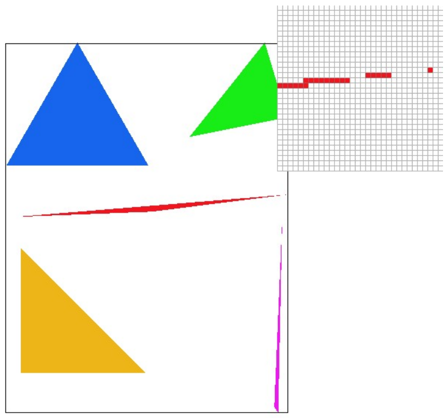
Task 1: Drawing Single-Color Triangles

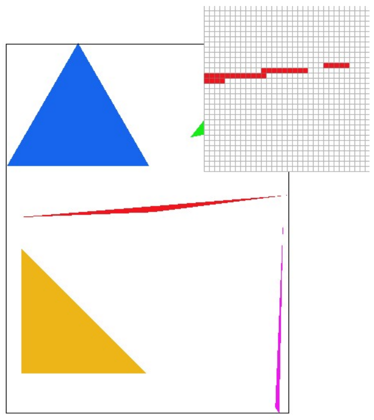
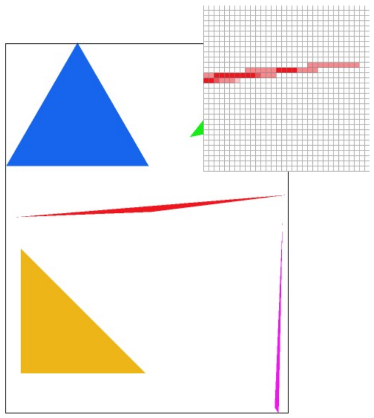
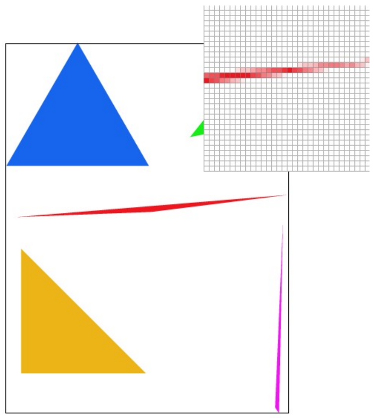
- Algorithm:
  - Rasterization:
    - Pick out the largest x, y and smallest x, y values from the triangle's vertices. Using these four values, we formulate a rectangle covering the whole triangle.
    - Loop through each pixel inside this rectangle and check if it is inside the triangle, call *fill_pixel* if this pixel lies inside the triangle.
  - How to check if a pixel is inside a triangle:
    - Formulate the line function *L(x, y)* using the vertices A and B.
    - Calculate *inside_sign* by plugging in C's coordinates.
    - Calculate *inside_flag* by plugging in the pixel's coordinates.
    - Compare their signs:
      - Match: repeat the procedure 2 more times with *L(x, y)* formulated using vertice B and C, A and C. If all match, this pixel is inside the triangle
      - Not match: this pixel is not within the triangle.
  - Efficiency: this algorithm only inspects x and y between the boundary of smallest and largest values provided. In other words, it only concerns with the pixels inside the bounding rectangle that the triangle fits in.



basic/test4.svg

Task 2: Antialiasing by Supersampling

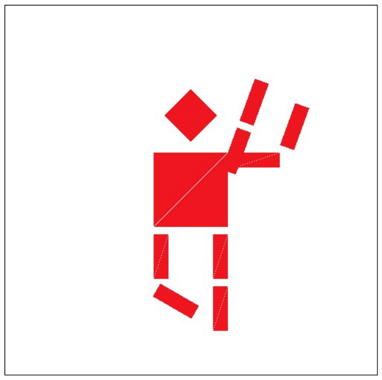
- Algorithm:
  - Rasterization:
    - Pick out the largest x, y and smallest x, y values from the triangle's vertices. Using these four values, we formulate a rectangle covering the whole triangle.
    - Loop through each pixel inside this rectangle. At each pixel, check if its subpixels are inside the triangle and call *fill_pixel* on the subpixels if needed.
  - How to check if a subpixel is inside a triangle:
    - Formulate the line function *L(x, y)* using the vertices A and B.
    - Calculate *inside_sign* by plugging in C's coordinates.
    - Calculate *inside_flag* by plugging in the pixel's coordinates.
    - Compare their signs:
      - Match: repeat the procedure 2 more times with *L(x, y)* formulated using vertice B and C, A and C. If all match, this pixel is inside the triangle
      - Not match: this pixel is not within the triangle.
  - How to color a pixel using its subpixels:
    - Loop through *sample_buffer* and average the color of all its subpixels.
    - Fill the pixel with this color.
  - Data structure: extend array *sample_buffer* to accommodate more subpixels of the original pixel.
  - Supersampling is useful when we need to smooth out the corners, points or diagonal lines, in short, reduce jaggies.
  - Modification: 2 additional loops were introduced so that the program can inspect individual subpixel at each point.

 *sample_rate == 1*	 *sample_rate == 4*	 *sample_rate == 16*

basic/test4.svg

- When *sample_rate = 1*, there are several "disconnected" pixels, the diagonal lines are also jaggies. Additionally, a pixel either has a very bright color or none at all (white). As *sample_rate* increases, the triangles are now seemingly "continuous". This happens because when *sample_rate = 1*, it's either hit or miss for a pixel to be considered inside a triangle, but when we take its subpixels into consideration, that pixel can have more than just binary values.

Task 3: Transforms

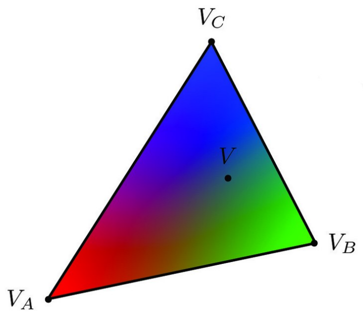


Dancing Robot

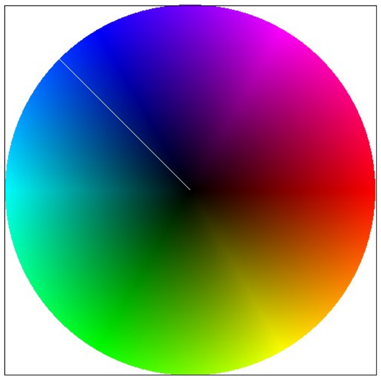
- This is a robot dancing with two hands clapping on one side.

Task 4: Barycentric coordinates

- Barycentric coordinates is an approximation of various attributes (namely color) based on relative coordinates.



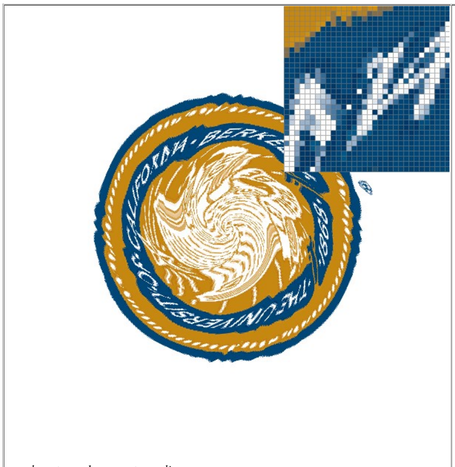
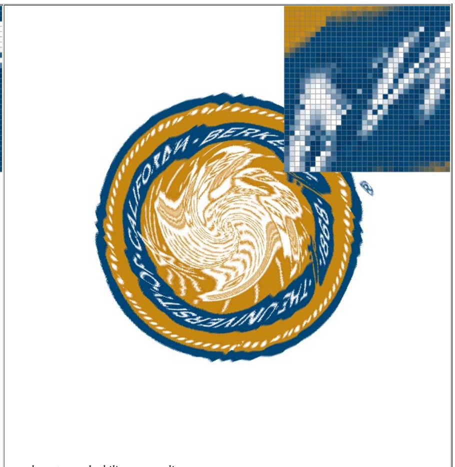
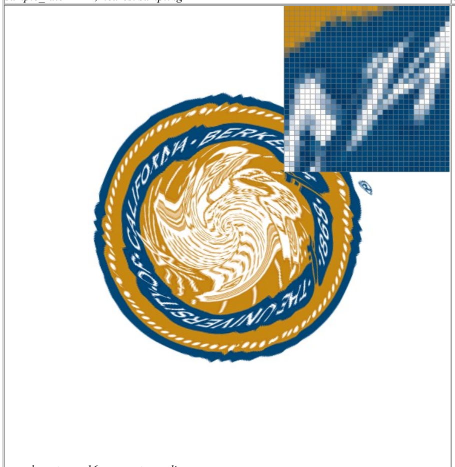
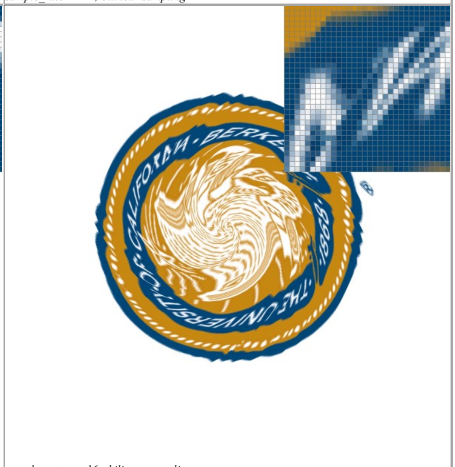
- Take this image as an example, we know the color at vertex A is red, B is green and C is blue. Our objective is to fill the whole triangle with color. Barycentric coordinates will help us "guess" the color at any given point inside the triangle.



basic/svg/test7.svg

Task 5: "Pixel sampling" for texture mapping

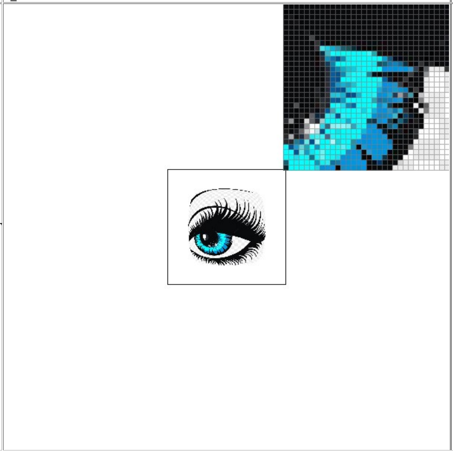
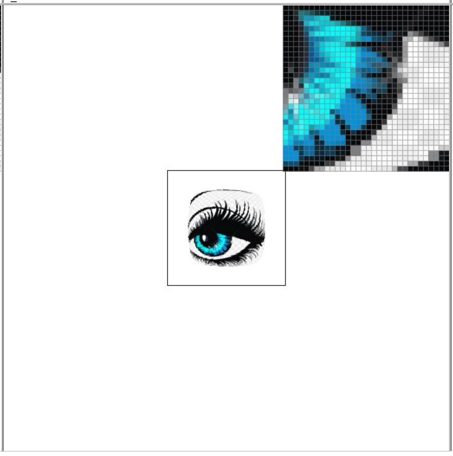
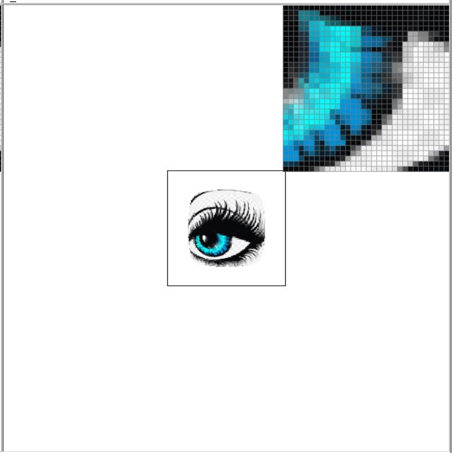
- What we have been doing so far is filling the whole triangle with a single color, or at most using a color derived from the relation between 2 colors. On the other hand, pixel sampling is when you “sample” a pixel’s color from the texture file, then transfer (sometimes with modification) that color over to the pixel at relative position on the triangle. It’s like cutting a picture into pieces then stretch, compress and glue them together to make a new, hopefully better but sometimes uglier, picture.
- There are two ways in which pixel sampling is done, they all start with translating the coordinates of the canvas, which we are drawing, to the coordinates of the pre-drawn picture (called texture file). Once we have the texture’s coordinates of the pixel, we have two options:
    - The first being getting color of the pixel nearest to the pixel being inspected, transfer that color without any modifications to the canvas - This is called Nearest Sampling.
    - The second requires much more work as we look at all four pixels around that point, average the color, then apply that onto the canvas - This is called Bilinear sampling.
- How I did it:
    - Rasterizing a triangle stays the same as described in *Task 1*
    - There is no preset color given this time, use barycentric coordinates to translate canvas' coordinates (x, y) into texture's coordinates (u, v).
    - *u* and *v*'s range is [0, 1]. Must multiply with texture's dimensions to get actual data point (tx, ty) in 'float' type.
    - There are 2 methods to get pixel's color from here:
        - Nearest Sampling: round (tx, ty) to the nearest integer, retrieve and return pixel's color from texture map.
        - Bilinear Sampling: collect and average colors (using linear interpolation) of all the adjacent pixels (4 to be exact), then return this color.

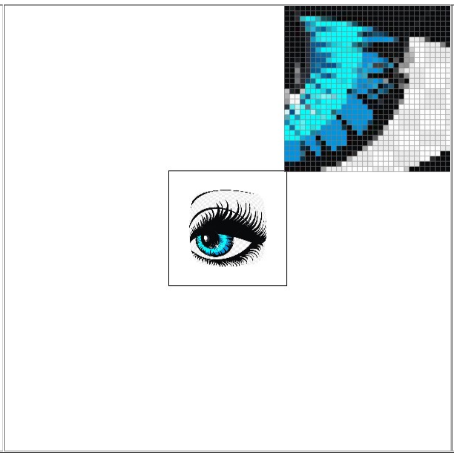
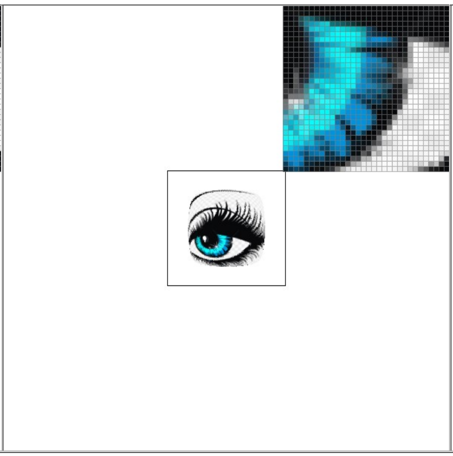
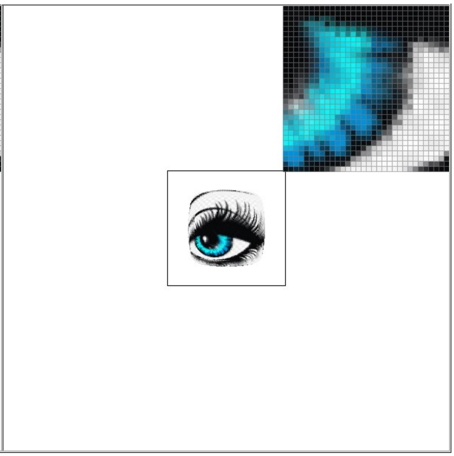
 *sample_rate == 1 ; nearest sampling*	 *sample_rate == 1 ; bilinear sampling*
 *sample_rate == 16 ; nearest sampling*	 *sample_rate == 16 ; bilinear sampling*

- While supersampling does help in smoothing out the edges, bilinear sampling method always seems to do a better job at making the image looking less "sharp". This is because supersampling relies on a more naive approach: averaging regardless of actual distance from the inquiring pixel to its neighbors, like how communism or group projects in high school go: resources / grades are equally divided upon every member regardless of real individual effort, it works but not necessarily the best option, still better than all or nothing as we can see in the example where *sample_rate == 1*. Bilinear uses a fancy mathematical approach - linear interpolation - to make an educated guess of how a pixel on the canvas should look like, thus, yields a much more acceptable result.

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

- Level sampling does exactly what pixel sampling does, but better! Level sampling not only prefilters the texture to reduce alias, but also improve the speed of sampling textures.
- As for the coding's side, all the fundamental steps are reserved from rasterization of triangles to using barycentric coordinates to retrieve texture's coordinates. The only difference is in the looking up pixels part because we now have multiple copies of a same texture at different resolutions. These texture files are stored in different "levels". Therefore, we must calculate which level we need access to using differential. Important to point out, we can be **inbetween** the levels. Once we know the "level":
  - Go to the nearest level and pick the texture file in the safe, then either use nearest sampling or bilinear sampling to sample pixels.
  - Pick both lower and upper level, get both texture files, use either nearest sampling or bilinear sampling to sample pixels, then interpolate the results from both textures. Finally, also apply linear interpolation on the "level" before merging the colors.
- Examples:

	L_ZERO	***L_NEAREST***	***L_LINEAR***
P_NEAREST			

P_LINEAR			

- Without a doubt, L_LINEAR needs the most memory since it must keep various copies of the same picture in the system. However, the rendered picture will always retain its general shape, as in easier to recognize when being zoomed out. With this in mind, we can say that the antialias attribute is strong when we use linear approach. But this comes at a cost of both memory usage and speed as there is much more computations to go through, namely barycentric coordinates. This approach is only suitable for powerful proccessors.
- The basic settings (L_ZERO, P_NEAREST) surely offers the highest render speed as well as quality when we need to inspect the picture/object closely. That being said, the picture will lose details quickly as we move further away (zoom out). This is definitely the most reasonable option for portable systems or some that have limited computing power.
- NEAREST approach seems to be the most rounded options as it also requires as much memory as LINEAR but does not need quite as much number crunching operations. Still more than L_ZERO case since this approach requires barycentric coordinates to figure out the appropriate level. In summary, it performs arguably well in any given situation, a jack of all trades.