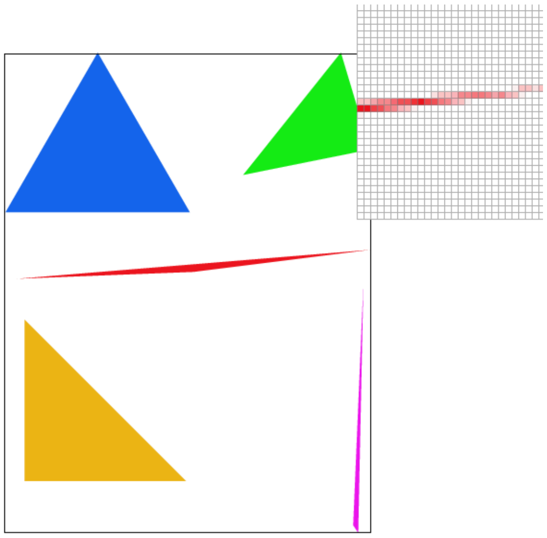
Task 2: Antialiasing by Supersampling

I allocate sample buffer that has the same number of pixels, but multiplied by a sample_rate. When rasterizing triangles, instead of drawing directly to the final framebuffer, I draw to a sample buffer. For every pixel in the triangle's bounding box, loop over a grid of √(sample_rate) * √(sample_rate) sample locations and perform the standard point-in-triangle test. If a sample falls inside the triangle, assign it the triangle's color in our sample buffer. After rasterizing all triangles loop over a sample buffer and average all the pixels in a box to get a final color in a final buffer. This will smooth out any jagged edges. Supersampling greatly reduces aliasing (jagged edges) by effectively rendering at a higher resolution. When these high-resolution samples are averaged, boundaries become smoother.


Sample rate of 1px

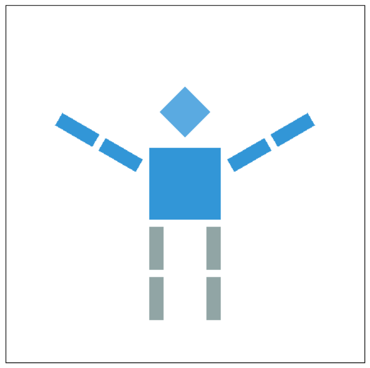

Sample rate of 4px


Sample rate of 9px

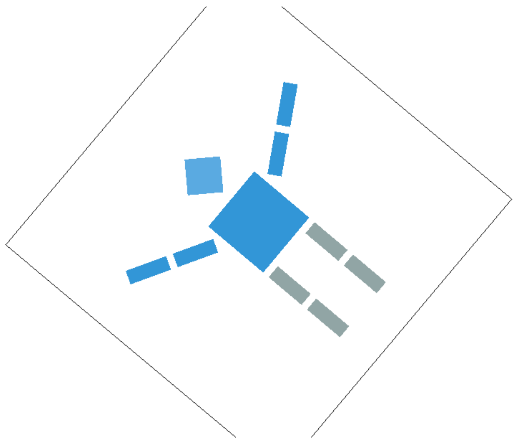

Sample rate of 16px

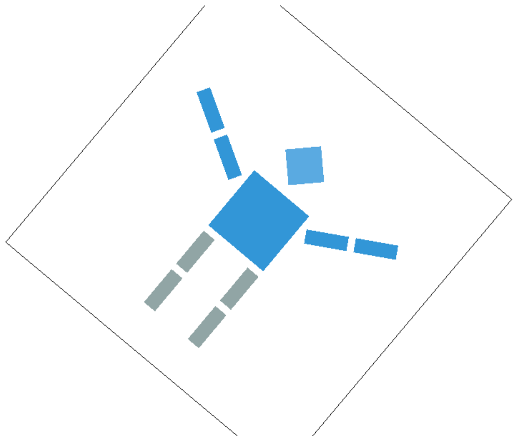
Task 3: Transforms

I updated the original robot.svg to create a more dynamic figure—a cubeman doing a friendly wave. I modified the transform attributes of both arms so they lift upward. For the right arm, I applied a rotation of −30. For the left arm, I applied a rotation of 30. Also, I changed the color of hands and legs.



Extra Credit: I modified the transformation matrices so that the mapping from SVG space to NDC and then to screen space now incorporates an extra rotation. This is achieved by composing the original transformation with a rotation matrix that rotates the view by a user-defined angle.





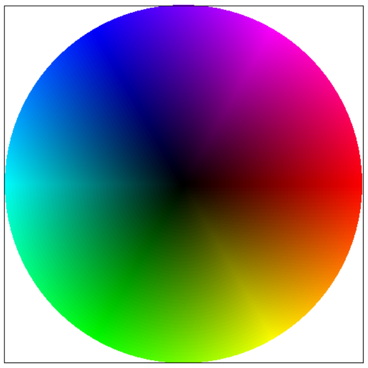
Additionally, I added keyboard events for the Q and E keys which increment or decrement this rotation angle.

Task 4: Barycentric coordinates

Barycentric coordinates allow us to describe any point inside a triangle as a weighted combination of its three vertices. Any point in a triangle can be written as

$$\text{color}(P) = \alpha \cdot \text{color}(A) + \beta \cdot \text{color}(B) + \gamma \cdot \text{color}(C)$$

We can think of those coefficients as proportional areas. In the image below we can think of Barycentric coordinates as a blending between red, green and blue colors.

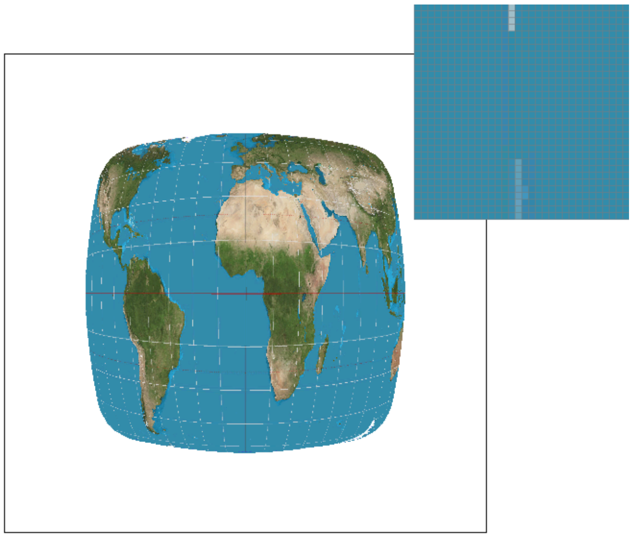


Task 5: "Pixel sampling" for texture mapping

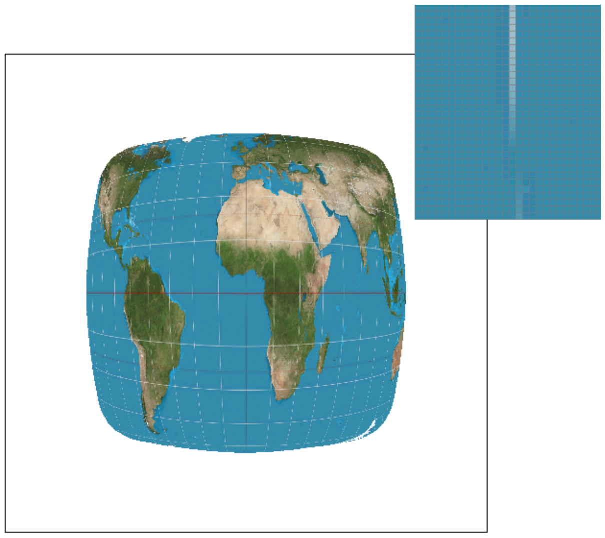
Pixel sampling is the process of determining what color to assign to a pixel by sampling from a texture. When rendering a textured triangle, we have texture coordinates (u,v) at each triangle vertex, and we need to determine appropriate texture colors for each pixel inside the triangle. I implemented texture mapping through these steps:

- For each pixel, I calculate if it's inside the triangle using edge functions
- If it's inside, I compute the barycentric coordinates (alpha, beta, gamma) to interpolate the texture coordinates (u,v) at that point
- With these interpolated (u,v) coordinates, I sample the texture to get a color

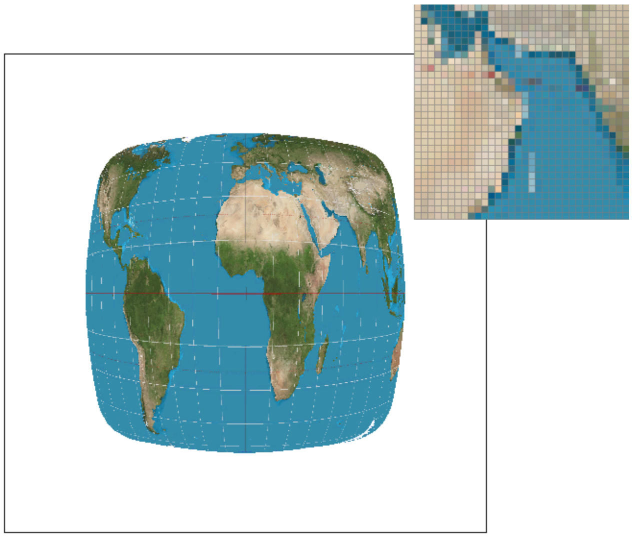
After I get (u,v) coordinates there are two methods avaliable. Nearest Neighbor Sampling and Bilinear Sampling. Nearest neighbor is the simplest approach to texture sampling. It takes the color of the nearest texel to the calculated (u,v) coordinate Bilinear sampling is a more sophisticated approach that samples the four nearest texels to the (u,v) coordinate and interpolates between them. Based on the images below. Nearest Neighbor produces the most jaggies at the sample rate of 1px. Bilinear sampling in the end produces a smooth line. This is because bilinear sampling is slightly more computationally expensive than nearest neighbor sampling. When sampling at 16px I don't see any noticable difference.



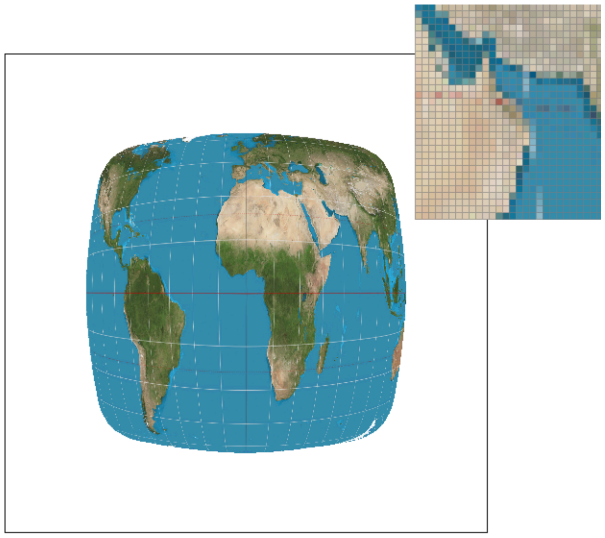
Thin line using Nearest Neighbor



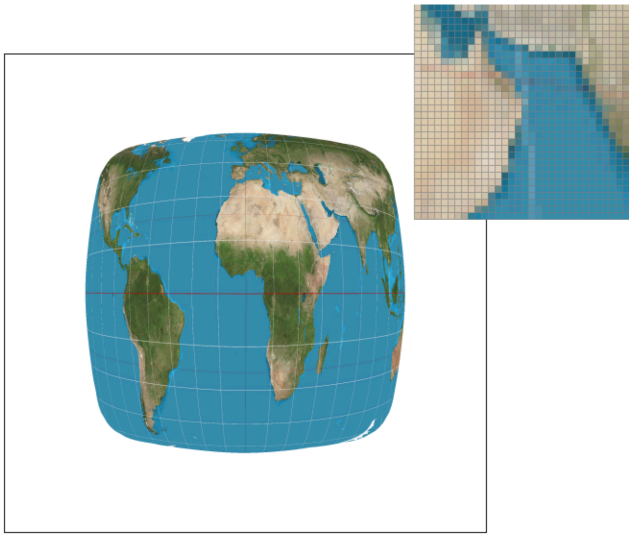
Thin line using Bilinear sampling



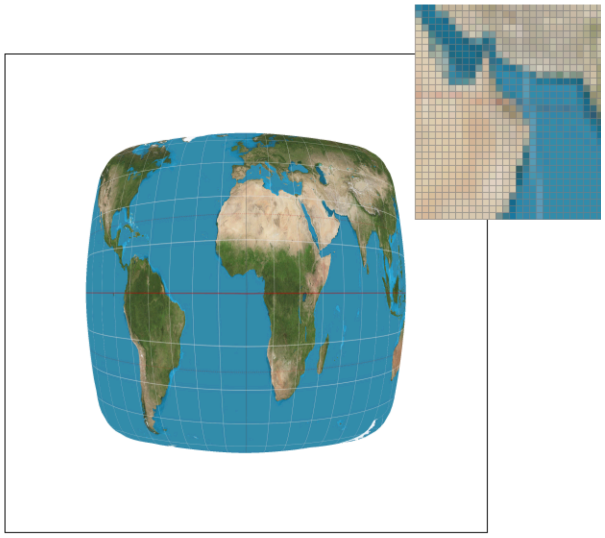
Sample rate of 1px (nearest neighbor)



Sample rate of 1px (bilinear)



Sample rate of 16px (nearest neighbor)



Sample rate of 16px (bilinear)

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

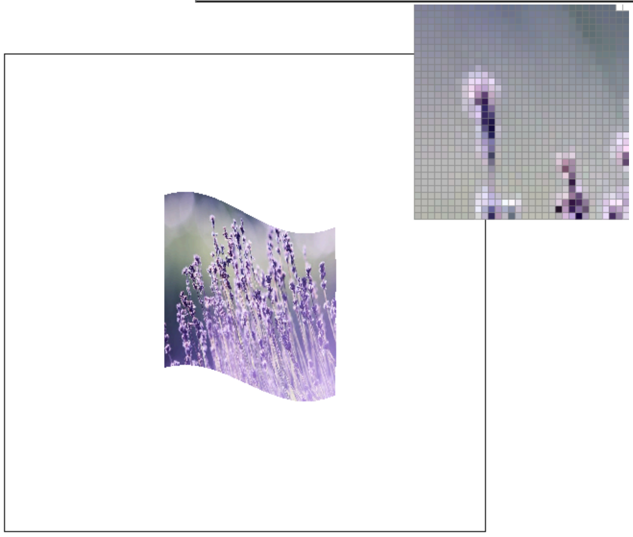
Level sampling is a technique used in texture mapping to address aliasing issues that occur when a screen pixel represents a varying amount of texture space. When we zoom out of a textured object, a single pixel on screen might need to represent many texels from the original texture. To implement level sampling, I created a mipmap hierarchy - a sequence of pre-filtered and downsampled versions of the original texture at different resolutions. Each level in this hierarchy is half the width and height of the previous level. The first level (level 0) is the original texture, level 1 is 1/2 the size, level 2 is 1/4 the size, and so on. The key part of my implementation was determining which mipmap level to sample from for each pixel. To do this:

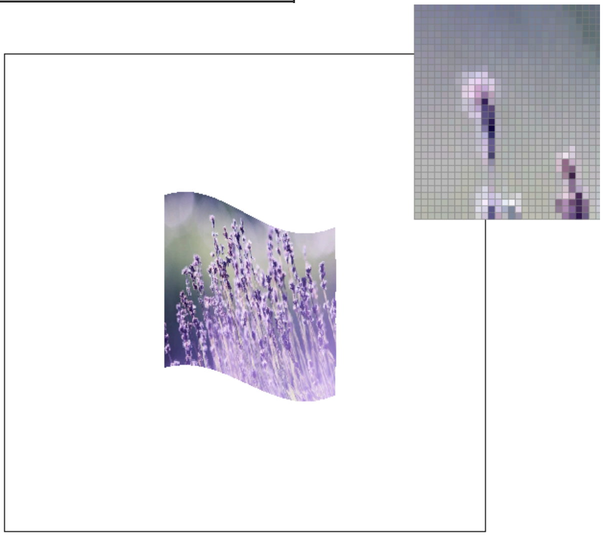
- I calculated the texture coordinate derivatives (how fast the texture coordinates change when moving one pixel in the x or y direction)
- These derivatives tell us how much the texture is being compressed onto the screen
- I used these derivatives to compute a "level" value using the formula $L = \log_2(\max(\sqrt{(du/dx)^2 + (dv/dx)^2}, \sqrt{(du/dy)^2 + (dv/dy)^2}))$
- Depending on the chosen level sampling method, I then used this level value differently:

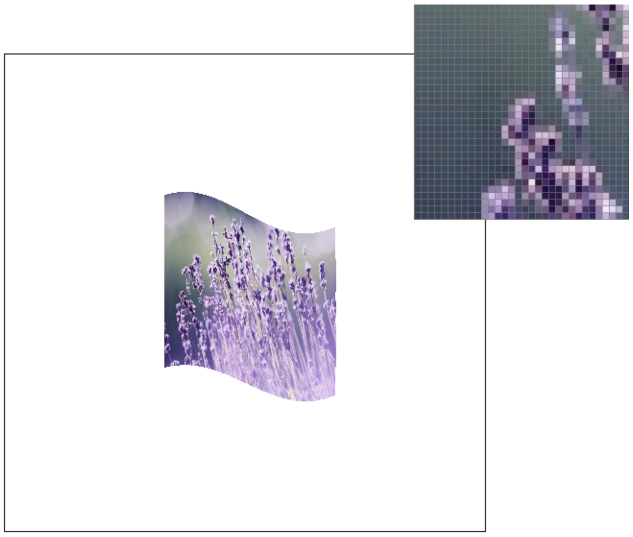
- For L_ZERO: Always use level 0
- For L_NEAREST: Round to the nearest integer level
- For L_LINEAR: Use the fractional level value to interpolate between adjacent levels

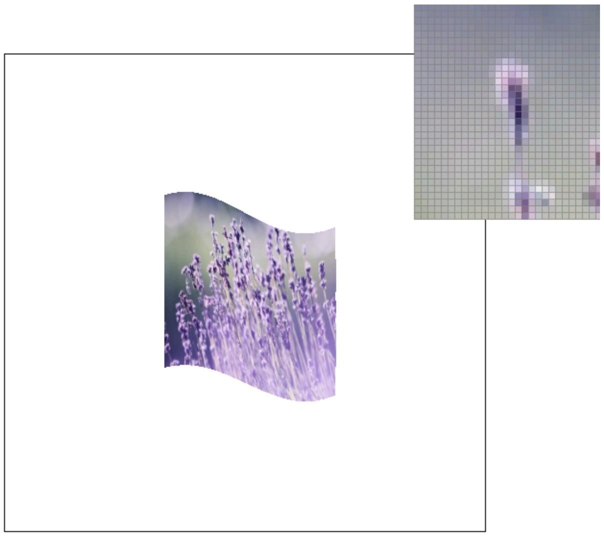
Sampling Method	Speed	Memory Usage	Quality
P_NEAREST	Fast - single texture lookup	Standard	Pixelated, visible aliasing
P_LINEAR (Bilinear)	Medium - 4 texture lookups	Standard	Smoother textures, reduced aliasing
L_ZERO	Fast - uses original texture only	Low - no mipmaps	Poor when zoomed out (aliasing)
L_NEAREST	Medium - requires level calculation	~33% more (mipmaps)	Good at distance, "popping"

			when zooming
L_LINEAR (Trilinear)	Slow - samples from two mipmap levels	~33% more (mipmaps)	Best quality, smooth transitions
Supersampling (1×)	Fast	Base memory	No edge antialiasing
Supersampling (16×)	Very slow - 16× calculations	16× sample buffer	Excellent edge antialiasing


Level 0 (nearest neighbor)


Level 0 (bilinear)


Nearest level (nearest neighbor)


Nearest level (bilinear)