Stanford CS248: Interactive Computer Graphics
Participation Exercise 3

Problem 1: Short Problems

A. Consider three surfaces having following three colors in RGB form.
C1=[1, 1, .5], C2=[1, 1, 1], C3=[1, 0, 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.

B. After taking CS248 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?

C. In class we've learned how a line in 2D can be defined by the equation $Ax + By + C = 0$. You'll soon learn that a plane in 3D can be defined by $Ax + By + Cz + D = 0$. In 2D the result of evaluating $f(x, y) = Ax + By + C$ told us which side of the line we were on. What do you think evaluating $f(x, y, z) = Ax + By + Cz + D$ tells us about the location of point (x, y, z) compared to a plane?

Problem 2: 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:

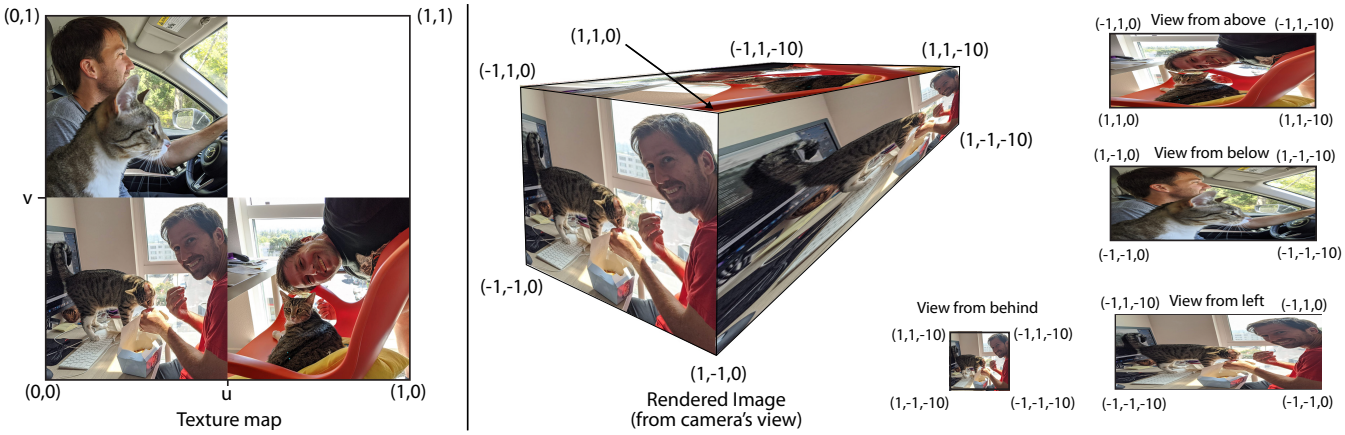
- Your rasterizer should process the triangles in the order they appear in the input array. No triangle sorting.
- Your rasterizer samples coverage at pixel centers.
- **HINT: Recall the perspective projection transform (`persp` in the code), followed by homogeneous divide (hint, hint) puts vertices 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









                }
        }
}
```

Problem 3: A Textured Cube

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).



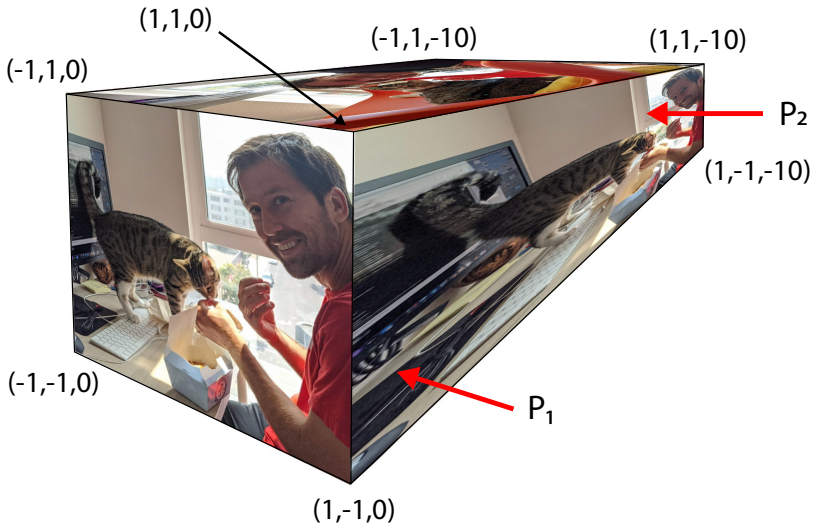
A. In class we talked about indexed mesh representations, where the vertices of each triangle are specified by an index into an array of 3D vertex positions. Below is a partial definition of an indexed mesh. **Please complete the specification of the mesh by filling in the missing indices for the triangles 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.