CS 284A - SPRING 2025 - HW1 - ABDULLAH ALRASHDAN

Task 1:

The idea behind rasetrize_triangles I implemented in this task is to compute the triangles' bounding box prior to checking individual pixels. The bounding box is the smallest axis-aligned rectangle that completely encloses the triangle. Then, we find the maximum and minimum x coordinate among the triangle's vertices as well as the maximum and minimum y coordinate among the triangle's vertices. We restrict our tests to pixels inside the rectangle in order to avoid unnecessary computations on pixels that lie outside the triangle. For every pixel within the bounding box we compute the sample point at the center of the pixel; this ensures that the sampling is consistent and to avoid aliasing artifacts. We then do a point-in triangle test using edge functions. For each sample point, we use an edge function to determine if the point is inside the triangle. Where edge function is defined as:

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

where AA and BB are two vertices of an edge, and PP is the point being tested.

For testing, we compute the edge function for each of the three edges of the triangle.
If all three computed values are either nonnegative or nonpositive, then the sample lies inside (or exactly on the boundary of) the triangle. Since the triangle's vertices might be specified in either clockwise or counterclockwise order, checking if all three values share the same sign (or are zero) ensures that the test works regardless of vertex order.

The approach we implemented is efficient and no worse than checking each sample in the bounding box. We claim this because we restrict our tests to only the pixels within the bounding box. In the worst-case scenario, every pixel in the bounding box is checked—which is the same as any algorithm that iterates over the bounding box. Also, the cost per pixel is only a few arithmetic operations (subtractions, multiplications, and comparisons), which are very efficient. For early rejection, If a sample point fails the edge tests, it is immediately rejected without further processing. This approach is also scalable for large triangles, although the bounding box might cover many pixels, the work done per pixel remains minimal. For small or skinny triangles, the bounding box is tight, reducing the number of pixels tested.

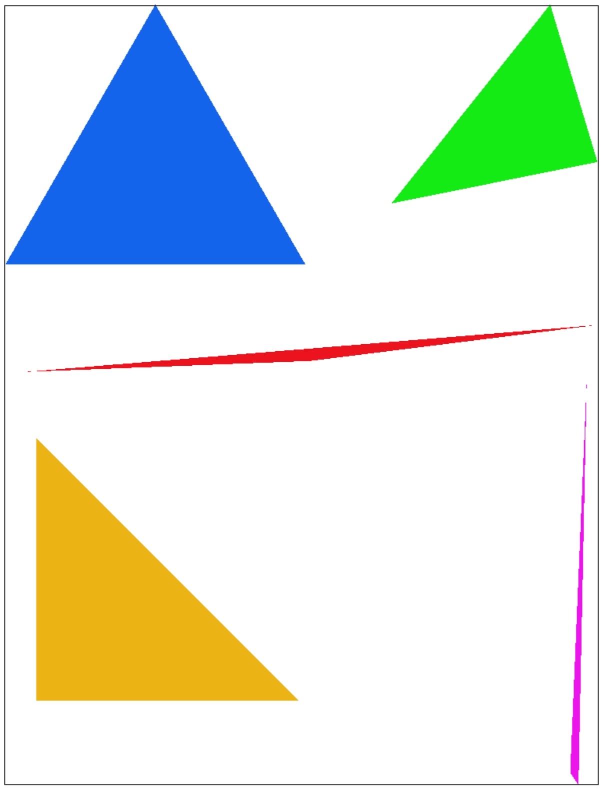

Figure 1: Task 1 deliverable (Basic/test4/svg)

Task 2:

Supersampling maintains a supersample buffer instead of storing a single color per pixel. This is a vector (or array) that holds multiple color samples per pixel. If the window is W×HW×H and we use a sample rate of Supersampling samples per pixel, the supersample buffer contains W×H×SW×H×S Color values. Typically, Supersampling is a perfect square (e.g., 4 samples per pixel arranged in a 2×22×2 grid). For each pixel, we subdivide it into an n×nn×n grid, where n=Sn=S. Instead of sampling at the pixel center (x+0.5,y+0.5)(x+0.5,y+0.5), we sample at multiple positions:

sample position= (x+i+0.5n,y+j+0.5n)sample position=(x+ni+0.5,y+nj+0.5)

where i,j range from 0 to n−1n−1. Each subpixel's center is tested to determine if it lies within the triangle.

Then, for storing and averaging samples; when a subpixel sample is determined to be inside the triangle (using our edge function-based point-in-triangle test), we store the triangle's color in the corresponding location in the supersample buffer. After processing all geometry, we resolve the supersample buffer by averaging the SS samples for each pixel and then converting the averaged floating-point color into the final 8-bit per-channel value that is written to the framebuffer.

Supersampling is useful for antialiasing. As aliasing appears as jagged edges or "staircase" effects along diagonal or curved edges. By taking multiple samples per pixel, supersampling averages the color over a pixel area, thereby smoothing the transition between inside and outside the triangle. This results in smoother edges and a higher-quality rendered image. In addition, supersampling promotes accurate coverage estimation; When a triangle only partially covers a pixel, supersampling helps capture this partial coverage. Instead of making a binary decision (inside/outside), the averaging process yields a color that represents the fractional coverage, which is crucial for antialiasing.

I made modifications to the data structure by introducing a supersampling buffer that is sized to hold W×H×SW×H×S Color objects. This buffer is dynamically resized whenever the window or sample rate changes. Also, I modified "fill_pixel" instead of writing a color to one sample, the updated "fill_pixel" now fills all supersampling subsamples for that pixel. Also, for subpixel loop; the triangle rasterization function was modified to include nested loops over the n×nn×n grid within each pixel. For each subpixel, the center position is computed, and a point-in-triangle test is performed and the same edge function test is applied to each subpixel. If the sample lies inside the triangle, the corresponding entry in the supersample buffer is updated with the triangle's color. Also, for the framebuffer resolution; at the end of the frame, a new pass is made over the supersample buffer where, for each pixel, the supersampling samples are averaged before they are converted to a final color format.

I used supersampling to antialias the triangles by smoothing edges. Since multiple sample points are evaluated within each pixel, pixels that are only partially covered by the triangle will receive a mix of colors from both inside and outside the triangle. Averaging these values leads to a smoother gradient at the edge of the triangle. Also, reducing staircase artifacts by capturing more detail on how the triangle covers the pixel area, the sharp, binary transitions that cause staircase artifacts are mitigated. Instead, the color transitions gradually, giving the illusion of a smoother, higher-resolution edge.

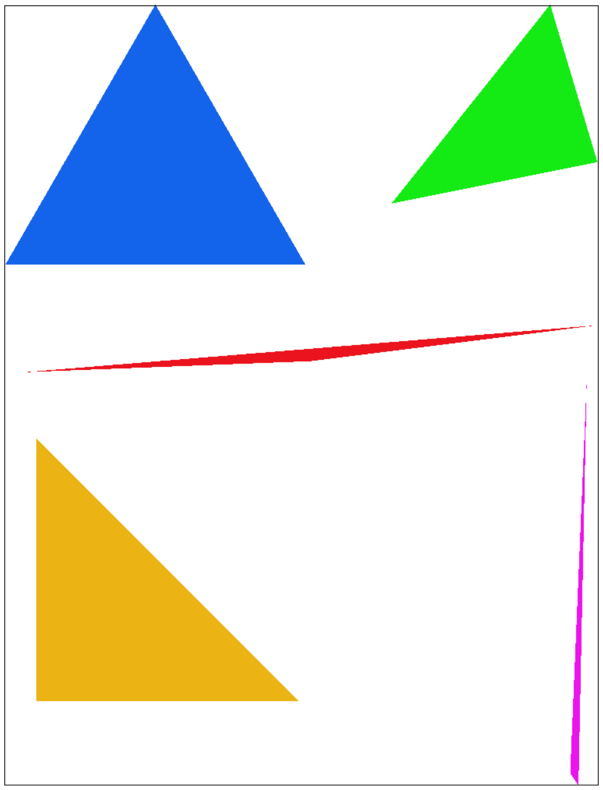

Figure 2: Task2 (basic/test4.svg w/ sample rate = 1).

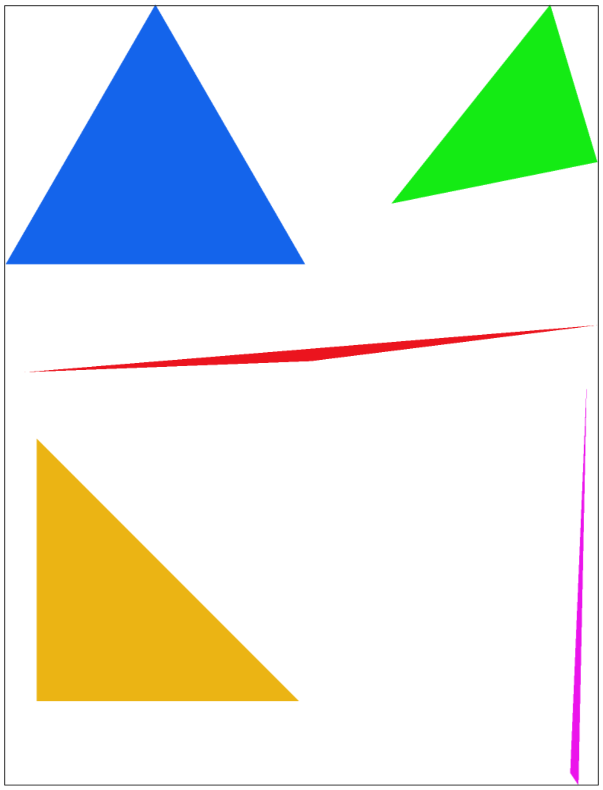

Figure 3: Task2 (basic/test4.svg w/ sample rate = 4).

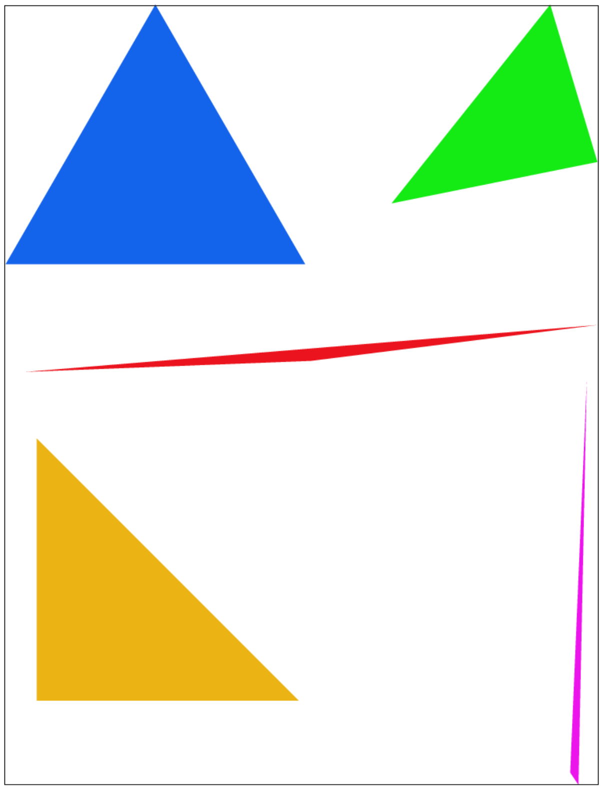

Figure 4: Task2 (basic/test4.svg w/ sample rate = 16).

Task 3:

After modifying the code to have the cubeman be filled with color (red). I tried to be creative more than just having him run or wave; I wanted him to lay down and relax. He seemed to be standing for a long time. To do this, I played with Matrix3x3 translate.

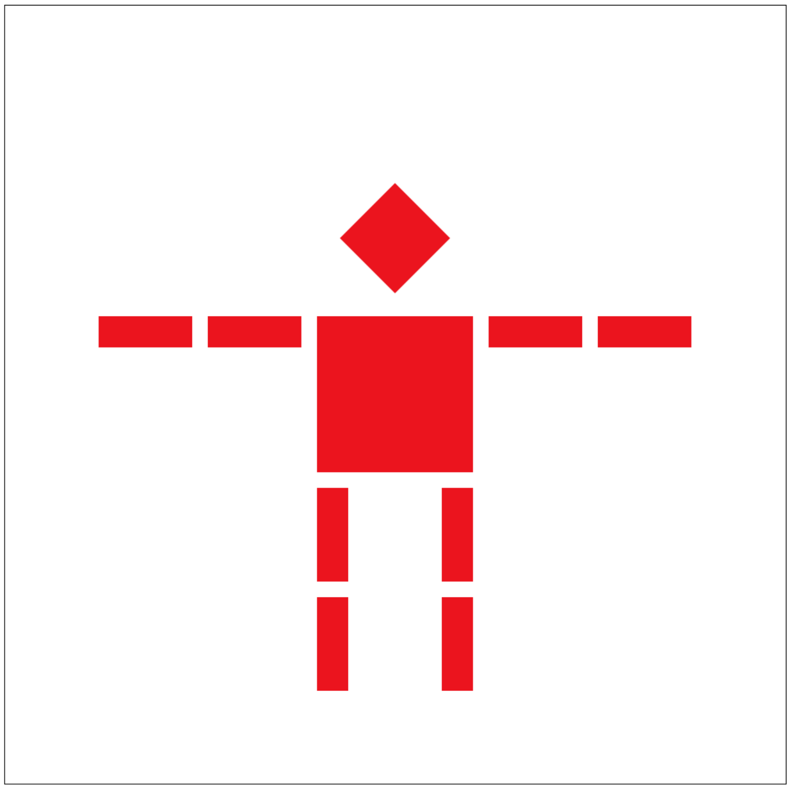

Figure 5: Task3 (svg/transforms/robot.svg w/ cubeman standing normal).

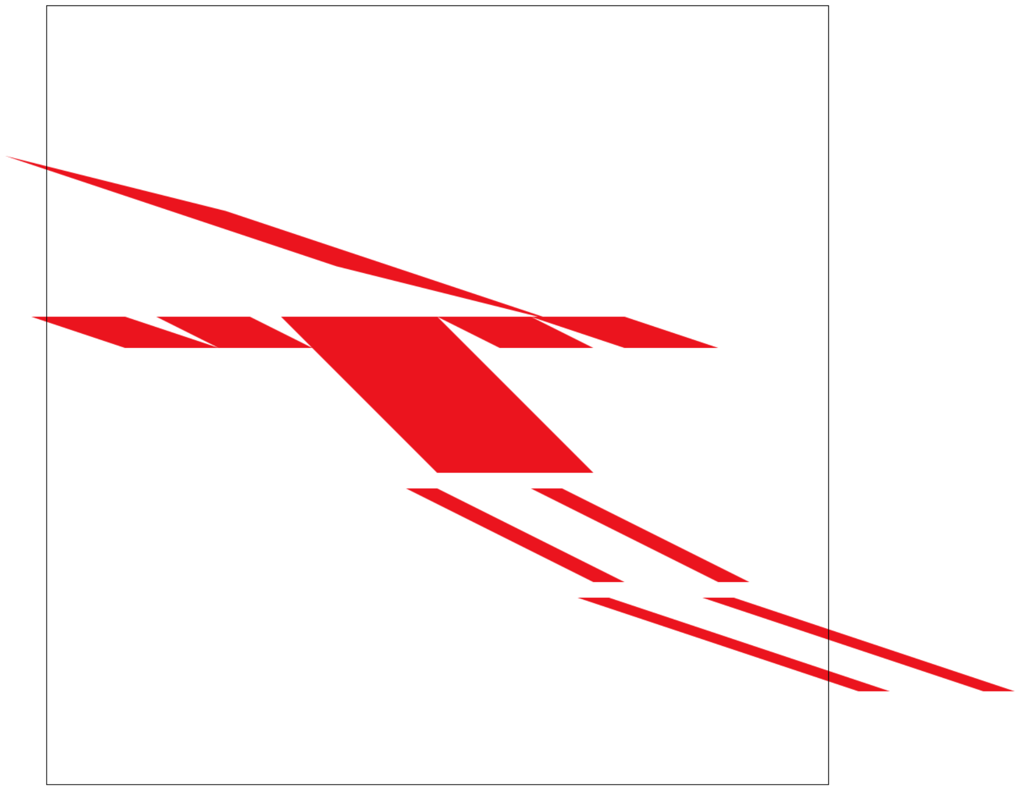

Figure 6: Task3 (svg/transforms/robot.svg w/ cubeman laying down).

Task 4:

Barycentric coordinates provide a way to represent any point inside a triangle as a weighted average of the triangle's vertices. In other words, given a triangle with vertices A, B, and C, any point P inside (or on the edge of) the triangle can be written as:

$P=\alpha A+\beta B+\gamma CP=\alpha A+\beta B+\gamma C$

with the constraints that

$\alpha+\beta+\gamma=1$and $\alpha, \beta, \gamma \geq 0$

The way it works is we used weights as influence by assigning the values α, β, and γ to represent the "influence" or "weight" of each vertex on the location of P. For example, if $\alpha=1\alpha=1$ (and the others are 0), then P is exactly at vertex A. Then we interpolate these coordinates which are especially useful for interpolating values (like color, texture coordinates, or normals) across the surface of the triangle. When you know the values at the vertices, you can interpolate them across the triangle using the barycentric weights. The weights can be thought of as areas of subtriangles that are formed by P with the edges of the triangle. If you draw lines from P to each vertex, you divide the triangle into three smaller triangles. The area of each smaller triangle (relative to the area of the full triangle) gives you the corresponding barycentric coordinate.

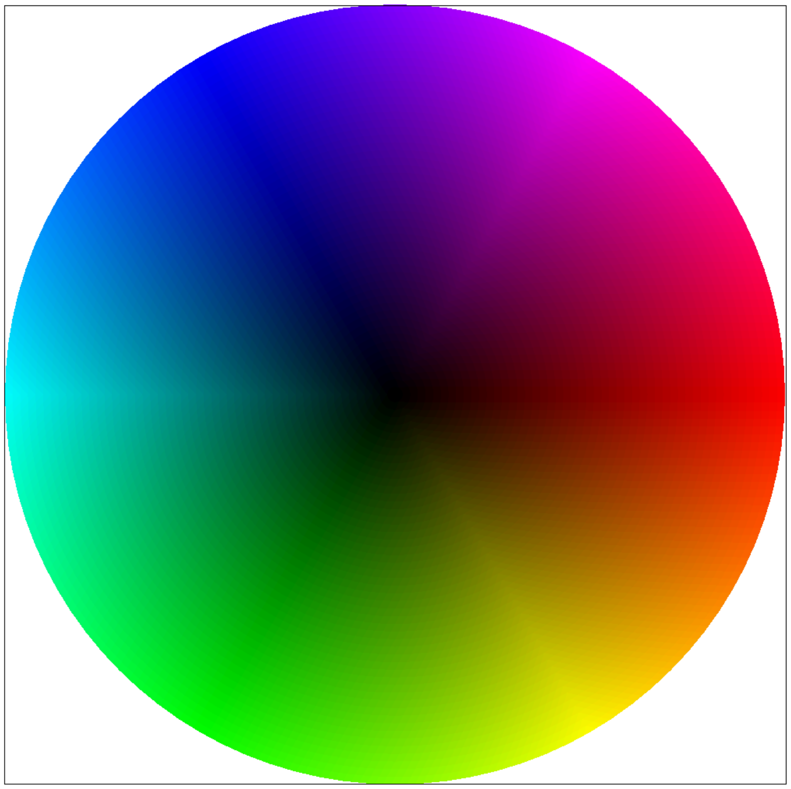

Figure 7: Task3 (svg/basic/test7.svg w/ default viewing and sample rate = 1).

Task 5:

Pixel sampling is the process of determining the color value for a pixel by “sampling” from a texture image. When mapping a texture onto a surface (like a triangle), each point on the surface is associated with a UV coordinate that corresponds to a position in the texture. However, because textures are stored as discrete arrays of texels (texture pixels), we must convert these continuous UV coordinates into discrete color values. Our Implementation for Texture Mapping used UV Mapping and interpolated the UV coordinates across the triangle using barycentric coordinates. This gives us a UV value for each sample point (or pixel) in the triangle. For the Sampling Methods; depending on the chosen PixelSampleMethod, we either can use nearest neighbor sampling or bilinear sampling.

Nearest Neighbor Sampling works by using the UV coordinate is scaled to the texture’s resolution, then rounded to the nearest integer. The texel at that integer coordinate is chosen as the sample.

Bilinear Sampling works by using the UV coordinate is scaled, and the fractional part is used to identify the four nearest texels. These four colors are then blended together using linear interpolation (first along one axis, then the other) based on the fractional parts.

We implemented both functions; in our triangle rasterization routine, after interpolating the UV coordinates using barycentrics, we call the appropriate sampling function (based on a toggle controlled by the GUI) to fetch the texture color. We also support supersampling by taking multiple subpixel samples per pixel, then averaging these to further reduce aliasing.

Using the pixel inspector and example SVG files from the svg/texmap/ directory, we observed clear differences between the sampling methods. In the screenshots below, we compare four scenarios. In nearest sampling at one sample per pixel, the texture appears blocky and pixelated because each pixel directly takes the color of its nearest texel. With nearest sampling at 16 samples per pixel, supersampling slightly smooths the image by averaging multiple samples, though the blockiness of the nearest neighbor method remains. Bilinear sampling at one sample per pixel already provides smoother gradients as it blends the colors of adjacent texels. The highest quality is achieved with bilinear sampling at 16 samples per pixel, where both bilinear interpolation and supersampling work together to create smooth transitions and significantly reduce aliasing.

When considering relative differences and their impact, high-frequency details and magnification play a key role. When a texture contains fine details or is magnified—such as when a small texture is stretched over a large area—nearest neighbor sampling introduces visible discontinuities and blockiness. Bilinear sampling, however, blends neighboring texels to produce a smoother gradient. The difference between low and high sample rates is also important. Increasing the number of samples per pixel, such as from 1 to 16, generally enhances quality by averaging out noise and aliasing artifacts. However, this improvement is more pronounced when using bilinear sampling, as the inherent interpolation already ensures smoothness, and supersampling further refines the result.

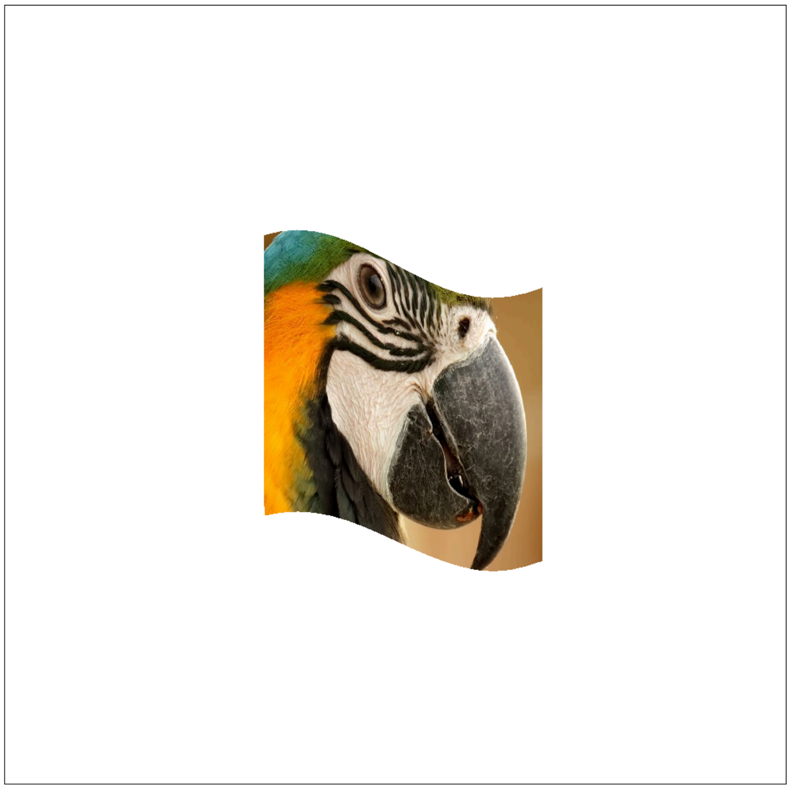

Figure 8: Task5 (svg/texmap/test4 w/ nearest sampling at 1).

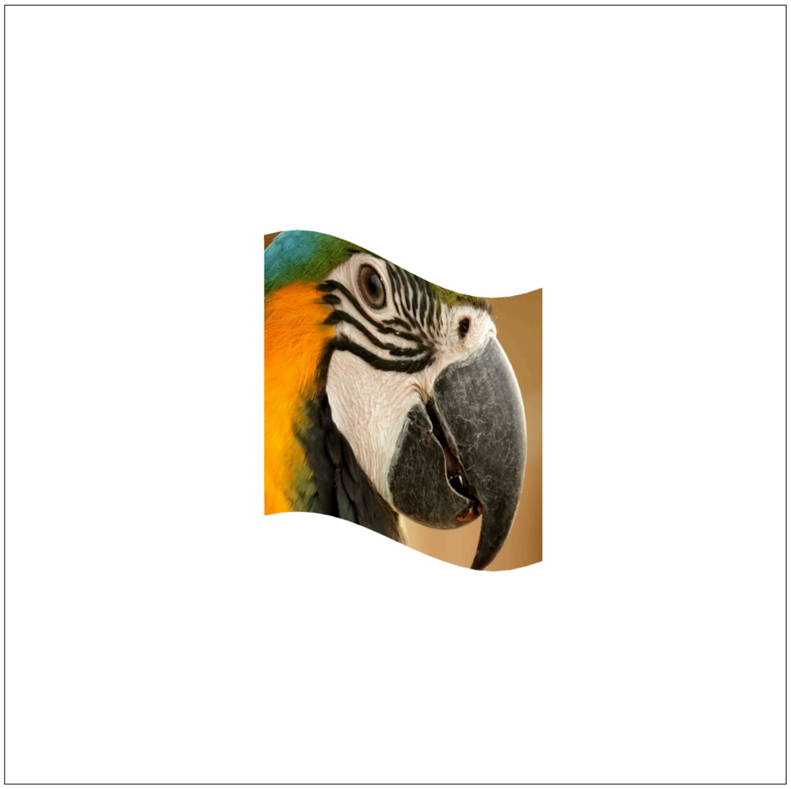

Figure 9: Task5 (svg/texmap/test4 w/ nearest sampling at 16).

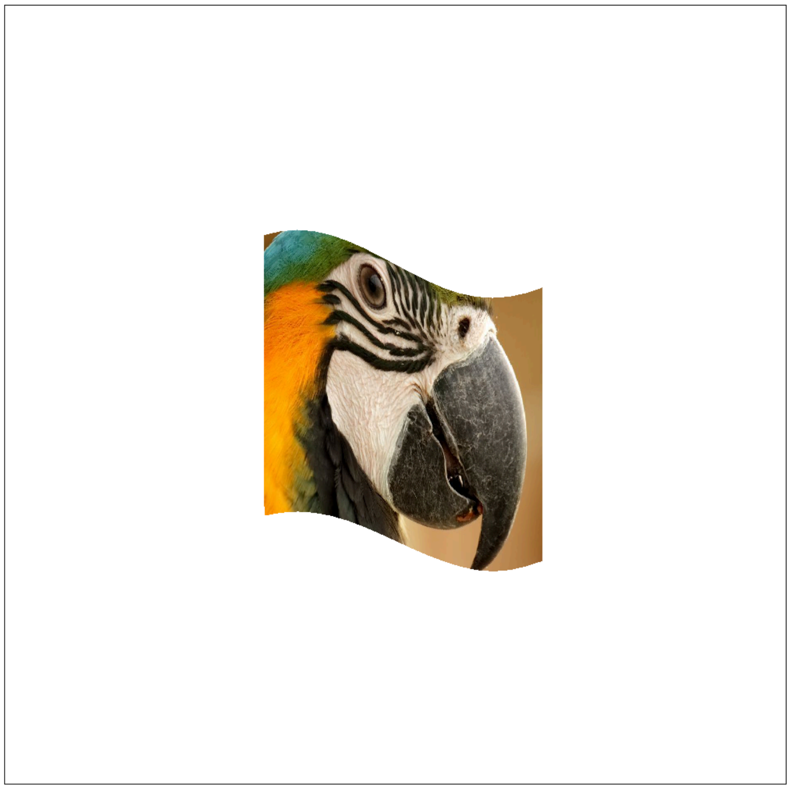

Figure 10: Task5 (svg/texmap/test4 w/ bilinear sampling at 1).

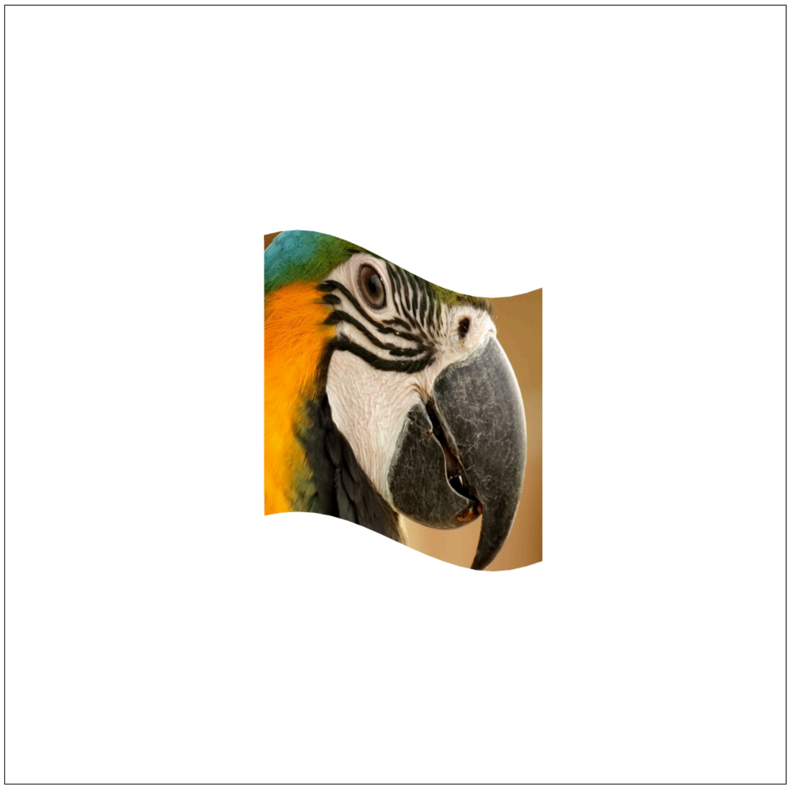

Figure 11: Task5 (svg/texmap/test4 w/ bilinear sampling at 16).

Task 6:

Level sampling is the process of selecting the appropriate level of detail (LOD) from a precomputed mipmap for texture mapping. Mipmaps are a series of pre-scaled versions of a texture, where each level is a downsampled (lower resolution) version of the original. The key idea is to sample from a texture level that best matches the size of the texture's footprint on the screen, thereby reducing aliasing and improving performance.

When a textured object is viewed from a distance or at a steep angle, many texels may map onto a single pixel. Sampling from the full-resolution texture (level 0) in such cases can cause severe aliasing—visual artifacts such as moiré patterns or shimmering. Level sampling uses the rate at which the UV coordinates change over screen space (i.e., the UV derivatives) to compute a continuous LOD value. This value determines which mipmap level to use, or how to blend between levels:

- L_ZERO: Always sample from level 0 (full resolution). This is the simplest approach but may cause aliasing when the texture is minified.
- L_NEAREST: Compute a continuous LOD from the UV derivatives and round it to the nearest integer. This selects one of the precomputed mipmap levels.
- L_LINEAR: Compute a continuous LOD and then linearly interpolate between the two nearest mipmap levels. This provides smooth transitions between levels, reducing abrupt changes.

In our implementation, level sampling begins with computing the differences in UV coordinates between adjacent screen samples. Specifically, we compute the differences as follows:

Δ^x_uv = p_dx_uv − p_uv
Δ^y_uv = p_dy_uv − p_uv

These differences are then scaled by the width and height of the full-resolution texture. The larger of the two scaled derivative magnitudes indicates how much the texture is being stretched or compressed on the screen. We then compute the level-of-detail (LOD) using the equation:

$$\text{LOD} = \log_2(\max(||\Delta^x\_u||, ||\Delta^y\_u||))$$

This calculation provides a continuous level value that reflects the appropriate mipmap level for the current texture sampling.

Once the LOD is computed, it is used to determine how to sample the texture. For the L_NEAREST mode, we round the continuous LOD to the nearest integer and then sample from that specific mipmap level, using either nearest-neighbor or bilinear sampling as determined by the pixel sampling method. In contrast, for the L_LINEAR mode, we take the two adjacent mipmap levels—the floor and the ceiling of the continuous LOD—and blend their results according to the fractional part of the LOD. This blending creates a smooth transition between mipmap levels, further reducing aliasing artifacts.

The texture sampling function has been updated to accept a SampleParams structure that contains not only the interpolated UV coordinate at the current sample point but also the UV coordinates for neighboring offsets (which are used to compute the derivatives). This integration allows the system to dynamically choose or blend between mipmap levels based on the view transformation and the level of texture detail required.

When considering tradeoffs between different techniques, pixel sampling, which involves a single sample per pixel, is the fastest method since it computes only one sample and stores only one color per pixel. However, it provides minimal antialiasing because it does not capture subpixel variations. Supersampling, on the other hand, subdivides each pixel into multiple subpixel samples (for example, 4 or 16 samples per pixel). Although this approach greatly increases computational load and memory usage—since the supersample buffer must store multiple color values per pixel—it offers excellent antialiasing by averaging the color variations across the pixel area, thereby reducing jagged edges and moiré artifacts.

Level sampling via mipmapping strikes a balance between performance and quality. It is often faster than full supersampling when textures are minified because sampling from a lower-resolution texture reduces both memory bandwidth and processing time. While mipmapping does require additional memory to store the mipmap chain, each successive level is smaller than the previous one, making the total extra memory relatively modest. In terms of antialiasing, level sampling is particularly effective at reducing aliasing in texture mapping, especially for distant objects or those viewed at steep angles. However, when used without supersampling, it may not capture subpixel variations as effectively as when the two techniques are combined.

Overall, pixel sampling is the fastest but least accurate, supersampling is the slowest due to the increased number of samples per pixel, and level sampling (using mipmapping) offers a balanced approach, particularly when textures are minified, as it avoids processing high-resolution textures unnecessarily. In terms of memory, pixel sampling uses minimal resources, supersampling requires a larger buffer proportional to the number of samples per pixel, and level sampling requires extra memory for the mipmap levels, though each level is smaller. Finally, regarding antialiasing quality, pixel sampling provides limited smoothing, supersampling significantly reduces aliasing by averaging multiple subpixel samples, and level sampling excels at reducing aliasing from texture minification—especially when combined with bilinear interpolation—to yield smooth, high-quality results without incurring the full cost of supersampling.

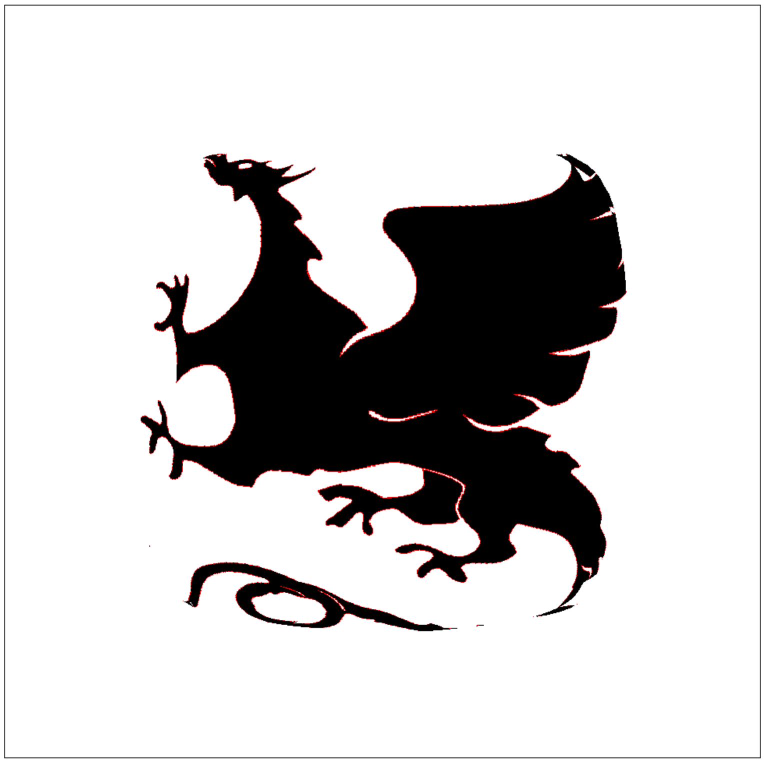

Figure 12: Task6 (L_ZERO & P_NEAREST at sampling rate = 1).

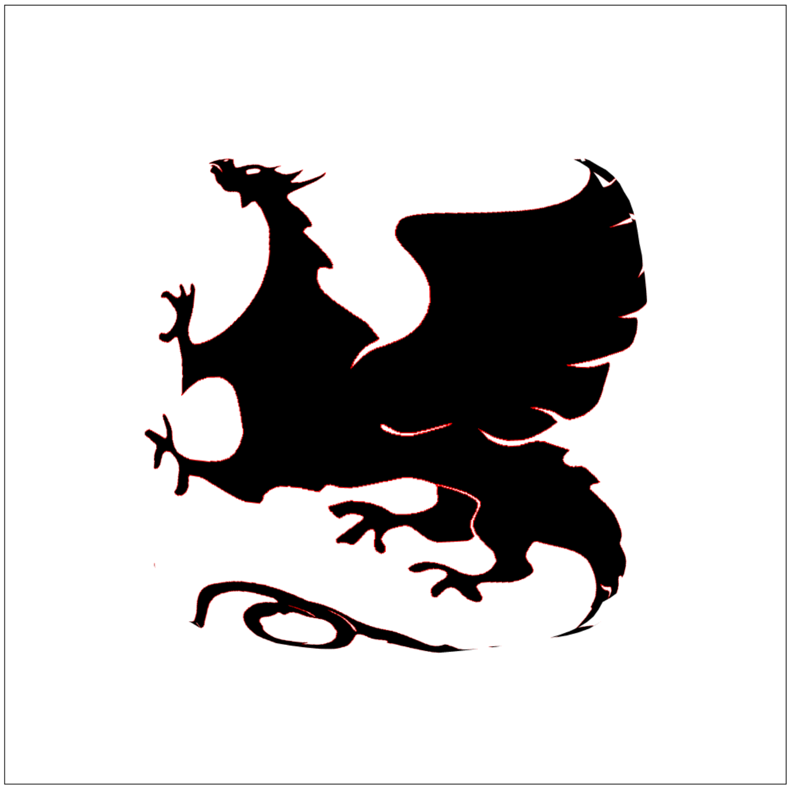

Figure 13: Task6 (L_ZERO & P_NEAREST at sampling rate = 16).

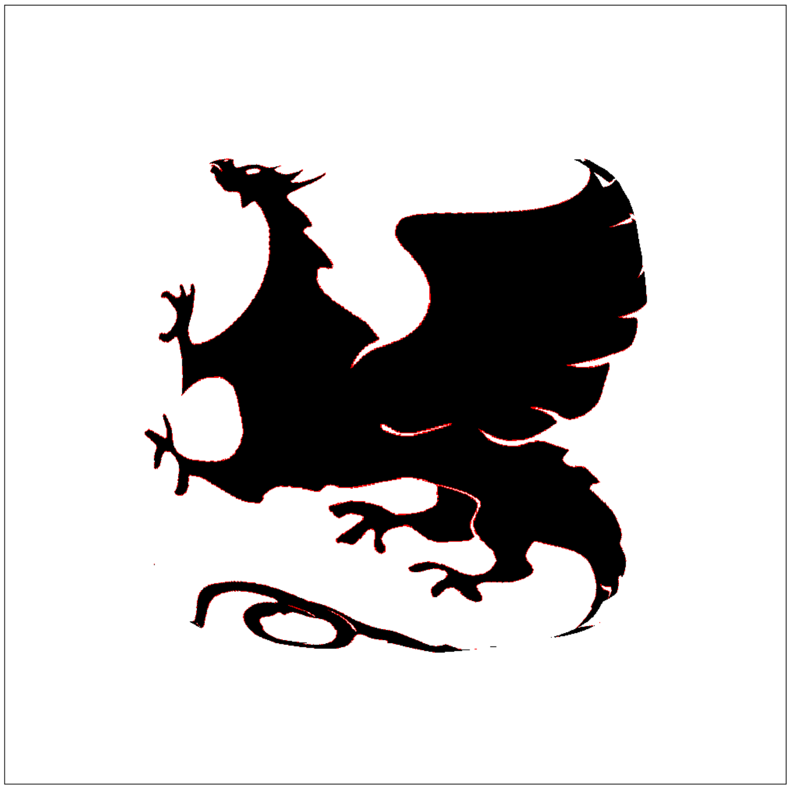

Figure 14: Task6 (L_ZERO & P_LINEAR at sampling rate = 1).

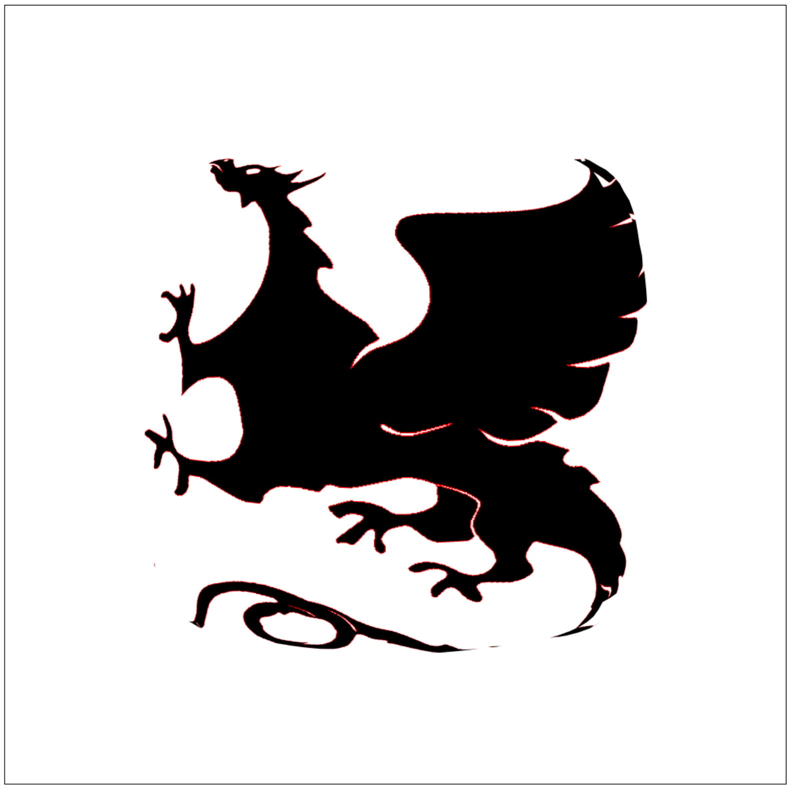

Figure 15: Task6 (L_ZERO & P_LINEAR at sampling rate = 16).

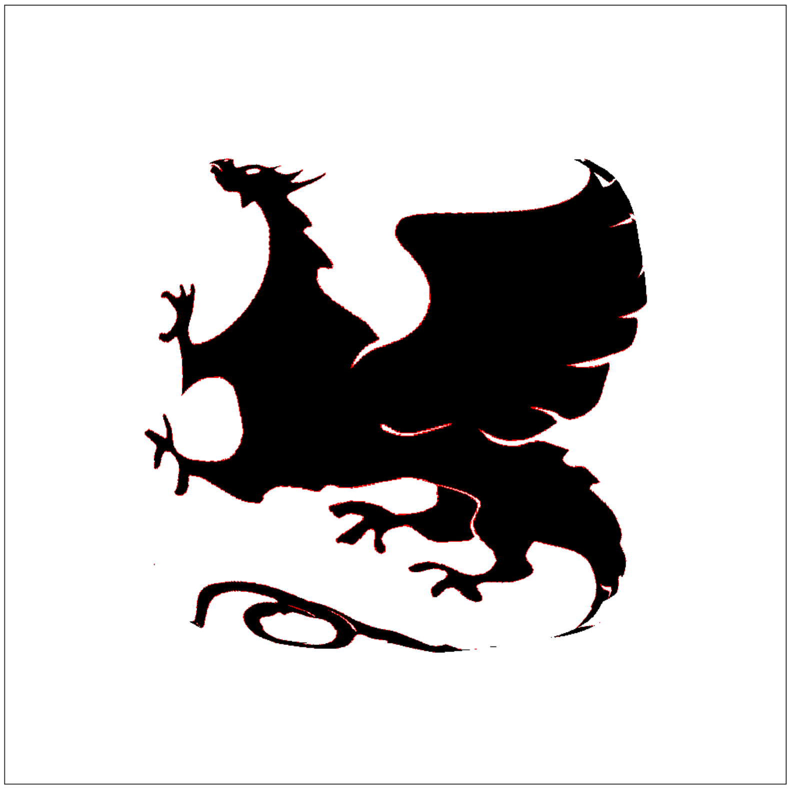

Figure 16: Task6 (L_NEAREST & P_NEAREST at sampling rate = 1).

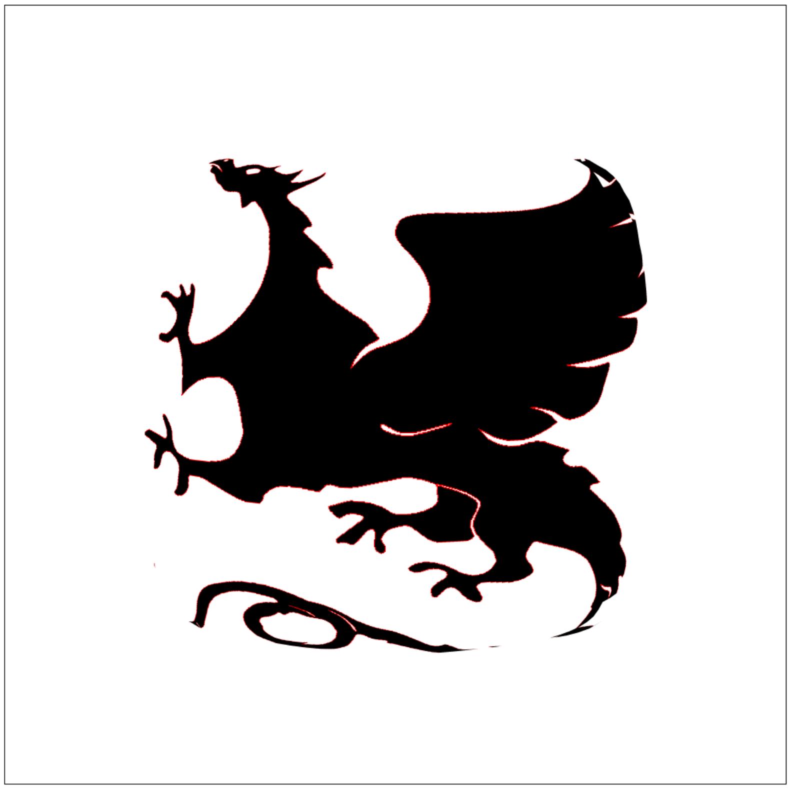

Figure 17: Task6 (L_NEAREST & P_NEAREST at sampling rate = 16).

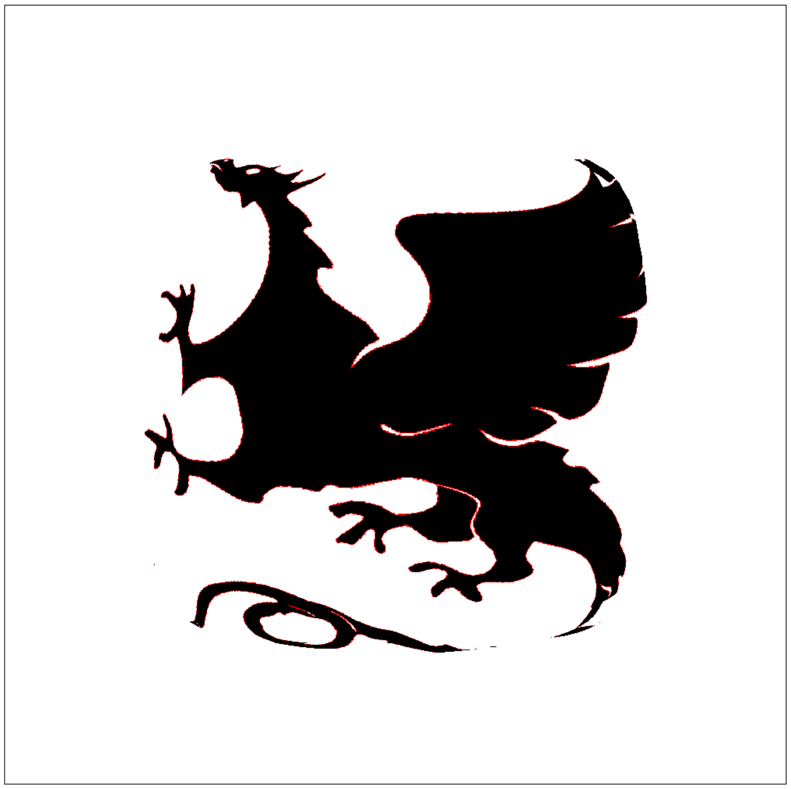

Figure 18: Task6 (L_NEAREST & P_LINEAR at sampling rate = 1).

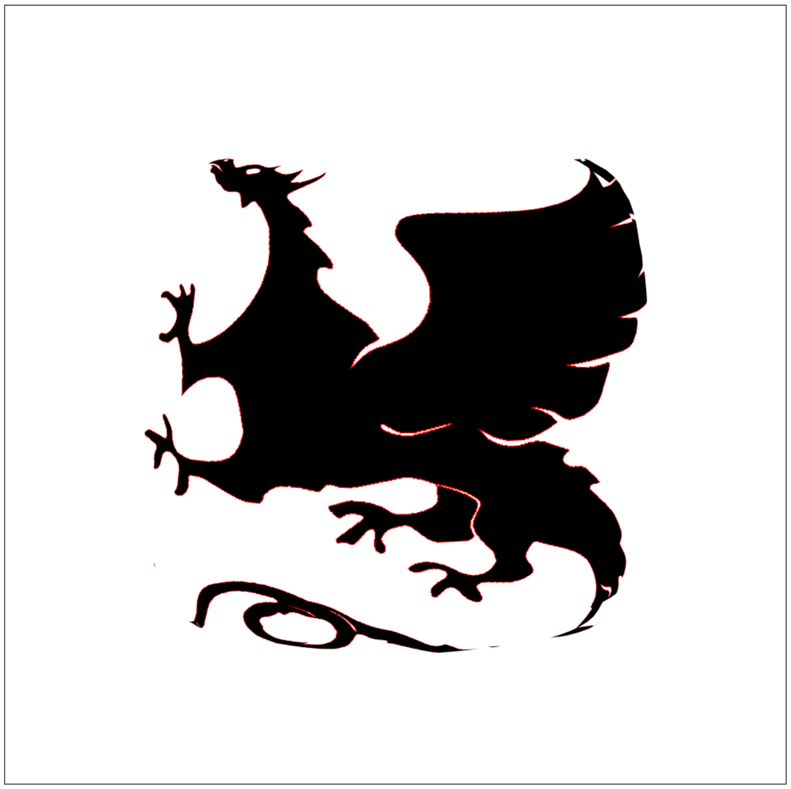

Figure 19: Task6 (L_NEAREST & P_LINEAR at sampling rate = 16).