Stanford CS248A: Computer Graphics
Participation Exercise 3

This exercise is graded for CREDIT ONLY. Serious attempts to answer the problems will be given full credit, even if the answers are incorrect.

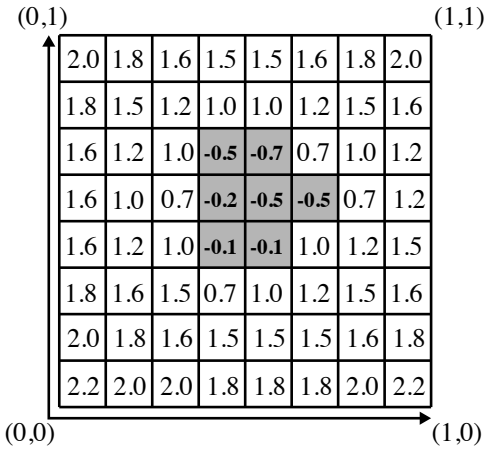
Problem 1: Alpha Compositing Consider three surfaces having following three colors in RGB form.
C1=[.5, 1, .5], C2=[.5, .5, 1], C3=[1, 1, 1]

Compute the **premultiplied alpha representation** of the result of compositing 30% opaque C1 OVER 40% opaque C2 OVER fully opaque C3. Please show intermediate work of the result of compositing C1 over C2. What does the color look like? (feel free to type your answer into a color visualization program on the internet).

Problem 2: Tinting Your Ride. After taking CS248A you start making serious bank at your graphics startup that dominates the admittedly niche, but surprisingly lucrative, SVG rendering market. To reward yourself you buy a limo and take it to the shop to receive a window tint. Tint is applied by fixing 10% opaque ($\alpha = 0.1$) layers of tint to your windows (more layers results in more light attenuation). If you want to have your windows reduce the amount of light entering the limo *by at least 20%*, what is the minimum number of layers of tint required?

Problem 3: 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?

OPTIONAL PROBLEM 4: Implementing a Mini Graphics Pipeline

Below you are given stub code for a very simple triangle rasterizer. The rasterizer is simple. It accepts an array of `NUM_TRIANGLES` triangles where the vertices for each triangle are given in 3D object space (`triVerts`). The rasterizer also accepts a RGB color per triangle (`triColors`). In the code on the next page, you need to fill out the implementation of the rasterizer so that is correctly renders the triangles. Note that:

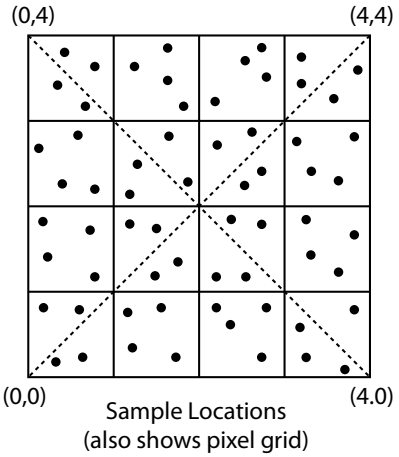
- Your rasterizer should process the triangles in the order they appear in the input array. No triangle sorting.
- Your rasterizer samples coverage at pixel centers.
- **HINT: Recall the perspective projection transform (`persp` in the code), followed by homogeneous divide (hint, hint) puts vertices in normalized screen space (also called normalized device space). In this problem, we'll define normalized screen space as follows: the bottom-left corner of the screen is (-1,-1) and the top-right is (1,1).**
- In pixel-coordinates, the bottom-left corner of the screen is (0,0) and the top-right is (WIDTH, HEIGHT).
- The code uses C-style array indexing. The first pixel is in the bottom-left of the screen, so the bottom-left pixel of the screen is given by `colorBuffer[0][0]` and `colorBuffer[HEIGHT-1][WIDTH-1]` is the top-right pixel.
- To make things simple (avoid interpolation) assume that after perspective projection, all vertices in a triangle are the same depth from the camera (have the same `.z` and `.w`). As a result, you can use the depth of the triangle at any projected vertex as the triangle's depth everywhere on the triangle.
- Your implementation should use the helper functions below.

```
// USEFUL FUNCTIONS TO USE IN YOUR SOLUTION...

// transforms input vertices in 3D object space by xform, putting the
// result in outputPos.
// IMPORTANT: Output positions are Vec4's because they are in homogeneous 3D space.
Vec4D vertexShader(Matrix4x4 xform, Vec4D inputPos);

// returns true if samplePos is INSIDE the triangle given by *2D screen space*
// vertices given by pos, false otherwise
// IMPORTANT: assumes vertex positions are in non-normalized screen space where
// the screen is (0,0) to (WIDTH, HEIGHT)
bool insideTri(Vec2D samplePos, Vec2 pos[3]);

// returns true if depth d1 is closer than d2
bool depthCheck(float d1, float d2) {
    return d1 < d2;
}
```

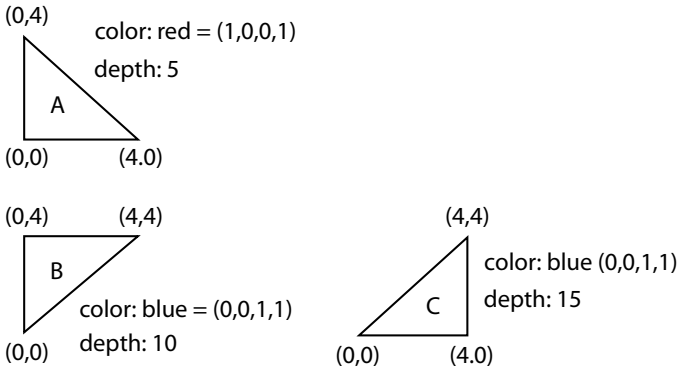
```
// COMPLETE THE IMPLEMENTATION BELOW
// Reminders:
//   -- Make sure you keep in mind what coordinate space your vertex positions
//      are in (homogeneous space? normalized screen space? screen pixel coordinates?)
//   -- You can use the depth at any vertex as the depth of the triangle

const int NUM_TRIANGLES = 32;

Matrix4x4 obj2Camera;             // assume initialized with obj-to-camera transform for the scene
Matrix4x4 proj;                   // assume initialized with the current perspective proj transform

Vec3D triVerts[NUM_TRIANGLES][3]; // assume initialized with vertex positions in 3D obj space
Vec3D triColors[NUM_TRIANGLES];   // assume initialized with per-triangle colors
                                  // (RGB only, so all triangles are opaque)

// outputs
Vec3D colorBuffer[HEIGHT][WIDTH]; // assume initialized to (0,0,0)
float depthBuffer[HEIGHT][WIDTH]; // assume initialized to very large number


for (int t=0; t<NUM_TRIANGLES; t++) {

    Vec2D screenPos[3];
    float triDepth;

    // TODO: compute triangle vertex positions and depth









        for (int y=0; y<HEIGHT; y++) {
            for (int x=0; x<WIDTH; x++) {

            // TODO: check coverage, update output buffers









                }
        }
}
```

OPTIONAL PROBLEM 5: Rasterizing Triangles

Consider rasterizing the three triangles (A, B, C) given below (at right) to a 4×4 pixel image with 4× supersampling shown at left.. The coordinates of image space are given on the figure (We're using the convention that (0,0) is in the *bottom left*.) Note that unlike assignment 1, the four sample positions per pixel are now placed at **random locations** in each pixel.

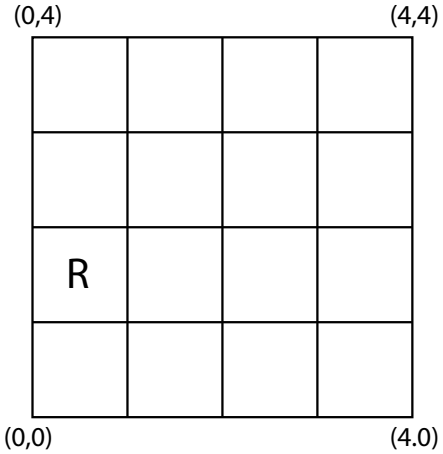


Sample Locations
(also shows pixel grid)



Triangles to Render

A. On the grid below, draw the final rendered output assuming that coverage and depth testing are performed at the provided sample locations, and that the supersample buffer is resampled to a final image by means of **convolving the supersample buffer with a 1-pixel wide box filter**. For simplicity, define the values of several color RGBA variables and just write the variable name in each pixel. (e.g., let R = (1,0,0,1), and one red pixel has been marked for you in the figure.)



B. How would the results in part A change if the resampling filter in part A was replaced with a 3-pixel wide box filter? You only need to answer in words – you do not need to illustrate a result. (You many ignore boundary conditions as well.)

C. Now assume triangle A's color is changed to be 75% opaque red. Recall that in a **non-multiplied alpha representation** this is $C = (1.0, 0, 0, 0.75)$.

Now, assume the renderer is changed to work in the following way... Instead of updating ALL SAMPLES COVERED BY A TRIANGLE that pass the coverage and depth tests, and doing so using the alpha blending equation $C_{\text{new}} = \alpha C_{\text{tri}} + (1 - \alpha)C_{\text{old}}$, the renderer discards a fraction of the triangle's covered samples according to α. Specifically, for a triangle with opacity 75%, the renderer **randomly discards** 1-.75=25% of the samples covered by the triangle, and for all other covered samples, treats the triangle as if it is fully opaque (e.g., has color (1.0, 0, 0, 1.0)).

Assuming the supersample buffer is resolved to a single sample per pixel using a 1-pixel box filter (as was done in Part A), **describe why the new rendering scheme results in the same answer as if alpha blending was used on all covered samples.**

Hint: To keep things simple, your answer need only consider the case where the transparent triangle covers a entire pixel. A clear answer will describe why proposed algorithm gives the same result as the equation above, showing that the math works out the same.

**** For extra credit, give (1) a clear explanation why, in expectation, this approach can rendering transparent surfaces in any order using regular depth testing and no alpha blending. (2) consider why it might not work so well with only a small number of samples per pixel.*

OPTIONAL PROBLEM 6: Order-Independent Transparency

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
```

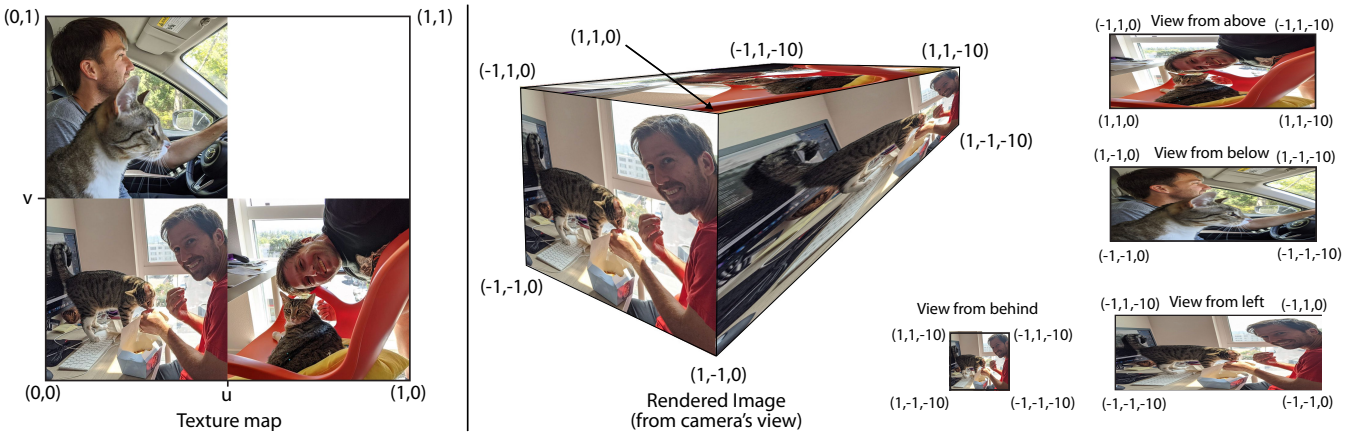
Now imagine you have the following two functions:

```
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)`).

OPTIONAL PROBLEM 7: A Textured Cube (Putting Texturing and Geometry Together)

In this problem you are rendering the textured box shown in the center of the figure below. The box is 2 units in width and height, and 10 units in depth. 3D world space vertex positions are shown on the figure. On the left of the figure is the texture image used. The front, right, back, and right sides of the box all display the bottom-left region of the texture. The top of the box is the bottom-right quadrant (see top view). The bottom of the box maps to the top-left quadrant of the texture (see bottom view).



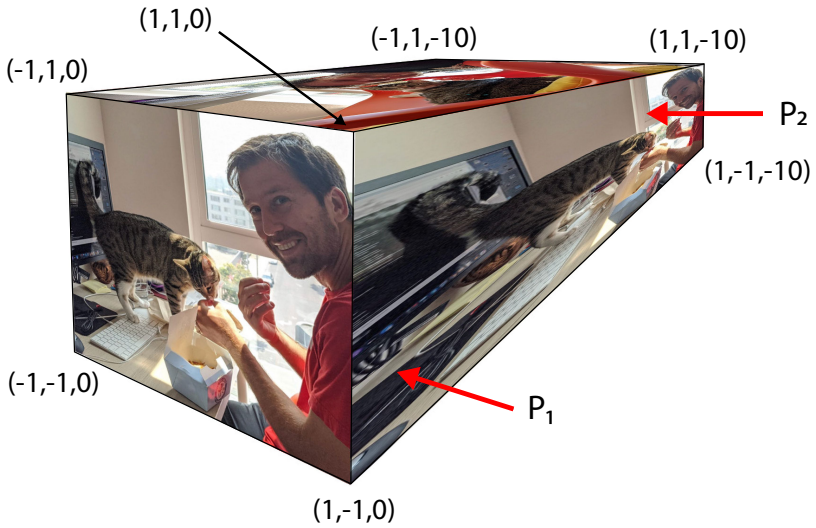
A. In class we talked about indexed mesh representations, where the vertices of each triangle are specified by an index into an array of 3D vertex positions. Below is a partial definition of an indexed mesh. **Please complete the specification of the mesh by filling in the missing indices for the triangles on the box's back and bottom faces. (three triangles are missing.)** Be careful to ensure your triangle "windings" are correct! (They should be consistent with the windings of the other faces and yield a normal that points away from the inside of the box.)

```
Vec3d positions[8] =
    { Vec3D(-1,-1,0),   Vec3D(1,-1,0),   Vec3D(1,1,0),   Vec3D(-1,1,0),
      Vec3D(-1,-1,-10), Vec3D(1,-1,-10), Vec3D(1,1,-10), Vec3D(-1,1,-10) };

int NUM_TRIANGLES = 12;
int posIndices[3 * NUM_TRIANGLES] =
       { 0,1,2,  0,2,3,       // triangles 0 and 1: front face
         1,5,6,  1,6,2,       // triangles 2 and 3: right face


                              // triangles 4 and 5: back face


         0,3,7,  0,7,4,       // triangles 6 and 7: left face
         3,2,6,  3,6,7,       // triangles 8 and 9: top face


         5,1,0,               // triangles 10 and 11: bottom face

       };
}
```

B. The same indexed representation can also apply to per-vertex texture coordinates as well. **Please complete the specification of the mesh texture coordinates by filling in the eight missing texture coordinate values.** Then provide texture coordinate indices corresponding to the vertices of triangle 0 (front face), triangle 2 (right face), triangle 8 (top face), and triangle 10 (bottom face). The result of rendering the box using these texture coordinates should be the image shown in the figure. *Hint: unlike with positions, the same vertex on the box may be different texture coordinates in different triangles!*

```
Vec2D uvCoords[8] =
    { Vec2D(    ,     ), Vec2D(    ,     ), Vec2D(    ,     ), Vec2D(    ,     ),
      Vec2D(    ,     ), Vec2D(    ,     ), Vec2D(    ,     ), Vec2D(    ,     ) };

int uvIndices[3 * NUM_TRIANGLES] =
       {
          // you don't need to fill out indices for all 12 triangles, but please
          // give the texture coordinate indices for triangles 0, 2, 8, and 10
          // (for the grader label them clearly, this doesn't have to be valid C code)









       };
```

C. Imagine that you render the image from the camera viewpoint shown below (it's the same figure copied from the figure on the previous page). Consider shading sample points located at P_1 and P_2 shown in the figure. You implement texture mapping using a mip-map. Which point will require sampling from a **HIGHER** mipmap level? **Please describe why.** (Recall that level 0 is the "bottom" of the mipmap, which corresponds to the full resolution texture.)



D. Consider the appearance of the rendered box when sampling texture color from the mipmap using TRILINEAR FILTERING vs. BILINEAR FILTERING. **In your description, describe what undesirable artifacts we might see on the right face of the box if only bilinear filtering is used.** Remember, in both the bilerp and trilerp cases the shader computes a mipmap level and uses the texture mipmap for sampling.