Full Name: ______________________________

Andrew ID: ______________________________

15-462/662, Spring 2020

Midterm Exam

March 4, 2020

Instructions:

- This exam is **closed book, closed notes, closed neighbor, closed cell phone, closed telepathy, closed internet**.
- You may however use a single 3in x 5in sticky note (or piece of paper) with any information you like written on both sides—-*except* for solutions to previous exams.
- If your work gets messy, please clearly indicate your final answer (by writing it in a box if possible).
- Partial credit will be awarded, *but only if we can understand your work!* So please try to write clearly, especially if you are uncertain about the final answer.

Problem	Your Score	Possible Points
1		25
2		20
3		25
4		30
Total		100

1 (25 points) Warm-up

(a) (6 points) Let $f : \mathbb{R}^4 \to \mathbb{R}^4$ be the transformation that takes a point $p = (x, y, z, 1)$ expressed in homogeneous coordinates to a point $f(p)$ that is the translation of p by a distance u, also expressed in homogeneous coordinates. Is f a:

- **A.** constant function
- **B.** linear function
- **C.** affine function
- **D.** neither linear nor affine
- **E.** translation cannot be expressed this way

(b) (6 points) Suppose you use a laser scanner to measure points on a person's face, connect these points up to form triangles, and then rasterize the triangles to generate a synthetic image of the face. Which of the following is a potential source of aliasing?

- **A.** the spatial resolution of the laser scanner
- **B.** the use of triangles to describe the surface of the face
- **C.** the fact that pixels are not perfect squares of color
- **D.** all of the above
- **E.** none of the above

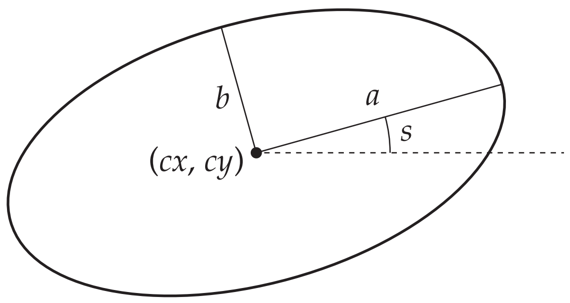
(c) (6 points) Suppose you've rasterized some polygons, and now have a supersample buffer containing both color and depth values. You now want to generate a MIP map of the depth buffer, so that you can do occlusion tests at a variety of resolutions. What operation is probably most natural when combining depth samples to form the next smallest level of the MIP map pyramid?

- **A.** sum
- **B.** average
- **C.** max
- **D.** min
- **E.** none of the above

(d) (7 points) Which of the following strategies will produce the correct image when rasterizing semi-transparent triangles in 3D?

- **A.** For each pixel, store a list of color and depth values. After all triangles have been drawn, sort color values from largest to smallest depth and apply the "over" operator in this order to get the final color value.
- **B.** Draw all the opaque triangles (in any order), using a depth buffer to keep track of the smallest depth at each pixel. Now sort the semi-transparent triangles from largest to smallest depth and rasterize them in this order, using the "over" operator to blend color values.
- **C.** First draw all the semi-transparent triangles from largest to smallest depth using the "over" operator, and without writing any depth values. Now enable depth testing and draw all the opaque triangles, again from largest to smallest depth.
- **D.** Both a and b will work.
- **E.** They will all work.

2 (20 points) Rasterizing an Ellipse



The standard rasterization pipeline is designed for drawing triangles really, really fast. Consider a different architecture where we instead want to draw little oriented ellipses, which will help us draw 3D point clouds from various points of view. We'll break our rasterizer down into two subroutines: one that does the inside-outside test for an ordinary circle, and another that transforms our query points in order to draw ellipses.

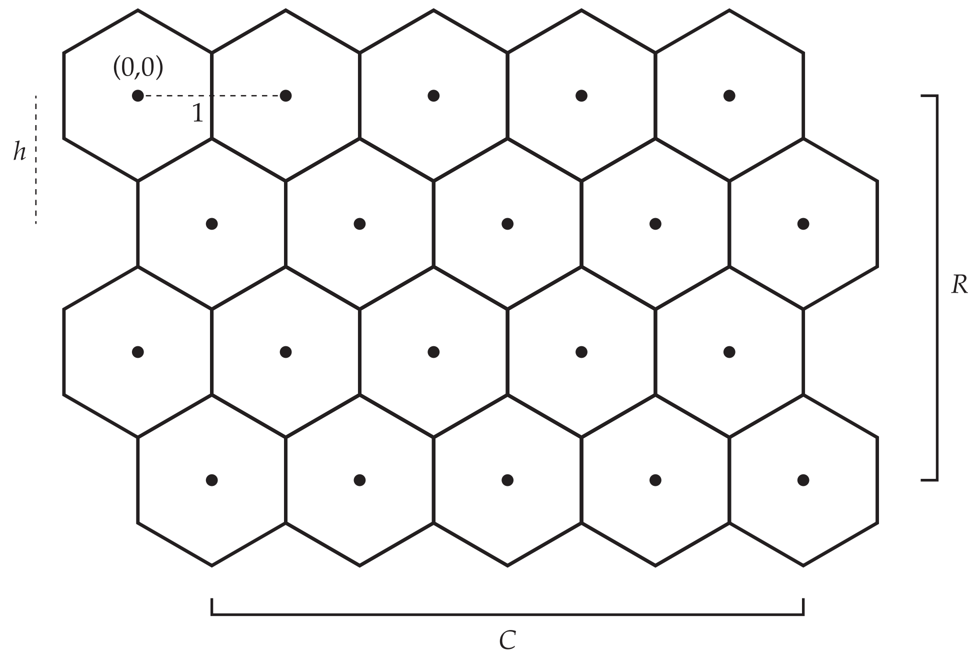
(a) (5 points) First, we just need to write a routine that returns `true` if the given point is inside a circle of the given radius, centered at the origin (or exactly on its boundary), and `false` if not. Fill out the function below; you may assume that standard library/math functions are available.

```
bool inside_circle(
   float r, // circle radius
   float x, y // query point ) {
   return x*x + y*y <= r*r;
}
```

(b) (15 points) Now suppose we want to use the routine above to draw ellipses. An ellipse is given by a center (cx, cy), two radii a, b, and a counter-clockwise angle of rotation. Write a routine that tests whether the given query point is inside an ellipse. This routine *must* call `inside_circle` (in a nontrivial way), but cannot make any modification to `inside_circle`.

```
bool inside_ellipse( float cx, float cy, // ellipse center
                     float a, b, // ellipse radii
                     float s, // ellipse rotation
                     float x, y // query point ) {
   // The ellipse can be obtained by scaling, then rotating,
   // then translating the circle.  Hence, to convert an
   // ellipse test into a circle test, we want to apply the
   // inverse of these operations, in the opposite order.

   float X, Y; // transformed test point

   // inverse translation
   x = x - cx;
   y = y - cy;

   // inverse rotation
   X = cos(-s)*x - sin(-s)*y;
   Y = sin(-s)*x + cos(-s)*y;

   // inverse scale
   X /= a;
   Y /= b;

   return inside_circle( 1, X, Y );
}
```

3 (25 points) A Bug-Eyed Display



In class we talked about the fact that some high-end displays use hexagonal pixels instead of square pixels in order to reduce aliasing.

(a) (8 points) Compared to an ordinary (square pixel) display with the same pixel density (i.e., pixels per square inch), what benefit will a hexagonal display provide?

A. better overall resolution

B. better color contrast

C. more efficient supersampling

D. [✓ less jagged line rendering]

E. none of the above

(b) (9 points) The high-level strategy for rasterization onto a hexagonal grid is very similar to rasterization onto a square grid: loop over all the pixels, and test if the pixel center is covered by the primitive. Assume you are given a hexagonal grid with R rows and C columns, the horizontal distance between columns is 1, and the horizontal distance between rows is h, and the upper-left pixel has a center at $(0, 0)$. Also assume that you are given a function `is_covered( float x, float y )` which returns 1 if the point (x, y) is covered by a primitive, and 0 otherwise. Write a routine that rasterizes this primitive onto a hexagonal grid, i.e., that sets the value of each pixel to 1 if it's covered, and 0 if it's not. You can assume that the `pixels` array has already been allocated, and stores grayscale values in row-major column-minor order (i.e., moving one column is an increment of 1).

```
void rasterize_hexagonal(
   int R, // number of rows
   int C, // number of columns
   float h, // vertical distance between rows
   float* pixels // array of pixel values
)
{
   for( int r = 0; r < R; r++ )
   for( int c = 0; c < C; c++ )
   {
      float x = c;
      float y = h*r;
      if( c%2 == 1 ) x += 0.5;
      pixels[ r*C + c ] = is_covered( x, y );
   }
}
```

(c) (8 points) Of course, to really build a rasterizer we need to implement the method `is_covered`. Suppose we've already implemented a version of this method that tests coverage for a triangle at the given point (x, y). To adapt this method to the hexagonal pixel grid we need to:

A. [✓ Do nothing. The method already works as-is.]

B. Modify the method to do half-plane tests for each of the 6 sides of the hexagon.

C. Modify the method to account for the vertical extent of the hexagonal pixels.

D. Modify the method to account for the horizontal shift of alternating rows.

E. Both c and d.

4 (30 points) Meshing Around

Let's consider a simplified halfedge data structure that stores only the `twin` and `next` pointers for each halfedge. To keep things simple, we'll assume we have a fixed number of halfedges, and the "pointers" are implemented as just indices into an array. For instance, if `twin[3] = 7` it means that halfedge 7 is the twin of halfedge 3. Likewise, if `next[9] = 2`, it means that halfedge 2 follows halfedge 9.

(a) (8 points) In class we talked about the fact that the connectivity of a halfedge mesh is valid as long as some basic invariants are preserved. Which of the invariants I1–I4 are sufficient to ensure that the two arrays (`twin` and `next`) describe a valid manifold polygon mesh?

I1. there are an even number of halfedges
I2. the twin of the twin of every halfedge is itself
I3. no halfedge is its own twin
I4. every halfedge points to some "next" halfedge

A. I1, I2, and I4 are enough

B. I1, I3, and I4 are enough

C. I2, I3, and I4 are enough

D. all four are needed

E. [✓ no combination of I1–14 is sufficient]

(b) (10 points) We can simplify our halfedge data structure even more by assigning consecutive indices to the two halfedges of each edge. For instance, the first edge has halfedges 0 and 1, the next edge has halfedges 2 and 3 and so forth. This way, we do not have to actually store the `twin` array: we know that `twin[i]` is always equal to `i+1` if `i` is even, and `i-1` if `i` is odd. Given this setup, which of the following halfedge meshes have valid *connectivity* (even if they might have crazy ugly geometry)?

M1:

`i`	0	1	2	3
`next[i]`	1	2	0	3

M2:

`i`	0	1	2	3	4	5
`next[i]`	5	0	3	1	5	2

M3:

`i`	0	1	2	3	4	5
`next[i]`	3	2	1	0	5	4

A. Only M1 and M2 are valid

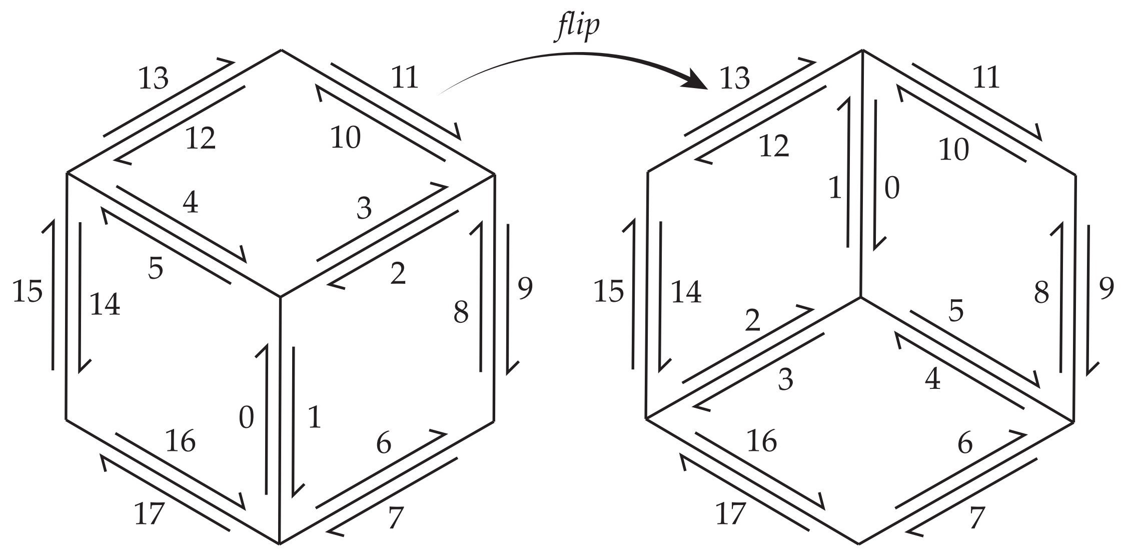
B. Only M2 and M3 are valid

C. [✓ Only M1 and M3 are valid]

D. All of them are valid

E. None of them are valid

(c) (12 points) One way to think about an edge flip in a triangle mesh is that in the original configuration you're looking at a tetrahedron from the front; in the new configuration you're looking at the same tetrahedron from the back. This perspective can be used to generalize the notion of a triangle-triangle edge flip to other types of polygons—for instance, if three quads meet at a vertex, we can flip them to three different quads meeting at the same vertex by viewing these configurations as the front and back of a cube. Implement this operation by making the minimal number of assignments to the `next` array, of the form `next[i] = j`. The initial state of `next` is given.



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
next[i]	5	6	1	10	3	14	8	17	2	7	12	9	4	11	16	13	0	15

```
next[1] = 12;
next[3] = 16;
next[5] = 8;
next[6] = 4;
next[8] = 10;
next[10] = 0;
next[12] = 14;
next[14] = 2;
next[16] = 6;
```