

Computer Graphics (15-462/662) Midterm Review

(Review of midterm exam from Fall 2020)

Question 1—Getting Warmed Up

Just a few assorted questions to get your brain in graphics mode!



- - **RGB**—encodes the intensity of red, green, and blue emission as values in [0, 1].

Question: Suppose we want to fade between two images by linearly interpolating either RGB or HSV values. Which choice will give a more natural fade? What artifacts might you see?



from Lecture 14: Color

Question 1a

(a) (6 points) In our lecture on spatial transformations, we saw that the choice of matrix decomposition made a big difference when interpolating between two poses—for instance, separately interpolating each component of the polar decomposition gave natural motion, whereas directly interpolating the original transformation matrix resulted in weird artifacts. Likewise, the choice of color space will make a big difference if we want to interpolate between color images. Consider two common color models:

• **HSV**—encodes hue as an angle $\theta \in [0, 360)$, and both saturation/value as values in [0, 1].







SOLUTION.

There are a lot of possible criteria for what might make a "good" color space for interpolation, but *linearly* interpolating in HSV space has one big flaw: since the hue values θ "wrap around" from 0 to 360, you can get a sudden jump in interpolated values that leads to ugly artifacts (see example below). You might be able to design a more intelligent, *nonlinear* interpolation scheme that takes the shortest path around the circle. But for linear interpolation, RGB is the more natural of the two given choices.

RGB



t=0

HSV



Question 1a — solution







the table below.

Question: Does the given data describe valid manifold connectivity? If so, how many vertices, edges, and faces does the mesh have? Is it possible to draw each face as a flat polygon in 3D, without causing any polygons to intersect or become degenerate?





Question 1b

(b) (7 points) A compact way to store the connectivity of a halfedge mesh is to index the halfedges from 0 to 2E - 1 (where E is the number of edges) and implicitly assume that the twin of each even halfedge n is n + 1, and likewise, the twin of each odd halfedge n is n - 1. So for instance, 0 and 1 are twins, 2 and 3 are twins, and so on. This way, we only have to explicitly store index of the next halfedge, as done in



A manifold polygon mesh has fans, not fins

- For polygonal surfaces just two easy conditions to check:
 - 1. Every edge is contained in only two polygons (no "fins")
 - 2. The polygons containing each vertex make a single "fan"



from Lecture 10: Meshes and Manifolds

Question 1b

Halfedge connectivity is *always* manifold

- **Consider simplified halfedge data structure**
- Require only "common-sense" conditions

```
struct Halfedge {
   Halfedge *next, *twin;
};
```

twin->twin == this twin != this every he is someone's "next"

- Keep following next, and you'll get faces.
- Keep following twin and you'll get edges.
- Keep following next->twin and you'll get vertices.



Q: Why, therefore, is it impossible to encode the red figures?





- positions! Just connectivity



from Lecture 10: Meshes and Manifolds

Question 1b





Question 1b — solution





(7 points) Suppose you're given a triangle mesh where you know the vertex positions **p**, the vertex normals **n**, and the connectivity. If you imagine this data was sampled from a smooth surface, then there's really no reason you *have* to interpolate it using straight line segments and flat triangles (as depicted below, left). For instance, you could draw a wireframe of the mesh using Bézier curves where (i) the curve for each edge *ij* interpolates the positions $\mathbf{p}_i, \mathbf{p}_j \in \mathbb{R}^3$ at the two endpoints and where (ii) the tangents of all curves meeting at a common vertex *i* lie in the plane perpendicular to the normal \mathbf{n}_i at this vertex (as depicted below, right).



Question: What's the lowest-degree Bézier curve you can use to interpolate the point and normal data as described above? Do the given constraints *uniquely* determine the interpolating curve? If not, what might you do to pin down a unique solution? (There are many possible answers for the final question—we just want you to think creatively!)

Question 1c







derivative of the curve subject to the given linear constraints on the endpoints^{*a*}.

^{*a*}This problem, by the way, amounts to minimizing a quadratic form subject to linear constraints, which itself can be solved via a linear system!



Question 1c — solution

Each curve must interpolate two endpoints, each of which has three scalar components (x, y, z), yielding six constraints. At each endpoint *i*, it must also satisfy the scalar condition $\langle \mathbf{t}_i, \mathbf{n}_i \rangle = 0$, giving two additional constraints for a total of eight constraints. A linear Bézier curve has only two control points, or six scalar degrees of freedom, which is not enough to interpolate the given data—which we can also see geometrically: in general, several line segments meeting at a point will not have a common tangent. However, a quadratic Bézier curve has three control points, or nine scalar degrees of freedom, which is more than enough. The final degree of freedom could be pinned down by, e.g., minimizing the second





Question 2—Cubing the Sphere



Often in this class we've presented rasterization and ray tracing as two "competing" ways to draw images on the screen. The reality is that these techniques are getting combined more and more to achieve the best of both worlds: beautiful effects at lightning speed. One nice example is drawing a very large number of spheres, which might be used to, say, nicely display vertices in a mesh editor, or render high-quality points in a point cloud (among many other things!).

The traditional way to draw a sphere using the rasterization pipeline is to tessellate it into a bunch of triangles (as shown at right), then rasterize these triangles as usual. An alternative route, which we'll explore here, is to rasterize a bounding box around the sphere (as illustrated above). For each pixel in the bounding box, we then trace a ray from the eye through the pixel center **x**, and see if and where this ray intersects the sphere. The intersection information is then used to update the color and depth buffers. In essence, rather draw the color and depth of the triangle itself, we treat the triangle as a "portal" that looks into a box where the sphere lives. We'll build up this procedure one small piece at a time.

Question 2







Real-world implementation in http://polyscope.run/

a sphere with center $\mathbf{c} \in \mathbb{R}^3$ and radius r > 0. You should assume that this matrix will be applied to the homogeneous coordinates for the eight vertices. Matrices are indexed as A[i][j], where i is the row index, and j is the column index, and Matrix4x4::Zero gives the matrix of all zeros.



Question 2a

(5 points) Build a matrix that will transform a cube with vertices $(\pm 1, \pm 1, \pm 1)$ to the bounding box for

3D Transformations in Homogeneous Coordinates

- Not much changes in three (or more) dimensions: just append one "homogeneous coordinate" to the first three
- Matrix representations of 3D linear transformations just get an additional identity row/column; translation is again a shear

from Lecture 5: Spatial Transformations

0 0 1

Matrix4x4				
	Mat	ri	L X 4	43
	// A[C	<i>a</i> ŗ)]	0]	13]
	// A[C A[1 A[2 A[3	<i>us</i>)] 2] 2]	se [3] [3] [3]	t]]
}	ret	:u1	n	P

Question 2a — solution

oboxTransform(**Vec3** c, // sphere center **double** r) { // sphere radius x4 A = Matrix4x4::Zero; // initialize to zero

y a uniform scaling along the diagonal = A[1][1] = A[2][2] = r;

the rightmost column to apply a translation = c.x;= c.y;

- = C.Z;
- = 1.0;

Α;

(6 points) Implement a method that performs barycentric interpolation of three given vertex coordinates in world coordinates, assuming we are given the barycentric coordinates **b** of a point in the 2D projection of the triangle.

Barycentric Coordinates

- No matter how you compute them, the values of the three functions $\phi_i(\mathbf{x}), \phi_i(\mathbf{x}), \phi_k(\mathbf{x})$ for a given point are called <u>barycentric coordinates</u>
- Can be used to interpolate any attribute associated with vertices. (color*, texture coordinates, etc.)
- Importantly, these same three values fall out of the half-plane tests used for triangle rasterization! (Why?)
- Hence, get them for "free" during rasterization

 $\operatorname{color}(x) = \operatorname{color}(x_i)\phi_i + \operatorname{color}(x_i)\phi_i + \operatorname{color}(x_k)\phi_k$

*Note: we haven't explained yet how to encode colors as numbers! We'll talk about that in a later lecture...

from Lecture 7: Perspective Projection and Texture Mapping

Question 2b

Vec3 interpolateWorldPosition (**Vec3** p0, **Vec3** p1, **Vec3** p2, // vertex world coordinates **Vec3** b) // barycentric coordinates of sample point

// since the triangle was projected into the // 2D plane, we have to use perspective-correct interpolation **double** Z0 = 1/p0.z;**double** Z1 = 1/p1.z;**double** $Z_{2} = 1/p_{2.z}$; **Vec3** P0 = p0/p0.z;**Vec3** P1 = p1/p1.z; **Vec3** P2 = p2/p2.z; **Vec3** Z = b[0] * Z0 + b[1] * Z1 + b[2] * Z2;**Vec3** P = b[0] * P0 + b[1] * P1 + b[2] * P2;return P/Z;

Question 2b — solution

(5 points) Implement a routine that intersects a ray $\mathbf{o} + t\mathbf{d}$ with a sphere of radius r > 0 centered at $\mathbf{c} \in \mathbb{R}^3$. This routine assumes that the ray direction **d** has unit magnitude, and should return the smallest *positive t* where the ray hits the sphere; if there is no such intersection, it should return -1.

Intersecting a ray with an implicit surface

- Recall implicit surfaces: all points x such that f(x) = 0
- Q: How do we find points where a ray pierces this surface?
- Well, we know all points along the ray: r(t) = o + td
- Idea: replace "x" with "r" in 1st equation, and solve for t
- **Example: unit sphere**

 $f(\mathbf{x}) = |\mathbf{x}|^2 - 1$ $\Rightarrow f(\mathbf{r}(t)) = |\mathbf{o} + t\mathbf{d}|^2 - 1$ $\underbrace{|\mathbf{d}|^2}_{\bullet} t^2 + \underbrace{2(\mathbf{o} \cdot \mathbf{d})}_{\bullet} t + \underbrace{|\mathbf{o}|^2 - 1}_{\bullet} = 0$ t =

from Lecture 12: Geometric Queries

Question 2c

Question 2c — solution

double r, // radius **Vec3** o, // ray origin **Vec3** d) // ray direction (unit)

(6 points) Ok, let's put it all together. To draw a super high-quality sphere, we will rasterize its bounding box, which has been diced into triangles. Your job is just to implement an "unusual" triangle rasterization routine drawBBoxTriangle that shades each pixel of the triangle according to the closest sphere-ray intersection (as discussed at the beginning). For each pixel covered by the triangle, your routine should figure out the location $\mathbf{x} \in \mathbb{R}^3$ of this pixel in world coordinates. It should then trace a ray from the eye through **x** to see if it hits the sphere. If the hit point is the closest thing seen so far, your routine should shade the pixel using the color of the *sphere*, rather than the color of the bounding box. It should also update the depth buffer so that subsequent objects are properly occluded by the sphere.

X₂

Question 2d

Implementation notes: You can (and should!) call the routines from the earlier parts of this question—and can assume these routines work correctly, independent of what answers you gave above. You may also assume you have two other basic routines (with inputs illustrated above):

- Vec2 pixelCenter(i, j, w, h) returns the location of pixel (i, j) for an image of width $w \times h$.
- Vec3 baryCoords (p, p1, p2, p3) returns the barycentric coordinates of a point p within a triangle with vertices $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$.

For the main routine drawBBoxTriangle, the three input vectors $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^3$ give the world coordinates of the triangle vertices, after the camera transformation but before being transformed into clip space. (Hence, you can assume that the camera is sitting at the origin, looking down the -z-axis.) The inputs $\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2 \in \mathbb{R}^2$ give the same three coordinates projected into the 2D image plane, and transformed into final 2D image coordinates $[0, w] \times [0, h]$. The depth and color buffers have size w*h and store a single value per pixel (hence, color is just a greyscale value rather than an RGB color). You do not need to worry about efficiency: it is ok to test every pixel in the image to see if it's covered by the triangle.

W Ч

pixelCenter

Question 2d

baryCoords


```
double sphereColor,
for(int i = 0; i < w; i++ )
for( int j = 0; j < h; j++ )</pre>
   // grab the pixel center and its barycentric
   // coordinates relative to the triangle
   Vec2 c = pixelCenter( i, j, w, h, I0, I1 );
   Vec3 b = baryCoords( c, u0, u1, u2 );
   // if any of the barycentric coordinates are
   // negative, then this pixel falls outside the
   // triangle and we can stop
   if( b[0] < 0 || b[1] < 0 || b[2] < 0 ) break;
   // otherwise, interpolate the world-space position
```

Question 2d — solution

void drawBBoxTriangle(Vec3 x0, Vec3 x1, Vec3 x2, // world coordinates **Vec2** u0, **Vec2** u1, **Vec2** u2, // projected coordinates **Vec3** c, double r, // sphere center/radius double* depth, double* color, // buffers int w, int h) // buffer width/height

Vec3 x = interpolateWorldPosition(x0, x1, x2, b);

// position with the given sphere **Vec3** u = x.unit();

if(t < 0) **break**;

Vec3 p = e + t * u;

// if the hit point is the closest // thing we've seen so far, replace the current // color/depth with values from the sphere **if**(p.z < depth[i][j]) color[i+w*j] = sphereColor; // update color depth[i+w*j] = p.z; // update depth // otherwise, leave the color/depth alone

// intersect a ray from the eye to the world-space

double t = intersectSphere(c, r, e, u);

// if we missed the sphere, don't modify color or depth

// otherwise, compute the hit point location

Question 2d — solution

(6 points) Suppose you want to rasterize a scene that combines your beautiful, pixel-perfect spheres with ordinary triangles. Will everything work out if you just rasterize triangles in the usual way? For instance, will you correctly resolve depth for spheres that intersect triangles? Why or why not?

Question 2e

Yes, everything will work out fine. Since depth is resolved on a pixel-by-pixel basis, it doesn't matter where these depth values are coming from—the rasterizer will always draw the color of the closest object (whether sphere or triangle). That's the beauty of depth buffering!

Question 2e — solution

(6 points) A completely different strategy is to just use *instancing* to draw a bunch of copies of a triangle mesh of a sphere (using standard triangle rasterization). What are some pros and cons of instancing relative to the mixed ray tracing/rasterization scheme we've devised above?

Instancing

- What if we want many copies of the same object in a scene?
- Rather than have many copies of the geometry, scene graph, etc., can just put a "pointer" node in our scene graph
- Like any other node, can specify a different transformation on each incoming edge

from Lecture 5: Spatial Transformations

For one thing, instancing doesn't help improve the quality of each sphere: if we use a small number of triangles, we'll have very jagged edges. We still have to do a lot of work to draw each sphere (transforming the vertices, projecting them into 2D, rasterizing the triangle, etc.).^{*a*}

^{*a*}Expert viewpoint: on the other hand, if we rasterize instanced triangles then we get more predictable depth values (which are bounded by the depth values at vertices). This can help with techniques like "Z-culling," which can preemptively discard, say, a 4x4 block of pixels by bounding its max/min depth values before per-pixel rasterization takes place.

Question 2f — solution

(6 points) Finally, the other obvious strategy is to just ray trace everything, i.e., shoot a ray through every pixel, that gets tested against every sphere (possibly using some kind of spatial data structure). What advantage(s) does our hybrid "bounding box" strategy provide—assuming an efficient implementation that does *not* test every pixel for every triangle? What advantage(s) does pure ray tracing provide? (Hint: consider situations where you have either a very small or very large number of spheres.)

Basic rasterization vs. ray casting

Rasterization:

- Proceeds in triangle order
- Store depth buffer (random access to regular structure of fixed size)

Ray casting:

- Proceeds in screen sample order

 - are encountered along the ray: front-to-back or back-to-front)
- Must store entire scene

- **Hierarchies of rays/samples**
- **Hierarchies of geometry**
- **Deferred shading**
- ...

Question 2g

Don't have to store entire scene in memory, naturally supports unbounded size scenes

Don't have to store closest depth so far for the entire screen (just current ray) Natural order for rendering transparent surfaces (process surfaces in the order the

Performance more strongly depends on distribution of primitives in scene High-performance implementations embody similar techniques:

CMU 15-462/662 from Lecture 13: Spatial Data Structures

For a very small number of spheres, the hybrid strategy may be more efficient, since it only has to consider the region of the screen where spheres appear. This region can be efficiently determined via ordinary triangle rasterization. In a sense, the rasterization process lets us rapidly determine where we need to focus computational effort. The story changes if we have (say) millions of spheres, covering most of the screen. Here, it's beneficial to use pure ray tracing since we can stop tracing the ray as soon as we find the first hit. Hence, we may only need to touch a tiny fraction of the total spheres in our scene—we never even have to load some of the spheres into memory. In contrast, the rasterization strategy must go through every single sphere one by one. In the worst case, rasterization also results in massive overdraw: suppose for instance we get unlucky and the spheres are sent down the pipeline in roughly back-to-front order. Then we're spending a lot of time rasterizing things we never actually see. In this sense, rasterization is "linear" whereas ray tracing is "logarithmic."

Question 2g — solution

Question 3—Step into the Shadow

https://gkjohnson.github.io/threejs-sandbox/shadow-volumes/index.html

Up until now, we've said that ray tracing is typically needed to render effects like reflections and shadows. But actually, there's a clever way to render exact shadows for a point light source using just a rasterizer. The basic idea is to explicitly construct a polygon mesh called the "shadow volume," corresponding to the shadowed region of space (see above). These polygons are then rasterized as usual—but with a twist: rather than rasterize directly into the image buffer, we rasterize them into a so-called *stencil buffer*. This buffer does not keep track of color values, but instead counts the *number* of shadow polygons that cover each pixel. More accurately: this number is *incremented* (+1) for shadow polygons that face the camera, and *decremented* (-1) for shadow polygons that face away from the camera. The stencil buffer can then be used in a final pass to shade only those pixels where the closest primitive is outside the shadow volume.

(5 points) In a conventional rasterization pipeline², what is the fundamental reason why we can't just directly evaluate whether a pixel is in/out of shadow during the triangle rasterization stage?

Question 3a

The rasterization pipeline processes one triangle at a time, so there's no way to know which other triangles might be occluding a given point.

Lecture 4: Drawing a Triangle & Intro to Sampling

Question 2g — solution

(5 points) Consider a triangle mesh that is manifold and has no boundary, and a point light source at a point $\mathbf{x} \in \mathbb{R}^3$. The *shadow contour* is the set of edges on the boundary between the shadowed and non-shadowed region. More precisely, an edge *ij* is part of the shadow contour if **x** is in front of one of the two triangles containing *ij*, and behind the other.

answer without such expressions.

Question 3b

Give an explicit mathematical expression (**not code**) that evaluates to true if *ij* is part of the shadow contour, and false otherwise. Your expression should use the light position $\mathbf{x} \in \mathbb{R}^3$, the two edge endpoints $\mathbf{p}_i, \mathbf{p}_j \in \mathbb{R}^3$, and the two normal vectors $\mathbf{n}_{ij}, \mathbf{n}_{ji} \in \mathbb{R}^3$ on either side of an edge *ij*. If you like, you may use logical operations (AND, OR, etc.) in your expression, but there is a nice way to write the

One possible expression is

 $\langle \mathbf{n}_{ij}, \mathbf{x} - \mathbf{p}_i \rangle$

Each inner product effectively tests whether the light source is in front of one of the two triangles' planes, by seeing if the vector toward the light is pointing toward (positive) or away from (negative) the normal. If the light is on the same side of both of the triangles, this product will be positive; otherwise only one of the terms will be negative and the product will be negative.

Question 3b — solution

$$\langle \mathbf{n}_{ji}, \mathbf{x} - \mathbf{p}_i \rangle < 0.$$

(5 points) Using your expression from the previous part, write some pseudocode to decide if the given edge is on the shadow contour. You may assume a basic halfedge data structure³, which provides methods like face->normal() and vertex->position(), but you must explicitly call these methods (rather than assuming that quantities like \mathbf{p}_i , \mathbf{n}_{ij} , etc., are already given). You may also assume basic vector operations (cross product, dot product, etc.). Note: you will *not* be penalized if your expression from the previous part is wrong, as long as the rest of your code is right.

Question 3c & solution

structure, and must explicitly access any data you need to perform the computation.

Question 3d

(5 points) For each edge *ij* on the shadow contour, we have to build a polygon that forms one "side" of the shadow volume⁴. This polygon is formed by the two endpoints p_i and p_j of the edge, which are extended along rays from the light source x to points \mathbf{q}_i , \mathbf{q}_j at infinity. Write a routine that computes the shadow polygon for a given edge, yielding four points in homogeneous coordinates. If you find it convenient, you can assume that Vec4 has a constructor that takes a Vec3 and a scalar as input (e.g., Vec4 u(v, 1), where u is a Vec3). As in part (c), you should assume a standard halfedge data

$$\lim_{s\to\infty}\frac{\mathbf{p}_i+s(\mathbf{q}_i-\mathbf{p}_i)}{|\mathbf{p}_i+s(\mathbf{q}_i-\mathbf{p}_i)|}=\frac{\mathbf{q}_i}{|\mathbf{q}_i-\mathbf{p}_i|}$$

Question 3d — solution

from Lecture 5: Spatial Transformations

 $-\mathbf{p}_i$ $-\mathbf{p}_i$

mesh that is manifold but does not have triangular faces?

In a general polygon mesh, polygons may not be planar. For instance, the four vertices of a quadrilateral may not all lie in a common plane. Hence, normals no longer have a clear definition; likewise, it's not clear anymore that the shadow contour will be along edges. (For instance, if we view a nonplanar quad as a bilinear patch, then the dot product of the normal with the light direction may change sign on the interior of the patch.)

Question 3e & solution

(5 points) What challenge might you run into when trying to build a shadow volume for a polygon

(5 points) For the shadow volume strategy to work, we need to make sure not to draw shadow polygons that are occluded by scene objects, and we need to make sure not to shade pixels that should be in shadow (*i.e.*, pixels where the final stencil buffer value is greater than zero). So, the general strategy is to rasterize the scene in multiple "passes." Each pass might render a different set of objects, and uses data from one buffer to decide what data should get written into another buffer.

Describe—in words, not code—any sequence of rasterization passes that will correctly draw shadowed pixels as black and lit pixels as white. The only three buffers you should consider are a *depth buffer* which keeps track of the closest primitive seen so far, a *stencil buffer* which counts the number of primitives drawn into each pixel (incrementing for camera-facing polygons, and decrementing otherwise), and a *color buffer* which stores the final color values (in this case just black or white). The only two sets of primitives you can rasterize are the scene primitives describing the objects in the scene, and the shadow primitives which are the polygons describing the shadow volume. All pixels that are neither lit nor in shadow should be given a background color (like grey).

Question 3f

One of many possible solutions is to first clear the color buffer to the background color. Then render the scene primitives into the depth buffer, and into the color buffer as black. Then render the shadow primitives, but only increment/decrement the stencil buffer if a pixel passes the depth test. This way, you count how many times you enter/exit a shadow volume before seeing the closest object. Finally, render the scene primitives again, but this time only write to the color buffer if the depth is equal to the value already in the depth buffer, and the stencil buffer has a value of zero.

Pass 1: render all pixels "in shadow" color depth

0

Question 3f — solution

stencil

(5 points) What happens if the camera is *inside* the shadow volume? Does your scheme correctly draw objects that are in shadow? If so, why? If not, why not? (Note: you are *not* required to come up with a procedure that produces correct shadows in this case! You merely need to be able to correctly analyze the behavior of your algorithm in this scenario.)

Image credit: Usta et al, "A Practical and Efficient Approach for Correct Z-Pass Stencil Shadow Volumes"

Question 3g

The proposed strategy will not work correctly when the camera is inside the shadow volume, since the count in the stencil buffer will be off: for instance, if we are sitting inside the shadow volume, looking at an object that is also inside the shadow volume, the count will be zero even though this object should be in shadow. There is a simple strategy (not described here) called "Z-fail" that correctly handles this case.

See: Usta et al, "A Practical and Efficient Approach for Correct Z-Pass Stencil Shadow Volumes"

Question 3g — solution

(5 points) Finally, let's put it all together—write a routine that rasterizes a polygon from the shadow volume, and updates the stencil buffer. Independent of how you designed your algorithm in part (f), this routine should take the four points of the polygon as input, and update the stencil buffer only if this polygon is closer than the closest object stored in the depth buffer. The four vertices of the polygon are given as points P0, P1, Q0, Q1 that have already been projected into 2D image coordinates; the third coordinate of each point gives its depth value, and all vertices are at finite locations (*i.e.*, not points at infinity).

Implementation notes: You may use any method from any other part of the exam—even if you did not complete that part. As before, you may also assume that you have the methods pixelCenter and baryCoords. All buffers have size w*h and store a single value per pixel (color is just a greyscale value rather than an RGB color). You do not need to worry about efficiency—to rasterize, you can just test coverage for every pixel in the entire image. This routine will be a bit longer/more complicated than the simple subroutines you wrote above. Some

questions to think about:

- What's the standard way to draw a quad via the rasterization pipeline?
- How do you check if a sample point is inside a triangle?
- How do you determine the depth at an arbitrary location inside a triangle?
- How do you know whether to increment or decrement the stencil buffer?

(Note: these questions are just to help you think about how to write the routine! You do not have to answer them directly.)

Question 3h


```
void drawShadowPolygon(
   int w, int h, // buffer width/height
   Vec2 I0, Vec2 I1 ) // image bounds
   // we're going to draw the quad as two triangles,
   // which for convenience we'll store in an array
   Vec3 tris[2][3] = {
      { P0, P1, Q1 }, // first triangle
      \{P0, Q1, Q0\} // second triangle
   };
   for(int i = 0; i < w; i++ )
   for(int j = 0; j < h; j++ )
     Vec2 c = getPixelCenter( i, j, w, h, I0, I1 );
      for( int k = 0; k < 2; k++ )
         array<Vec3, 3>& P = tris[k];
```

Question 3h — solution

Vec3 P0, Vec3 P1, Vec3 Q0, Vec3 Q1, // polygon vertices double* depth, double* stencil, double* color, // buffers

// iterate over the two triangles making up the quad

// grab a reference to the current triangle

// get the barycentric coordinates of the // pixel center, relative to this triangle **Vec3** b = baryCoords(c, P[0], P[1], P[2]);

 $if(b1[0] > 0 \&\& b1[1] > 0 \&\& b1[2] > 0) {$ d = b1[0] * T[k][0].z +b1[1]*T[k][1].z + b1[2]*T[k][2].z ; **if**(d < depth[i+w*j]) { **if**(o > 0) stencil[i+w*j]++;

```
// check if this pixel is inside this triangle
   // get the triangle's depth at (x, y) by interpolating
   // the corner depths via barycentric coordinates
   // check whether the stencil polygon is closer
   // than the closest primitive in the scene
      // get the triangle's orientation by taking the cross
      // product of two of its edge vectors
      o = (P[1]-P[0]) \cdot x * (P[2]-P[0]) \cdot y -
          (P[1]-P[0]).y*(P[2]-P[0]).x;
```

```
// increment the stencil buffer if the triangle is
// pointing toward us; otherwise, decrement it
      else stencil[i+w*j]--;
```

Question 3h — solution

