Stanford CS248: Interactive Computer Graphics
Exercise 3

It's Good to be Transparent

A. In class we talked about alpha compositing, and stressed that in order to get correct transparency, scene objects overlapping the same sample points must be rendered in strict *back-to-front* order. Assuming that the current state of the frame buffer is given by `frame_buffer` and the incoming (new) fragment from the current triangle has RGBA value `new_tri`, the correct compositing equation is (note this assumes premultiplied alpha):

```
frame_buffer.r = new_tri.r + (1.0 - new_tri.a) * frame_buffer.r;
frame_buffer.g = new_tri.g + (1.0 - new_tri.a) * frame_buffer.g;
frame_buffer.b = new_tri.b + (1.0 - new_tri.a) * frame_buffer.b;
frame_buffer.a = new_tri.a + (1.0 - new_tri.a) * frame_buffer.a;
```

Now assume that triangles are rendered in *front-to-back* order. Please give a new compositing equation that also results in correct transparent rendering.

B. Describe a scene containing two semi-transparent triangles that *cannot be rendered correctly* using a rasterization-based approach where a single RGBA value is stored in the frame buffer (regardless of whether the triangles are rendered in front-to-back or back-to-front order. (Drawing a picture might help.)

C. 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
```

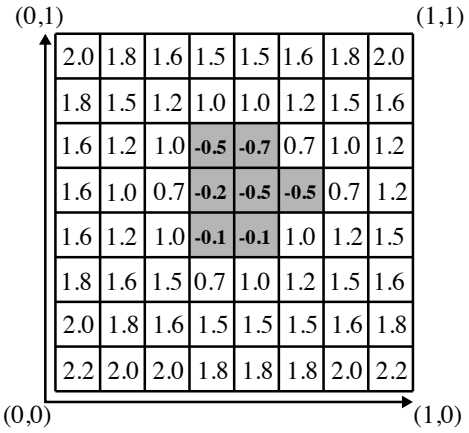
Now imagine you have the following two functions:

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

`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)`).

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. (NOTE THIS IS A DIFFERENT CONVENTION THAN ASSIGNMENT 1!) 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? (Careful! We're asking you to draw the surfac defined by the level set, we are not asking you to draw the 8×8 image that defines that defines the level set. The answer to this question will require you to understand the difference.)

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?