CSC4140 Assignment 3

Computer Graphics

March 19, 2022

Transformation

This assignment is 18% of the total mark.

Strict Due Date: 11:59PM, Mar 19th, 2022

Student ID: 119010265

Student Name: Shi Wenlan

This assignment represents my own work in accordance with University regulations.

Signature: Shi Wenlan

1 How to deal with tasks

1.1 Draw a single color triangles (20 points)

Firstly, find the minimum x, minimum y, maximum x and maximum y of three vertices of the given triangle using *min*() and *max*(). The AABB bounding box is from (minimum x, minimum y) to (maximum x, maximum y).

Secondly, traverse all the pixels in the bounding box. Then use Sample coordinates = the integer point + (0.5,0.5) to check whether it is in the triangle using *insideTriangle*(). So for the traversed coordinates, the value of x should be in the range of [minimum x, maximum x), and the value of y should be in the range of [minimum y, maximum y). Since this algorithm does not visit any pixel out of the bounding box, it should be no worse than one that checks each sample within the bounding box of the triangle.

Thirdly, if the pixel is in the triangle, then draw the given color into the pixel in the *sample_buffer* through *fill_pixel*.

How to implement *insideTriangle*(): Firstly, I convert the three vertices of the triangle (a, b ,c) and the point to be judged (p) into homogeneous coordinates. If (p and c are on the same side of ab) and (p and b are on the same side of ac) and (p and a are on the same side of bc), then p is in the triangle. To compute the condition, I get edge vectors ab, bc, ca, ap, bp, cp through points subtraction. Take the first part of the condition as the example, if cross product of ab and bc and the cross product of ab and ap have the same direction, then p and c are on the same side of ab. To check whether the directions are the same or not, I compute the dot product of two cross products. If the dot product is larger than 0, then two cross products have the same direction.

Bug found and Improvement of *insideTriangle*(): While testing the program, I found that the 7th test case of the basic folder and the two test cases of the hardcore folder have a white line from the upper left corner to the lower right corner. After I looked at the svg file, I found that it was because the sampling points on the diagonal happened to be on the side where the two triangles coincided at the same time. According to the definition written above, the pixels on the sides are not inside the triangle, so the pixels on the diagonal are not colored. So I changed the judgment condition from greater than 0 to greater than or equal to 0, so that to color the pixels on the sides of the triangle and fix the bug.

Crop: The png screenshot of basic/test4.svg with the default viewing parameters is as following:

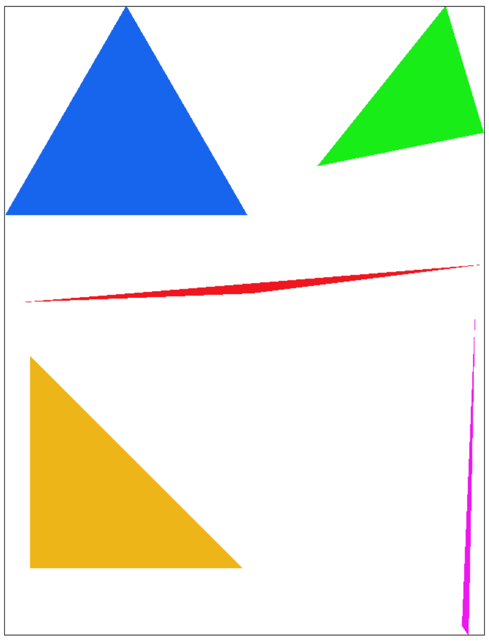


Figure 1: screenshot of basic/test4.svg

1.2 Antialiasing (20 points + 10 extra)

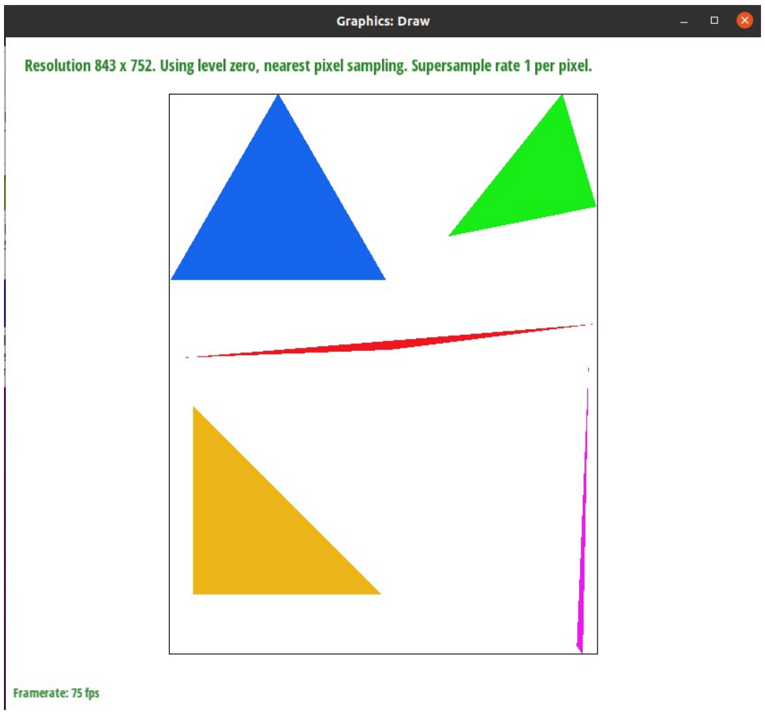
Firstly, find the AABB bounding box as last subsection. Then for every pixel in the bounding box, evenly distribute sqrt($sample\_rate$) * sqrt($sample\_rate$) sampling points.

Secondly, for every sampling point, check whether it is in the triangle using $insideTriangle()$. If the sampling point is in the triangle, color the sampling point with the given color.

Thirdly, for every pixel, take the average of the colors of all its sampling points as the color of the pixel.

Why is supersampling useful: When the sampling frequency is low, the image will have obvious aliasing. After increasing the sampling points and averaging the sampling points, some of the pixels near the edge will become faded, so that the image will not be so obvious from the macro perspective, so as to have the effect of anti-aliasing.

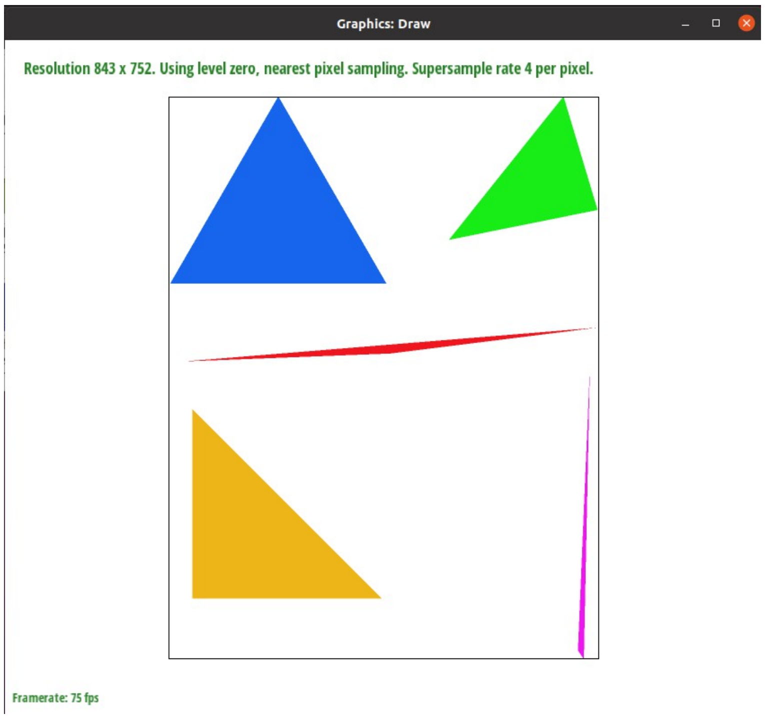
Crops: The screenshots of basic/test4.svg with the default viewing parameters and sample rates 1, 4, and 16 are as following:


Graphics: Draw

Resolution 843 x 752. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 75 fps

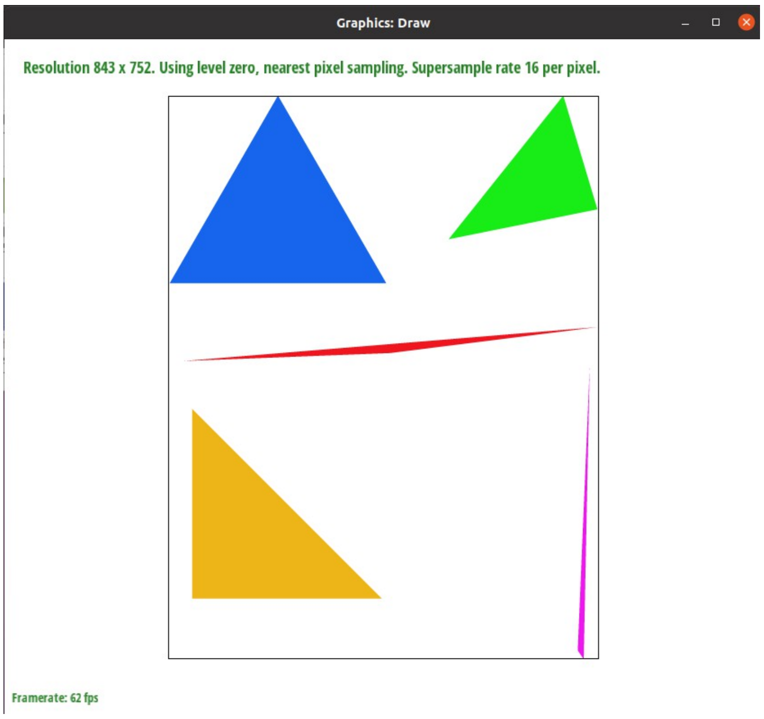
Figure 2: screenshot of sample rates 1


Graphics: Draw

Resolution 843 x 752. Using level zero, nearest pixel sampling. Supersample rate 4 per pixel.

Framerate: 75 fps

Figure 3: screenshot of sample rates 4



Resolution 843 x 752. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.

Framerate: 62 fps

Figure 4: screenshot of sample rates 16

Why these results are observed: The more sampling points, the more accurate the proportion of each edge pixel that is inside the triangle, and after averaging the color of the pixels, the more natural the color of the edge pixels blends with the background from a macro perspective.

Credit - how to implement it with no extra memory: For each pixel, when traversing its sample points, *count* plus 1 for each sample point within the triangle. After traversing the sample points, mix the foreground (the given color) and background colors in a $count$:$(sample\_rate - count)$ ratio. This eliminates the need to store additional sampled point color data. The cost of this is that white seams will appear when multiple adjacent triangles are put together (details will be explained in subsection 1.5 paragraph "Bug of white seams" with figures).

1.3 Transforms (10 points)

Complete the function with formulas following:

➢ Scale

$$\mathbf{S}(s_x, s_y) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

➢ Rotation

$$\mathbf{R}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

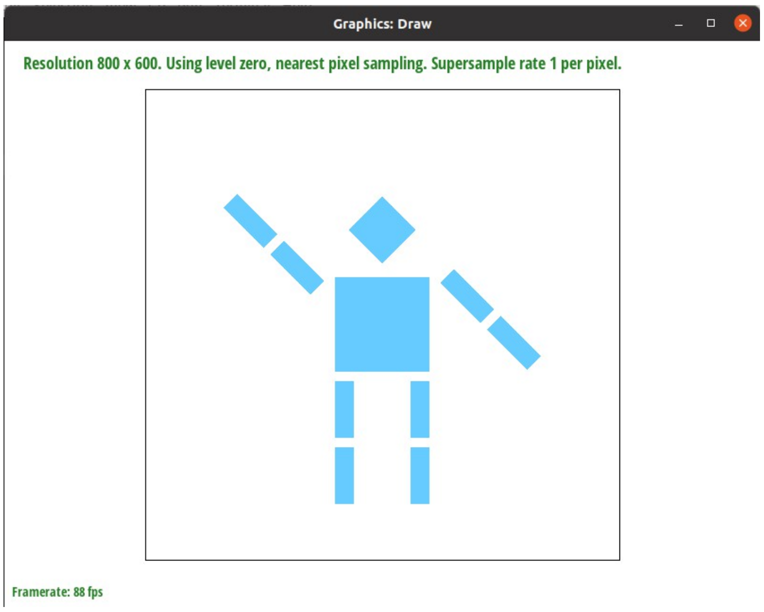
➢ Translate

$$\mathbf{T}(t_x, t_y) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 5: formulas of transformation

Note to convert degree to radiant before passing the parameter to the function $cos()$ and $sin()$.

Crop: The screenshot of my rendered drawing of *docs/my_robot.svg* is as following:



Graphics: Draw

Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 88 fps

Figure 6: my waving cubeman

1.4 Barycentric coordinates (10 points)

Given the coordinates of three vertices A, B, C of a triangle, a point (x, y) in the plane can be written as a linear combination of the coordinates of the three vertices, that is $(x, y) = \alpha A + \beta B + \gamma C$ and satisfy $\alpha + \beta + \gamma = 1$, then the weight α, β, γ of the three coordinates A, B, C is called the **barycentric coordinate** of point (x, y).

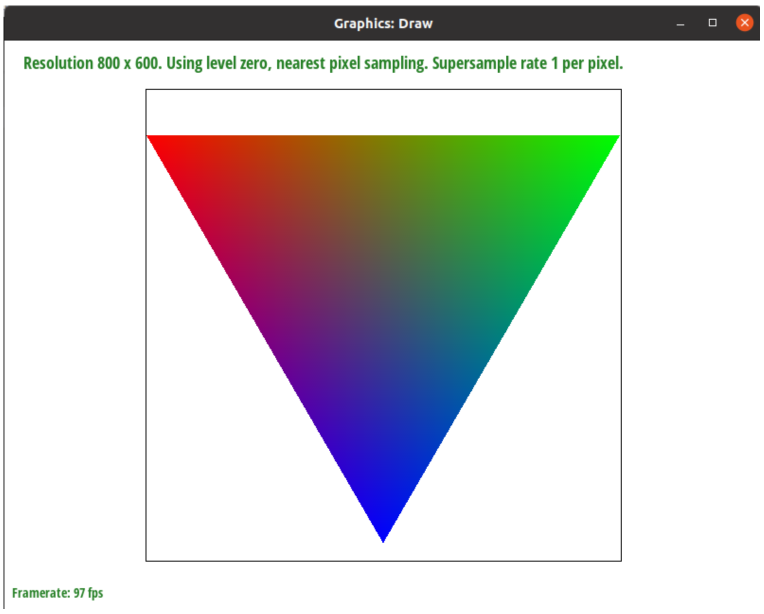
The barycentric coordinate of point (x, y) can be compute using following formula:

$$\gamma = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}.$$

$$\beta = \frac{(y_a - y_c)x + (x_c - x_a)y + x_a y_c - x_c y_a}{(y_a - y_c)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a},$$
$$\alpha = 1 - \beta - \gamma.$$

Figure 7: formula of barycentric coordinate

For each sampling point, compute the barycentric coordinate respect to the given triangle, and blend the color of three vertices using the proportion of barycentric coordinate. Take docs/barycentric.svg as an example:

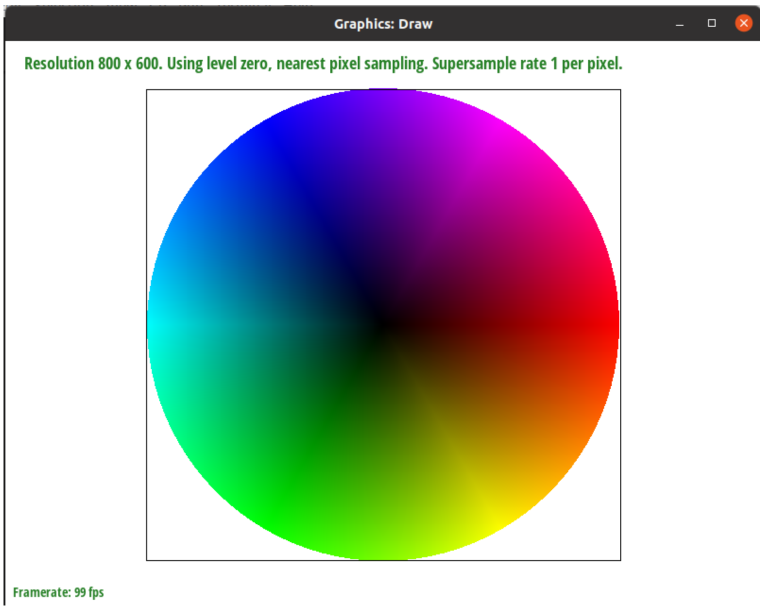


Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 97 fps

Figure 8: barycentric coordinate example

Crop: The png screenshot of basic/test7.svg with the default viewing parameters is as followings and sample rate 1:



Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 99 fps

Figure 9: screenshot of basic/test7.svg

1.5 Pixel sampling for texture mapping (10 points)

Pixel sampling is the operation of converting a continuous image (analog image) in the spatial domain into a discrete sampling point (pixels) set (digital image), and these values can be uniformly spaced or non-uniformly spaced. To implement texture mapping, I used following steps:

Firstly, I assign discrete sampling points to each pixel in the bounding box according to the sampling rate, and use the barycentric coordinates to calculate the uv coordinates corresponding to the sampling points.

Secondly, u and v are in the range of [1, 0], and the actual coordinates on the texture is (u * width of the texture, v * height of the texture).

Thirdly, using actual coordinates on the texture to sample the color using required sampling method on the texture as the color of the sampling point.

Sampling methods - nearest: The actual coordinates on the texture are floating point numbers, and they should be converted to integer numbers so as to get the color of the specific pixel on the texture. This sampling method rounds the coordinates of floating point numbers to integers (i.e. if the fractional part is less than or equal to 0.5, it will be discarded, and if it is greater than 0.5, it is regarded as 1 and added to the integer part).

Sampling methods - bilinear: Take 4 nearest pixels of the actual coordinate, then compute the blend color using formulas as following:

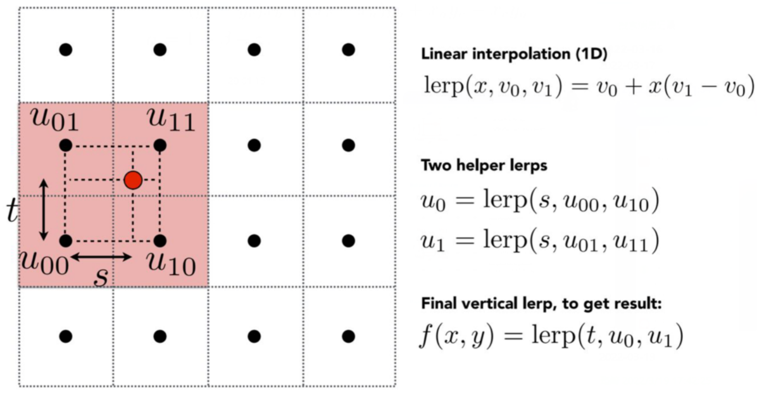
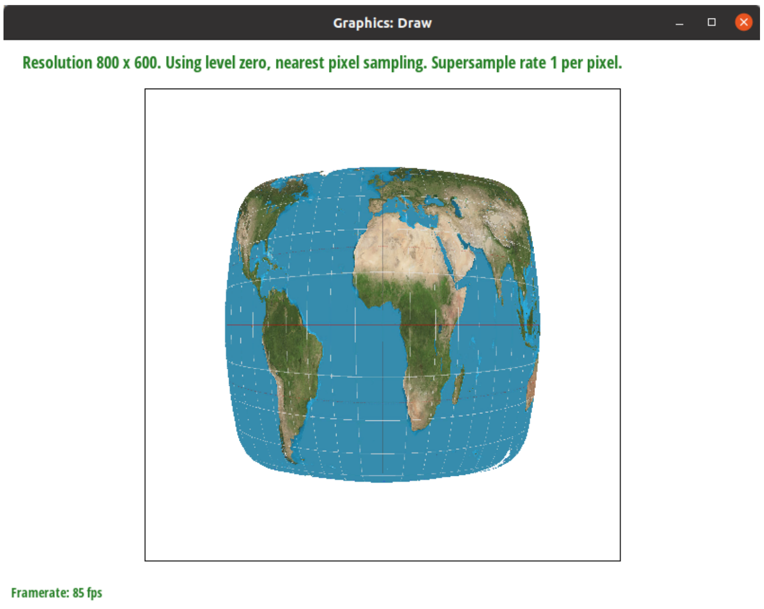

u_{01}
u_{11}
t
u_{00}
s
u_{10}
Linear interpolation (1D)
$\mathrm{lerp}(x, v_0, v_1) = v_0 + x(v_1 - v_0)$
Two helper lerps
$u_0 = \mathrm{lerp}(s, u_{00}, u_{10})$
$u_1 = \mathrm{lerp}(s, u_{01}, u_{11})$
Final vertical lerp, to get result:
$f(x, y) = \mathrm{lerp}(t, u_0, u_1)$

Figure 10: formula of bilinear sampling

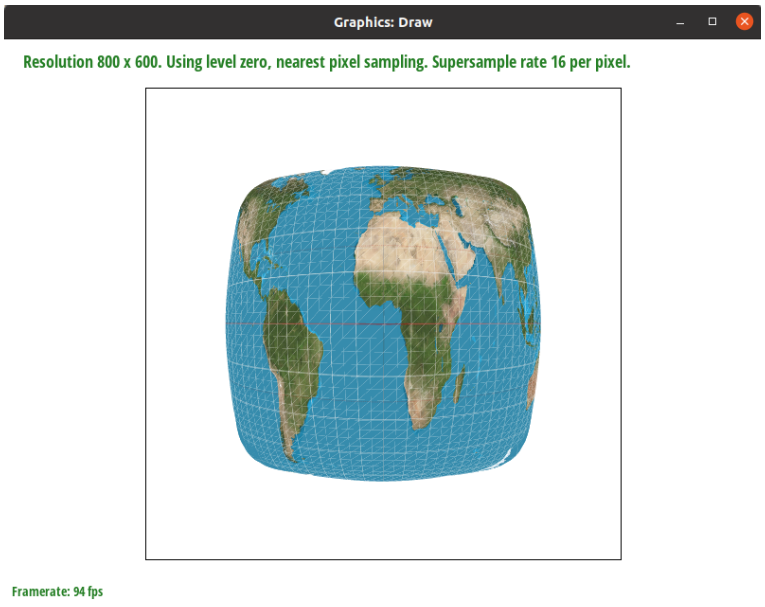
Crops: The screenshots using nearest sampling at 1 sample per pixel, nearest sampling at 16 samples per pixel, bilinear sampling at 1 sample per pixel, and bilinear sampling at 16 samples per pixel are as following:



Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 85 fps

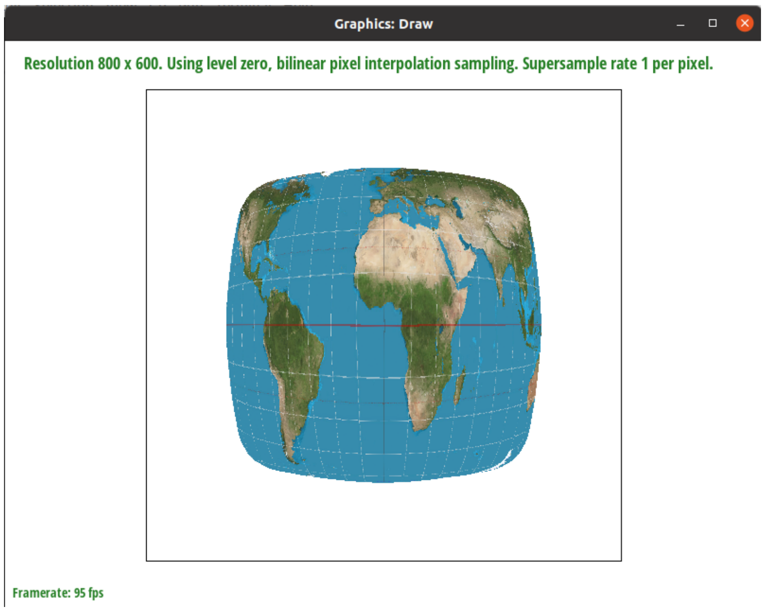
Figure 11: nearest sampling at 1 sample per pixel


Graphics: Draw

Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 16 per pixel.

Framerate: 94 fps

Figure 12: nearest sampling at 16 sample per pixel


Graphics: Draw

Resolution 800 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.

Framerate: 95 fps

Figure 13: bilinear sampling at 1 sample per pixel

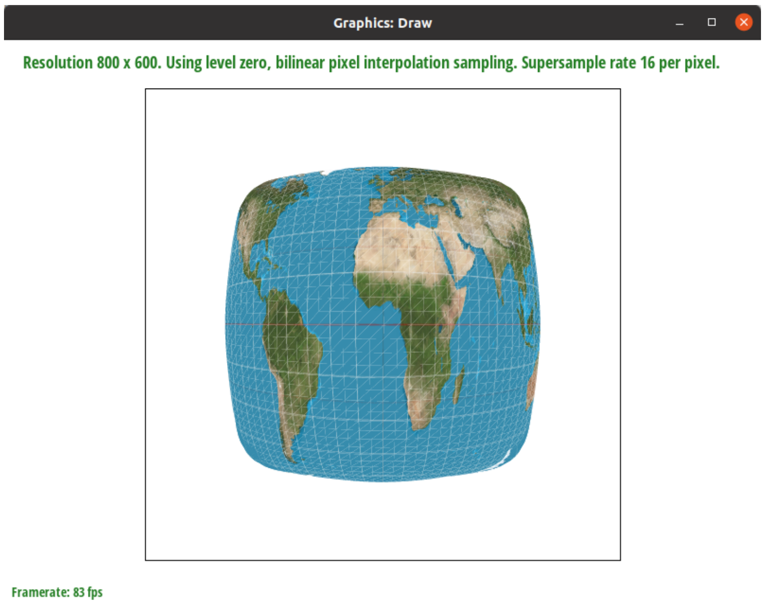

Graphics: Draw
Resolution 800 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 16 per pixel.
Framerate: 83 fps

Figure 14: bilinear sampling at 16 sample per pixel

When the sampling frequency is low and there are relatively fine details in the texture, the nearest sampling method is easy to miss these details during sampling, while the bilinear sampling method can take into account some details. As in the above figures, there are relatively thin vertical white lines in the texture. Figure 11 loses more vertical white line information, while Figure 13 retains relatively more vertical white line information. This is because the sampling span is larger than the size of the details. The nearest sampling method only samples one pixel, making it easier for details to be completely skipped during sampling, while the bilinear sampling method samples 4 adjacent pixels at a time, so details are more likely to be retained.

When the sampling rate is high, both sampling methods can take into account the details, but because each sampling result of bilinear is a linear combination of four pixels, the color position information is retained more accurately, so the image is smoother than that generated by nearest sampling method. Figure 12 and Figure 14 are almost the same at first glance, but if you look closely at the part of the green area, you can see that Figure 14 is smoother than Figure 12.

Bug of white seams: White seams appear between closely assembled triangles in both Figures 12 and 14. This is because in order to save memory overhead during supersampling, each time a triangle is drawn, the edge is mixed with the foreground color and the background color. If a triangle is stacked on top of another triangle, the pixels on the edge will only do one effective blend of the foreground color and the background color, and the display will be normal. However, if the two triangles are closely assembled, and the edge pixels would do two effective blends of the foreground color and the background color, the proportion of the background color in the final displayed color will be higher than expected. As shown in Figure 12 and Figure 14, the edge of the triangle is is biased to the background color (white), resulting in a white seam. I drew a detailed example to explain, see the following figure:

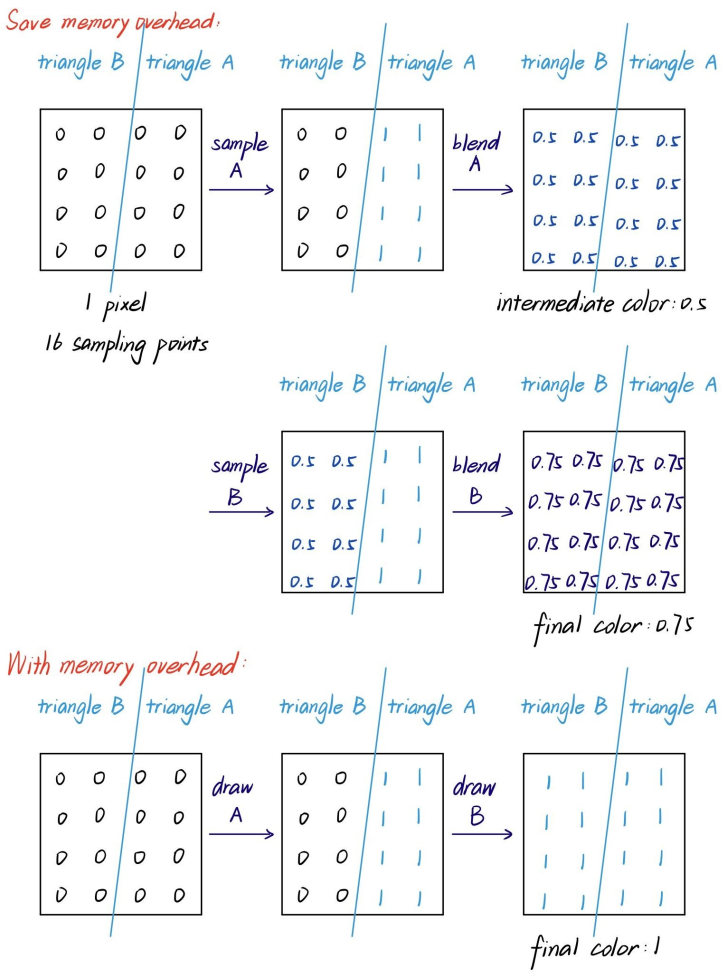


Figure 15: detailed example to explain the bug

1.6 Mipmapping (30 Points)

Area of screen pixel maps to different size of region of texture. Some pixels in a picture have a large footprint in the texture space, which is under-sampling, and moiré will appear and reduce the quality of the picture. If supersampling with a high sampling rate is used in order to maintain the overall high quality of the image, the computational cost is too high. To reduce the cost, the mipmapping method precomputes different levels of texture. Level 0 represents the original texture, which is also the texture with the highest precision. As the level increases, the average of 4 adjacent pixels is combined into one pixel for each level, so the higher the level, the greater the The footprint of the area query. The next thing to do is to select textures of different levels according to the footprint size of the screen pixels, and then perform a point query, which is actually equivalent to performing an area query on the original texture. This process of selecting the appropriate level is called **level sampling**. To compute desired level number, following steps are needed:

Firstly, for each sampling point (x, y), use the barycentric coordinates to calculate the uv coordinates corresponding to (x, y), (x + 1, y), and (x, y + 1).

Secondly, u and v are in the range of [1, 0], and the actual coordinates on the texture is (u * width of the texture, v * height of the texture).

Thirdly, compute du/dx, dv/dx, du/dy, dv/dy trough subtractions of actual coordinates. And compute desired level D using following formula:

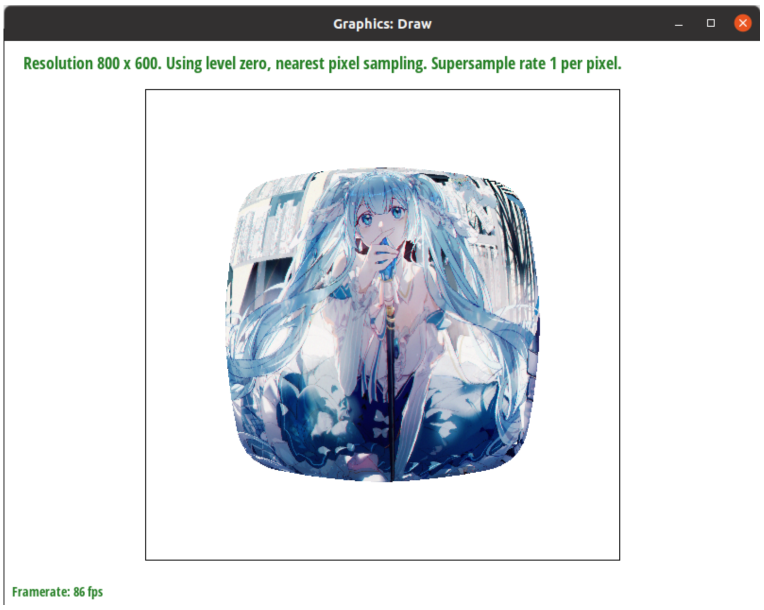
Figure 16: formula of level computing

Then we can sample color from level D from pixel (u * D.width, v * D.height).

Trade-offs:

technique	speed	memory usage	antialiasing power
bilinear pixel sampling	relatively slow	small	strong
nearest pixel sampling	fast	small	relatively weak
bilinear level sampling	relatively slow	small	strong
nearest level sampling	fast	small	relatively weak
supersampling(my code)	slow	small	strong
supersampling(traditional)	slow	big	strong

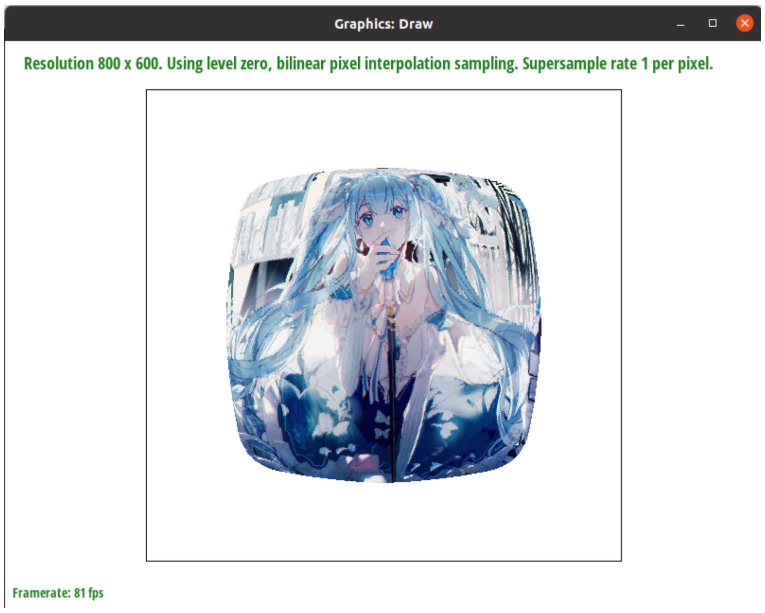
Crops: The screenshots of four versions of the image are as following:



Resolution 800 x 600. Using level zero, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 86 fps

Figure 17: L_ZERO and P_NEAREST

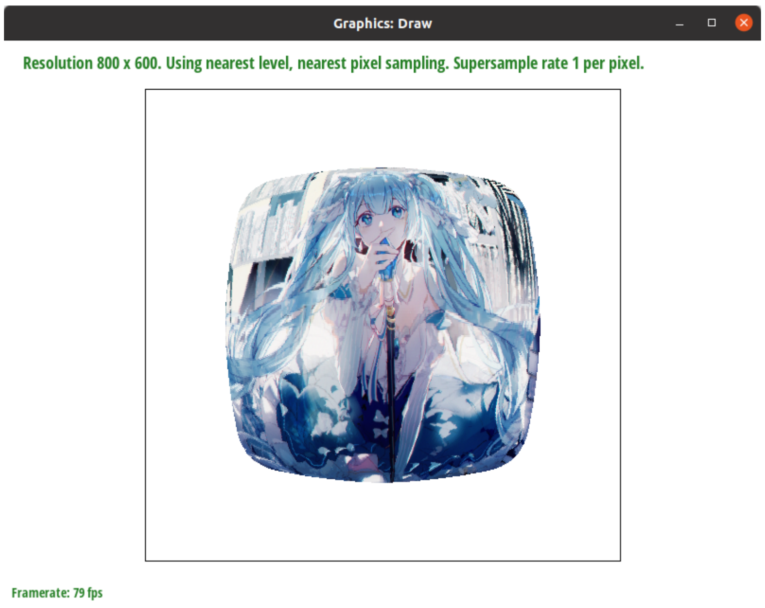


Graphics: Draw

Resolution 800 x 600. Using level zero, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.

Framerate: 81 fps

Figure 18: L_ZERO and P_LINEAR

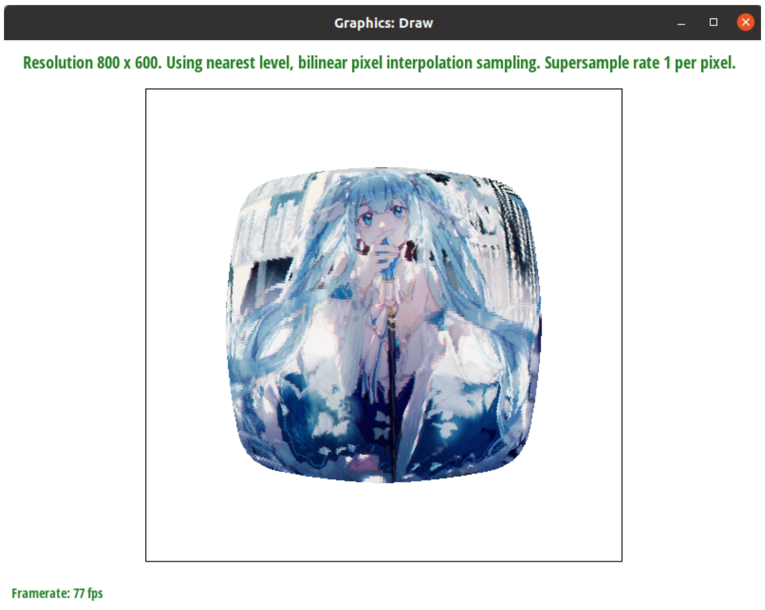


Graphics: Draw

Resolution 800 x 600. Using nearest level, nearest pixel sampling. Supersample rate 1 per pixel.

Framerate: 79 fps

Figure 19: L_NEAREST and P_NEAREST



Resolution 800 x 600. Using nearest level, bilinear pixel interpolation sampling. Supersample rate 1 per pixel.

Framerate: 77 fps

Figure 20: L_NEAREST and P_LINEAR

There is no absolute good or bad method, only suitable and not suitable.