Full Name: _____

Andrew ID: _____

15-462/662, Fall 2017

Midterm Exam

October 18, 2017

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 3in 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		25
3		25
4		25
EC		20
Total		100

1 (25 points) Getting the Party Started: Miscellaneous Short Problems

A. (5 points) Consider a function $f : \mathbb{R}^m \to \mathbb{R}^n$. What does it mean for f to be a *linear function*? How can you represent the *affine* function $f(x, y) = ax + by + c$ as a linear function in homogeneous coordinates?

SOLUTION.

The function f is linear if two conditions hold:

- For every pair of vectors $x, y \in \mathbb{R}^m$, $f(x + y) = f(x) + f(y)$.
- For every vector $x \in \mathbb{R}^m$ and scalar $a \in \mathbb{R}$, $f(ax) = af(x)$.

The given affine function can be represented by assigning each point (x, y) homogeneous coordinates $(x, y, 1)$ and applying the linear function $\tilde{f}(x, y, w) := ax + by + cw$.

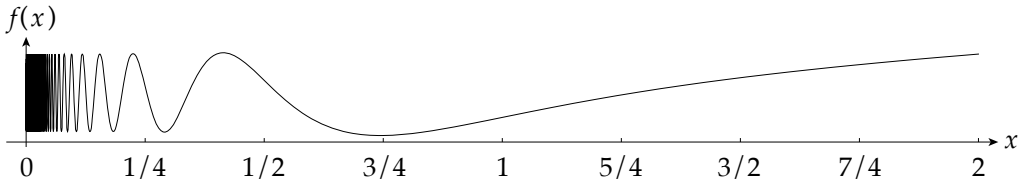
B. (5 points) In your own words, what is *aliasing?* Give two examples of aliasing that show up either in graphics algorithms or in real life.

SOLUTION.

Generally speaking, *aliasing* occurs whenever poor sampling gives a false impression of a signal, *i.e.*, it appears to be something that it is not. There are many possible examples:

- **Algorithmic Examples:** Sampling triangle coverage onto a pixel grid yields jagged (rather than smooth) edges; minification of a high-resolution texture (*i.e.*, sampling a very large texture over a small region) can yield "crawling" artifacts; using too few vertices to encode a piece of geometry can "round out" sharp features.
- **Real life examples:** A quickly spinning tire can appear to turn backwards (under artificial light); a wire basket can appear to exhibit circular patterns (called a *moiré effect*); the pitch of an ambulance siren will change depending on whether it is traveling toward or away from you (called the *doppler effect*).

C. (5 points) **Sampling the Topologist's Sine Curve**



Consider the curve above, which is given by the function

$$f(x) := 2x\cos(\pi/x) + \pi\sin(\pi/x) + 4.$$

For each quarter-interval, *roughly* how many samples would you guess are needed to get an accurate reconstruction of this function (*e.g.*, using linear interpolation)? Why?

Suppose now that rather than linear interpolation, you want to generate a high-quality 1D texture representing this function (*i.e.*, a single strip of texels, with one texel per quarter interval). Other than sampling, how could you obtain intensity values for these texels? **Bonus points:** give the exact value for the first texel in this image (as a fraction).

SOLUTION.

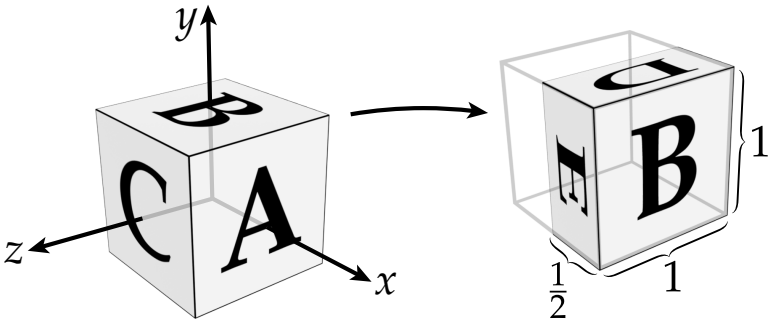
The Nyquist-Shannon sampling theorem says that for an ideal (sinc) filter we should take samples whose spacing is about half the shortest wavelength of any frequency in the signal, so we probably need a few more to get a good reconstruction with linear interpolation. Eyeballing the picture, it looks like most intervals don't contain many high-frequencies—it's probably enough to use about 2–4 samples for each interval past the first one. The first interval contains arbitrarily high frequencies, and no number of samples will be sufficient to accurately reconstruct the function. To get our texel intensities we could instead directly integrate the function. In particular, it has antiderivative

$$F(x) = x^2\cos(\pi/x) + 4x$$

which means its integral over the first interval is

$$F(1/4) - F(0) = (1/4)^2\cos(4\pi) + 1 - 0 = 1/17.$$

D. (5 points) Describe in words any sequence of transformations whose composition produces the overall transformation depicted in the figure below. The initial shape is an axis-aligned cube centered at the origin with all side lengths equal to 1; the gray outline on the right indicates the location of the original cube relative to the new cuboid.



SOLUTION.

One way to do it: rotate 90 degrees around the x-axis, then scale by $1/2$ along the z-axis, then rotate 90 degrees around the y-axis, then translate by $1/4$ along the x-axis.

E. (5 points) Suppose we have a surface with no boundary (*i.e.*, no holes) described by an explicit triangle mesh, and need to do many inside/outside tests. What is a simple implicit representation that makes these queries more efficient, and how might we perform the conversion from explicit to implicit? There are several reasonable solutions here; your solution should attempt to make the *queries themselves* as efficient as possible, *i.e.*, smallest amortized cost to do a single query. You do not need to write any code; just give a high-level outline.

SOLUTION.

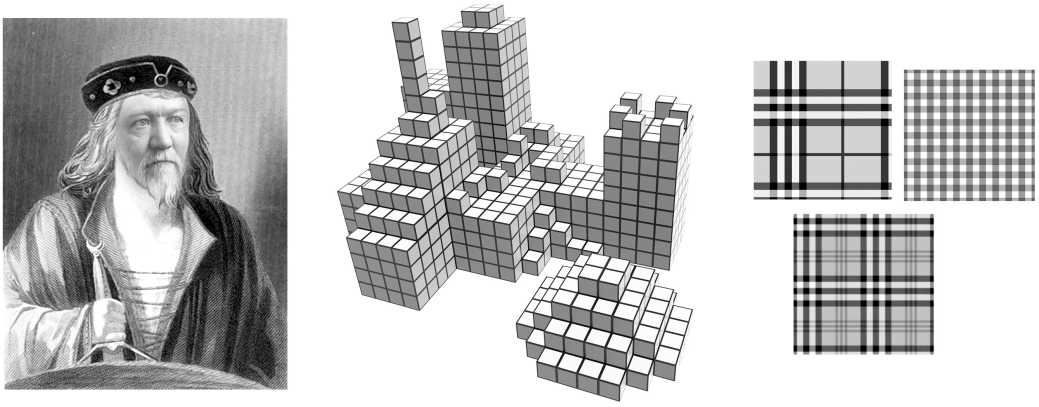
One solution: If we had the signed distance to the mesh, then we could do inside/outside tests in $O(1)$ time by simply checking whether the sign at the query point is negative or positive (respectively). To do this, we could first build a bounding volume hierarchy to accelerate closest point queries, which takes $O(n \log n)$ time. For each cell center c of a regular grid, we could then compute the closest point p on the mesh; the sign could be determined by taking the dot product of the surface normal with the vector from p to c. Subsequent distance queries will take amortized $O(1)$ time (just a lookup into the grid); query points outside the grid bounds will all evalute to "outside."

F. (5 pts — EXTRA CREDIT) Your solution to the previous question may not be exact for query points very close to the original surface, *i.e.*, it may classify points that are "inside" as "outside" and vice versa. How can you modify your solution to provide exact queries near the surface? How can you make this fallback efficient?

SOLUTION.

First, we mark grid cells that contain triangles from the input surface. If the query point is inside one of these cells, we do a ray-mesh intersection using a random ray. If the ray pierces the surface in an even number of points (including zero) then we're outside; if it pierces it in an odd number of points then we're inside. Using a BVH, the amortized cost of these queries is $O(\log n)$.

2 (25 points) TartanCraft



In the massively multiplayer online game of *TartanCraft*™, players create 3D worlds out of cubes textured with lovely plaid patterns. Each cube is determined by a center location (i, j, k), as well as a index specifying which pattern to use.

A. (5 points) Consider the difference between rendering this scene using a rasterizer and a ray tracer: specifically think about the cost of determining which cube is visible at each screen sample in a rasterizer, or along each camera ray in a ray tracer. Do you think determining occlusion using the Z-buffer algorithm or via ray tracing is a better solution in this scenario? Why? You can assume that in the case of a ray tracer, you have a prebuilt spatial acceleration hierarchy. Also assume that the geometry is really a large collection of cubes (including those "deep inside" objects), rather than just those on the outer boundary.

SOLUTION.

I would use a ray tracer because then I could benefit from an "early exit," significantly reducing the number of intersection tests required and making the amortized cost of one visibility test $O(\log n)$. With rasterization, I would waste time drawing many cubes that are not seen. (Sorting and drawing back-to-front might reduce the amount of rasterization work, but I would need to re-sort every time the view changes.)

B. (5 points) His Imperial Majesty Andrew Carnegie, Emperor of the Tartan Realm, wants a panoramic view of everything going on in the world of *TartanCraft* from a *fixed* viewpoint far above the ground. Suppose that cubes from all players are being streamed over the network to his machine, and he wishes to watch the world as it is being built. Cube data can arrive at any time, in any order. You can assume that cubes are created, but never destroyed. What rendering strategy might Emperor Carnegie use to render these millions of cubes: rasterization or ray tracing? Why? (Think again about how visibility will be determined.)

SOLUTION.

Since at any given moment we don't have all the scene geometry, it becomes harder (though not impossible!) to build a spatial acceleration hierarchy like a BVH or KD-tree. Also, building such a structure would require us to hold all the cubes in memory at once. Since the camera never moves, we can instead just rasterize into a depth buffer, and discard the geometry of each cube after we're done rendering it. In this case, the memory cost is just proportional to the size of the depth buffer itself.

C. (5 points) Suppose that in one of the scenarios above we chose to use ray tracing rather than rasterization. Given that we're always rendering cubes, what's a small optimization we can make that (slightly) speeds up our ray intersection tests?

SOLUTION.

Recall that when we intersect a ray $\mathbf{r}(t) := \mathbf{o} + t\mathbf{d}$ with a plane $\mathbf{n}^\mathsf{T}\mathbf{x} = c$, we get an intersection time of

$$t = \frac{c - \mathbf{n}^\mathsf{T}\mathbf{o}}{\mathbf{n}^\mathsf{T}\mathbf{d}}.$$

For cubes, the normal $\mathbf{n}$ is always a unit vector along the x, y, or z axis, so we can skip the dot product and simply write

$$t = \frac{c \pm \mathbf{o}_x}{\pm \mathbf{d}_x}$$

for faces in the yz plane, and so forth.

D. (5 points) Again suppose we're using ray tracing, rather than rasterization. What kind (or kinds) of spatial acceleration data structure(s) would you use for our little world made of cubes, and why?

SOLUTION.

Since the world is made of cubes, it is very tempting to use a regular grid. One reason for doing so, perhaps, is that there is no additional cost for ray intersection beyond traversal of the grid itself. However, we may still spend a huge amount of time traversing empty space. In this sense, the tradeoffs between different data structures don't change all that much: a camel walking across a vast desert plane might demand a bounding volume hierarchy, whereas a field of voxelized grass blades might be served well by a regular grid. Since we do know that everything is ultimately made of little blocks, perhaps the best of both worlds is an adaptive structure (like a kd-tree or BVH) with fairly dense regular grids in its leaf nodes.

E. (5 points) Finally, we have to gift-wrap our cubes with lovely plaid patterns (of the kind shown above). Given that we're rendering exclusively in plaid, what kind of texture magnification filter would you use to nicely preserve the appearance of these textures?

SOLUTION.

Rather than using standard bilinear filtering, or mip mapping, I would use a simple nearest neighbor filter (*a.k.a.*box filter). Since all the texture features are aligned with the horizontal/vertical pixel grid, we won't get aliasing. In fact, using something like bilinear filtering will blur out the edges of our plaid pattern, which we know should be sharp no matter how close we are to the cube.

3 (25 points) Validating Meshes

An important part of writing robust code is verifying the validity of the input. Suppose you are given an adjacency list representation of a triangle mesh and are asked to convert it to a halfedge mesh. In particular, you are provided with the data:

```
double pts[nVertices][3]; // coordinates (x,y,z) for each vertex
int tris[nTriangles][3]; // triangles as triples of 0-based indices into pts
```

A. (6 points) What must be true about the `pts` list in order to convert this data into a valid halfedge mesh?

SOLUTION.

Nothing, except that the number of vertices must be at least as big as the largest index in the `tris` list (plus one, due to 0-based indexing). Otherwise, a halfedge mesh can store vertices in any location.

B. (6 points) Outline a simple strategy for simultaneously verifying that (i) no edge in the given mesh is nonmanifold, and (ii) all triangles have a consistent orientation (or "winding"). What is the cost of your strategy, assuming you have an efficient data structure that can map a *ordered* pair of integers to a stored integer value in $O(1)$ time (*i.e.*, a hash table / associative array)? You do not have to write any code.

SOLUTION.

Iterate over the triangles; for each triple (i, j, k) increment the values associated with the three ordered edges (i, j), (j, k), and (k, i). If any of these values become larger than 1, the mesh has either two inconsistently oriented triangles (since the shared edge has the same orientation for both triangles) or a nonmanifold edge (since if there are more than two triangles with the same edge, there must be at least two with the same edge orientation).

C. (7 points) Suppose now that your code is running on the deep space probe *New Horizons II* which has been taking 3D scans of objects floating around in the Kuiper belt. You already have a nicely scanned and reconstructed halfedge mesh, stored in the data structure below. However, in outer space it is quite common for high-energy *cosmic rays* to hit the RAM of your machine, randomly invalidating bits. Write a routine that does an exhaustive consistency check on your halfedge mesh—a mesh that passes this test should describe a valid polygonal (though not necessarily triangular) surface. You may assume that a preliminary check has already been run to ensure that all pointers point to some valid element of the mesh, *i.e.*, not NULL or some totally random address. (Note that you do *not* need to check for non-manifoldness, since a halfedge mesh cannot represent a nonmanifold mesh!)

```
struct HalfedgeMesh
{
   vector<Halfedge> halfedges;
   vector<Vertex>   vertices;
   vector<Edge>     edges;
   vector<Face>     faces;
};
```

```
struct Halfedge
{
   Halfedge* t; // twin
   Halfedge* n; // next
   Vertex*   v; // vertex
   Edge*     e; // edge
   Face*     f; // face
};
```

```
struct Vertex
{
   Halfedge* h;
   Vector3D coords;
};
```

```
struct Edge
{
   Halfedge* h;
}
```

```
struct Face
{
   Halfedge* h;
}
```

```
bool isValid( HalfedgeMesh* mesh )
{
```

SOLUTION.

```
for( Halfedge h : mesh->halfedges ) {
   // first just make sure that the
   // twin of the twin is itself
   if( h->t->t != &h ) return false;

   HalfedgeIter i;
   bool isValid;

   // check that one of the outgoing
   // halfedges of h's vertex is h itself
   isValid = false;
   i = h.v->h;
   do {
      i = i->t->n;
      if( i == &h ) isValid = true;
   }
   while( hv != h.v->h );
   if( !isValid ) return false;

   // check that one of the two halfedges
   // of h's edge is equal to h
   if( h.e->h != h && h.e->h->t != h ) return false;

   // check that one of the halfedges
   // around h's face is equal to h
   isValid = false;
   i = h.f->h;
   do {
      i = i->n;
      if( i == &h ) isValid = true;
   }
   while( hv != h.f->h );
   if( !isValid ) return false;
}

// for all other mesh elements, just check that their halfedge
// points back to themselves
for( Vertex v : mesh->vertices ) { if( v.h->v != &v ) return false; }
for(   Edge e : mesh->edges    ) { if( e.h->e != &e ) return false; }
for(   Face f : mesh->edges    ) { if( f.h->f != &f ) return false; }

return true;
```

}

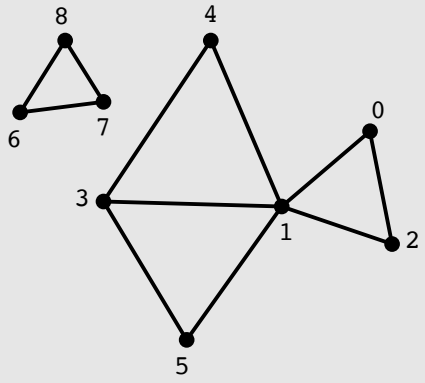
D. (6 points) Sending data across outer space is expensive. Below you are given the bare minimum encoding for the connectivity of two different meshes: one in adjacency list format, the other as the list of "next" pointers for the halfedges in a halfedge mesh. The twin pointers are encoded by a simple rule: the twin of a halfedge with *even* index i is $i + 1$; the twin of a halfedge with *odd* index j is $j - 1$ (so that consecutive even and odd indices are paired). Draw a picture of each mesh, and indicate whether each one is manifold (and why, in the case of a nonmanifold mesh). How many polygons does each mesh encode? How much storage is needed for each mesh?

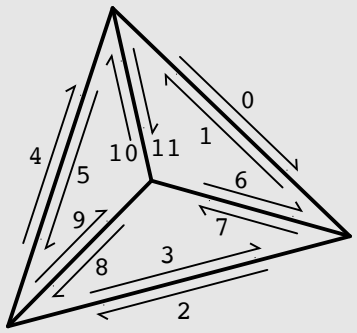
```
0 1 2
3 1 4
1 3 5
6 7 8
```

```
    NEXT
0  ---> 2
1  ---> 11
2  ---> 4
3  ---> 7
4  ---> 0
5  ---> 9
6  ---> 1
7  ---> 8
8  ---> 3
9  ---> 10
10 ---> 5
11 ---> 6
```

SOLUTION.

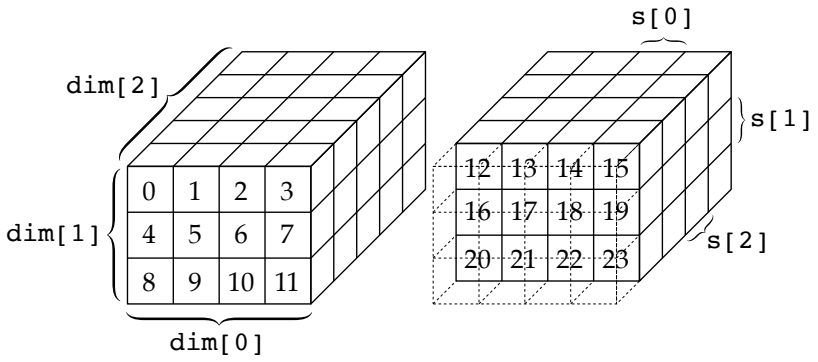
The mesh described by the adjacency list is nonmanifold at vertex 1; the halfedge mesh is manifold. Both meshes encode four triangles using 12 integers.





4 (25 points) Generalized Sampling

Uh oh. Your beloved pet terrier (named "Scotty", oddly enough) ate your most recent quiz for 15-462. To recover the solution, you take Scotty to get a CT scan, which lets you peek inside your trusted companion.



A. (5 points) Your first task is to implement a routine that simply looks up the sample value stored at a given integer index. In particular, the data is given as a linear array `data` of scalar (*i.e.*, grayscale) values. The main challenge here is translating a requested 3D index (i, j, k) into a single index I corresponding to the location of the requested sample value. The dimensions of the data are specified by the array `dim[3]`, which gives the number of samples along the x, y, and z axes, respectively. The data itself is packed into the linear array in *x-minor order*. In other words, incrementing by 1 moves along the x-axis; incrementing by `dim[0]` moves along the y-axis, and so forth (see the figure above for an example). For indices outside the array dimensions, you should just return zero. All indices start at zero (not one).

```
double lookup3D( double* data, // sample values
                 int dim[3],   // number of samples along each axis
                 int i, int j, int k ) // requested sample index
{
```

SOLUTION.

```
// first make sure index is in bounds
if( i < 0 || j < 0 || k < 0 ||
i > dim[0]-1 || j > dim[1]-1 || k > dim[2]-1 )
return 0;

// compute index of requested value and return it
int I = i + dim[0]*j + dim[0]*dim[1]*k;
return data[ I ];
```

```
}
```

B. (6 pts) Since the CT scan is fairly low-resolution, you will need to upsample it using trilinear sampling. However, due to the nature of the hardware that measured this signal, the data is sampled at equal-size intervals in x and y, but at much coarser intervals in z. In other words, the distance between samples is **NOT** the same along all three axes—these distances are provided in the array `s[3]`. Your routine should take a sample location (x, y, z) and return the interpolated value at this point. You must make use of the routine `lookup3D` from the previous part (whether or not you completed this question). Since this routine returns zero for values outside the domain, you do not need to worry too much about special handling of values near the boundary. The center of grid cell $(0, 0, 0)$ is at the origin.

```
double sampleTrilinear( double* data, // sample values
                        int dim[3],   // number of samples along each axis
                        int s[3],     // spacing between samples along each axis
                        double p[3] ) // coordinates of the sample point
{
```

SOLUTION.

```
// scale sample coordinates by grid spacing so
// that 1 unit = 1 cell width for all three axes
for( int n = 0; n < 3; n++ )
{
   p[n] /= s[n];
}

// compute fractional part of the normalized sample
// coordinate, which will be used to interpolate
// between adjacent sample values
double a = p[0] - i0;
double b = p[1] - j0;
double c = p[2] - k0;

// compute the starting index of the 2x2x2 grid of
// sample values that will be used to interpolate
int i0 = floor( p[0] );
int j0 = floor( p[1] );
int k0 = floor( p[2] );

// grab the sample values
double v[2][2][2];
for( int i = 0; i < 2; i++ )
for( int j = 0; j < 2; j++ )
for( int k = 0; k < 2; k++ )
{
   double v[i][j][k] = lookup3D( data, dim, i0+i, j0+j, k0+k );
}

// return the trilinearly interpolated value
return
(1-c) * (
   (1-b) * ( (1-a)*v[0][0][0] + a*v[1][0][0] ) +
      b  * ( (1-a)*v[0][1][0] + a*v[1][1][0] )
) +
c * (
   (1-b) * ( (1-a)*v[0][0][1] + a*v[1][0][1] ) +
      b  * ( (1-a)*v[0][1][1] + a*v[1][1][1] )
) ;
```

}

C. (4 pts) Based on the current treatment of values near the boundary, what will the edges of the interpolated signal look like? How might the interpolation strategy be modified to avoid this behavior?

SOLUTION.

It will get darker near the edges, since samples near the boundary will get interpolated with zero. We could avoid this behavior by (for instance) duplicating the values along the boundary (rather than using zero), so that the interpolated values are closer to the boundary values.

D. (4 pts) Suppose that instead of a static 3D scan you now have a movie showing the CT scan over time, stored as a 4D grid (3D + time). How many samples would you need to evaluate *quadrilinear* sampling of this data, and what would your basic strategy be? (You do not need to write any code here.)

SOLUTION.

We will now need two samples along each of four axes, or $2^4 = 16$ total samples. The basic strategy would be to simply linearly interpolate the trilinearly sampled values from two consecutive samples in time.

E. (6 pts) Implement a method which, given a query point p, interpolates the three values u, v, w at the corners a, b, c (respectively) of a triangle in 2D using barycentric interpolation. If the query point is outside the triangle, you should return zero. All coordinates are specified via a `Vector2D`, which has coordinates `p.x` and `p.y`, as well as all the usual vector operations (addition, subtraction, dot product, cross product, *etc.*). You may use whatever names you like for these methods, as long as the meaning is clear.

```
double sampleBarycentric( Vector2D p, // sample point
                          Vector2D a, Vector2D b, Vector2D c, // triangle corners
                          double u, double v, double w ) // values at corners
{
```

SOLUTION.

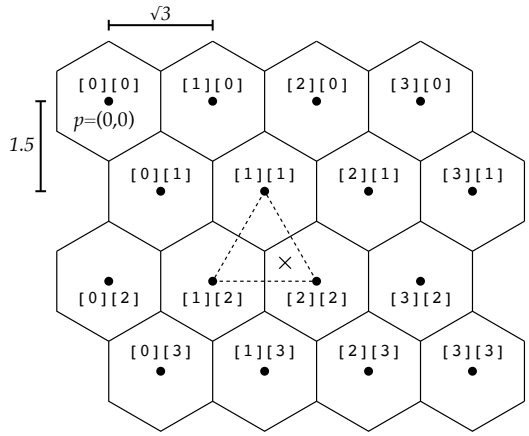
```
// there are many ways to compute barycentric
// coordinates; one is to compute the ratio of
// each subtriangle area over the total area
double A  = cross( b-a, c-a ) / 2;
double Aa = cross( p-b, p-c ) / 2;
double Ab = cross( p-c, p-a ) / 2;
double Ac = cross( p-a, p-b ) / 2;
double ta = Aa/A;
double tb = Ab/A;
double tc = Ac/A;

if( ta < 0 || tb < 0 || tc < 0 ||
    ta > 1 || tb > 1 || tc > 1 )
{
   return 0;
}

// Interpolate values using these coordinates
return ta*u + tb*v + tc*w;
```

```
}
```

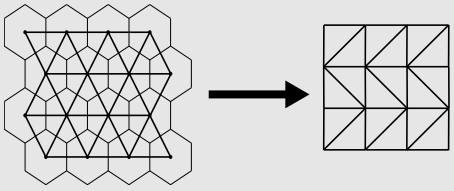
EXTRA CREDIT: *This final question is challenging and should be completed for* extra credit only *(you are not required to answer it in order to get full points. You may wish to try it after completing the others.*



(15 pts) Once the homework was located inside your dog, the doctor took a high-resolution image of your quiz solution using a special, high-res 2D camera. To reduce directional aliasing, this 2D sensor has pixels arranged in a regular *hexagonal* grid rather than a rectangular grid. The grid is comprised of equilateral triangles with unit side length, which means the horizontal spacing between sample centers is $\sqrt{3}$ and the vertical spacing is 1.5; the center of grid cell $(0, 0)$ is at the origin. To reconstruct the image, you must now interpolate values on this grid. To do so, you will perform barycentric interpolation using the three sample values closest to a given point (marked by an "X" in the image above). This time, you may assume that a method `lookup2DHex( data, i, j )` has already been defined, which returns the value of `data` at index (i, j)—cells are indexed as indicated in the example above. You do not need to worry about the width or height of the grid; values outside the valid range will just return zero. You may use the method `sampleBarycentric` defined in the previous question, though you may find it easier to just compute the interpolation directly.

```
double sampleHexGrid( double** data, // sample values
                      int nCols,     // number of horizontal samples
                      int nRows,     // number of vertical samples
                      Vector2D p )   // sample point
{
```

SOLUTION.



```
// first, just normalize the sample point coordinates
// by the grid spacing, so that the horizontal and
// vertical distance between samples is effectively 1
x /= sqrt(3);
y /= 1.5;

// do the easy part first: get the index of the closest
// row above this sample, and the fractional part t
// between this row and the next one
int j0 = floor(y);
double t = y - j0;

// now make life simpler by displacing x according to the
// fractional y value, effectively making the grid square
if( j0%2 == 0 )
   x -= .5*t;
else
   x -= .5*(1-t);
int i0 = floor(x);
double s = x - i0;

// get the four samples on the regular grid
double v00 = lookup2DHex( data, i0+0, j0+0 );
double v01 = lookup2DHex( data, i0+0, j0+1 );
double v10 = lookup2DHex( data, i0+0, j1+0 );
double v11 = lookup2DHex( data, i0+0, j1+1 );

// apply barycentric interpolation, depending on
// which type of triangle the sample point is in
if( j0%2 == 0 )
{
   if( s+t <= 1 ) return (1-s-t)*v00 + s*v10 + t*v01;
   else           return (1-(1-s)-(1-t))*v11 + (1-s)*v01 + (1-t)*v10;
}
else
{
   if( s <= t ) return (1-s-(1-t))*v01 + s*v11 + (1 - t)*v00;
   else         return (1-(1-s)-t)*v10 + t*v11 + (1 - s)*v00;
}
```

}