Full Name: ______________________________

Andrew ID: ______________________________

15-462/662, Fall 2018

Midterm Exam

October 17, 2018

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		30
2		25
3		33
4		12
EC		5
Total		100

1 (30 points) Sampling and Aliasing

A. (5 points) In your own words, what is aliasing? How does it relate to sampling and reconstruction?

SOLUTION.

In general, *aliasing* is a catch-all term describing any situation where the way a signal is perceived or interpreted provides a false impression of what that signal really looks like. Aliasing can occur both due to undersampling (*e.g.*, sampling far below the Nyquist frequency), and reconstruction (*e.g.*, using a box filter to reconstruct a smooth function).

B. (5 points) Give an example of aliasing found in computer graphics. (This example must be different from the "jagged edges" that occur in line rasterization.)

SOLUTION.

One (of many) examples of aliasing in computer graphics is *texture magnification*: when we're sampling the texture at a higher rate than the finest level of the MIP map hierarchy, we're very likely missing out on high-frequency features, causing the resulting image to look "blurry." (Consider for instance using a tiny 32×32 image to texture map a whole zebra—you'll definitely miss some stripes, and just end up with a blurry mess!)

C. (5 points) Give an example of aliasing found *outside* of computer graphics/computer science. (This example must be different from the "wagon wheel" effect described in class.)

SOLUTION.

One (of many) examples of aliasing outside computer graphics is the *moiré* pattern seen when two regular patterns cross, such as the front and back of a wire wastebasket. We perceive bands of darker/lighter stripes, even though they're not really there. This phenomenon has to do with human perception, and nothing to do with computers or computer graphics.

D. (5 points) You've been hired to work on the next generation of video distribution software for YouTube. The iPhone XII has just been released, and by default records video at 480Hz. However, having taken 15-462 you know that most video will look good if played back at 60Hz—in fact, the new YouTube player you're designing *only* supports 60Hz playback, independent of the original source rate. Taking both bandwidth and amortized computational effort into account, what is a more appropriate strategy for the software you're writing: *supersampling* or *prefiltering?* Why?

> **SOLUTION.**
>
> I would definitely use prefiltering; specifically, I would just downsample all the users' videos to 60Hz before uploading them to the YouTube servers, by doing some kind of time-averaging of video frames. Doing so will save on bandwidth (less data to transmit to viewers over the network), and also will save dramatically on overall computational cost: by downsampling the video ahead of time, each viewer won't have to perform this downsampling on their own machine.

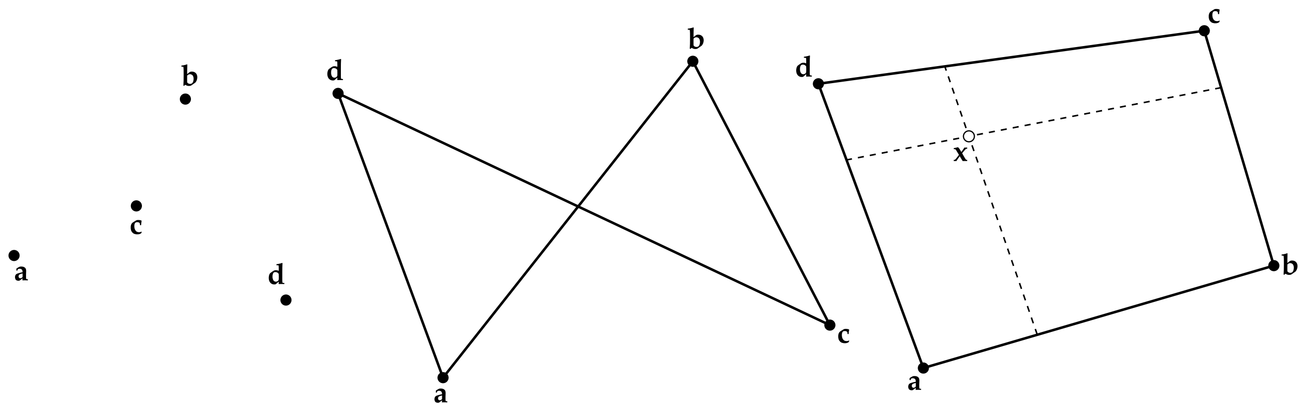
E. (5 points) In the YouTube scenario described in the previous question, why isn't *subsampling* a good strategy? What kinds of videos will look particularly bad if you use subsampling?

> **SOLUTION.**
>
> Subsampling, *i.e.*, just dropping 7 out of every 8 frames, is a bad idea because it will lead to aliasing. This effect will be especially bad when there is fast motion (*e.g.*, a video of an extreme snowboarder being chased by a snow leopard).

F. (5 points) Since you were hired at YouTube, personal assistants like Alexa, Siri, and Cortana have been adding robotic legs and low-quality surveillance cameras. Scary. Since these cameras record video at only 24Hz, how could you generate 60Hz video that could be played back in the YouTube video player? Try to be specific about your strategy. (Deep learning is not allowed.)

> **SOLUTION.**
>
> I would upsample the video to 60Hz by using linear interpolation. In particular, for a given time T where I want to generate a frame, I would find the nearest frames at times $t_0 < T < t_1$ and compute the value $t := (T - t_0)/(t_1 - t_0)$. I would then use this t value to linearly interpolate between pixels in the two frames.

2 (25 points) Quadsterization



quadVidia has decided that triangles are no longer cool, and quads are the next big thing[1]. Your graphics consulting company, PinkFongGFX, has been hired to develop the rasterization algorithms for quadVidia's next-generation architecture. The basic setup is that you're given four vertices $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{R}^2$, in any order. quadVidia has asked that your algorithms correctly rasterize any *convex* quad, but may cull (reject) nonconvex quads, which are not considered to be valid input.

A. (5 points) To simplify assumptions at later stages of the pipeline, the first thing you want to implement is the culling stage. How would you test that the four given points describe a convex quad?

> **SOLUTION.**
>
> I would simply check if any vertex is inside the triangle made by the other three, by doing three half-plane tests for each triangle. If so, these points don't describe a convex quad, and can be culled.

B. (5 points) Now that you know the points define a convex quad, how would you test whether they're in a consistent counter-clockwise order? If they're not, how would you fix the ordering so that they are?

> **SOLUTION.**
>
> I'd first verify that triangle abc has counter-clockwise orientation, by checking that the sign of the cross product $(b - a) \times (c - a)$ is positive. If not, I would swap b and c. I would then repeat this procedure for triangle acd.

[1]Historical aside: the original NVIDIA graphics chip, the NV1, actually *did* rasterize quads... nobody bought it!

C. (5 points) Now that you know you have a convex quad with points $\mathbf{a}, \mathbf{b}, \mathbf{c}$ and $\mathbf{d}$ in counter-clockwise order, how would you test that a given sample point x is inside the quad?

> **SOLUTION.**
>
> I would perform four half-plane tests, one for each edge, and return true if and only if all four tests pass. In particular, for an edge with vertices $\mathbf{p}, \mathbf{q}$, I would compute the vector $\mathbf{u} := (\mathbf{q} - \mathbf{p})/|\mathbf{q} - \mathbf{p}|$, rotate it by 90 degrees in the counter-clockwise direction (by swapping its two components and negating the first one), and then check whether $\langle \mathbf{u}, \mathbf{x} \rangle > \langle \mathbf{u}, \mathbf{p} \rangle$ to see if $\mathbf{x}$ is inside this half plane.

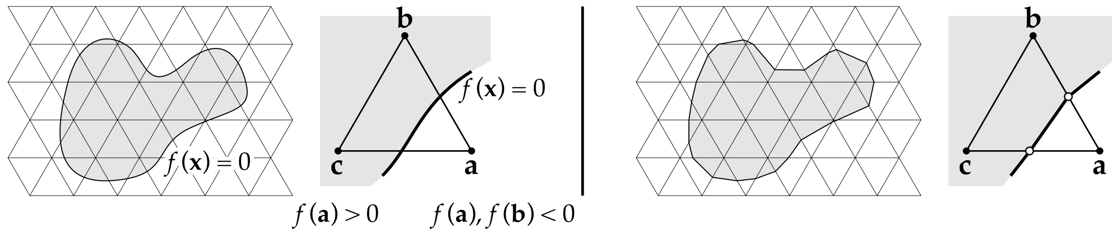
D. (5 points) Finally, quadVidia wants your algorithm to provide coordinates for interpolating vertex attributes like colors and texture coordinates. Why might splitting the quad into two triangles and using barycentric interpolation in each triangle be a bad idea?

> **SOLUTION.**
>
> You'll get noticeable artifacts along the diagonal, since you're using two different linear functions (with different slopes) to interpolate; this is especially problematic, since there's not one "correct" diagonal to split along. The effect will look like the errors we saw in barycentric interpolation when you don't correctly account for perspective.

E. (5 points) Since barycentric coordinates are no good, you've decided to use *bilinear coordinates*, *i.e.*, the coordinates such that $\mathbf{x}$ is a bilinear interpolation of the four corner vertex coordinates. Write down the system you would have to solve in order to obtain these coordinates. How many equations are there, and how many unknowns? ***Extra credit (5 points):*** *solve it!*

> **SOLUTION.**
>
> I would simply need to look for two values $s, t \in \mathbb{R}$ that satisfy the equation
>
> $$(1-s)((1-t)\mathbf{a} + t\mathbf{b}) + s((1-t)\mathbf{c} + t\mathbf{d}) = \mathbf{x}.$$
>
> Since each of the corners is a point in $\mathbb{R}^2$, this equation actually describes two *real* equations in two unknowns. However, these equations are not linear; they are *bilinear*, but can still be solved (as described below).
> **Extra credit:** To solve for the bilinear coordinates, I would rearrange the equation above as
>
> $$\mathbf{a} - \mathbf{x} + t(\mathbf{b} - \mathbf{a}) + s(\mathbf{d} - \mathbf{a}) + st(\mathbf{a} - \mathbf{b} + \mathbf{c} - \mathbf{d}) = 0$$
>
> to isolate the powers of s and t. Letting $\mathbf{P} := \mathbf{a} - \mathbf{x}$, $\mathbf{Q} := \mathbf{b} - \mathbf{a}$, $\mathbf{R} := \mathbf{d} - \mathbf{a}$, and $\mathbf{S} := \mathbf{a} - \mathbf{b} + \mathbf{c} - \mathbf{d}$, we get $\mathbf{P} + t\mathbf{Q} + s\mathbf{R} + st\mathbf{S} = 0$. Letting x_1 and x_2 denote the first and second component of any vector $\mathbf{x}$, I would then solve the first scalar equation for s:
>
> $$s = (-P_1 - Q_1 t)/(R_1 + S_1 t).$$
>
> Plugging this value into the second scalar equation yields
>
> $$(P_2R_1 - P_1R_2 + (Q_2R_1 - Q_1R_2 + P_2S_1 - P_1S_2)t + (Q_2S_1 - Q_1S_2)t^2)/(R_1 + S_1 t) = 0,$$
>
> and if the denominator is nonzero (which it should be for any point inside the quad) then the numerator is a quadratic equation which can be solved for t. This t value can then be plugged back in to the expression for s.

3 (33 points) Converting Geometry



In class we spent a lot of time talking about various explicit and implicit representations of geometry. Of course, no one representation will work well for all algorithms, and so in practice you'll have to convert between the two. Here we'll run through a few exercises where we translate between representations.

A. (5 points) Suppose we're given a curve represented as the zero set of a function $f(\mathbf{x})$, and want to convert this zero set into a polygon. Are each of these representations explicit or implicit, and why?

> **SOLUTION.**
>
> The set $f(\mathbf{x}) = 0$ is an *implicit* representation of shape, since we don't have an explicit list of points that belong to the shape; just a "test" we can run to check if a given point is included. The polygon, on the other hand, is an *explicit* representation, since we have an explicit list of points (the vertices), and an explicit map taking any parameter value $t \in [0, 1]$ along an edge to a point on the polygon (via linear interpolation).

B. (6 points) One algorithm for approximating a zero level set by a polygon is called *"marching triangles."* The idea is to lay down a grid of triangles over space, and evaluate the function f at each vertex. If the sign of f is different on two neighboring vertices $\mathbf{a}$ and $\mathbf{b}$ (*e.g.*, positive at one end; negative at the other end), then the zero set must cross the edge somewhere between $\mathbf{a}$ and $\mathbf{b}$. In this situation, the marching triangles algorithm inserts a crossing at the location where the linearly interpolated value of f is equal to zero. Write a simple routine that computes the crossing location, given the locations of the two endpoints, and the values of f at the two endpoints. This method should store the solution in $\mathbf{p}$, and return false if and only if there is no crossing along the edge.

```
bool crossing( Vector2D a, Vector 2D b,
                   double fa, double fb,
                   Vector2D& p )
{
```

> **SOLUTION.**
>
> ```
> double t = -fa/(fb-fa); // t value where interpolated function equals zero
> if( t < 0 || t > 1 ) return false;
> p = (1-t)*a + t*b; // interpolate endpoints
> ```

```
}
```

C. (6 points) Now that we know how to find the crossing point for a single edge, we just have to connect up crossings within each triangle. (Think: how many different ways can this happen, given that we're only sampling f at the vertices?) Implement a routine that takes as input the three vertex coordinates $\mathbf{a}, \mathbf{b}, \mathbf{c}$, and the three function values $f(\mathbf{a}), f(\mathbf{b}), f(\mathbf{c})$, and appends any segments passing through this triangle to a list of pairs. You can initialize a pair by writing `P = pair(p,q)`, and you can append a pair `P` to a list by writing `list.append(P)`. Note that your routine should call the `crossing()` method (even if you did not implement it).

```
bool marchTriangle( Vector2D a, Vector2D b, Vector2D c,
                    double fa, double fb, double fc,
                    list<pair>& segments )
{
```

SOLUTION.

```
Vector p, q;
if( fa<0 && fb<0 && fc<0 ) return;
if( fa>0 && fb>0 && fc>0 ) return;
if( fa*fb>0 ) {
   crossing( c, a, fc, fa, p );
   crossing( c, b, fc, fb, q );
}
if( fb*fc>0 ) {
   crossing( a, b, fa, fb, p );
   crossing( a, c, fa, fc, q );
}
if( fc*fa>0 ) {
   crossing( b, c, fb, fc, p );
   crossing( b, a, fb, fa, q );
}
segments.append( pair( p, q ) );
```

```
}
```

D. (5 points) If we run `marchTriangle` on every triangle in our grid, we get a list of segments that make up our polygon. At this point, however, we really just have "segment soup": we don't know which segments are connected to which other segments. What data structure might you use to quickly locate segments with equal endpoints? Why?

SOLUTION.

Any spatial data structure would help with this task. Since we know our crossing points are regularly sampled, a regular grid could be a good choice for this task. If our zero level set occupies a very small or irregular region of the overall grid, it may be better to use something spatially adaptive (*e.g.*, BVH, kd-tree, or octree). In practice, of course, there's a much simpler solution: since there's at most one crossing per edge, we can assign a unique index to each edge with a crossing, and then store segments as pairs of indices rather than pairs of crossing locations; this way we don't have to "glue together" segments after the fact.

E. (5 points) Suppose we want to go the other direction: we have a polygon, and want to compute a signed distance value $f(\mathbf{a}$ for every vertex $\mathbf{a}$ of a regular triangular grid. What data structure would you use to accelerate computation of these values? If there are m total vertices on the grid, and n total vertices on the polygon, what would be the total cost of computing the signed distance function (using your chosen data structure)? You should assume that m is *much* larger than n, and give the amortized asymptotic cost.

> **SOLUTION.**
>
> Again, any spatial data structure would help—here something spatially adaptive like a BVH or kd-tree might make most sense, since the polygon can in general be irregularly sampled and have irregular shape. The cost of building one of these data structures would depend only on the size of the polygon, asymptotically $O(n \log n)$. Assuming each leaf node contains a constant number of segments, the cost of each subsequent distance query is $O(\log n)$. Since there are m nodes on the grid the overall amortized cost of distance evaluation would be $O(m \log n)$. (Since m is much larger than n, the cost of building the tree doesn't contribute to the amortized cost.)

F. (6 points) Suppose we start with a function $f(\mathbf{x}) = 0$, run marching triangles to get a polygon, and then sample the distance to this polygon back onto the original grid to get values $f'(\mathbf{a})$ at each vertex $\mathbf{a}$. Are these distance values the same as the original ones? In other words, does $f'(\mathbf{a}) = f(\mathbf{a})$ for all $\mathbf{a}$? What if we run through this whole loop one more time, *i.e.*, we run marching triangles on the function f', then compute the distance f'' to the resulting polygon. Does $f'(\mathbf{a}) = f''(\mathbf{a})$? If there is any loss of information, at which step(s) does it occur, and what name would you give to this phenomenon?

> **SOLUTION.**
>
> The values change both times, *i.e.*, $f \neq f'$, and $f' \neq f''$. In the first loop, the polygon generated by marching triangles will not (in general) exactly match the zero level set of the distance function f. In the second loop, the polygon we get exactly represents the zero set of the function f'; however, the distance to this polygon is not a piecewise linear function (consider what it looks like in the vicinity of a convex vertex), hence the sampled function f'' will not be the same as f'. The name I would give to this phenomenon is *aliasing*: we are getting a false impression of what the original signal looks like (*i.e.*, the original function f) due to issues of sampling and reconstruction.

4 (12 points) Transformations

A. (6 points) Of the five sequences of 3D transformations listed below, two pairs are equivalent. Which pairs are the same, and which one is different from the other four?

(a) Rotate by 90 degrees around the z-axis, then translate by a distance 2 along the y-axis, then scale by a factor 2.

(b) Rotate by 90 degrees around the z-axis, then scale by a factor 2, then translate by a distance -2 along the x-axis.

(c) Translate by a distance 2 along the y axis, then rotate by 90 degrees around the z-axis, then scale by a factor 2.

(d) Rotate by -270 degrees around the z-axis, then translate by a distance 2 along the x-axis, then scale by a factor 2, then translate by a distance -8 along the x-axis.

(e) Scale by a factor 2, then rotate by 90 degrees around the z-axis, then translate by a distance 4 along the y-axis.

SOLUTION.

Transformations (a) and (e) are the same; transformations (c) and (d) are the same; transformation (b) is different from the rest.

B. (6 points) Write down a 4×4 matrix that encodes transformation (a) from the previous question in homogeneous coordinates. Apply it to the three points $\mathbf{p} := (0,0,0)$, $\mathbf{q} = (1,0,0)$, $\mathbf{r} := (0,1,0)$, and to the unit normal N of the triangle made by points $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{r}$. You should assume this normal points in the same direction as the vector $(\mathbf{q}-\mathbf{p}) \times (\mathbf{r}-\mathbf{p})$.

SOLUTION.

Note that rotating by 90 degrees around the z-axis, then translating by a distance 2 along the y-axis, then scaling by a factor 2 is the same as first rotating by 90-degrees around the z-axis, then scaling by 2, then translating by a distance 4 along the y-axis. Therefore, we can write the whole transformation as a single matrix

$$\begin{bmatrix} 2\cos 90^\circ & 2\sin 90^\circ & 0 & 0 \\ -2\sin 90^\circ & 2\cos 90^\circ & 0 & 4 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 2 & 0 & 0 \\ -2 & 0 & 0 & 4 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Notice that what we've done here is put the usual 2D rotation matrix in the upper-left 2x2 block, which keeps the z coordinate fixed while rotating the x and y coordinates; the factor 2 in front of cos and sin applies the desired scaling, and the rightmost column applies a translation by 4 in the y direction.

To apply this transformation to our data, we just give the points a homogeneous coordinate 1 and the normal vector, $N = (0,0,1)$, a homogeneous coordinate of zero since it should be rotated but not translated. For brevity, we can encode all four homogeneous vectors in a single matrix and perform a matrix-matrix multiplication:

$$\begin{bmatrix} 0 & 2 & 0 & 0 \\ -2 & 0 & 0 & 4 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 2 & 0 \\ 4 & 2 & 4 & 0 \\ 0 & 0 & 0 & 2 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

After homogeneous division, we get rotated coordinates $\mathbf{p}' = (0,4,0)$, $\mathbf{q}' = (0,2,0)$, and $\mathbf{r}' = (2,4,0)$; the direction of the normal is unchanged (and we don't care about its magnitude).