CS 184 Assignment 1

Hector Li

July 2025

Contents

0 Logistics Note **2**

1 Task 1 **2**
1.1 Rasterization Process . . . 2
1.2 Bounded Box Optimization . . . 3
1.3 Float Operation Rounding Errors and Handling . . . 4
1.4 Picture Examples . . . 5
1.5 Possible Improvements . . . 9

2 Task 2 **9**
2.1 Supersampling Algorithm . . . 9
2.2 Picture Examples . . . 10
2.3 Comparison with Jittered Sampling . . . 20

3 Task 3 **25**

4 Task 4 **25**
4.1 Barycentric Coordinates . . . 25
4.2 Picture Examples . . . 26

5 Task 5 **31**
5.1 Pixel Sampling . . . 31
5.2 Picture Examples . . . 32

6 Task 6 **37**
6.1 Level Sampling . . . 37
6.2 Picture Examples . . . 39
6.3 Anisotropic Sampling . . . 46

0 Logistics Note

The PDF of this write-up and the original png files used in this document are all available on the website https://sigmundr6.github.io.

1 Task 1

1.1 Rasterization Process

When rasterizing triangles, we want to decide unambiguously whether each individual pixel is inside or outside the graph consisting of all these triangles. We know that if a pixel is inside a collection of triangles, it has to be inside one of the triangles in the collection. Therefore, it is safe to rasterize each triangle in the collection respectively and then take a union of the pixels inside. We can define a binary function $inside(t, x, y)$ which turns 1 if the point (x, y) is inside the triangle t, and 0 otherwise. Then the rasterization process is sampling the value of the inside function at the centre of each pixel.

By Euclidean geometry, we know that a triangle is planar and is the intersection of three lines. From optimization, we know that we can perform line tests to determine which side of the line a point falls upon. Specifically, for a line $l : Ax + By + C = 0$, we can define a function $L : \mathbb{R}^2 \to \mathbb{R}$ such that $L(x, y) = Ax + By + C$. For a point $P(x_0, y_0) \in \mathbb{R}^2$, When $A > 0$, we have

- If $L(x_0, y_0) > 0$, the point P is on the right side of the line l.
- If $L(x_0, y_0) = 0$, the point P is on the line l.
- If $L(x_0, y_0) < 0$, the point P is on the left side of the line l.

When $A = 0$ and $B > 0$, we have a horizontal line, and

- If $L(x_0, y_0) > 0$, the point P is above the line l.
- If $L(x_0, y_0) = 0$, the point P is on the line l.
- If $L(x_0, y_0) < 0$, the point P is below the line l.

For a triangle consisting of the three vertices $P_0(x_0, y_0)$, $P_1(x_1, y_1)$, and $P_2(x_2, y_2)$ in $\mathbb{R}^2$, we can uniquely determine the orientation of the vertices with the cross product $\overrightarrow{P1P0} \times \overrightarrow{P2P0} = (0, 0, (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0))$. The first two components are zero since by the definition of the cross product, the resulting vector has to be perpendicular to both starting vectors. Let z be the third component $(x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$. The absolute value $|z|$ is twice the area of the triangle $P_0P_1P_2$, If z is positive, the three vertices P_0, P_1, P_2 are in anticlockwise orientation. If z is negative, the three vertices are in clockwise orientation. If z is zero, the three vertices are on the same line and they do not form a triangle.

From optimization, we have the following result. Consider the triangle consisting of vertices $P_i = (x_i, y_i)$ where $i \in \mathbf{Z}_3$ and $x_i, y_i \in \mathbb{R}$ and the point

$P = (x, y)$. Let $L_i(x, y) = -(x - x_i)(y_{i+1} - y_i) + (y - y_i)(x_{i+1} - x_i)$. If the vertices are in anticlockwise orientation, then the point P is inside the triangle if all three L_i are positive, on the edge if two L_i are positive and the remaining one is zero, and on one vertex if two L_i are zero and the remaining one is positive. If the three vertices are in clockwise orientation, then we have a similar result if we flip the sign. Namely, if the vertices of the triangle are in clockwise orientation, then the point P is inside the triangle if all three L_i are negative, on the edge if two L_i are negative and the remaining one is zero, and on one vertex if two L_i are zero and the remaining one is negative. To eliminate the impact of the sign of z, we use the normalized line test $l_i = \frac{L_i}{z}$. We can conclude that P is inside the triangle if all three l_i are positive, on the edge if two l_i are positive and the remaining one is zero, and on one vertex if two l_i are zero and the remaining one is positive.

Let us consider one theoretical situation where P is on one edge of the triangle. If this edge is shared by another triangle, we might run into the situation where two triangles try to paint the same pixel with different colors. As a result, we could have a flickering pixel. Therefore, we can set up a tie-breaker rule that a point P on the edge e is inside the triangle if e is a top or left edge. Let the two vertices of e be (x_0, y_0) and (x_1, y_1) where $x_0, x_1, y_0, y_1 \in \mathbb{R}$. We consider this question in the computer graphics coordinate rather than the ordinary coordinate (y-axis and the orientation of the triangles are flipped). In a hypothetical triangle whose vertices are in anticlockwise orientation (clockwise orientation in the computer graphics coordinate system), we call an edge a top edge (it is above the other two edges) if it goes from left to right ($y_0 = y_1$ and $x_0 < x_1$). We call an edge a left edge if y decreases ($y_1 < y_0$). Even though we used a hypothetical well-oriented triangle to help us understand the concept, we can see that whether an edge is a top or left edge is only dependent on the coordinate of the two vertices of the edge. This situation is much less relevant in practice since the line tests are seldom exactly zero due to the float operation rounding errors and the tie-breaker rule has the potential to underdraw a triangle if it is separate from others.

1.2 Bounded Box Optimization

Even though we have the mathematical tools to uniquely determine whether a point is inside a triangle, but we do not want to check every pixel in the frame buffer. If a triangle is small and is on the one corner, it is not necessary to check the pixels on the opposite corner. For a triangle consisting of the three vertices $P_0(x_0, y_0)$, $P_1(x_1, y_1)$, and $P_2(x_2, y_2)$ in $\mathbb{R}^2$, we only need to check the pixels in a bounded box where x is in the interval $[\min(x_0, x_1, x_2), \max(x_0, x_1, x_2)]$ and y is in the interval $[\min(y_0, y_1, y_2), \max(y_0, y_1, y_2)]$. This bounded box is the smallest rectangle with horizontal and vertical edges containing the triangle, since moving any edge inwards would cause some x or y fall outside the bounded box. In C++, we can enlarge this rectangle a little by using the floor and ceiling functions followed by static casting to make the vertices of the bounded box integer tuples to simulate the pixels in real practice. In this way, we only need

to iterate through and perform line tests (and the tie-breaking rule) on sample within the bounded box of the triangle without going through the whole frame buffer. In Task 1, we can use the `fill_pixel` to set the color of the pixel in the sample buffer to the color we want for the single-color polygons.

1.3 Float Operation Rounding Errors and Handling

In section 1.1 we introduced the tie-breaker rule that checks whether an edge is a top edge or a left edge when the point lies on the edge and discussed its limitations in practice due to the errors introduced by float operations. In fact, the impact of float operation rounding errors surpasses the tie-breaker rule.

When a point $P = (x_0, y_0) \in \mathbb{R}^2$ is close to a line $l : Ax + By + C = 0$, the unnormalized line test $L(x, y) = Ax + By + C$ will be close to 0 at P because L is continuous. If the error introduced by float operations flips the sign of the line test, it is possible that a successful line test (greater than or equal to zero) will be considered to be failed or a failed line test (smaller than zero) will be considered successful. Since P has to succeed three line tests to be considered to be inside the triangle, while failing one line test is sufficient to exclude the point outside the triangle, it is more likely for the float operation rounding errors to exclude points that should be inside the triangle than include the points that are not inside the triangle. In practice, the picture rendered might have some white (or the default color of the frame buffer) pixels that should be filled next to the edges and some fewer glitch pixels painted with the color of the neighboring triangles. We will see an example of this artifact in Task 4.

It seems to me that handling such errors is very difficult. The float type in C++ uses 32 bits, where 1 bit is used for sign, 8 bits are used for sign (with a bias of -127), and 23 bits are used for mantissa. If we introduce an $\epsilon \in \mathbb{R}$ to be around 10^{-7} (C++ has `std::numeric_limits<float>::epsilon()` which is around 1.192×10^{-7}) and relax the line test check to be greater than $-\epsilon$, the line test tie-breaker rule becomes redundant because in this scenario, the relaxed line test will include any point that returns zero. The relaxed line test will also increase the chance of a pixel on the edge to be considered inside two neighboring triangles at the same time.

Another possible solution is to uphold the requirement that the line test has to be greater than zero for a point to be strictly inside the triangle, but to relax the line test requirement of a point being on the edge to be greater than $-\epsilon$ rather than equal 0. Since the top and left edge tier-breaker rule still applies, we significantly reduces the chance of a pixel being painted over twice by two neighboring triangles. However, even this relaxation would still increase the likelihood of some points outside the top or left edge to be considered inside the triangle.

Both of the possible error handling strategies try to address the blank pixel problem near the edge but both increased the chance of flickering pixels being painted over twice. In addition, neither strategy gives an answer to the fewer glitched pixels of the wrong color near the edge.

1.4 Picture Examples

In this section, we include one sample picture rendered with our rasterization program. Even without zooming in, we immediately noticing some artifacts. For example, the left and right edge of the purple triangle at the bottom right have high slope, and the jaggies are clearly visible. In addition, the middle right red triangle and the bottom right purple triangle have disconnected pixels near the vertices. Figure 2 and Figure 3 highlights those disconnected pixels in the pixel inspector.

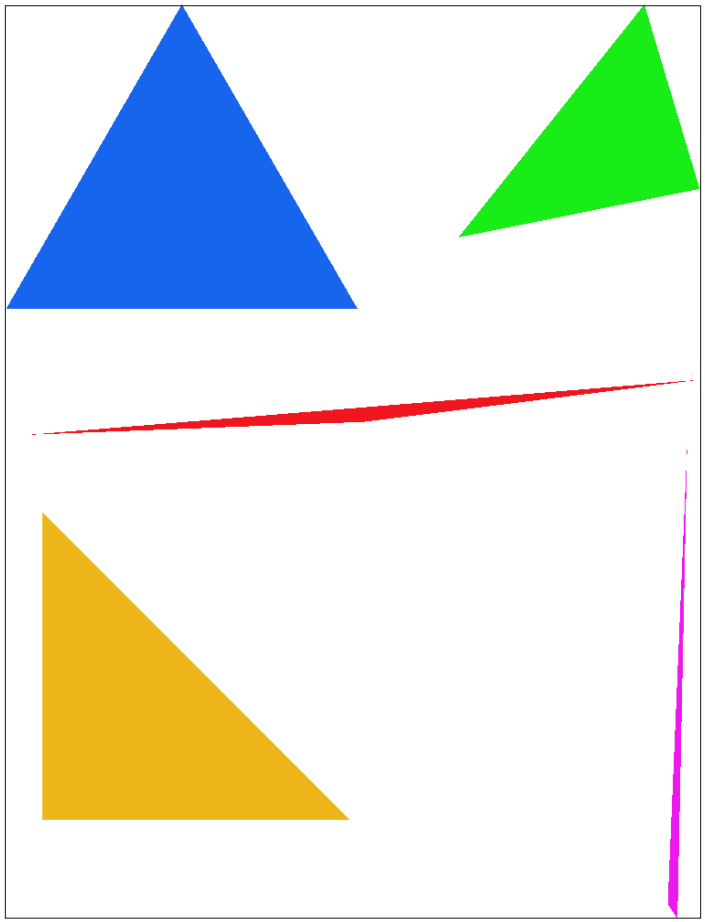


Figure 1: basic/test4.svg with default viewing parameters.

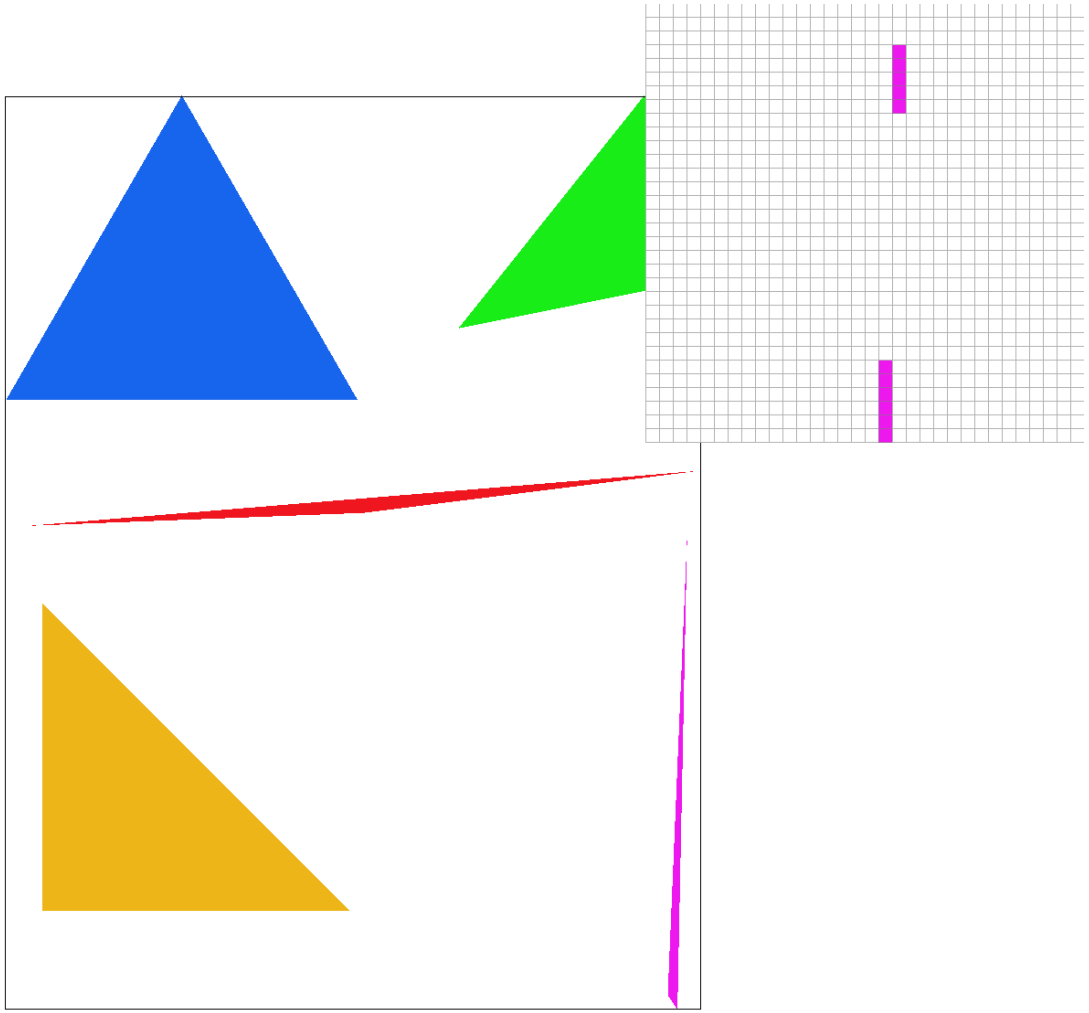


Figure 2: Pixel inspector showing disconnected pixels near a vertex.

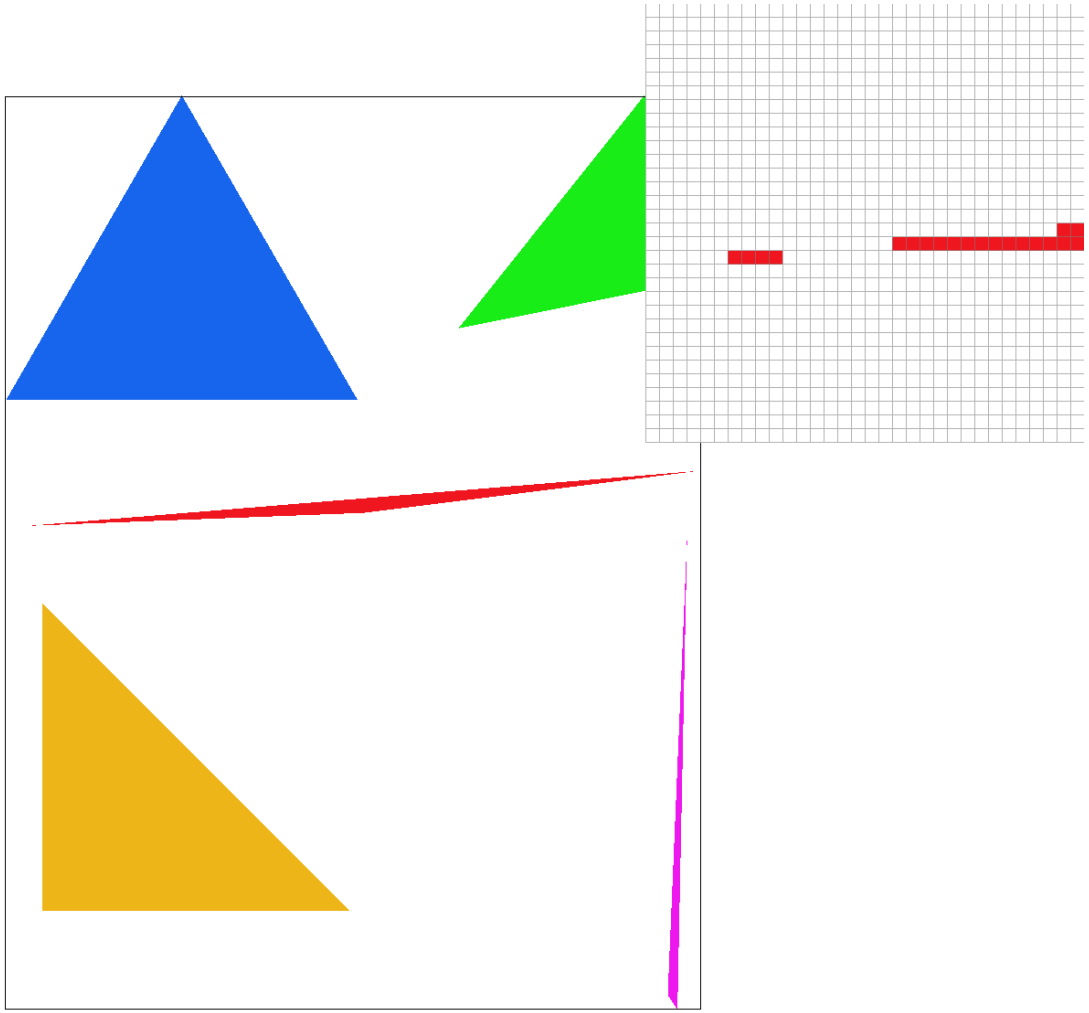


Figure 3: Pixel inspector showing disconnected pixels near a vertex.

1.5 Possible Improvements

As we have discussed in 1.3, the float operation rounding errors can cause pixels to be incorrectly included in or excluded from the triangle. It is more likely for a pixel to incorrectly fail one line test and be excluded than for a pixel to succeed three line tests by mistake to be included. Relaxing the line tests by $\epsilon \in \mathbb{R}$ to compensate could fill these pixels, but it makes the tie-breaker rule irrelevant, and can overdraw triangles, causing pixel flickering in nearby triangles. In our program, we decide to keep the blank pixels without implementing ϵ offset.

Another possible improvement is changing the loop order of x and y. In mathematics, we might be conceptually more inclined to list x first and then y, and thus making x the loop variable in the outer loop. However, if we take a look at the `fill_pixel` method, it uses the index `y * width + x`. We can see that if we keep the current loop structure, the indices are discontinuous and jump around. However, if we change `x` into the inner loop, we get continuous access on the memory. By the design of memory hierarchy, we know computer caches are optimized with the principles of spatial and spatial locality. If we access continuous blocks of data within a short time interval, we can improve the hit rate on caches and make the computation faster. On the other hand, indices jumping around can cause thrashing and lower the hit rate on caches and cause longer memory access time. We use drawing `test3.svg` as a test. If we keep `x` the outer variable, the time it takes for our program to complete the task is around 1.9 ms. However, if we move `x` to the inner loop, the average time to finish drawing goes down to around 1.2 ms.

2 Task 2

2.1 Supersampling Algorithm

Recall the Nyquist-Shannon sampling theorem from the lecture.

Theorem 2.1 (Nyquist–Shannon sampling theorem) *If a function $x(t)$ contains no frequencies higher than B hertz, then it can be completely determined from its ordinates at a sequence of points spaced less than $\frac{1}{2B}$ seconds apart.*

In other words, we can only accurately capture the signals of half the frequency of the sampling frequency (i.e. the Nyquist frequency). In order to address the aliasing such as jaggies in the Task 1, we can increase the sampling frequency to increase the Nyquist frequency.

However, our screens (frame buffers) only have a fixed number of pixels. If we sample at a higher frequency, we have to resolve the sample result to the target frame buffer. In Section 1, we sampled at each pixel of the frame buffer (technically only the bounded boxes of the triangles, but the result was equivalent to sampling at each pixel, since it is easy to see that the pixels outside the bounded box are not inside its triangle). We can see that the frame buffer is the same as the sample buffer.

In supersampling with the sampling rate $r \in \mathbb{Z}$, specifically, when r is a perfect square, we set up a sample buffer whose width and height are $\sqrt{r}$ times the width and height of the target frame buffer respectively. Intuitively, we are sampling the inside function in Section 1 at the centres of the r subpixels of each pixel, and then resolve this pixel by averaging over the sampling results of all its subpixels. Note that it is possible to achieve supersampling without creating a sample buffer, if we just sample r subpixels of a pixel and store the average of their sampling result directly to the target frame buffer without storing them in a separate sample buffer. However, this implementation is linear and poor for parallel optimization.

In our implementation of the supersampling, instead of using the flattened vector `y * width + x` and changing the `x, y, width` accordingly, where same `y` could be subpixels of different pixels, we store the sampling result of the subpixels of one pixel in one continuous block, with each subpixel result stored at the index `(y * width + x) * sample_rate + j * square_rate + i`. We implement the functions `sample_buffer, set_sample_rate(), clear_buffers(), set_framebuffer_target()` in the `RasterizerImp` namespace to set up the sample buffer and target buffer, `resolve_to_framebuffer()` to average all the sample results of the subpixels and store the value to pixel they belong in the target frame buffer. We then update the `rasterize_triangle()` method to run the line tests on each subpixel and instead of using the `fill_pixel` method, we directly save the sampling results to the sample frame buffer. However, we do need to update the `fill_pixel` method with the sample rate to ensure that the points and the lines in the original pictures render correctly.

2.2 Picture Examples

In this section, we show the same sample picture at different sampling rates produced by our program. As we go from sample rate 1 to 4, we can immediately see that the jaggies on the edges, especially ones with high slope, are reduced significantly. The disconnected pixels around the vertices also appear less apart from the triangles now. With the pixel inspector, we can see that the triangles are no longer of the uniform color near the edge. Many pixels appear visibly paler. In addition, the gap between the disconnected pixels from the vertices become narrower. As we go from sample rate 4 to 16, we can see that the jaggies on the edges are further reduced, and the disconnected pixels at lower sample rate seem to be connected to the triangles they belong to now. We can confirm with the pixel inspector that the gap does disappear.

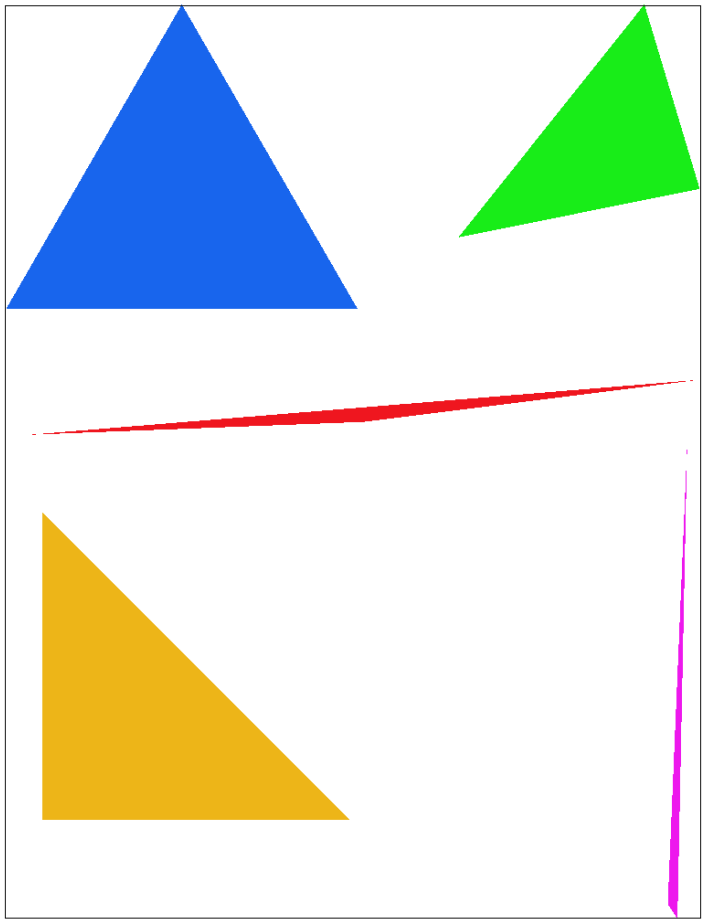


Figure 4: basic/test4.svg with default viewing parameters and sample rate 1.

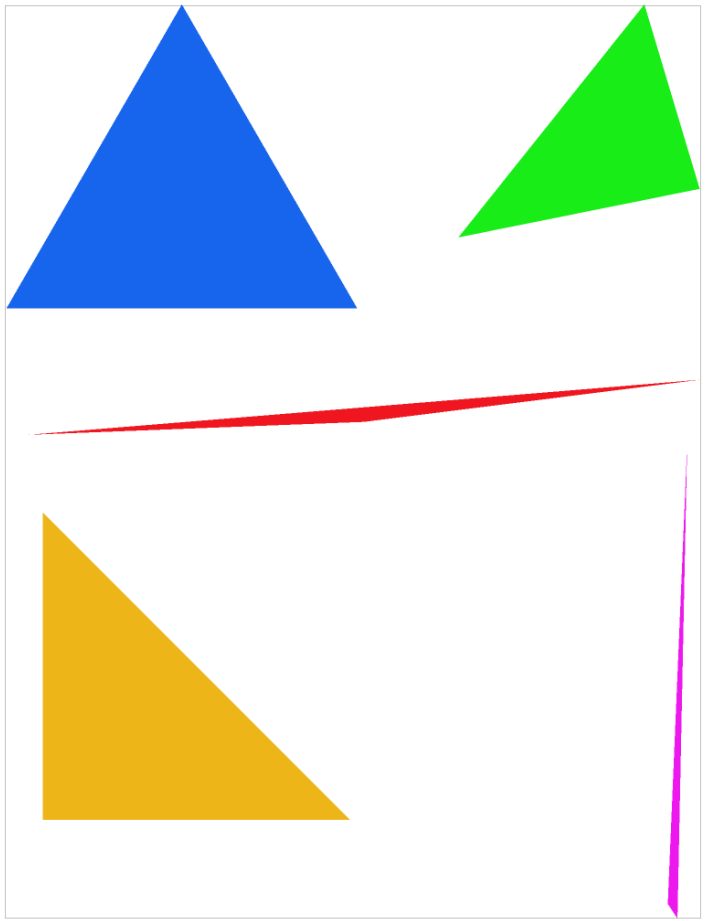


Figure 5: basic/test4.svg with default viewing parameters and sample rate 4.

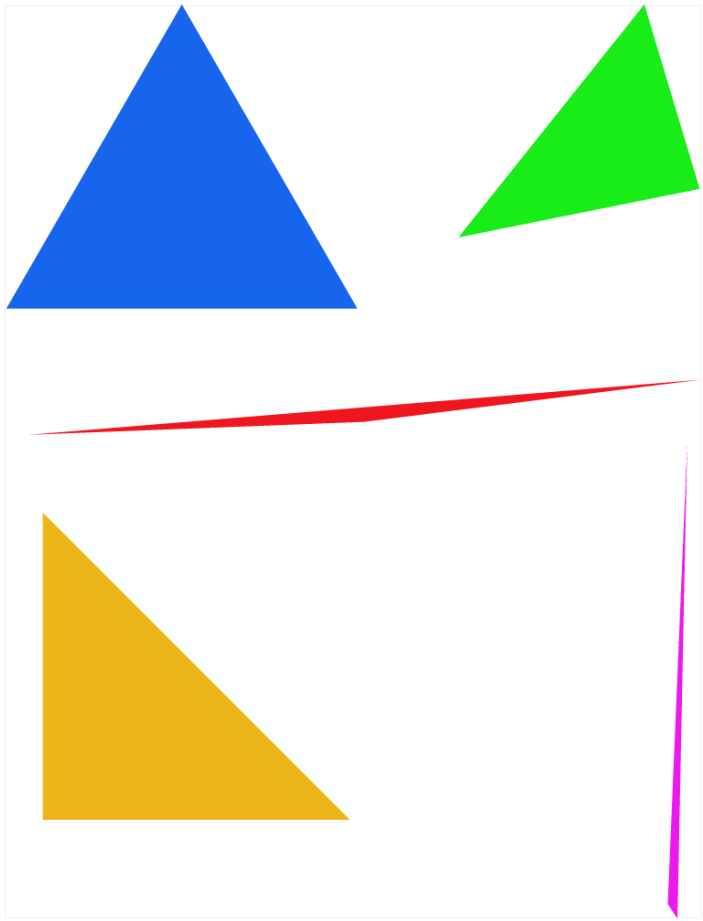


Figure 6: basic/test4.svg with default viewing parameters and sample rate 16.

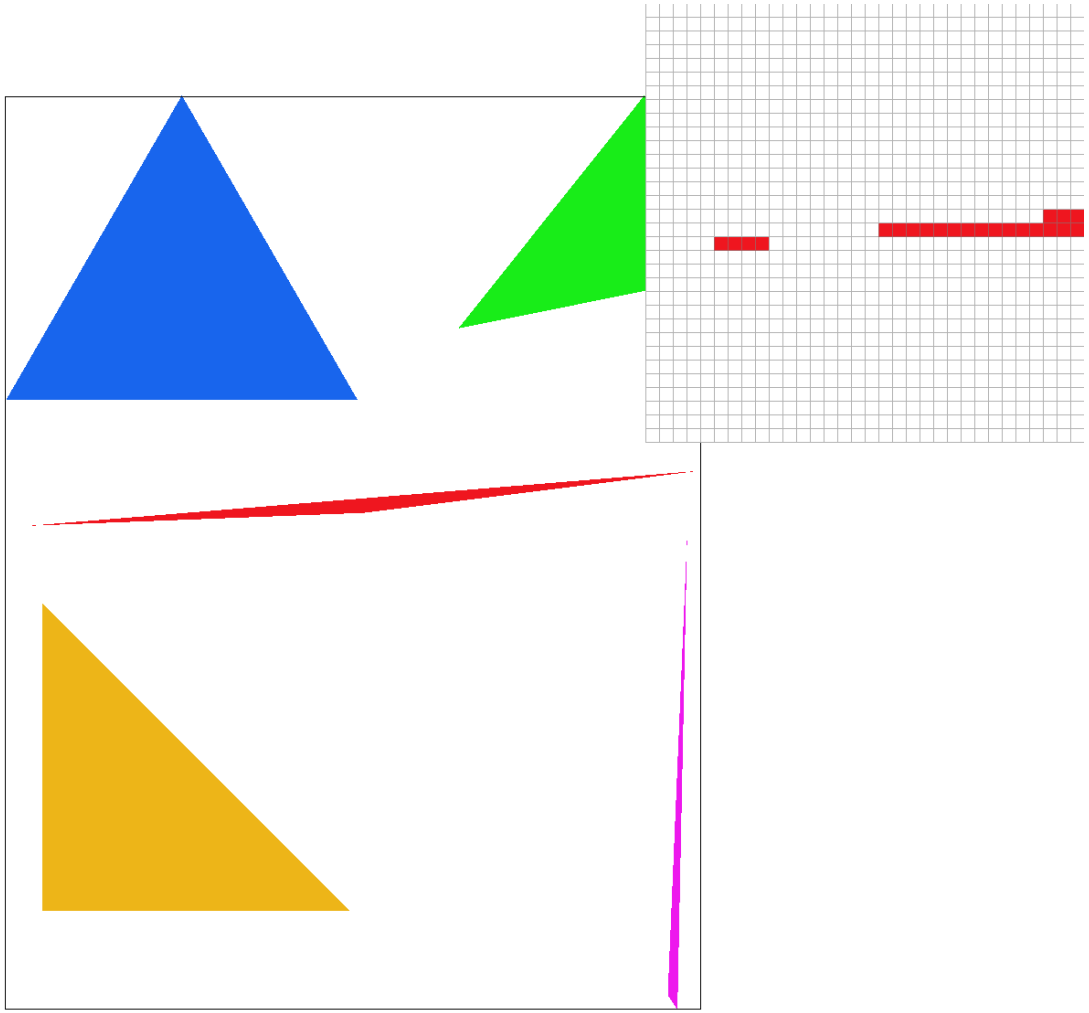


Figure 7: Pixel inspector showing disconnected pixels near a vertex at sample rate 1.

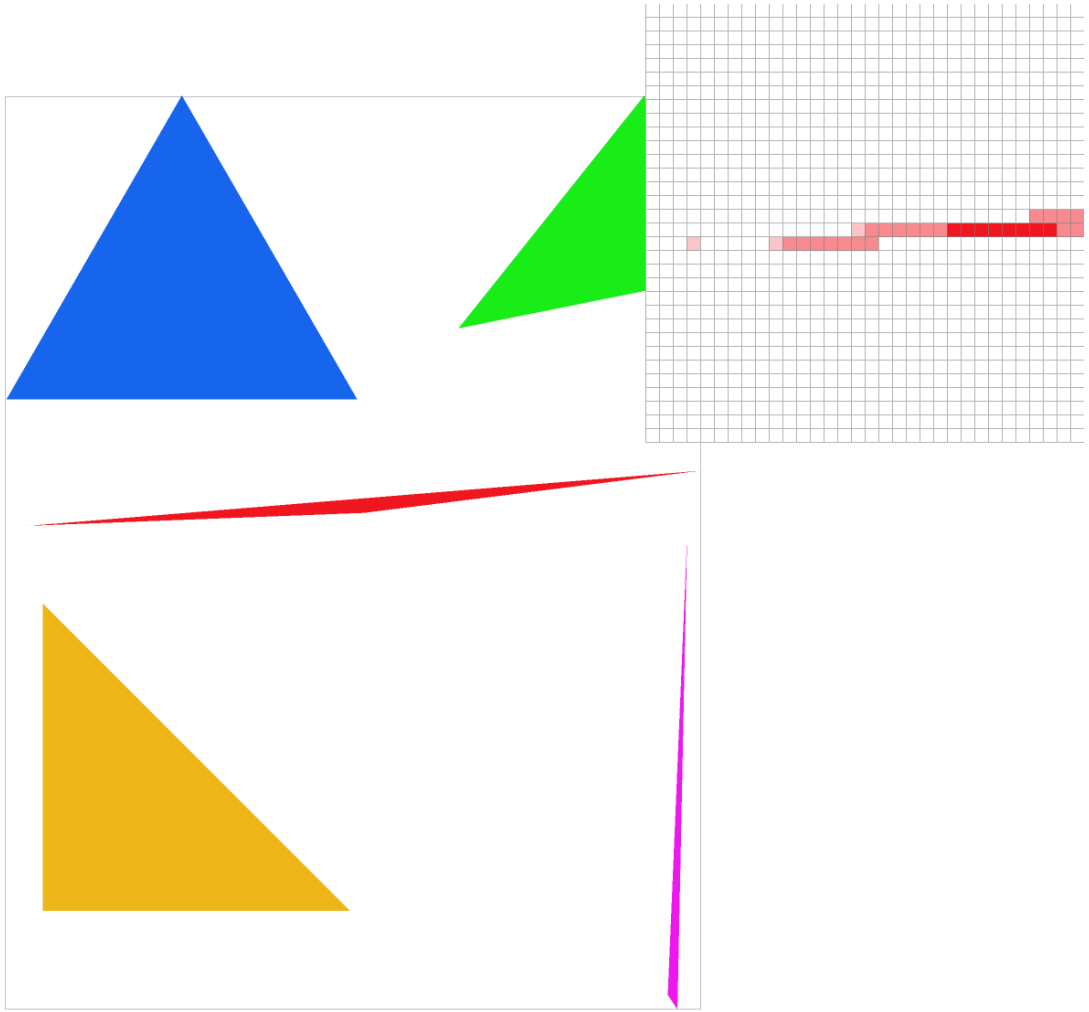


Figure 8: Pixel inspector showing disconnected pixels near a vertex at sample rate 4. The gap between the pixel becomes narrower.

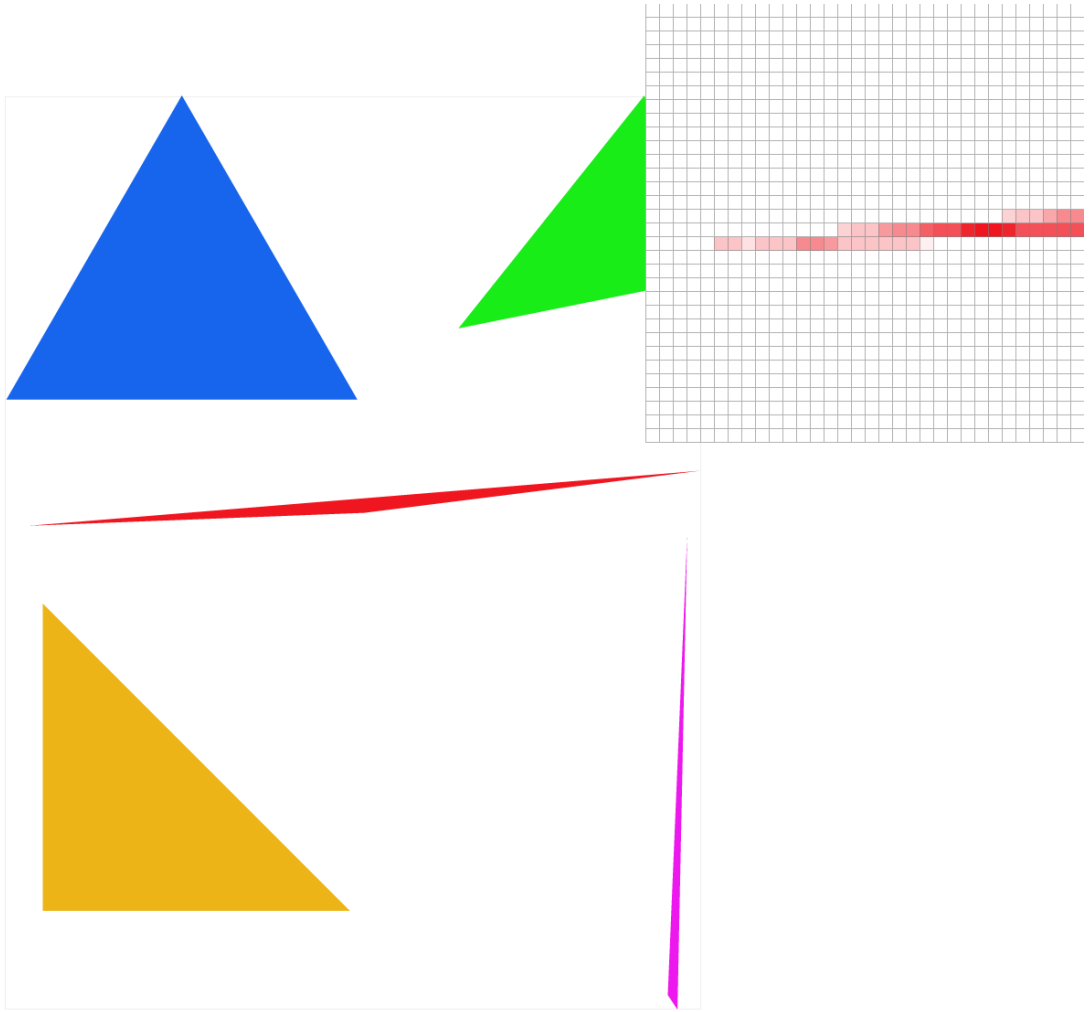


Figure 9: Pixel inspector showing disconnected pixels near a vertex at sample rate 16. The pixels are fully connected now.

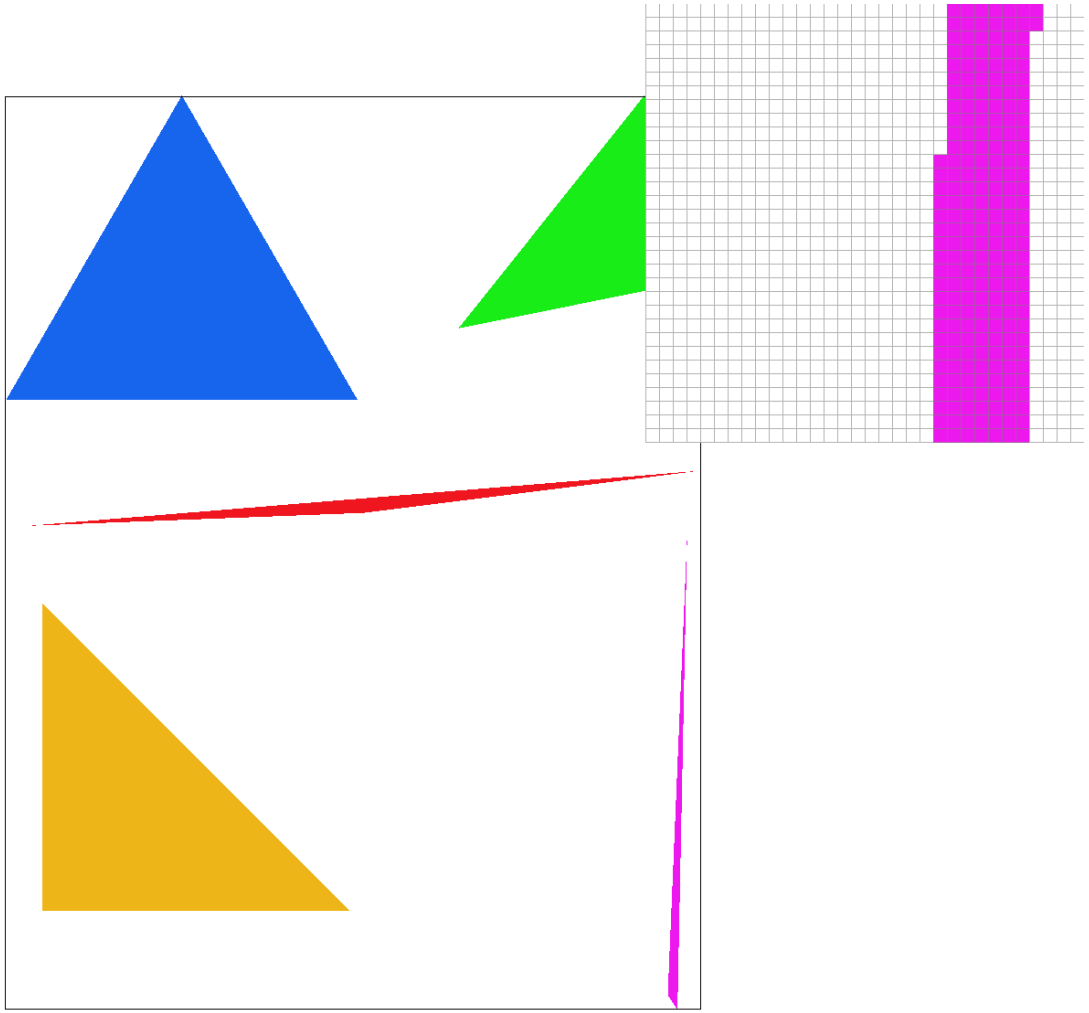


Figure 10: Pixel inspector showing jaggies on a line with high slope at sample rate 1.

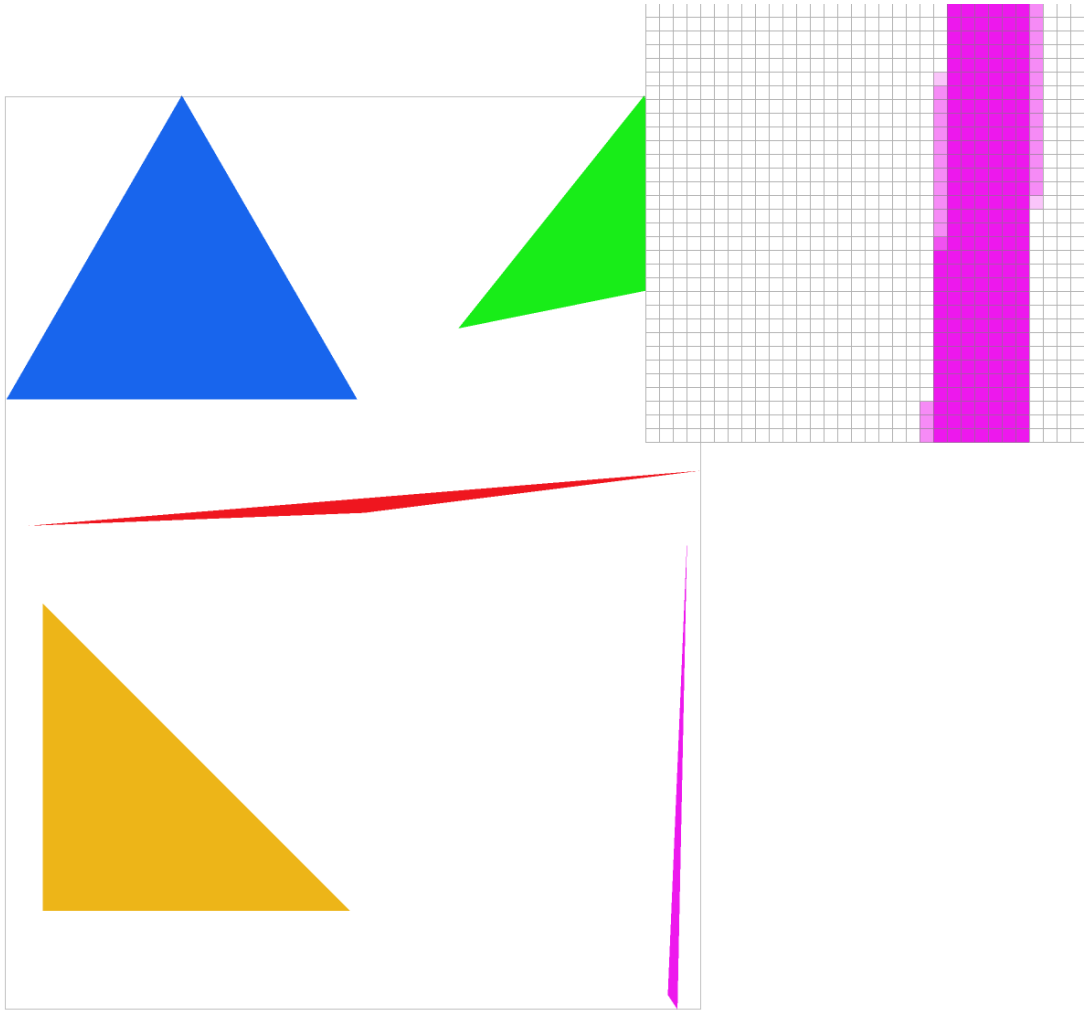


Figure 11: Pixel inspector showing jaggies on a line with high slope at sample rate 4. The jaggies are reduced.

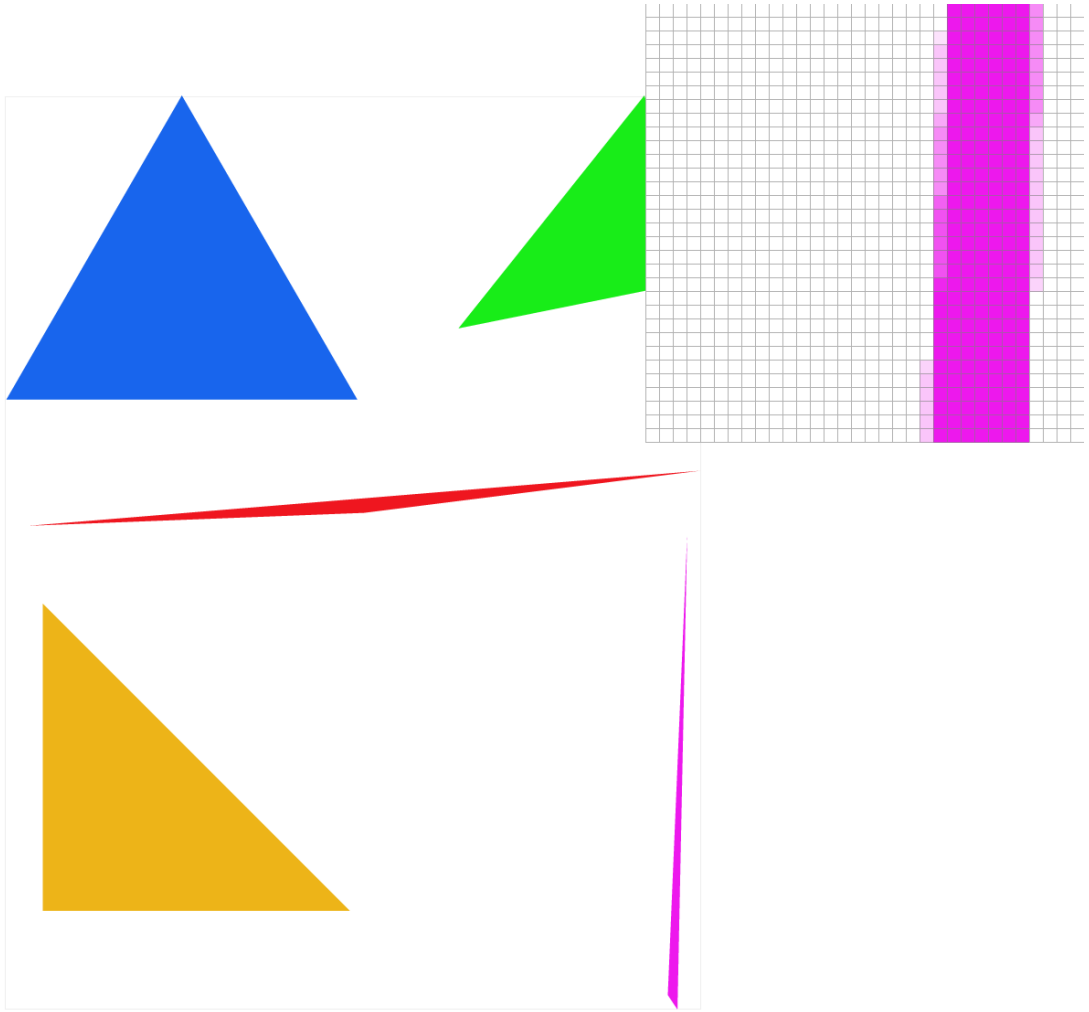


Figure 12: Pixel inspector showing jaggies on a line with high slope at sample rate 16. The jaggies are reduced even further.

2.3 Comparison with Jittered Sampling

In our implementation of rasterization at various sampling rates, we have always been sampling at the centres of pixels or subpixels. However, the uniform sampling intervals can cause interference with signals at high frequency, causing moiré patterns. If we choose to randomize the point inside each pixel at which we sample, we can reduce the interference.

In our program, we can just change the `0.5f` offset to `rand()` for on both the `x` and `y` coordinates to achieve jittered sampling. Since rasterization in practice are run on multiple pixels in parallel, the tie-breaker rule becomes much less relevant, because neighboring triangles might sample the same pixel at different points. Therefore, jittered sampling increases the chance of pixel flickering.

Since default jittered sampling samples once per pixel, it is cheaper than supersampling at higher sample rate. However, it is totally possible to combine supersampling with jittered sampling by sampling each subpixel at a random point inside and then resolve the sample buffer to the target buffer.

Below we included the same picture of lines arranged in a circular pattern with normal sampling and jittered sampling at sample rate 1 and 16. At the same sample rate, we can see that the jaggies on the lines become much less noticeable. One possible explanation is that since the irregularities appear in a much less uniform pattern, they do not manifest as easily to human eyes as jaggies. At sample rate 16, we can see that the color of the pictures become much more faded. This exposes a flaw in supersampling when the lines are very thin, where downsampling process can make the lines seem much less well-defined. Unfortunately, when combined with supersampling, jittered sampling also exhibits this problem.

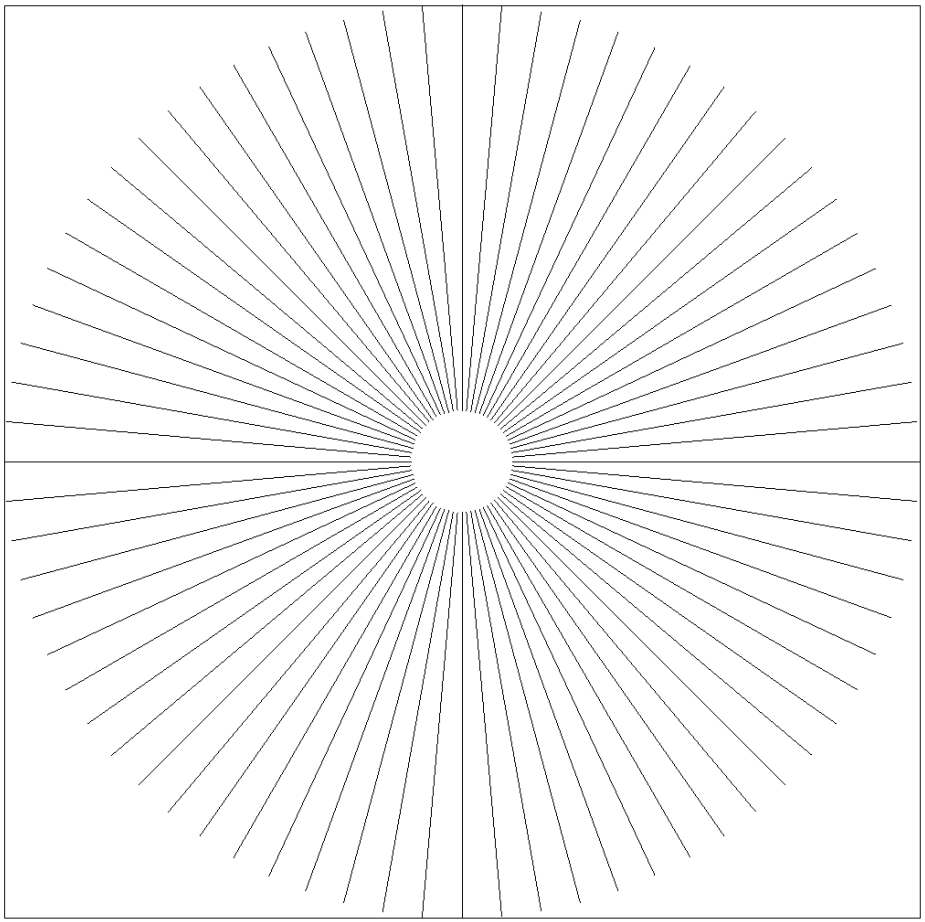


Figure 13: basic/test1.svg with default viewing parameters and sampling rate 1.

Figure 14: Default viewing parameters and sampling rate 16.

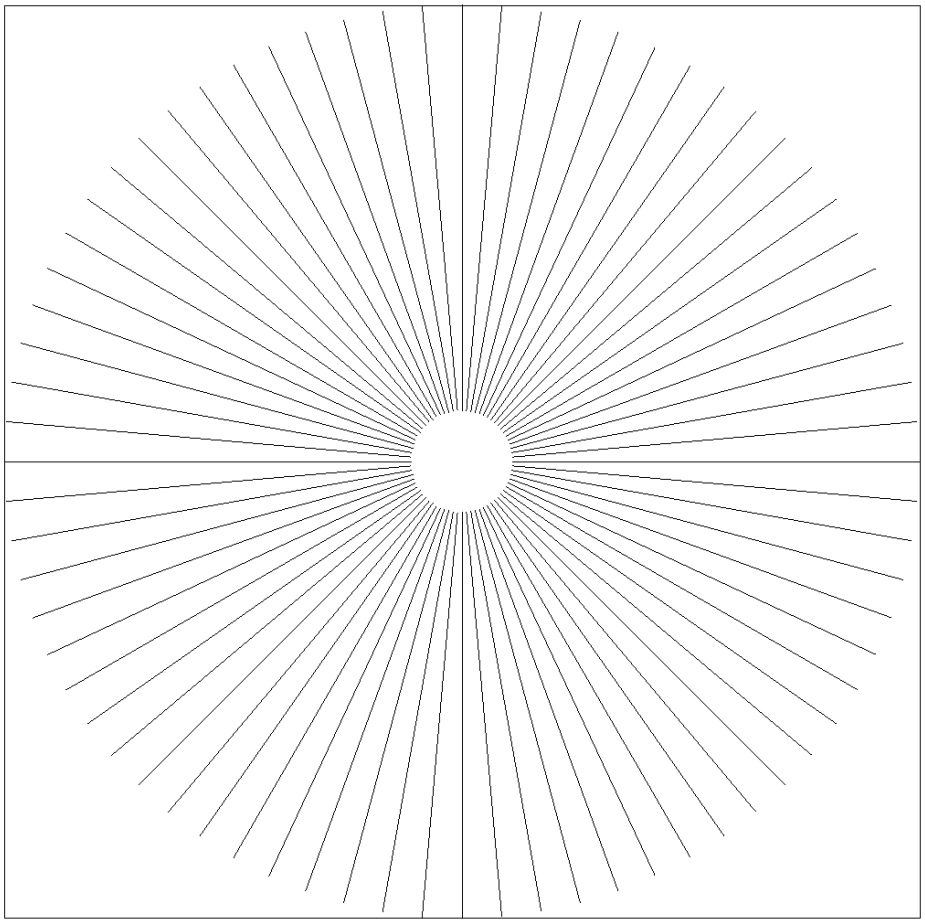


Figure 15: Jittered sampling with sampling rate 1.

Figure 16: Jittered sampling with sampling rate 1.

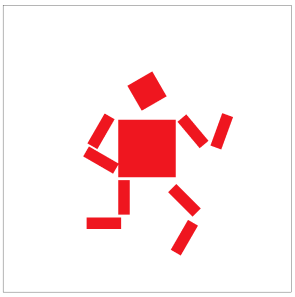


Figure 17: Running cubeman.

3 Task 3

In this task, we implemented the `translate`, `scale`, and `rotate` methods in C++. By applying these functions in the svg file parsed by C++, we are able to draw a cubeman rushing with all force to the right with his head slightly leaning forward.

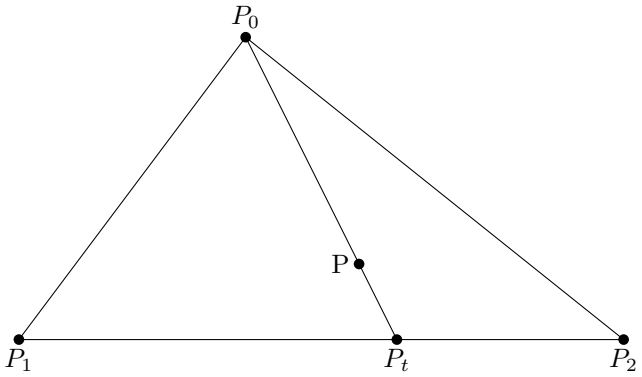
4 Task 4

4.1 Barycentric Coordinates

Consider the linear interpolation $\text{lerp}(A, B, t) = (1-t)A + tB$ where $t \in [0, 1]$ and $A, B \in \mathbb{R}^n$ for some $n \in \mathbb{Z}_+$. The linear interpolation creates a bijection between the interval $[0, 1]$ and the line segment between A and B. If $t < 0$ or $t > 1$, then we can see that the point lies outside the line segment. In addition, the derivative of lerp with respect to t is constant, so the rate of change is constant. Consider a continuous map $f : \mathbb{R}^n \to \mathbb{R}^m$ for some $m \in \mathbb{Z}_+$. $\text{lerp}(f(A), f(B), t) = (1-t)f(A) + tf(B)$ gives an approximation of the value of $f((1-t)A + tB)$ based on the assumption that the value changes linearly from $f(A)$ to $f(B)$. Note that if f is a linear map, then this approximation is exact.

Now we want to consider the linear interpolation between $P_0, P_1, P_2 \in \mathbb{R}^n$. One possible way to represent a point inside the triangle $\Delta P_0P_1P_2$ is to interpolate a point P_t on the line segment P_1 and P_2 such that $P_t = (1-t)P_1 + tP_2$ where $t \in [0, 1]$. Then we can interpolate a point P between P_0 and P_t such that $P = (1-s)P_0 + sP_t = (1-s)P_0 + (s-st)P_1 + stP_2$ where $s, t \in [0, 1]$. We can see that $1 - s + s - st + st = 1$, and $1 - s, s - st, st \in [0, 1]$. Since s and t are parameters and $1-s$, $s-st$ and st are monotonous and continuous, we can do a reparameterization $P = \alpha P_0 + \beta P_1 + \gamma P_2$ where $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \in [0, 1]$. This reparameterization always uniquely exists since for any P inside triangle $\Delta P_0P_1P_2$, the line P_0P always intersects inside the line segment P_1P_2. Since α, β, γ uniquely determines a point inside the triangle, we can use the tuple

(α, β, γ), i.e. the barycentric coordinates, to represent the point. Consider a continuous map $f : \mathbb{R}^n \to \mathbb{R}^m$ for some $m \in \mathbb{Z}_+$. Then $\alpha f(P_0)+\beta f(P_1)+\gamma f(P_2)$ gives an approximation of $f(\alpha P_0 + \beta P_1 + \gamma P_2)$ that assumes the rate of change of f inside the triangle is constant. The approximation is exact if f is linear.



By the geometrical discussion in the lecture, we know that for any point $P = (x, y) \in \mathbb{R}^2$ inside the triangle ΔABC, we have $\alpha = \frac{-(x-x_B)(y_C-y_B)+(y-y_B)(x_C-x_B)}{-(x_A-x_B)(y_C-y_B)+(y_A-y_B)(x_C-x_B)}$, $\beta = \frac{-(x-x_C)(y_A-y_C)+(y-y_C)(x_A-x_C)}{-(x_B-x_C)(y_A-y_C)+(y_B-y_C)(x_A-x_C)}$, and $\gamma = 1 - \alpha - \beta$. Note that the numerators in α and β are exactly the unnormalized line test in Section 1. The denominators are the z-component of the cross products. Since the magnitude of the cross product is twice the size of the triangle between the two vectors, the denominators have the same absolute value. Upon a closer inspection, we can see that α, β, γ are exactly the normalized line tests in Section 1. Therefore, we can create a static method `barycentric(x0, y0, x1, y1, x2, y2, x, y)` that returns a `Vector3D` struct containing α, β, γ to perform line tests and compute the barycentric coordinates at the same time.

From our previous discussion, we can see that the barycentric coordinates are linear interpolation between three points. Therefore, if there is a texture map f that maps the triangle ΔABC to the texture, given the texture $f(A), f(B), f(C)$ at the vertices A, B, C, we can use the $(\alpha f(A) + \beta f(B) + \gamma f(C))$ to interpolate the texture at $(\alpha A + \beta B + \gamma C)$.

4.2 Picture Examples

In this section, we show the pictures of color gradation interpolated with barycentric triangles. Note that there are white pixels appearing along a line segment. In Section 1.3 we already discussed that one possible cause of these white pixels is the float operation rounding error. In fact, even if we completely remove the tie-breaker top-left edge checks in our program, these white pixels persist. If we increase the sample rate, we can see that the effect of these white pixels become much less visible.

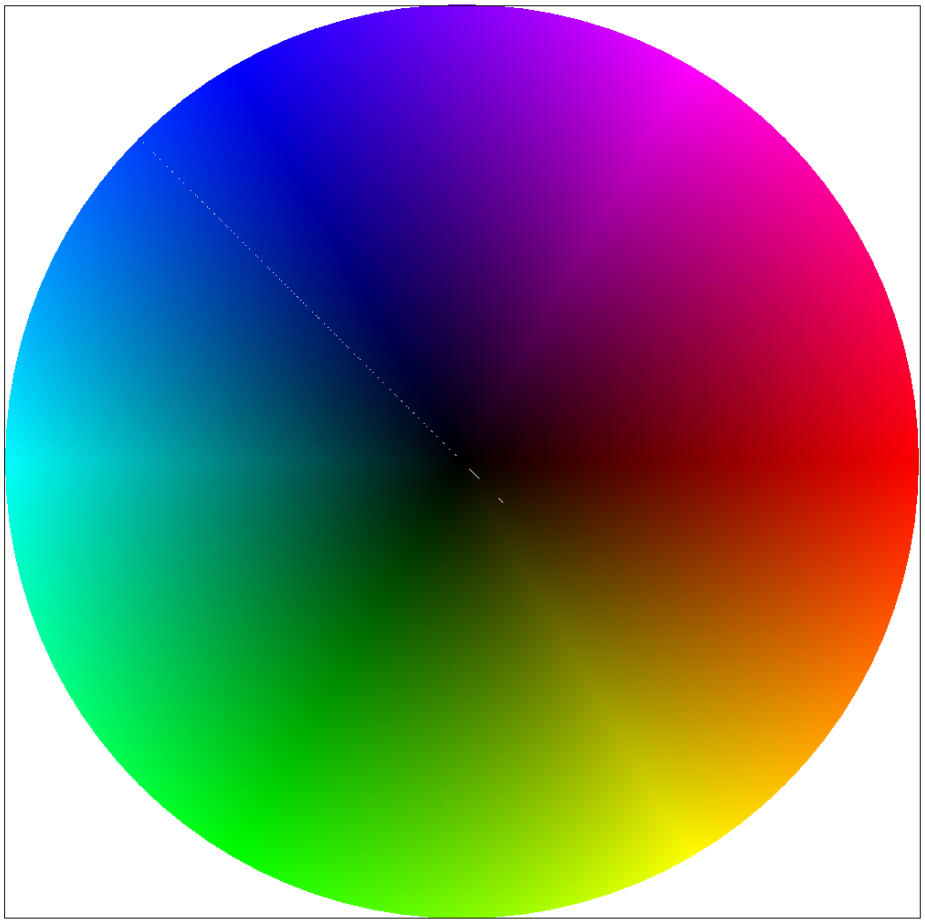


Figure 18: svg/basic/test7.svg with default viewing parameters and sample rate 1. Note the white pixels along a line segment.

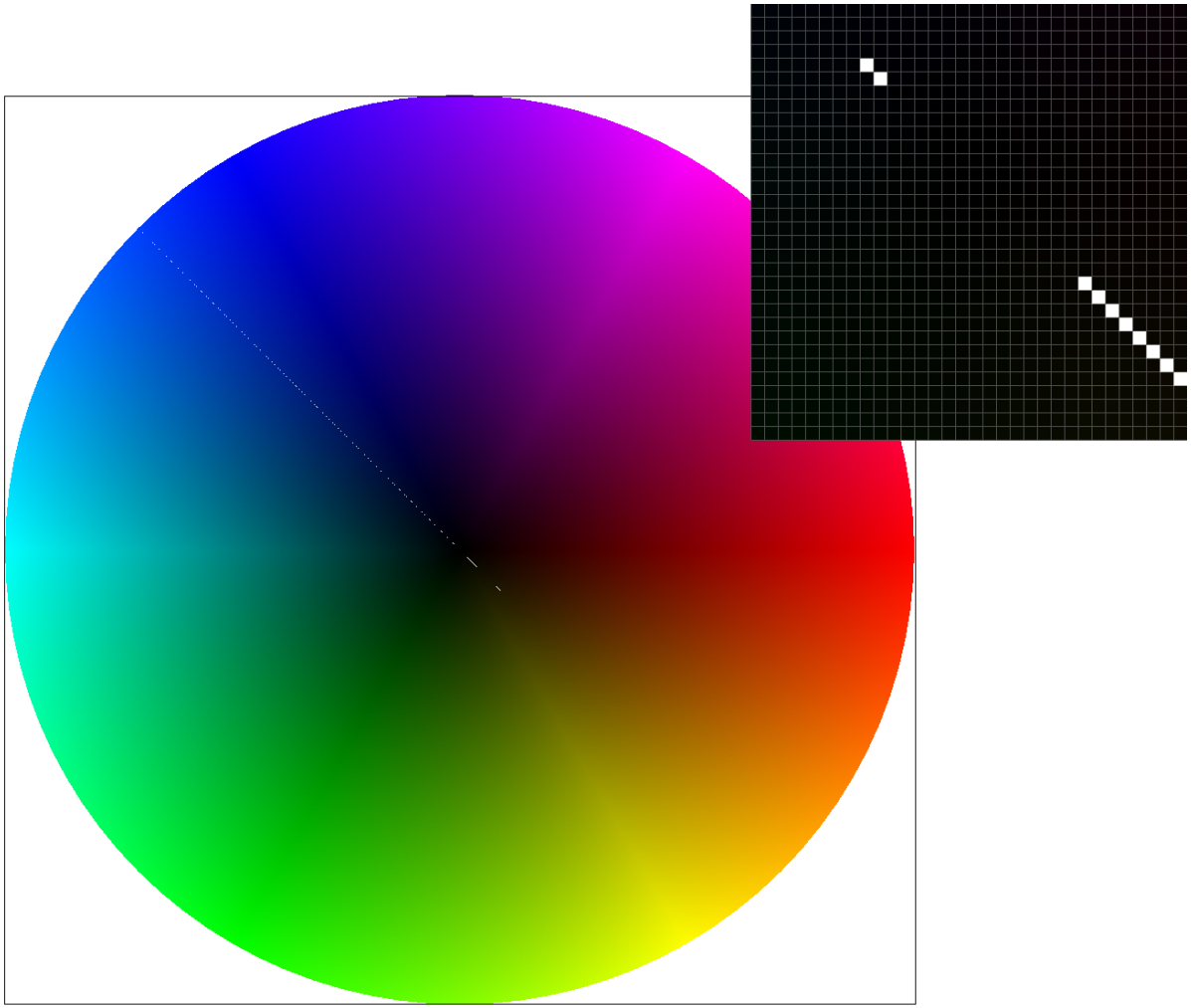


Figure 19: Pixel inspector highlighting the white pixels.

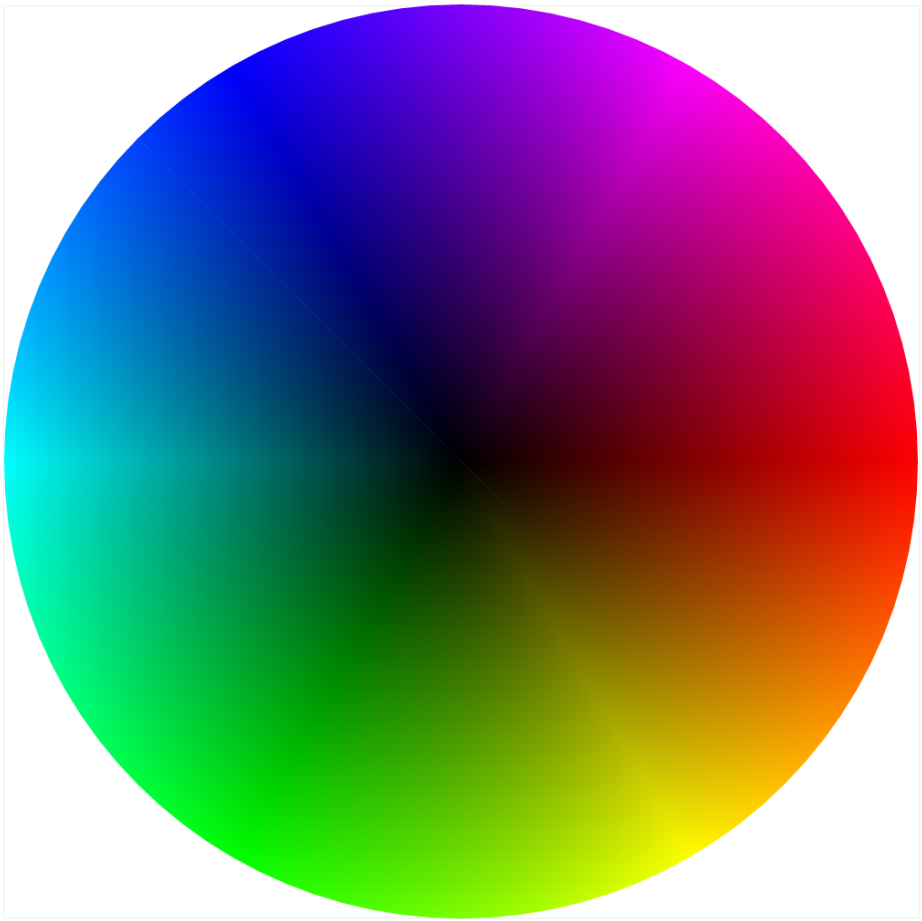


Figure 20: Supersampling at sample rate 16. The white pixels are significantly reduced.

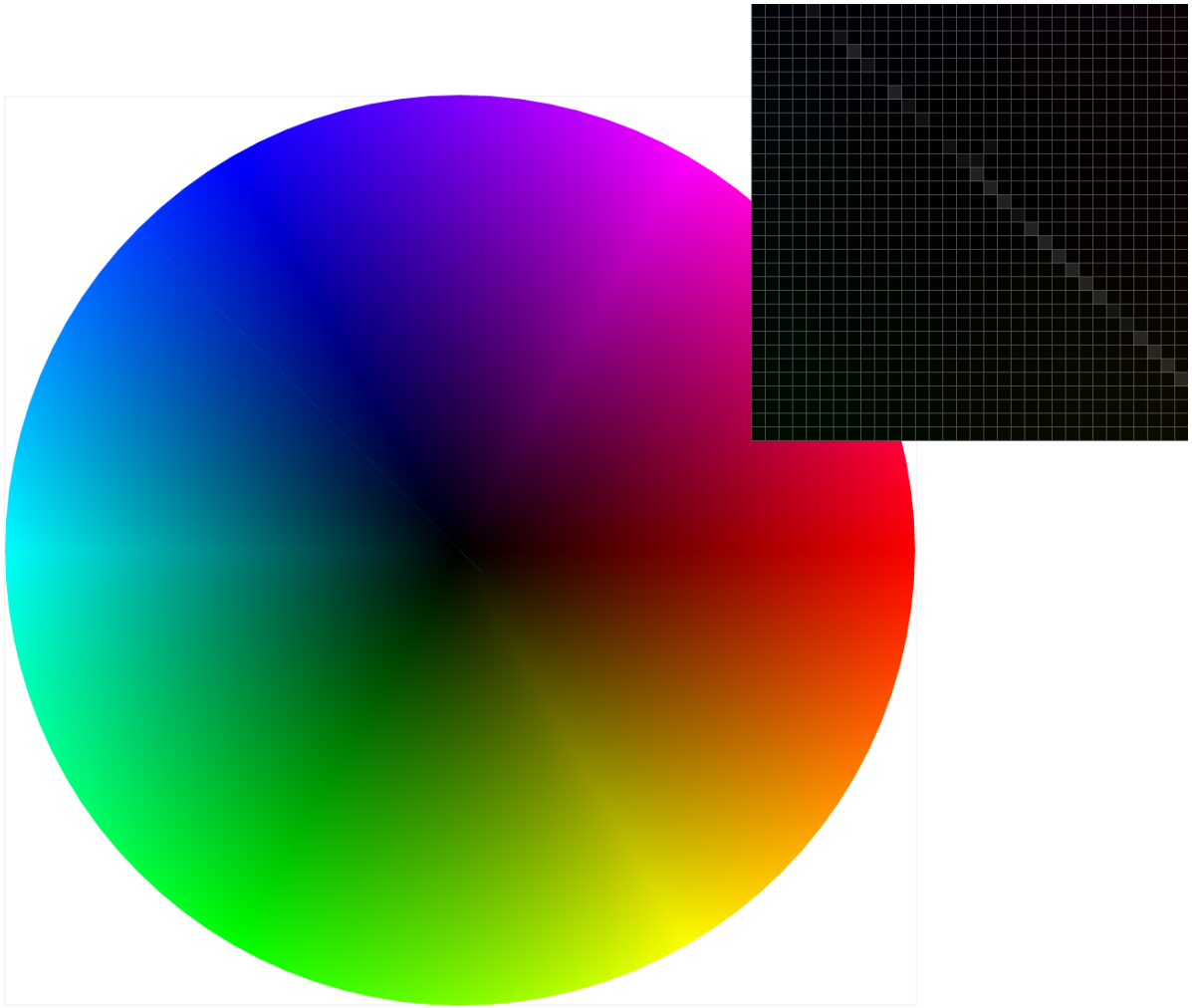


Figure 21: Pixel inspector showing where the white pixels were at sample rate 16.

5 Task 5

5.1 Pixel Sampling

Let us consider the situation where we want to map whole texture onto a triangle rather than just interpolating the texture from the given texture at three vertices as we discussed in 4.1. Let $f : \mathbb{R}^2 \to \mathbb{R}^2$ where $f(x, y) = (u, v)$ that maps pixels on the frame buffer to points in the texture space, and let $g(u, v)$ be the sampling function that returns the color of a given point (u, v) in the texture space. For points $P_0(x_0, y_0), P_1(x_1, y_1), P_2(x_2, y_2) \in \mathbb{R}^2$ and $P(x, y)$ inside $\Delta P_0P_1P_2$, we can see that the interpolation model we introduced in 4.1 uses the approximation $g(f(x, y)) = \alpha g(f(x_0, y_0)) + \beta g(f(x_1, y_1)) + \gamma g(f(x_2, y_2))$. This model makes the assumption that both the texture mapping f and the sampling function g are linear. If g were linear, the texture we work with would be gradations. However, the real texture we use only have sharp edges and many discontinuities that cannot be approximated well with linear interpolation.

Pixel sampling on a triangle is coloring the pixels inside the triangle $\Delta P_0P_1P_2$ with the sampled color $g(f(\alpha(x_0, y_0)+\beta(x_1, y_1)+\gamma(x_2, y_2)))$. If our triangle is in a well-defined mesh and maps to a region relatively small compared to the whole texture, we can make an approximation that f is locally affine, namely $f(x, y) = A(x, y)+b$ for some $A \in \mathbb{R}^{2\times2}$ and $b \in \mathbb{R}^2$. Based on this assumption, if we have $f(x_0, y_0) = (u_0, v_0), f(x_1, y_1) = (u_1, v_1), f(x_2, y_2) = (u_2, v_2)$, then $f(\alpha(x_0, y_0) + \beta(x_1, y_1)+\gamma(x_2, y_2)) = A(\alpha(x_0, y_0)+\beta(x_1, y_1)+\gamma(x_2, y_2))+\mathbf{b} = \alpha(Af(x_0, y_0)+\mathbf{b})+\beta(Af(x_1, y_1)+\mathbf{b})+\gamma(Af(x_2, y_2)+\mathbf{b}) = \alpha f(x_0, y_0)+\beta f(x_1, y_1)+\gamma f(x_2, y_2) = \alpha(u_0, v_0) + \beta(u_1, v_1) + \gamma(u_2, v_2)$.

In our C++ program, we compute the barycentric coordinates to determine whether each pixel in the bounded box is inside the triangle. Since the (u, v) coordinates have been given by the parameters of `rasterize_textured_triangle`, we can use the barycentric coordinates to compute the point in the texture space where the centre of each pixel inside the triangle maps to with `u = alpha * u0 + beta * u1 + gamma * u2` and `v = alpha * v0 + beta * v1 + gamma * v2`.

In practice, however, the textures we use in computer graphics are hardly continuous. They have been tessellated into texels. How do we deal with points with non-integer coordinates inside the texture space? Let us discuss the two most common sampling methods, nearest and bilinear.

Since the texture space consists of texels, the most obvious solution is to use the color of texel in which the point falls. Note that in our C++ program, the (u, v) given are coordinates normalized by the width and height of the texture (this normalized version will become useful when we scale the texture later). In order to use the computed (u, v) data from earlier, we need to multiply u with the width and v with the height. Then we can compute the nearest u and nearest v with `int nearest_u = static_cast<int>(std::floor(u))` and `int nearest_v = static_cast<int>(std::floor(v))`, and return the texel color at that point. Note that the nearest sampling method does not return the texel color of the texel whose integer coordinates are the closest, but the color of the

texel whose centre is the closest.

The bilinear sampling method acknowledges that the texels we have in the texture space are discrete approximation of the origin texture, so instead of considering only the texel where the point falls into, this approach considers the four texels whose centres the point is closest to. Again, we use the width and height to compute the actual coordinates of the point, and then get the closest four texels $(u_0, v_0), (u_0, v_1), (u_1, v_0), (u_1, v_1)$ with the formulae `int u0 = static_cast<int>(std::floor(u - 0.5f))` and `int v0 = static_cast<int>(std::floor(v - 0.5f))`. Then we can compute u_1, v_1 since they are just neighboring texels with `int u1 = u0 + 1` and `int v1 = v0 + 1`. Now we want to linearly interpolate the color of this point by how far they are from the centres of these four texels. Since we assume that in this small patch, the color of the texture changes linearly with u and v, this sampling method is called bilinear. We compute our interpolation coefficients `float s = u - (u0 + 0.5f)` and `float t = v - (v0 + 0.5f)`. We get the color `cv0` by interpolate the texel color at (u_0, v_0) and (u_1, v_0) with the coefficient s and the color `cv1` by interpolate the texel color at (u_0, v_1) and (u_1, v_1) with the coefficient s. Finally we return the interpolated color between `cv0` and `cv1` with the coefficient t: `(cv0 * (1 - t) + cv1 * t)`.

5.2 Picture Examples

We include the same sample pictures with nearest pixel sampling and bilinear pixel sampling at sample rate 1 and 16. At sample rate 1, we can clearly see that the grid lines are disconnected and show jaggies in the nearest pixel sampling picture. The pixel inspector highlights the gap on the grid line. The grid lines show jaggies in the bilinear pixel sampling picture, but they are connected. At sample rate 16, both nearest sampling and bilinear sampling produce grid lines of a much higher quality. However, from the pixel inspector we can see that the grid lines show a clear color contrast with the blue background in the nearest pixel sampling picture, while the color changes gradually in the bilinear sampling picture. Without pixel inspector, both pictures at sample rate 16 are of high quality and show very little difference.

The methods probably differ the most at edges with drastic color change. These edges have high frequency that are usually higher than the Nyquist frequency. Bilinear pixel sampling filter out the high frequency component to reduce the aliasing.

Figure 22: Nearest pixel sampling with sample rate 1. The lines of the grid are disconnected. Pixel inspector highlights this gap.

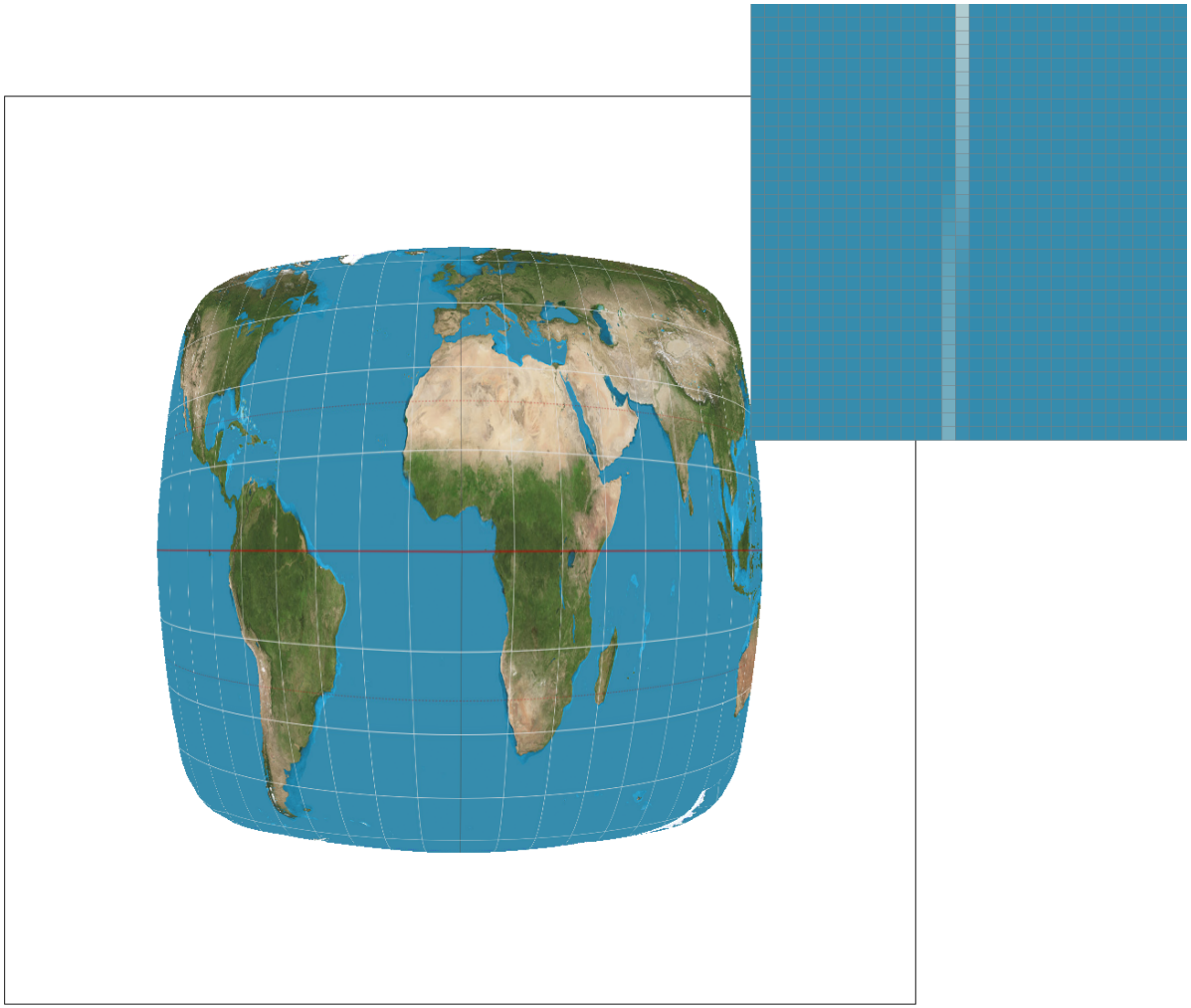


Figure 23: Bilinear pixel sampling with sample rate 1. The lines of the grid show visible jaggies but are clearly connected. Pixel inspector highlights the same region.

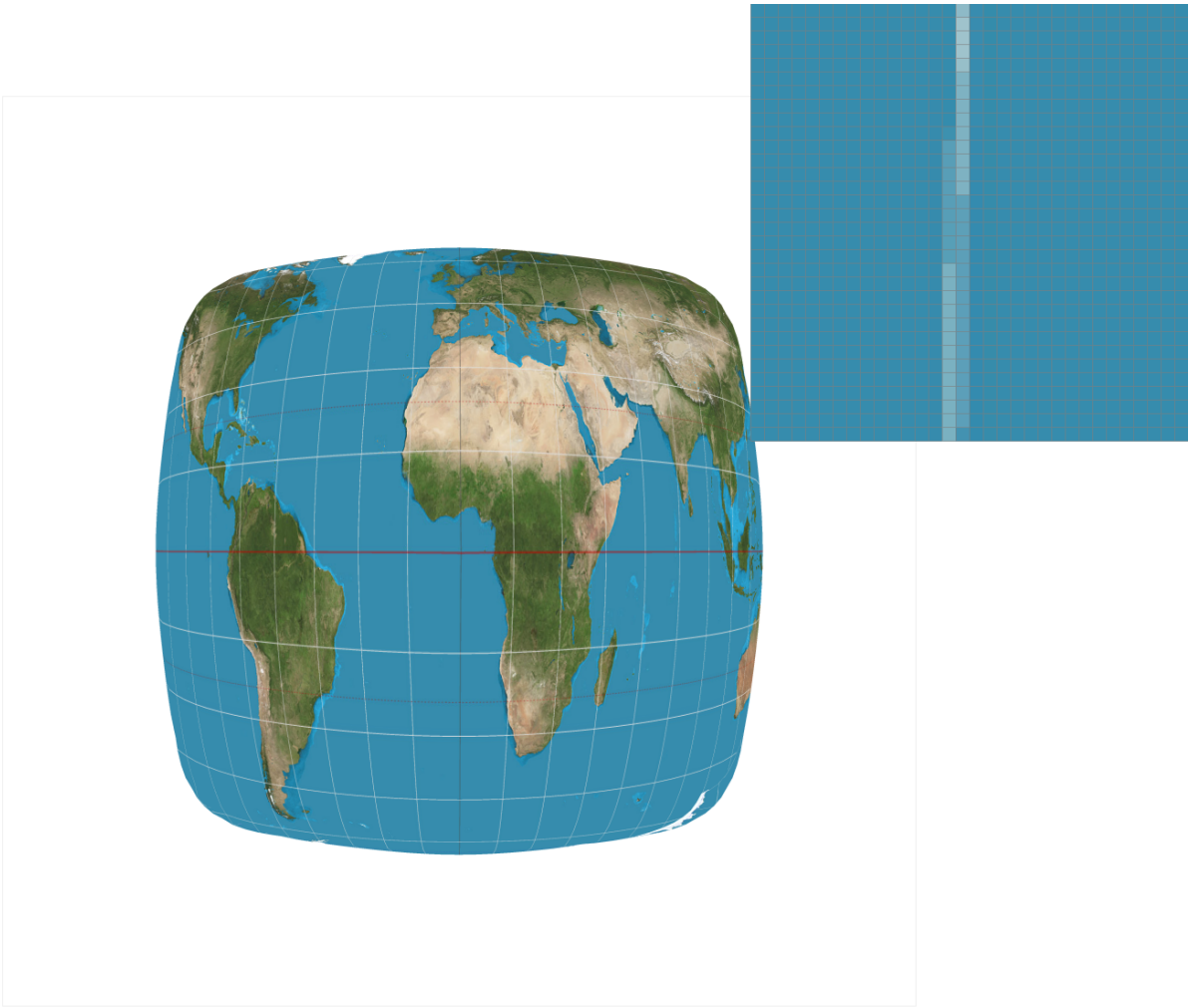


Figure 24: Nearest pixel sampling with sample rate 16. The lines of the grids are connected. Pixel inspector shows clear contrast of color compared to the blue background.

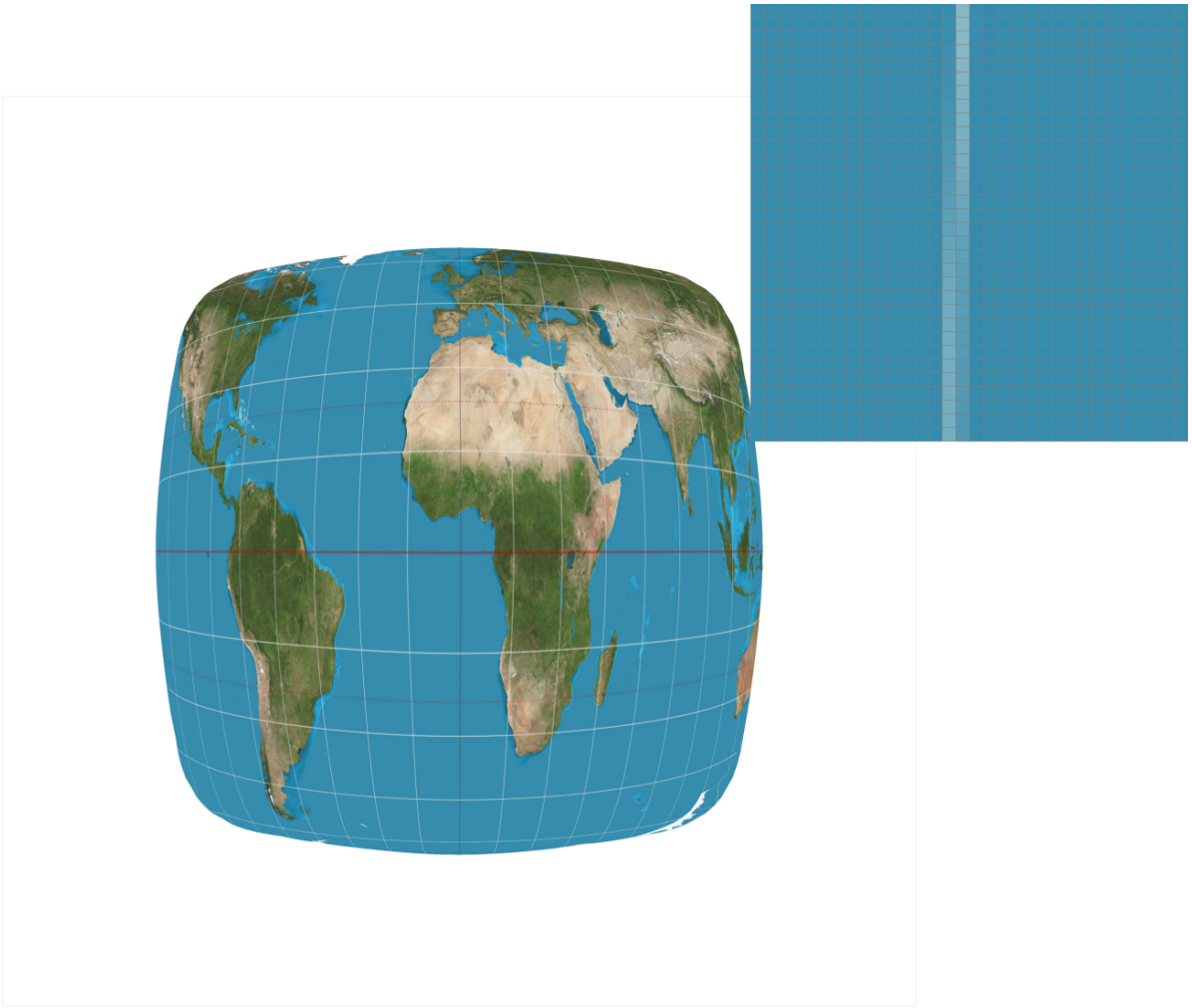


Figure 25: Bilinear pixel sampling with sample rate 16. The lines of the grids are of high quality. Pixel inspector shows a gradual change of color that blends into the blue background.

6 Task 6

6.1 Level Sampling

The discussion in Section 5.1 approximates f inside each triangle with the affine transformation. However, as we have seen from the rendered grids in the lecture, the line segments of the same length map to a but shorter segment in the nearby grids than those in the distant grids. Mathematically, this difference could be accounted by the norm $\|A\|$ of the affine transformation matrix A used to approximate f locally. Recall from Section 2.1 that we are only able to accurately capture the signals of frequency less than or equal to half of the sampling frequency. The fact that a short line segment in the distant grids could map to a large distance in the texture space means that our sampling frequency is too low. The low sampling rate could result in aliasing effects such as moiré patterns in distant grids. One potential solution is to filter out the high frequency with a low-pass filter. However, at the close grids, where the norm $\|A\|$ is small, the sampling rate in the texture space is sufficiently high to keep high frequency signals filtered out.

As is discussed in the lecture, one common practice to address this discrepancy at different parts of the picture is to generate different levels of mipmaps (multum in parvo), where level n mipmap ($n \in \mathbb{Z}_+$) is generated by filtering out high frequency signals and downsampling by a scale of 4^n in area.

The general philosophy of level sampling is choosing the mipmap of the most appropriate level, and sample the texel color in that mipmap. Since different mipmaps have different lengths and widths, we have to store coordinate (u, v) as the version normalized by the length and width of the mipmap. When need to get the actual (u, v) coordinate in the mipmap, we just need to multiply the normalized coordinates with the length and the width.

As we have discussed before, the level of the mipmap chosen is related to the norm of matrix A in the affine transformation $Ax + \mathbf{b}$ used to approximate f locally. By linear algebra, we know that if f is smooth (as a map between screen space and the texture space should be) and we try to approximate f in a neighborhood near the point $(x_0, y_0) \in \mathbb{R}^2$, then A is the Jacobian matrix $\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}$ at (x_0, y_0). In our implementation, we will approximate the partial derivative terms with the discrete changes of u and v when x or y changes by 1. We construct a struct `s` of `SampleParam`. When we have determined a point (x, y) to be inside the triangle, we can compute its barycentric coordinate with our static method barycentric introduced in Section 4. As we discussed before, for a small triangle patch, we use affine approximation for the map from the screen space to the texture space, so we can approximate (u, v) described in Section 5.1. We make a `Vector2D` of (u, v) and make it the `s.p_uv` field of the struct. Then we compute the barycentric coordinates of $(x + 1, y)$, use them to compute the (u, v) and store it in the `s.p_dx_uv` field. Repeat the same computation and save the new (u, v) coordinates for $(x, y + 1)$ and save it in the `s.p_dy_uv`. Now we want to set the field `s.psm` to be pixel sampling method

described in Section 5, and `s.lsm` the level sampling method we are going to discuss below. Now the struct `s` contains all the information we need to sample this pixel (x, y) in the texture space, so we can use `tex.sample(s)` to get the texture color and save it in the sample buffer for rendering.

The most obvious implementation of level sampling is always using the level 0 mipmap, which is the original texture space. Then the result we get is exactly the same as method described in Section 5 (it was conducted on level 0 mipmap). In fact, the pixel sampling methods in Section 5 are independent of the level sampling, and could be conducted on mipmaps of any level (so we have two separate fields `s.psm` and `s.lsm` in the struct).

If we want to have a non-trivial level sampling, we need to determine the appropriate mipmap level for the points in the small triangle patch. Let us denote `(sp.p_dx_uv.x - sp.p_uv.x) * width` as Δu_x and `(sp.p_dy_uv.x - sp.p_uv.x) * width` as Δu_y for the sake of discussion (due to the implementation of `Vector2D` struct, `.x` accesses `u` and `.y` accesses `v`). We can definte Δv_x and Δv_y similarly.

Using the apprximation we discussed above, we have the approximate Jacobian matrix $A = \begin{bmatrix} \Delta u_x & \Delta u_y \\ \Delta v_x & \Delta v_y \end{bmatrix}$. From linear algebra, we know that all norms are equivalent in the finite dimensional vector spaces up to a factor of a constant, so we can just choose the norm returns the length of the column with the largest magnitude. Therefore, we have $\|A\|^2 = \max\{\Delta u_x^2 + \Delta v_x^2, \Delta u_y^2 + \Delta v_y^2\}$. By the construction of the mipmaps, we know the width and height scale down by a factor of $\frac{1}{2^n}$ at n-th level. In practice, we can see that $l = \log_2 \|A\| = \frac{1}{2} \log_2 \|A\|^2$ (the logarithm operation here will reduce the constant factor to a constant offset if we use another norm definition) returns a value closest to the mipmap level we want, so we implement a method `get_level` that takes a `SampleParams` struct and returns this value.

Notice that for small norms, the logarithm function can return a negative value. This situation can happen when the screen has high resolution while the texture has low resolution. Conversely, if the norm is very large, when a small line segment in the screen space maps to a large section in the texture space, the logarithm function might return a value larger than the highest level of mipmaps generated. Therefore, it is necessary to use the `std::min` and `std::max` functions in C++ to clamp the value l between 0 and the highest level of mipmaps generated minus 1.

In addition, the logarithm function most likely returns a value l that is not an integer. There are two approaches to determine which level of mipmap we should use. If we take the nearest integer n of clamped l with `static_cast<int>(l + 0.5f)` and use it as the level to sample the texture space, we repeat the pixel sampling process described in Section 5.1, except this time on the n-th level of mipmap and compute (u, v) with its width and height instead of at level 0. This method is called the nearest level sampling. However, if we clamp l to be between 0 and highest mipmap level minus 2, and let l_1 and l_2 be the two integers clamped l falls between with `int l1 = static_cast<int>(std::floor(l))` and `int l2 = l1 + 1`, we can use $l - l_1$ as the coefficient to linearly interpolate the texel

color sampled at l_1-th and l_2-th level mipmaps. This method is called linear level sampling. In particular, if we use bilinear pixel sampling and linear level sampling, we will get a color continuous changing rather than flips abruptly. This combined pixel sampling and level sampling approach is called trilinear filtering.

Zero level sampling uses the original texture. If its implementation does not generate different levels of mipmaps, then it has the highest speed, lowest memory usage, but the worst antialiasing power because it fails to filter out high frequency for far away objects. However, if we also generate all levels of mipmaps and only use the level zero mipmap for the zero level sampling, then it has roughly the same memory usage as the other two methods since mipmaps are shared and most of the memory use comes from storing the sample buffer and mipmaps. The speed advantage becomes minimal, since the sampling logic differs only slightly from the other methods.

Nearest level sampling has slightly higher speed and lower memory usage than the linear level sampling since it involves fewer computations and directly takes the closest integer value for the mipmap level. The antialiasing power is stronger than zero level sampling but weaker than linear level sampling because the changes are not continuous. If the object moves, the level can change fast and cause distracting flickering or abrupt color changes.

Linear level sampling has the lowest speed due to the extra computations involved to interpolate between different levels of mipmaps and roughly the same memory usage as nearest level sampling since mipmaps are shared and most of the memory use comes from storing the sample buffer and mipmaps. However, it has the best antialiasing power. Even if you move the objects, the color change continuously, giving a smooth feeling.

However, these comparisons are largely theoretical. The psychological effects of the viewers also impact how people think about the image quality. For example, pictures with zero level sampling often appear sharper at the cost of some distracting artifacts. Even though they might have more aliasing, some people might still prefer those pictures.

6.2 Picture Examples

In this section, we have included pictures of the US map with bilinear pixel sampling and different level samplings at sample rate 1. As we can see, state names interfere with each other and exhibit moiré patterns in the picture with zero level sampling. The state boundaries and names also show up disconnected even though they should be connected. Those aliasing effects are consistent with our discussion of the issues associated with using only the level zero mipmap. In the picture with nearest level sampling, the state boundaries and names are correctly connected. However, we can see that at the intersection of curves, some boundaries appear very blurred (e.g. the boundary between SD and NE). One possible explanation is that those segments are an edge case between two mipmaps of different levels, a small perturbation changes the mipmap used, so the pixels are not closely related to its neighbors and appear "confused." In the

picture with the linear level sampling, we can see that the state boundaries and names are correctly connected. In addition, the blurred curves in the picture with nearest level sampling are well defined with its color changing smoothly.

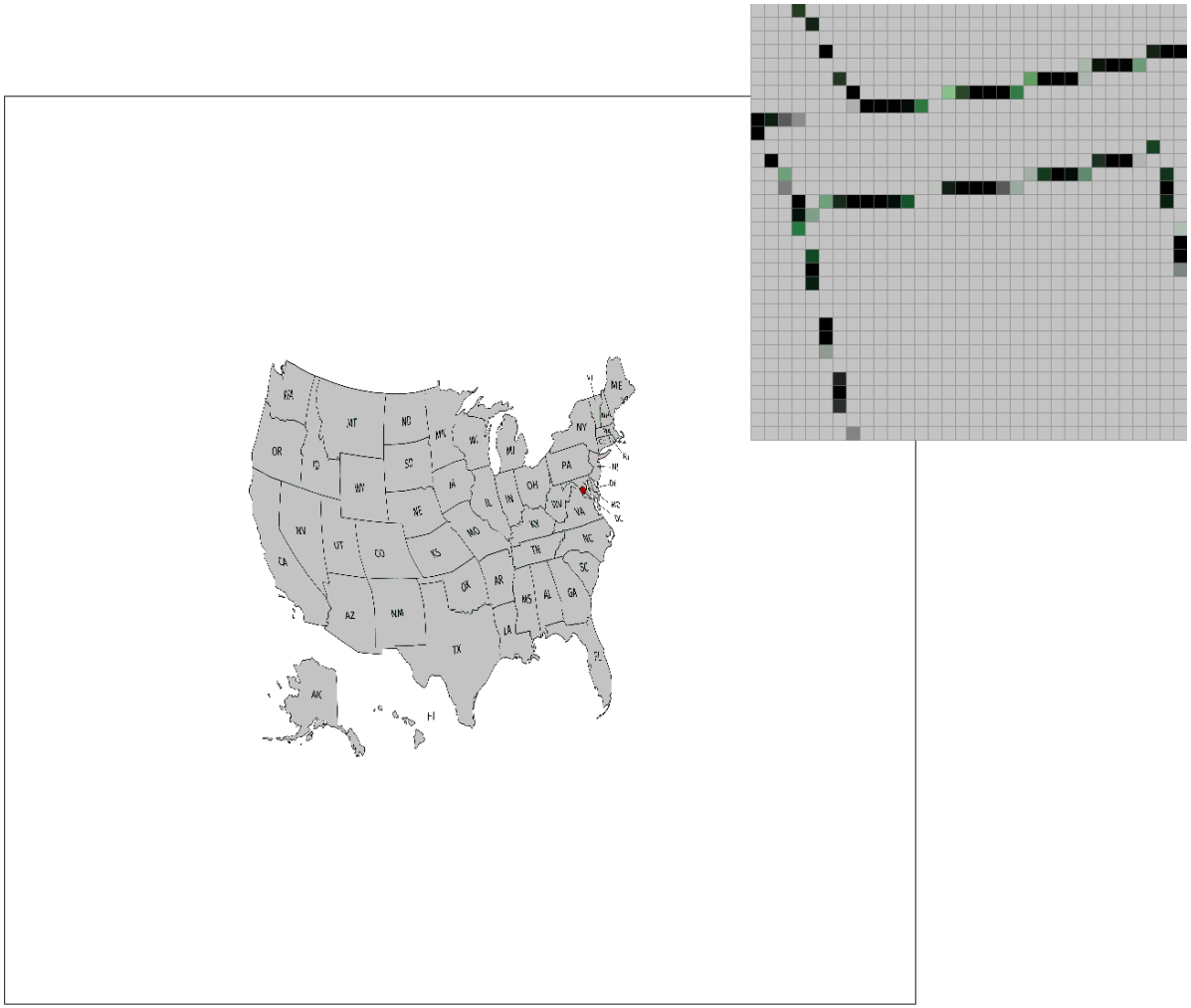


Figure 26: Bilinear pixel sampling with zero level sampling at sample rate 1. The state names interfere with each other and exhibit moiré patterns. The state boundaries and state names show up disconnected. Pixel inspector highlights one such gap.

Figure 27: Bilinear pixel sampling with nearest level sampling at sample rate 1. The state boundaries and state names are connected now.

Figure 28: Bilinear pixel sampling with linear level sampling at sample rate 1. The state boundaries and state names are connected. Pixel inspector shows that the color of the boundary changes more continuously.

Figure 29: Bilinear pixel sampling with nearest level sampling at sample rate 1. At curve intersections some lines are very blurred. Pixel inspector highlights the blurred segment.

Figure 30: Bilinear pixel sampling with linear level sampling at sample rate 1. Pixel inspector shows that the segments are much better defined and the color changes smoothly.

6.3 Anisotropic Sampling

Recall in section 2.1, we increased the sample rate with supersampling in the texture space to raise the Nyquist frequency. In that way, we were able to capture higher frequency signals and reduce aliasing. From the discussion in Section 6.1, we know that the matrix $A = \begin{bmatrix} \Delta u_x & \Delta u_y \\ \Delta v_x & \Delta v_y \end{bmatrix}$ gives us an approximate of the Jacobian matrix to approximate f as an affine function locally in the small triangle patch. A larger norm $\|A\|$ means that a small change in the texture space causes a huge change in the texture space. What if we use the same philosophy of supersampling and increase our sampling rate in the texture space?

In our earlier rasterization examples without texture mapping, we have $(x, y) \in \mathbb{R}^2$ changing uniformly in the screen space. However, due to the map f from the screen space to the texture space, our $(u, v) \in \mathbb{R}^2$ do not change uniformly in the texture space. Since our Nyquist frequency is lowest along the direction where f changes fastest, we want to increase sample in the texture space along this line, namely in the direction $\mathbf{d}$ such that $|A\mathbf{d}|$ is the largest. Notice this description is exactly the definition of the norm of the matrix. For our approximation of the Jacobian matrix, $\mathbf{d}$ is in the same direction as $(\Delta u_x, \Delta v_x)$ or $(\Delta u_y, \Delta v_y)$. Let $\mathbf{d} \in \mathbb{R}^2$ be the vector between $(\Delta u_x, \Delta v_x)$ or $(\Delta u_y, \Delta v_y)$ that has the larger norm. In our program, we want to take `SAMPLE_SIZE` samples between $-0.5\mathbf{d}$ and $0.5\mathbf{d}$ for every point `p_uv` in the texture space. Eventually, we average the texel color at every sample to get the color for the screen space. Note that anisotropic sampling is independent of pixel sampling method and the mipmap level. We can sample each point in the texture space with nearest or bilinear pixel sampling. We can also perform the sampling on different mipmap levels, using the zero, nearest, or linear level sampling methods.

Below we have included the same US map with or without ansiotropic filter. We can see that the picture with ansiotropic filter appears much sharper. The state names such as WY are blurry in all other pictures, but they are sharp in the image with the ansiotropic filter.

However, ansiotropic filter adds another layer of computation and slows down the rendering. We chose `SAMPLE_SIZE = 64` for our US map, and the program noticeably slowed down. In our implementation, we resolved the color without storing the samples in an extra buffer, so the memory usage is only slightly higher.

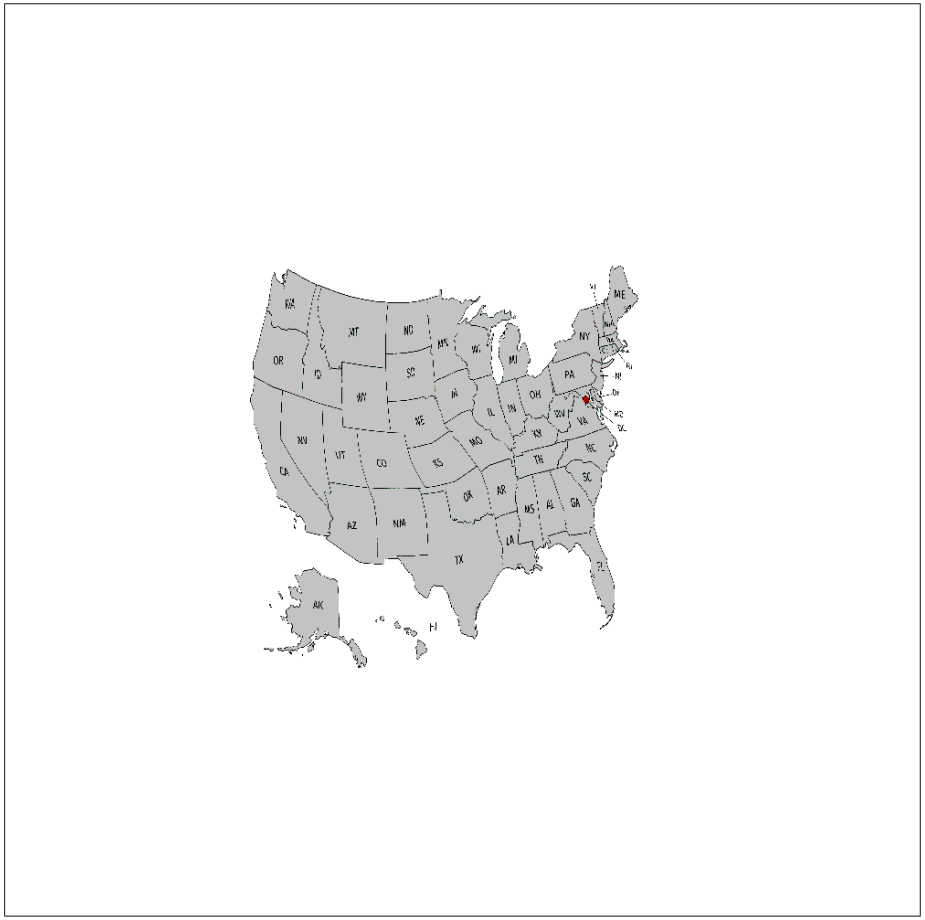


Figure 31: Zero level sampling and nearest pixel sampling. The state names and boundaries are disconnected

Figure 32: Nearest level sampling and nearest pixel sampling. The state names and boundaries are correctly connected, but there are blurry curve intersections.

Figure 33: Linear level sampling and nearest pixel sampling. The state names and boundaries are correctly connected. The blurry curve intersections disappeared.

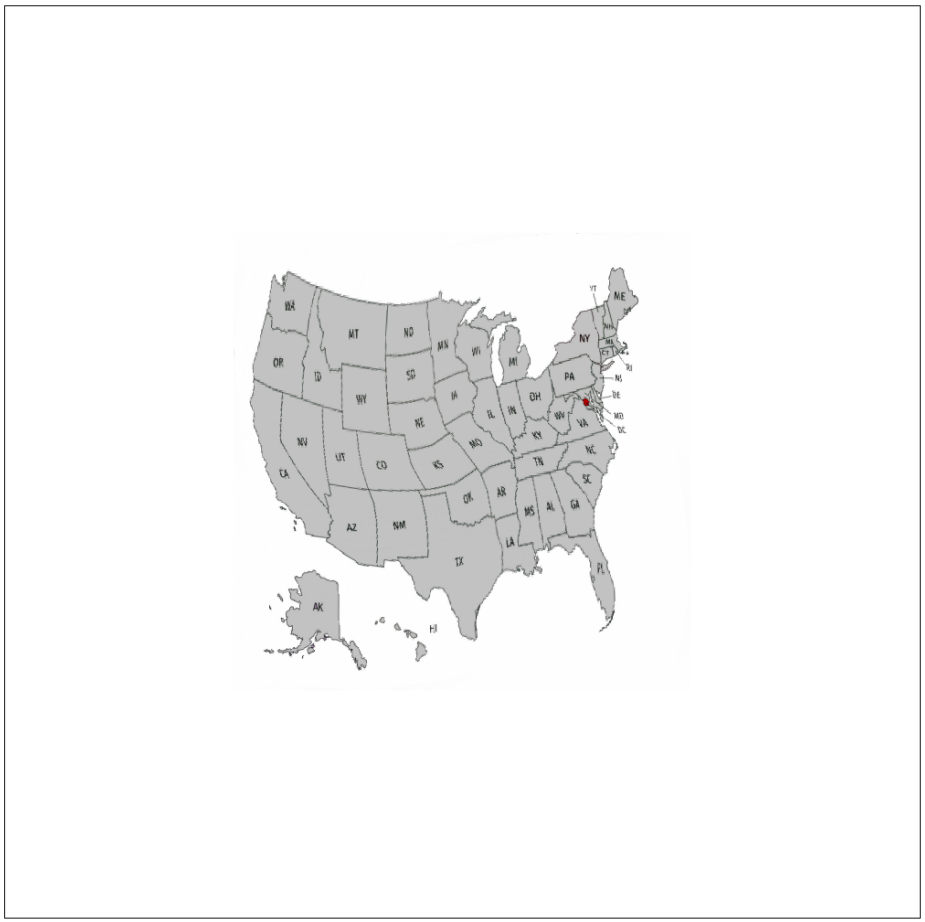


Figure 34: Linear level sampling, nearest pixel sampling, and anisotropic sampling. The image looks much sharper. The state names are noticeably less blurry.