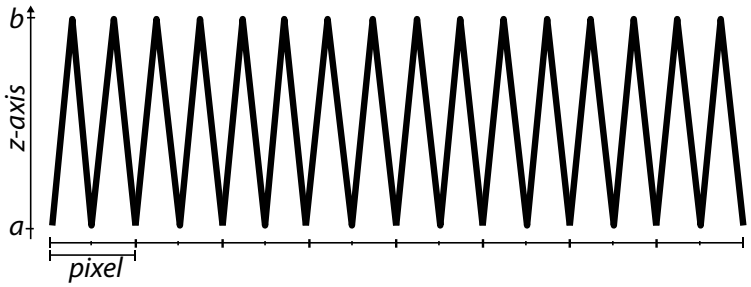
Stanford CS248: Interactive Computer Graphics
Participation Exercise 2

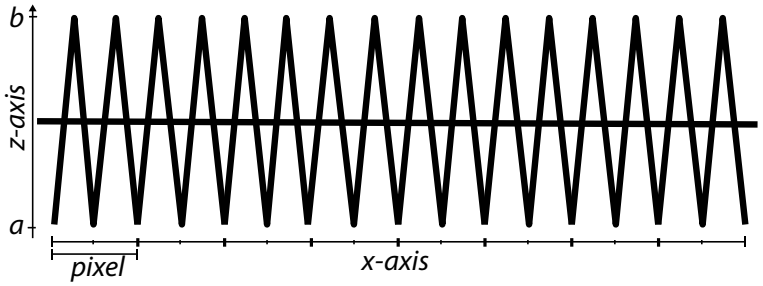
Problem 1: Rasterizing a Comb

A. Your project partner has just finished up a new version of the rasterizer that renders 3D geometry (not just 2D SVG files). They try out their new system by rendering a very high resolution mesh, as shown below. In the figure below, we're showing you the final location of the geometry after all transformations. The X-axis in the figure is the X-axis of the screen, with pixel locations shown. The Z axis is also shown, illustrating that the depth of the geometry varies rapidly in a pixel.



Your partner renders this scene with a sample rate of *one-sample per pixel, with sample locations at pixel centers*. As a debugging exercise, they look at the resulting Z-buffer and find the farthest sample. "Hmm, says your partner. I know that in this scene, the farthest point should be at depth 'b', but in my Z-buffer, I don't see any values of that depth. I must have a bug!" Is the Z-buffer correct given the sampling method? If it is incorrect, state what the correct Z-buffer values should be. If it is correct, explain why your partner does not see 'b' value in the Z-buffer.

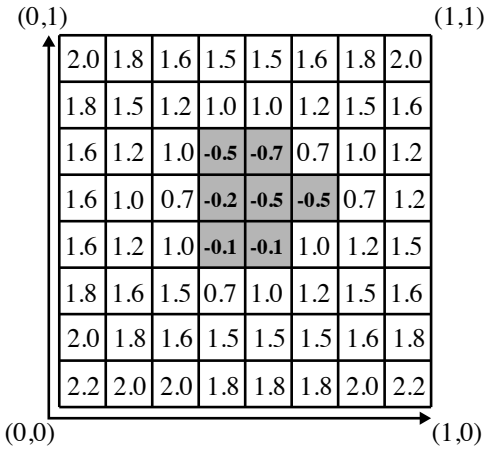
B. In the above example, imagine your partner added a new piece of geometry to the scene that was a **red-colored** plane, parallel with the image plane at depth $(a+b)/2$ as shown below. Imagine that the original geometry was colored **green**. Assuming that the rasterizer continues to sample the scene at pixel centers, and renders the scene with depth buffer to resolve occlusions, what will the rendered picture look like?



C. Now imagine the scene is supersampled at two samples per pixel, with the samples in pixel (i, j) chosen to reside at $(i + 0.25, j + 0.5)$ and $(i + 0.5, j + 0.5)$. Assuming that the final image is resolved using a pixel-sized box filter, describe the appearance of the final image. To be precise, if possible give the numerical RGB colors for pixels. **Note: although the Y-axis is not shown in the figure, just assume that the geometry at a given X, has the same depth along Y.**

Problem 2: Rasterizing a Level Set

In class we talked about a level set surface representation where each cell in a grid stores the value of a function sampled at the center of each grid cell. The surface is given by the zero-crossing of this function when **it is reconstructed using bilinear interpolation**. For example, consider the surface defined on the following 2D $[0, 1]^2$ domain, which is encoded as a 8×8 array of samples.



Now imagine that you want to extend your SVG renderer implementation to also render level set primitives. Assume that all level set primitives are associated with a transform $\mathbf{T}$ that describes how to transform points in the domain of the level set to points in 2D canvas space, which is defined with (0,0) in the BOTTOM-LEFT of the screen and (W,H) in the TOP-RIGHT of the screen. You may assume that you also have the transform $\mathbf{T}^{-1}$.

A. Please describe an algorithm for rasterizing the level set. Color the screen black if it is INSIDE the level set (the function's value is less than zero), and white otherwise. You may assume that `getSamplePos(px,py)` returns the screen (canvas) sample point for pixel (x,y). You may also assume that you have access to a function `bilerp(s, t, i, j)`, which evaluates the value of the level set function via bilinear interpolation of the samples at level set cells (i,j), (i+1,j), (i,j+1), (i+1,j+1) according to coefficients s and t. You need not worry about algorithm efficiency, or edge-case behavior near the edges of the level-set.

B. Consider the case where the output image size is 1024×1024 and the corners of the level set object map to screen coordinates (512,512), (1024, 512), (1024,1024), and (512,1024). Given your algorithm in part A, will the object described by the level set look blurry on screen, or will it have a sharp edge at the boundary of the object? Why or why not?

C. Imagine you wanted to extend all shapes in your 2D SVG renderer to carry an additional value "depth" which is the distance of the shape from the "camera" (lower depths are closer objects). In this case all shapes are contained within a single Z plane. You decide to implement occlusion calculations with a depth-buffer as described in class. Your friend looks at you as says "hey, while that's a correct implementation, that's not necessary to correctly render pictures with correct occlusion in this case." Given your renderer implementation in assignment 1 (which draws all objects in the order it is given), describe a method to get correct occlusion without using a depth-buffer.

D. Imagine that we extended the level set representation to also maintain a per-cell DEPTH value, so that the depth of the surface at a point in the domain was also determined by bilinear interpolation. Given your algorithm in part A, could a depth buffer be used to correctly handle occlusion in a scene with multiple level sets, as well as multiple triangles with different depths? Why or why not?

Problem 3: Texture Mapping (One More Time!)

Consider a 1024×1024 texture map whose value at pixel (x,y) is white if x mod 2 = 0, and black otherwise. This texture is used to texture a single triangle with vertices p0=(0,0), p1=(1,0), and p2=(0,1) and uv texture coordinates uv0=(0,0), uv1=(1,0), and uv2=(0,1)

Consider rendering this triangle to a 512×512 image, where the **background color is 50% gray**. The scene viewport is set up so that scene coordinate (0,0) is in the bottom left of the image, and (1,1) in the top-right corner.

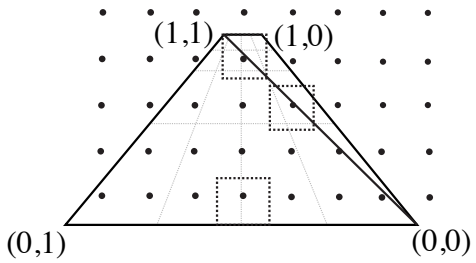
Also assume that screen and texture sample points are at pixel centers (as was the case in assignment 1), and that texture mapping uses **nearest neighbor filtering WITHOUT a MIPMAP.**

A. Describe the image that you will see when you render the scene. Describe both the position of the triangle on screen and what the triangle looks like. (A simple sketch would suffice.)

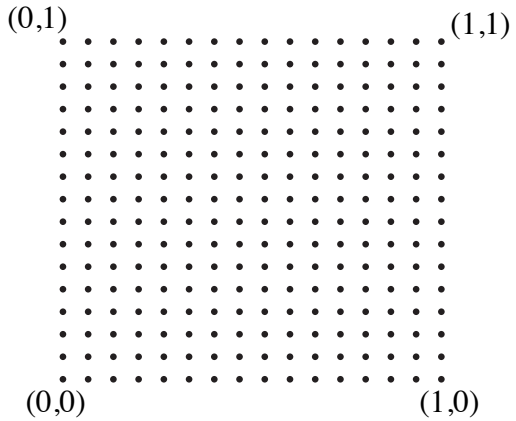
B. Now assume that the rendering mode is changed to **bilinear interpolation** and that the rendered image size is changed to 1024×1024. Describe how you might move the camera (aka pan the viewpoint) to make the triangle *disappear!*

C. **This problem is unrelated to parts A and B.**

Consider rendering the two triangles under perspective projection shown at left in the figure below. Per-vertex texture coordinates for the four vertices are given, and the dots indicate the position of screen sample points during rasterization. Now consider the computation to compute the color of the scene at the highlighted screen sample point, which requires a texture lookup into the 16×16 texture shown at right.



Rendered Image



16 x 16 texture (color samples stored at dots)

Three sample points are highlighted in the left side of the above figure, along with dotted boxes showing the extent of the corresponding pixel. In the figure at right, draw the corresponding polygons that correspond to the texture space extent of these screen regions. **BE CAREFUL! Pay attention to the texture coordinate values.**

D. Assume that texture mapping is performed using *bilinear filtering with a mipmap*. Will texture mapping operations to compute the color of the triangles near the top of the rendered image access higher levels of the mipmap (lower resolution textures) or lower levels of the mipmap? Why?

E. Consider the compute cost of texture mapping operations (using mipmapping and bilinear filtering as in part D) for samples at the top of the image or the bottom of the rendered image. Is the cost of texture mapping higher at the top or bottom, or the same? Why?

Problem 4. **THIS PROBLEM IS OPTIONAL AND WILL NOT BE GRADED FOR PARTICIPATION.** In class we talked about the limitations of rendering transparent triangles using rasterization. First, to get correct output, the triangles need to be drawn in front-to-back (or back-to-front) order. Second, if two triangles interpenetrate, it's actually impossible to order drawing so that the ordering of the triangles is the same for all sample points.

Now consider a modified rendering algorithm where instead of there being a single RGBA and depth value stored at each sample point, there is an array of up to 16 values. The frame buffer also stores the number of fragments stored in the frame buffer at each sample point, as shown below.

```
struct Sample {
  float r,g,b,a,z;
};

Sample frame_buffer[WIDTH][HEIGHT][16];  // all samples initialized to (0,0,0,0,INFINITY)
int    num_values[WIDTH][HEIGHT];        // initialized to 0
```

Now imagine you have the following two functions:

```
void process_fragment(Sample new_frag, int x, int y)
void done_rendering(Sample result[WIDTH][HEIGHT])
```

Recall that a "fragment" is the name given to a sample of a triangle. `process_fragment` is called for each fragment generated by each rasterized triangle. It can modify `frame_buffer` and `num_values` as needed. `done_rendering()` is called after all triangles in the scene have been processed. When `done_rendering` returns, the final image pixel values should be written to the buffer `result`. **Assume that the scene has at most 16 triangles**, all triangles are semi-transparent, and that you can make no assumptions about the depth order of the triangles when rendering. In rough pseudocode, describe an implementation of `process_fragment` and `done_rendering` that results in a correct alpha composited image. You may assume that you have handy helper functions that sort an array, and composite two samples on top of each other and return the result (`Sample OVER(Sample s1, s2)`).