Full Name: _____

Andrew Id: _____

0.1in

15-462/662, Fall 2015

Final Exam

Dec 14, 2015

Instructions:

- This exam is CLOSED BOOK, CLOSED NOTES (with the exception of your one post-it note).
- The exam has a maximum score of **100** points. Unlike your midterm, you should try to answer *all* of the questions. Don't worry if you can't finish everything—keep in mind that everyone else is on the same clock, and will be graded on the same curve as you.
- If your work gets messy, please clearly indicate your final answer.

Problem	Your Score	Possible Points
1		15
2		15
3		18
4		10
5		7
6		10
7		10
8		15
Total		100

Getting the Party Started: Miscellaneous Short Problems

Problem 1. (15 points):

A. (2.5 pts) Give one reason why color representations that explicitly separate the luminance (brightness) and chroma components of a color (e.g., YCbCr or hue-saturation-brightness (HSB)) can be useful representations.

Solution:*This representation is useful for compression since the human visual system is less sensitive to spatial variation in chroma, allowing for lower-resolution sampling of these components. Another reason is that these representations can be more intuitive when describing or selecting colors.*

B. (2.5 pts) Imagine the human visual system could directly measure and interpret the full spectrum of incident light. (That is, your brain received and used full spectral information $L(\lambda)$ rather than just the response of S,M,L-cones). How would this make recording and displaying digital images and rendering pictures far more challenging?

Solution:*Graphics would get tough. Computers would have to record, synthesize, or generate entire spectrums that match those encountered in the real world. Otherwise, the images produced by computers would not be perceived by our brains to contain the same colors as what we see in the real world.*

C. (2.5 pts) In class we discussed how the cosine basis was a very useful basis for compression for **real-world images** since high-frequencies are less common in nature. Describe the properties of an image that would be more amenable to compression when represented in the **pixel basis** than the cosine basis. (Recall from the lecture 24 quiz that basis image B_{ij} in the pixel basis is the image with all pixels black except for pixel (i,j).)

Solution:*We accepted solutions where non-zeros in the pixel basis were sparse. For example, an image with only a few colored pixels and the rest black. We also accepted solutions that described large flat regions of color, since they could easily be run-length encoded.*

D. (2.5 pts) Suppose you wanted to simulate a popcorn popping in a popcorn machine. What would be a more appropriate spatial discretization: Lagrangian or Eulerian? Why?

Solution: *I would probably track individual popcorn kernels as Lagrangian particles, so that I could track their individual trajectories. However, for a truly massive amount of popcorn (e.g., filling up an entire house) it might be useful to approximate the gross motion of the popcorn using some kind of continuum model. Also, the popping of an individual popcorn kernel might be better simulated on an Eulerian grid.* [Note to the graders: any answer is good here as along as it is logical and thoughtful.]

E. (2.5 pts) What's an example of a function that has infinitely many local minima but no global minimum?

Solution: *There are many possible answers here; one is* $f : \mathbb{R} \to \mathbb{R}; x \mapsto \cos x - x$.

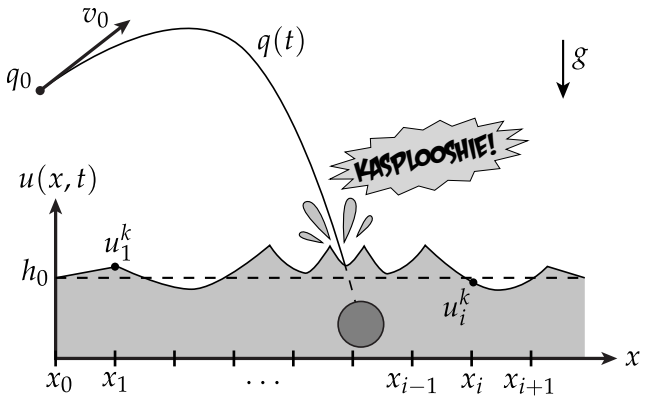
F. (2.5 pts) Label each of the following linear transformations, and justify your answer by saying which quantity was preserved.

(Original)

Solution: *From left to right: rotation (because angles, lengths, and orientation was preserved); linear isomorphism (because lines were preserved); uniform scale (because everything but scale was preserved).*

Making a Splash in Computer Graphics

Problem 2. (15 points):



You are working on a key scene from the next Pixar movie, where in desperation the main character chucks her favorite spherical rock into her favorite one-dimensional pond. The artists are gearing up to animate each little splash by hand, but suddenly you remember all the amazing stuff you learned in 15-462. Give the artists a break by building the solver described below.

A. (2.5 pts) First, you need to animate the trajectory $q(t)$ of the rock as it flies through space, starting at a position $q(0) = q_0$ with velocity $\dot{q}(0) = v_0$. The trajectory of a spherical rock with uniform mass density is given by the 2nd-order ODE $\ddot{q} = g$, where g is a downward-pointing vector giving the acceleration due to gravity. Rewrite this equation as a pair of first-order ODEs, so that it can be more easily integrated.

Solution: *Letting v denote the velocity of the rock, we get $\dot{q} = v$ and $\dot{v} = g$.*

B. (2.5 pts) Now you need to convert your two differential equations into a finite procedure that can be executed on the compute farm. Pick a time discretization scheme (forward Euler, backward Euler, or symplectic Euler), and write down the corresponding update rule for your rock. Make sure you specify how all your variables are initialized. Give some justification for why you picked this particular scheme for this animation shot.

Solution: *I chose to use symplectic Euler, because I wanted the artists to have predictable control over how big the splash is—therefore, I don't want the rock to gain or lose energy due to numerical integration. Let $v(t)$ be the velocity of the particle as a function of time. Then symplectic Euler yields the update rule*

$$\begin{aligned} v^{k+1} &= v^k + \tau g, \\ q^{k+1} &= q^k + \tau v_{k+1}, \end{aligned}$$

*where τ is the time step, and the second equation is evaluated after the first one. The initial values of q and v are simply given by the initial conditions q_0 and v_0. [**NOTE:** Any choice of integrator is fine here as long as the answer is meaningfully justified.]*

C. (2.5 pts) Now you hit "play" and the rock flies beautifully through the air. Sadly, the animation lacks a climax because the rock simply passes through the surface of the pond, which remains calm and placid. Next, you will need to solve for the height $u(x,t)$ of the water as a function of space and time. The first question to answer, though, is: where and when does the rock impact the water? You may assume that the rock always starts out above the water, which initially sits at a constant height $u(x,0) = H_0$. It may help to work things out in terms of coordinates $q = (x,y)$. Your answer should provide both the location and time of impact, and should depend only on values that come out of the numerical integrator. [1] Try to be as accurate as possible—this is Pixar, after all.

Solution: *Since we only have samples of the trajectory, we need to interpolate it somehow to find where it intersects the water surface. The simplest scheme is linear interpolation. In particular, we're looking for the moment at which a line segment from q_k to q_{k+1} intersects a horizontal line with height H_0. We first need to find the index k such that $y_k > H_0$ and $y_{k+1} \leq 0$, which we can simply run as a check during our ODE integration. Next, we need to find the point along this segment that intercepts the line $y = H_0$. If we write the height of the segment as*

$$y(\alpha) = (1-\alpha)y^k + \alpha y^{k+1},$$

then we just have to solve $y(\alpha) = H_0$ for α, which yields

$$\alpha = \frac{y^k - H_0}{y^k - y^{k+1}}.$$

The point of impact is then $q^* = (1-\alpha)q_k + \alpha q_{k+1}$, and the time of impact is $t^* = t_k + \alpha\tau$.

D. (2.5 pts) Now that we know where and when the rock hits the pond, it needs to somehow affect the motion of the water. Suppose that we use an *Eulerian* discretization, i.e., we have a fixed grid at positions $x_0, x_1, \ldots, x_n$, and at each point we keep track of the corresponding height of the water $u_0^k, u_1^k, \ldots, u_n^k$, where the subscript denotes the grid point, and the superscript k denotes the time step. Roughly sketch out some way we might set the values of u such that it looks like our rock is hitting the water. You do *not* have to give a precise mathematical/algorithmic description here. There are several possibilities; any solution is ok as long as you can provide a compelling justification.

Solution: *The simplest thing we might do is simply find the nearest time step k and the nearest grid point i, and change the value of the corresponding height u_i^k; for instance, we could set it to something like half the rest height H_0. However, this scheme would not be particularly accurate (especially on a very coarse grid), because we are rounding to the nearest location in space/time. Instead, we might find, say, the two closest nodes on the grid, and adjust both of their heights. Still, this scheme leaves something to be desired since it does not account for the velocity of the rock. So, we might do something like (i) grab the velocity of the rock from our ODE integrator, and (ii) set the* difference *of heights over time to be proportional to this velocity. If we really wanted to get fancy, we could try deriving a scheme based on conservation of momentum, i.e., making sure that the total momentum of the rock-water system before and after impact is preserved. Lots of possibilities.*

[1] In other words: for those of you more comfortable with ODEs, do *not* simply use the analytical solution, because in general analytic solutions cannot be computed! This approach is therefore forbidden.

E. (2.5 pts) *I like to move it! Move it!* ...Whoops, wrong studio. Anyway, it's time to add some motion to your pond. At this point, everything is pretty much in place: you have a grid with water height u_i^k for the ith grid point at the kth time step, and some of these values have already been set according to where and when the rock makes an impact. The only thing left to do is to update the grid locations at all subsequent points in time so that you get beautiful rippling motion. For this, you'll want to integrate the *wave equation*, which in 1D amounts to just

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2},$$

where the constant c is the *wave speed*, i.e., how fast the wave moves outward from the point of impact. Because film runs at only 24 frames per second, you'll want to take pretty big time steps. Therefore, you decide to discretize this equation using the *backward Euler* method. But rather than split this equation into two pieces that are first-order in time (as we did with the rock), you're going to simply use second-order finite differences for each of the second derivatives (in space and time). Write down the backward Euler update rule at grid node i and time k, assuming a fixed time step τ and a fixed grid spacing h. You do not have to *solve* for the new variables; you just have to write down the rule. (Hint: if you can't remember the rule for a second-order finite difference, just try taking the first-order difference of a first-order difference!)

Solution: *The first-order finite difference in time is*

$$\frac{u_i^{k+1} - u_i^k}{\tau},$$

which means the second-order difference in time is

$$\frac{\frac{u_i^{k+1} - u_i^k}{\tau} - \frac{u_i^k}{u_i^{k-1}}}{\tau} = \frac{u_i^{k+1} - 2u_i^k + u_i^{k-1}}{\tau^2}.$$

Likewise, the second-order difference in space is

$$\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{h^2}.$$

For backward Euler, we want to evaluate the velocity function at the next *point in time $(k+1)$, which means our update rule is*

$$\frac{u_i^{k+1} - 2u_i^k + u_i^{k-1}}{\tau^2} = c^2 \frac{u_{i+1}^{k+1} - 2u_i^{k+1} + u_{i-1}^{k+1}}{h^2}.$$

which we want to solve for the unknown variables u_i^{k+1} at all grid notes i at time step $k+1$.

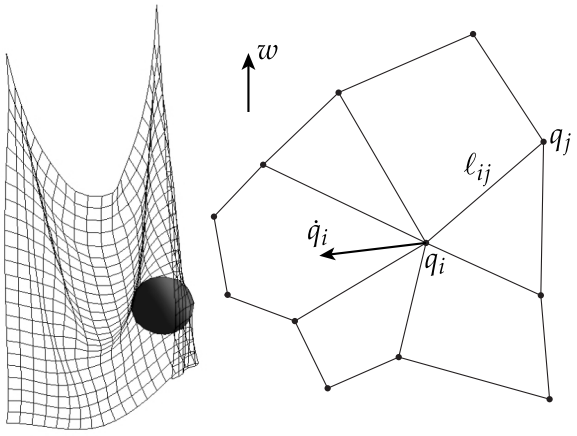
F. (2.5 pts) We now have a big system of linear equations (one for each grid node) that we want to solve for all the unknown variables u_i^{k+1} at the next time step $k+1$. The artists at Pixar are pretty bad at solving linear equations, so instead you're going to just call a routine in a numerical linear algebra library. Question: should you use a dense matrix or a sparse matrix for this system, and why?

Solution: *A sparse matrix is most appropriate here, because each of our linear equations involves just a few variables—therefore, the corresponding matrix will contain mostly zeros.*

Mesh-Spring System

Problem 3. (18 points):

Your startup is building a cutting-edge platform for *virtual try-on*: slip on a VR helmet, and you can visualize yourself wearing all the latest fashionable hoodies. The great part about this system is that the customer can buy clothes from the comfort of their own home, without having to put on anything at all—kind of a modern version of *The Emperor's New Clothes.* Your job is to build a fast, stable system for virtual cloth that meets the low-latency demands of modern VR. Since clothing can come in many shapes and sizes your basic idea is to turn any polygon mesh with vertices V, edges E, and faces F into a dynamic piece of cloth, by treating each edge as a spring in a giant mass-spring system. In other words, between each pair of neighboring vertices, there is a force preventing the edge from getting compressed or stretched-out too much. Your task will be fleshing out this basic idea into a dynamic simulation algorithm.



A. (2 pts) From your intro physics class, you probably remember that the potential energy of a spring is $\frac{1}{2}\alpha x^2$, where x is the displacement of the spring from its rest length, i.e., x is negative when the spring is compressed, positive when the spring is stretched-out, and zero when it is at its rest length. The constant $\alpha > 0$ determines the spring stiffness. Let ℓ_{ij} be the rest length for the edge between vertices i and j of your mesh (corresponding to the edge length in the original file), and use q_i, q_j to denote the position of the vertices. What is the spring potential energy for edge ij? What then is the total spring potential energy U_{spring} for the entire mesh?

Solution: *For a given edge $ij \in E$, the displacement x is just the difference between the current length and the rest length: $|q_j - q_i| - \ell_{ij}$. The total spring potential energy is then just the sum of $\frac{1}{2}\alpha x^2$ over all edges: $U_{\text{spring}} = \frac{\alpha}{2}\sum_{ij\in E}(|q_j - q_i| - \ell_{ij})^2$.*

B. (2 pts) Most of your customers will be wearing their clothes on a planet with gravity. What is the total gravitational potential U_{gravity} of the cloth in terms of the vertex positions q_i? You may assume that each vertex has mass m and that the gravitational constant is g; the springs connecting vertices can be assumed to be massless. You should also assume that the user provides an arbitrary unit vector w corresponding to the "up" direction, the direction along which gravitational acceleration is applied. (*Hint:* think carefully about how to measure the "height" along an arbitrary direction.)

Solution: *Since the potential for a single vertex is mgh, where h is the height along the direction w, the total potential is $U_{\text{gravity}} = mg\sum_{i\in V}\langle q_i, w\rangle$, where the inner product extracts the height.*

C. (2 pts) At this point, you might be able to build a system that shows how cloth drapes over the body: just find the configuration of the mesh that minimizes potential energy, keeping some of the vertices fixed (e.g., those that come into contact with the body). However, most customers also care about how the cloth *moves*. Therefore, you will also need an expression for the total kinetic energy K. Again you may assume that each vertex has mass m; use $\dot{q}_i$ to denote the velocity of vertex i.

Solution: *The kinetic energy for a single vertex is $\frac{1}{2}m|\dot{q}_i|^2$, so the total kinetic energy is $K = \frac{m}{2}\sum_{i\in V}|\dot{q}_i|^2$.*

D. (2 pts) Now that you have the kinetic and potential energy, you can easily formulate the Lagrangian $\mathcal{L}(q, \dot{q})$. Write out an expression for $\mathcal{L}$ using the energies you derived above.

Solution: *The Lagrangian is just the total kinetic energy minus the total potential energy, which we can write as*

$$\mathcal{L} = K - U = K - (U_{\text{spring}} + U_{\text{gravity}}) = m \sum_{i \in V} (\tfrac{1}{2}|\dot{q}_i|^2 - g\langle q_i, w \rangle) - \tfrac{\alpha}{2} \sum_{ij \in E} (|q_j - q_i| - \ell_{ij})^2.$$

E. (2 pts) Now that you have the Lagrangian, you can just "plug and chug" to get the equations of motion for our cloth. In particular, you have to evaluate the *Euler-Lagrange equation*

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} = \frac{\partial \mathcal{L}}{\partial q_i},$$

which gives an implicit description of the motion of any vertex $i \in V$. First work out an explicit expression for the left-hand side, i.e., take the derivative of $\mathcal{L}$ with respect to the velocity $\dot{q}_k$ of some particular vertex $k \in V$, then take the time derivative.

Solution: *The partial derivative of the Lagrangian with respect to the velocity of vertex k is given by*

$$\frac{\partial \mathcal{L}}{\partial \dot{q}_k} = \frac{\partial}{\partial \dot{q}_k} \sum_{i \in V} \tfrac{m}{2} |\dot{q}_i|^2 = m\dot{q}_k,$$

since only the velocity of vertex k affects the kinetic energy of vertex k. The time derivative is then just

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_k} = \frac{d}{dt}(m\dot{q}_k) = m\ddot{q}_k.$$

F. (2 pts) Next, you will need to work out an explicit expression for the right-hand side of the Euler-Lagrange equation for vertex k. Start by taking the derivative of the gravitational potential U_{gravity} with respect to q_k.

Solution: *The derivative of the gravitational potential with respect to q_k yields*

$$\frac{\partial}{\partial q_k} U_{\text{gravity}} = \frac{\partial}{\partial q_k} \sum_{i \in V} mg\langle q_i, w\rangle = mgw,$$

since only one of the terms in the sum involves q_k.

G. (2 pts) Finally, to complete the right-hand side of the Euler-Lagrange equation, take the derivative of the elastic potential energy with respect to the position q_k of vertex k. Be careful to think about which edges from the potential energy affect the motion of vertex k. The key geometric question to think about here is: what is the gradient of the norm $|q_j - q_i|$ with respect to, say, q_j? In other words, if we want to increase this norm as quickly as possible, in which direction should we move q_j? And if we move q_j a unit distance in that direction, how much will the norm change? These two questions should quickly lead you to an expression for the gradient (though you are free to grind it out algebraically if you prefer!). From there, everything else is just careful application of the chain rule. (*Hint:* in 1D, the gradient of spring energy $\frac{1}{2}kx^2$ with respect to the displacement x yields the spring force kx. In 3D the force will now be a vector rather than a scalar, but this 1D expression can help you check if you're on the right track.)

Solution: *First, let's think about the gradient of the norm $|q_j - q_i|$ with respect to q_j. The gradient will move q_j away from q_i as quickly as possible, i.e., along the direction $q_j - q_i$. Moreover, moving a unit distance along this direction will yield a unit change in the norm. Therefore, the gradient is just the corresponding unit vector*

$$\frac{q_j - q_i}{|q_j - q_i|}.$$

The derivative of the spring potential is then just an application of the chain rule, using the expression above:

$$\frac{\partial}{\partial q_k} U_{\text{spring}} = \tfrac{\alpha}{2} \frac{\partial}{\partial q_k} \sum_{ij \in E} (|q_j - q_i| - \ell_{ij})^2 = \alpha \sum_{kj \in E} (|q_j - q_k| - \ell_{kj}) \frac{q_j - q_k}{|q_j - q_k|}.$$

In other words, we just sum up the unit edge vectors "sticking out of" vertex k, multiplied by the spring stiffness and the displacement relative to the rest length—this quantity is just like the 1D case, except that our force vectors now have a direction, given by the direction of the edges.

H. (2 pts) Combine the answers to your previous solutions to get the equations of motion for a single vertex. To make the next step easier, isolate the acceleration $\ddot{q}_k$ on the left-hand side of your final answer. *Note that this question continues on the next page!*

Solution: *Overall, the equations of motion at vertex k are given by*

$$\ddot{q}_k = -gw - \frac{\alpha}{m} \sum_{kj} (|q_j - q_k| - \ell_{kj}) \frac{q_j - q_k}{|q_j - q_k|}.$$

I. (2 pts) You've done the hard work of deriving the equations of motion. Now for the payoff: plug them into your code, and see the clothing flap around! You can assume that someone else at your VR startup is responsible for actually implementing the time integrator; the only thing you'll need to do is implement the method `Vertex::acceleration()`, which provides the acceleration of a vertex in a halfedge mesh. Note that you will **not** be penalized for getting the equations of motion wrong on this question, as long as your implementation matches the acceleration you derived the (possibly incorrect) in the previous part. [2] You should explicitly define any of the constants you need (α, w, etc.), but are free to set these constants to arbitrary values (these values will probably have to be tuned anyway, depending on the type of garment). The method should return the acceleration as a `Vector3D`. Remember that in a halfedge mesh, each halfedge knows about its `next` and `twin` halfedge, and each vertex, edge, and face points to a single `halfedge`; the vertex position is stored in `Vertex::position`, and the rest edge length is stored in `Edge::length`.

```
Vector3D Vertex::acceleration( void )
{

   const double m = 1.;
   const double g = 9.81;
   const double alpha = 1.23;
   const Vector3D w( 0., 1., 0. );

   Vector3D qk = position;
   Vector3D a( 0., 0., 0. ); // acceleration

   a -= g * w; // add contribution due to gravity

   HalfedgeCIter h = halfedge(); // loop over incident edges
   do
   {
      double lkj = h->edge()->length;
      Vector3D qj = h->twin()->vertex()->position;
      Vector3D rkj = qj - k;

      // add spring acceleration for this edge
      a -= (alpha/m) * ( rkj.norm() - lkj ) * rkj / rkj.norm();

      h = h->twin()->next();
   }
   while( h != halfedge() );

   return a;

}
```

[2] For you jokers in the crowd: we will *not* accept an answer where you write something obviously bogus for the acceleration in the previous part, like $\ddot{q} = 0$.

A Highly Irregular Rasterizer

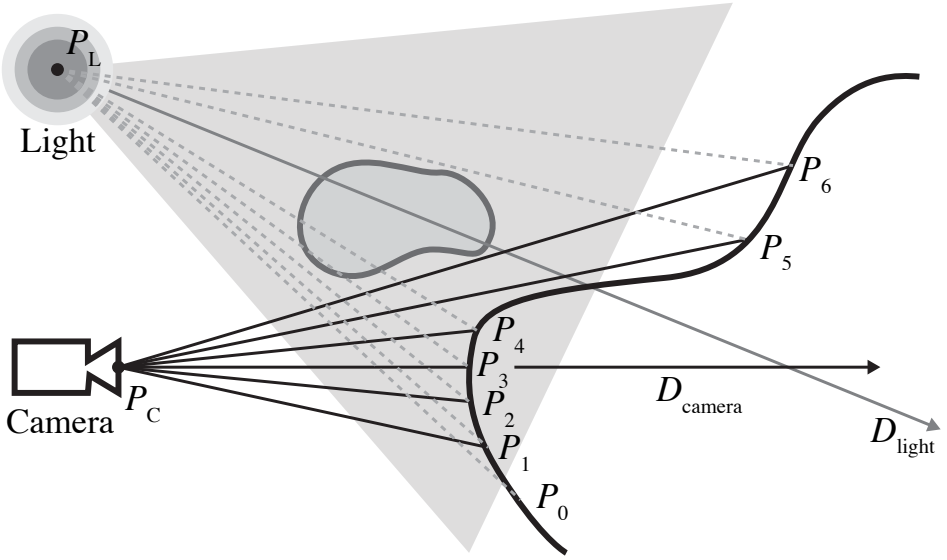
Problem 4. (10 points):

Imagine that you have a special kind of rasterizer which doesn't evaluate depth/coverage at uniformly spaced screen sample points, instead it evaluates depth/coverage **at a list of arbitrary 2D screen sample points provided by the application**. An example of using this rasterizer is given below. In this problem you should assume that depths returned by `fancyRasterize` are in WORLD SPACE UNITS.

```
vector<Point2D>  myPoints;  // list of points to eval coverage in [-1,1]^2
vector<Triangle> geometry;  // list of triangles
Transform perspProj;        // perspective projection from camera space
                            // to [-1,1]^3 cube

// this call returns the distance of the closest hit (assume infinity if no hit)
vector<float> depths = fancyRasterize(geometry, myPoints, perspProj);
```

You are now going to use `FancyRasterize` to render images with shadows. Consider the setup of a camera, scene objects, and a light source as illustrated below.



A. (4 pts) You use a traditional rasterizer to compute the depth of the closest point of the scene at each screen sample point. In the figure, the closest point visible under each sample when the camera is placed at position P_C and looking in the direction D_{camera} is given by P_i. All points in the figure are given in **world space!**

Assume you are given a world space to light space transform `world2Light` and a perspective projection transform `lightProj` that performs perspective projection of points in light space into the canonical $[-1, 1]^3$ cube in which `FancyRasterize` performs coverage testing. (Light space is the coordinate space whose origin in world space is P_L and whose -Z axis is in the direction D_{light}.)

Provide pseudocode for an algorithm that computes, for each point P_i, if the point is in shadow from a point light source located at P_L. The algorithm should call `fancyRasterize` only once. (No, you are not allowed to just implement a ray tracer from scratch.) Your algorithm accepts as input an array of points P_i, and also has access to points P_C, P_l, and transforms `world2Light`, `lightProj`.

Hint: Be careful, `fancyRasterize` wants points in 2D (represented in a space defined by the $[-1, 1]^2$ "image plane") so your solution will need to convert from a 3D or homogeneous 3D representation to this representation.

Solution:

First transform all hit points into projected light space. (Note: we're using $\mathrm{T}_{\text{lightProj}}$ *and* $\mathrm{T}_{\text{lightProj}}$ *to refer to the transforms described above.*

$$Q_i = \mathrm{T}_{\text{lightProj}}\mathrm{T}_{\text{world2light}}P_i$$

Then perform homogeneous divide to obtain a 2D sample point for the call to `fancyRasterize`. Note that the homogeneous divide is divide by w after applying the perspective projection transform, not divide by z.

$$(x_i, y_i) = (Q_i[x]/Q_i[w], Q_i[y]/Q_i[w])$$

For each input point x_i, y_i, `fancyRasterize` *produces the output tuple* $(depth_i, covered_i)$. *A point* P_i *is in shadow if the point is farther from the light source than what* `fancyRasterize` *indicates is the distance to the first hit. Specifically,* P_i *is in shadow if:* $|P_i - P_L| > depths_i$.

B. (3 pts) Does the algorithm you proposed above generate "hard" or "soft" shadows? Why? (please assume your solution successfully does what is asked in part A)

Solution: *The algorithm is a hard shadow algorithm because it is computing shadowing due to a point source. Many students answered that the shadows were hard because only one shadow ray was shot to the light (one light sample taken). This is not correct, since you can still get a soft shadow (albeit a very high variance one) but taking one illumination sample from an area light source. The point is that the light source is a point source!*

C. (3 pts) Prof. Kayvon quickly looks at the algorithm you devised above and says "remember I told you in class that shadow mapping is such a hack", you should just write a really fast ray tracer that accurately computes shadows in real time. Prof. Keenan jumps in and says, wait a minute, this algorithm seems perfectly accurate to me. Who is correct? Why?

Solution: *Prof. Keenan is correct. In this case, we're using a modified form of a rasterizer to exactly compute the first hit along the exact shadow rays that would be shot by a ray tracer towards a point source. The hard shadows produced by this algorithm (which uses irregular rasterization) is exactly the same as the output of a ray tracer. In fact, you can read more about a real implementation in the paper, "The Irregular Z-Buffer: Hardware Acceleration for Irregular Data Structures" Johnson et al. 2005.*

Good Thing He Always Wears Hoodies

Problem 5. (7 points):

In class we discussed how to implement an edge detector via discrete convolution with a filter $f(i, j)$, where the weights were specially designed for the task of edge detection. Now we're going to up-level a bit and implement a Prof. Kayvon detector.

Imagine you have a 32×32 image of Prof. Kayvon, such as the one below. Now you are given a new, 1000×1000 photo $I(x, y)$ whose contents are unknown. Assuming that (1) Prof. Kayvon always strikes this a pose in photos (he does), (2) that Prof. Kayvon always wears the same hoodie (the class claims he does), and (3) that Prof. Kayvon is always projects to a 32×32 size region in 1000×1000 photos taken of him, give an algorithm that, takes as input the image I, detects whether Prof. Kayvon is in the photo and **returns the top-left corner of a** 32×32 **pixel bounding box of the region containing Prof. Kayvon.** (Or returns nothing if the algorithm thinks Prof. Kayvon is not in the photo). Rough pseudocode is fine.

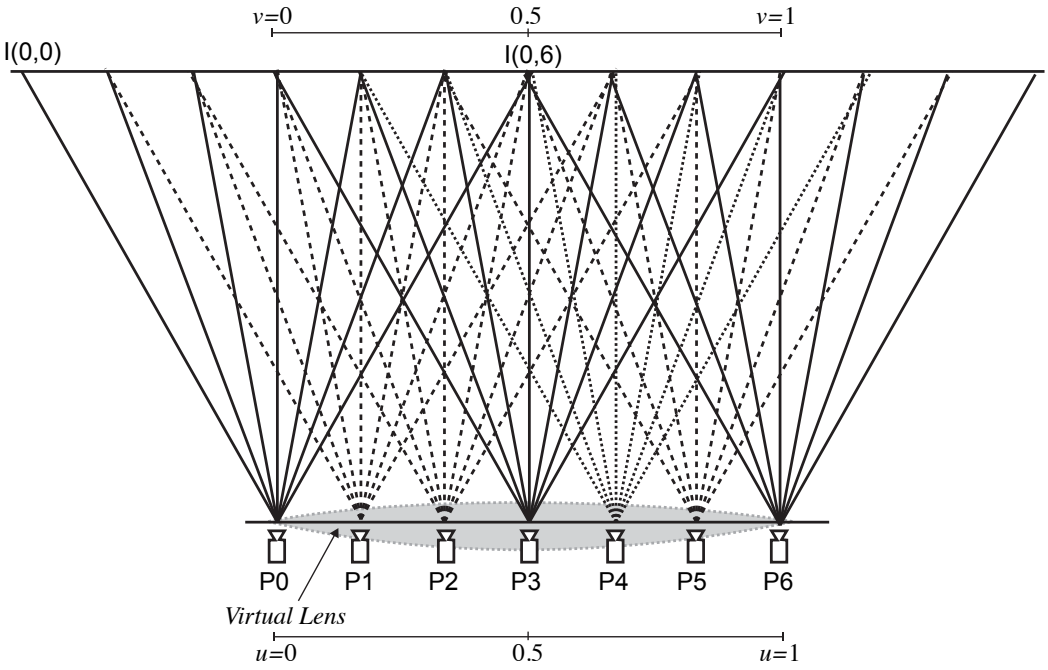
Hint: Keep it simple, several answers are possible. It may be helpful to first think about an edge detector that returns 1 if a pixel is part of an an edge, or 0 otherwise. **+2 points of extra credit for a detection algorithm that is robust to Prof. Kayvon projecting to ANY SIZED SQUARE region in the input photo.**

Solution: *Many answers are possible to this question. For example, for each pixel in the image treat the* 32×32 *region around the pixel as a vector and compute the L2-distance between this vector and the image of Prof. Kayvon. Then pick the pixel with the smallest distance (provided one exists under a detection threshold value) and use that as the center of the bounding box returned as the detection. (You have to subtract (16,16) to get the top-left corner of the detection box.)*

An easy solution to the extra credit is to downsample (or upsample) the input image to many scales (e.g., build a mip-hierarchy) and repeat the above algorithm for all scaled images.

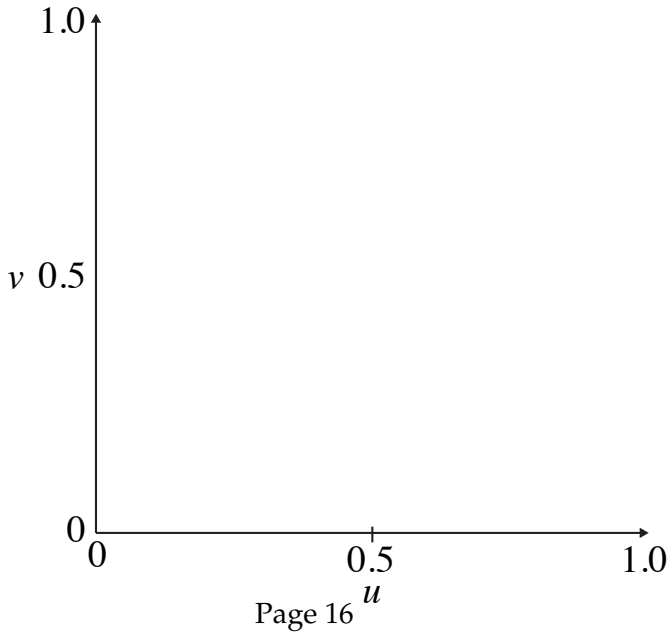
That's a Really Big Aperture You've Got There

Problem 6. (10 points):

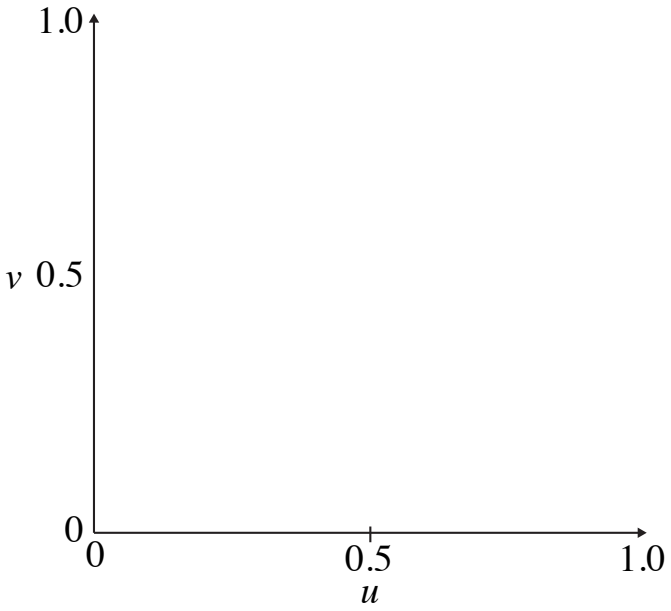


As part of studying for the final, the class decides to measure the light field in the classroom. Seven students line up along one wall at positions $P0 - P6$, and take pictures of the opposite wall with pinhole cameras as shown in the figure. Each camera is a 7-pixel camera, and the image acquried by camera i in pixel j is encoded at $I(i, j)$ (For ease of illustration we are only measuring a 2D light field with 1D cameras in this question).

A. (2.5 pts) Each pixel of each camera integrates light from a small number of rays in the light field. In the ray-space plot below, plot the rays measured by the pixels for cameras at P0, P3, and P6. (You can ignore rays that fall outside the $[0, 1]^2$ region.)

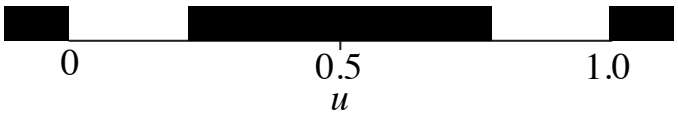


B. (2.5 pts) Kayvon and Keenan walk in and say, hey, nice work everyone. Given this light field, you can now simulate the picture that would be formed by a camera that had a **really, really large aperture** that spanned the entire u-plane wall (from $u = 0$ to $u = 1$) and **is focused on the far wall**. Give an expression that computes the value of the center pixel of such a camera in terms of the measured $I(i, j)$'s. Also plot the pixels used on the ray-space diagram below.



Solution: *You need to add up: I(0,6), I(1,5), I(2,4), I(3,3), I(4,2), I(5,1), I(6,0).* The ray-space diagram will consist of 7 points all at $v = 0.5$.

C. (2.5 pts) Given a light field, it's also possible to simulate very exotic types of cameras. The students now decide to simulate a camera whose aperture still extends from $u = 0$ to $u = 1$, but now has a blocker of light between $u = 0.25$ to $u = 0.75$. In other words, the aperture looks like the following:



Describe how to compute the center pixel of an image formed by a camera with this aperture. The camera is still focused on the back wall.

Solution: *The light from samples I(2,4), I(3,3), and I(4,2) is blocked by the new aperture, so the value of the center pixel is now the sum of I(0,6), I(1,5), I(5,1), I(6,0).*

D. (2.5 pts) Now assume that we have a much higher resolution sampling of the light field obtained by using 1000's of cameras on the near wall, each taking 1000 pixel images. We use this new light field to simulate a **camera with the same aperture as the one in the previous subproblem**, except now the simulated camera is focused on a plane a little in front of the far wall. If the far wall was all white, except for a small, red dot in the center of the wall (around $v = 0.5$), describe or give a rough sketch of the image that would result from this simulated large, (but odd) aperture camera. (Don't just say it will be blurry.)

Solution: *The image of the dot takes the shape of the aperture. Therefore, in this 1D example, the point is "burred" out to form two disjoint horizontal lines. If you answered the problem thinking of a real 2D image, you would have said the result is an image with a red ring.*

The Reflectance Equation

Problem 7. (10 points):

This semester we talked at length about the *reflectance equation*, which describes the radiance $L(\mathrm{p}, \omega_o)$ along a ray due to light reflected off a point p on a surface. (The ray's direction is given by the angle ω_o about the surface's normal at point p.) The reflectance equation is given below:

$$L(\mathrm{p}, \omega_o) = \int_{H^2} f(\mathrm{p}, \omega_o, \omega_i)\, L_i(\mathrm{p}, \omega_i)\, cos\, \theta_i \, \mathrm{d}\omega_i$$

A. (3 pts) What is role of the term $f(\mathrm{p}, \omega_o, \omega_i)$ in this equation?

Solution: *This term is the BRDF, a function that represents the ratio of energy reflected off the surface in the direction ω_o to that arriving at the surface from direction ω_i. Specifically, the BRDF provides a ratio of outgoing radiance to incident irradiance.*

B. (3 pts) Why are the limits of integration the hemisphere about the normal? (e.g., why not the whole sphere of directions? why not something less?) (Assume the surface is fully opaque.)

Solution: *We need to integrate reflected light from all incident directions, however, light incident on the opposite side of the surface will not make it through the surface (the surface is opaque). Therefore, the equation needs to only consider light incident from the hemisphere about the normal.*

C. (4 pts) Why does the reflectance equation have a $\cos\theta_i$ term in the integral?

Solution: *The* $\cos\ \theta_i$ *term is the conversion of incident radiance $L(\mathrm{p}, \theta_i)$ into incident differential irradiance on the surface. Recall the BRDF is defined to be a ratio of exit radiance in direction θ_o to incident irradiance from θ_i.*

!@#$ That Noise!

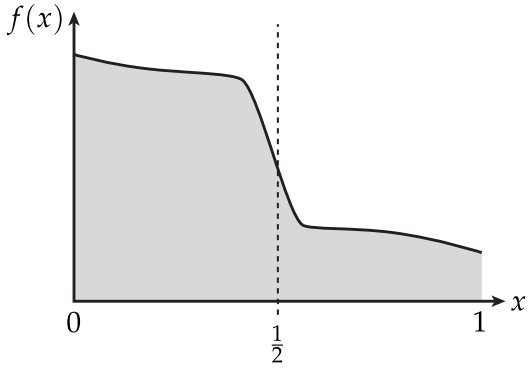
Problem 8. (15 points):

As you saw in your third assignment, noise is a pesky issue in photorealistic rendering. Fortunately, there are many techniques for reducing noise; here we will explore a few, tying together your knowledge of rendering, geometry, and numerical methods.

A. (2.5 pts) *Stratified Sampling.* The "noise" in a rendered image can be quantified in terms of the *variance* of a Monte Carlo estimator. Recall that the variance of any random variable X is defined as

$$\text{Var}[X] = E[(X - E[X])^2],$$

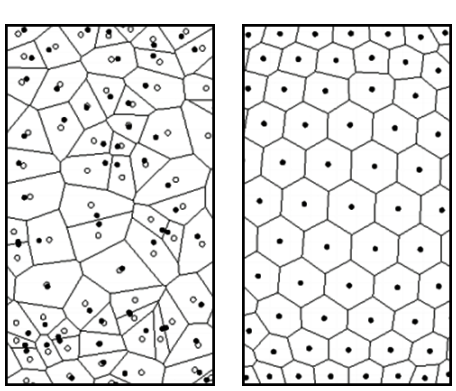
i.e., as the expected deviation from the expected value, squared. In a typical Monte Carlo estimator, we take samples uniformly across the entire domain. Recall that the basic idea behind *stratified sampling* is to instead divide the domain into several pieces, and uniformly sample within each piece. For instance, consider the following example:



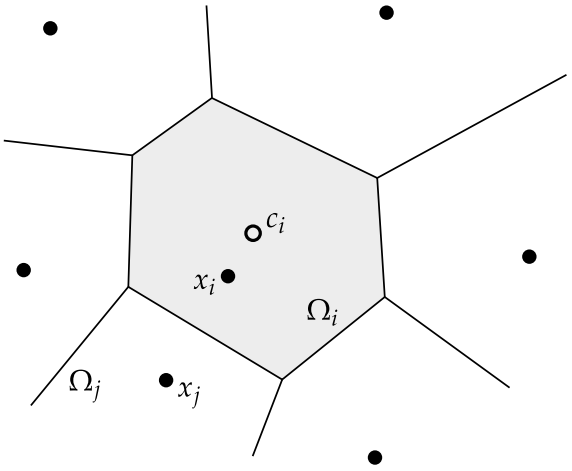
Here, we could imagine two different strategies: either (i) pick $2N$ samples uniformly from the range $[0, 1]$, or (ii) pick N samples from the range $[0, 1/2]$ and another N samples from the range $(1/2, 1]$. The former represents standard Monte Carlo; the latter represents stratified sampling. Give a rough, intuitive argument for why stratified sampling will tend to have lower variance than standard Monte Carlo, and can never have worse variance. (You do not need to do any calculations here!) What's an extreme example of a function $f : [0, 1] \to \mathbb{R}$ where the stratified strategy will do "infinitely" better than standard Monte Carlo?

Solution: *For most function, the variance in each stratum (i.e., the range of values the function can take) will be less than the variance over the whole domain. Therefore, the overall variance of the stratified estimator will never be greater than the variance of standard Monte Carlo. An extreme example is a function that is* constant *over each half of the interval—in this case, we are guaranteed to get the exact integral in just two samples using the stratified estimator, whereas the standard estimator will gradually approach the true integral as we take more and more samples.*

B. (2.5 pts) *Blue Noise Sampling.* As we just saw, getting a nice distribution of points is essential to reducing variance. Beyond stratified sampling, how can you get a good point sample in 2D? One way is to use a *centroidal Voronoi tessellation (CVT)*, which, roughly speaking, is a collection of evenly-sized cells that provides a decent "blue noise" distribution. A standard Voronoi diagram takes a collection of *sites* $x_1, \ldots, x_n \in \mathbb{R}^2$, and partitions the plane into *cells* $\Omega_i \subset \mathbb{R}^2$ such that every point x inside Ω_i is closer to x_i than to any other site. However, the site x_i can be very far from the geometric cell center $c_i := \frac{1}{A_i} \int_{\Omega_i} x \, dx$ where A_i is the cell area, as indicated in the leftmost figure below:



Voronoi *Centroidal Voronoi*



You may notice that sites in the standard Voronoi diagram have a rather nonuniform distribution, leading to a fairly noisy estimator. The idea of a *centroidal* Voronoi tessellation is to make sure every site sits exactly at the geometric center, i.e., for all cells Ω_i, we want $x_i = c_i$. To make this happen, you can run gradient descent on the energy

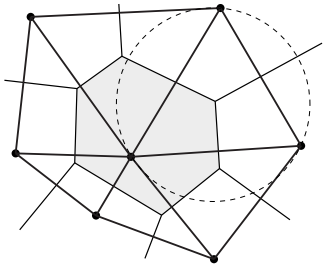
$$E_{CVT} := \tfrac{1}{2} \sum_k \int_{\Omega_k} |x_k - x|^2 dx,$$

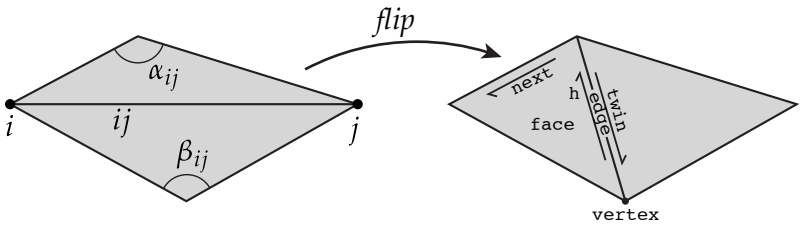
which measures the total distance from the site to all other points in the cell. Intuitively, cells that are nice, round, and "compact" will have smaller energy than cells that are long, skinny, and off-center. Show that the gradient of E_{CVT} with respect to site x_i can be expressed as $\nabla_{x_i} E_{CVT} = A_i(x_i - c_i)$ (and that therefore, local minima will satisfy $x_i = c_i$).

Solution: *Since only one term in the sum depends on x_i, we have*

$$\begin{aligned}
\nabla_{x_i} E_{CVT} &= \tfrac{1}{2} \nabla_{x_i} \textstyle\int_{\Omega_i} |x_i - x|^2 \, dx \\
&= \tfrac{1}{2} \textstyle\int_{\Omega_i} \nabla_{x_i} |x_i - x|^2 \, dx \\
&= \textstyle\int_{\Omega_i} x_i - x \, dx \\
&= \textstyle\int_{\Omega_i} x_i \, dx - \int_{\Omega_i} x \, dx \\
&= x_i \textstyle\int_{\Omega_i} dx - \int_{\Omega_i} x \, dx \\
&= A_i(x_i - c_i).
\end{aligned}$$

C. (2.5 pts) *Time to flip out!*





Before computing a *centroidal* Voronoi tessellation, we need to initialize our algorithm with a standard Voronoi tessellation. An important fact about Voronoi diagrams is that their sites define the vertices of a *Delaunay triangulation*, which in turn is any triangulation such that no vertex is contained in the interior of a circumcircle of one of the triangles. (Such triangulations have many nice properties in computer graphics, far beyond our blue noise example.) An equivalent characterization is that for each edge $ij \in E$ the sum of opposite angles α_{ij} and β_{ij} is less than or equal to π, where α_{ij}, β_{ij} are the two angles opposite ij. This characterization suggests a dead-simple algorithm for finding a Voronoi diagram: start with *any* triangulation in the plane, and keep flipping greedily until $\alpha_{ij} + \beta_{ij} \leq \pi$ for all edges. Connecting up the circumcenters of the triangles then yields a Voronoi diagram. One can show that this algorithm terminates in at most $O(n^2)$ flips, but in practice tends to be much faster. Your job is to implement this algorithm using a halfedge mesh, assuming that `Mesh::flipEdge()` has already been implemented (and does not change any pointers). Recall that in a halfedge mesh, each halfedge knows about its `next` and `twin` halfedge, and each vertex, edge, and face points to a single `halfedge`; the vertex positions are stored in `Vertex::position`.

```
double Halfedge::angle( void ) const
// returns the angle opposite this halfedge
{

   Vector3D a = next()->next()->vertex()->position;
   Vector3D b = vertex()->position;
   Vector3D c = next->vertex()->position;

   Vector3D u = b-a;
   Vector3D v = c-a;

   return atan2( cross(u,v).norm(), dot(u,v) );

}
```

```
void Mesh::makeDelaunay( void )
// greedily flips edges until the mesh is Delaunay
{

   bool isDelaunay = true;
   do
   {
      for( EdgeIter e = edges.begin(); e != edges.end(); e++ )
      {
         double alpha = e->halfedge()->angle();
         double beta  = e->halfedge()->twin()->angle();
         if( alpha + beta > M_PI )
         {
            flipEdge( e );
            isDelaunay = false;
         }
      }
   }
   while( !isDelaunay );

}
```

D. (2.5 pts) Now that you know how to make a Delaunay triangulation (and hence a Voronoi diagram), you can apply gradient descent to the energy E_{CVT} to improve the quality of your Voronoi diagram, ultimately converging on a centroidal Voronoi tessellation. You can assume that the methods `Vertex::cellArea()` and `Vertex::cellCentroid()` have already been implemented for you, returning the current area A_i and centroid c_i of the Voronoi cell associated with a given vertex of the triangulation (or equivalently, site in the Voronoi diagram). You can also assume that there is a member `Vertex::newPosition`, which can be used however you see fit.

```
void Mesh::gradientDescentCVT( double tau )
// takes a single gradient descent step of size tau on the CVT energy
{

   for( VertexIter v = vertices.begin(); v != vertices.end(); v++ )
   {
      v->newPosition = v->position - tau * v->cellArea() * ( v->position - v->cen
   }
   for( VertexIter v = vertices.begin(); v != vertices.end(); v++ )
   {
      v->position = v->newPosition;
   }

}
```

E. (2.5 pts) The sites from your centroidal Voronoi tessellation provide a good general-purpose sampling pattern for an unknown function, but suppose now that we know *a priori* which function we're going to sample. In particular, imagine that we want to use a known 2D image as an area light source. How might you modify the definition of the energy E_{CVT} to concentrate more samples in regions that are bright? How would this modification affect your algorithm for constructing the CVT?

Solution: *One simple thing we could do is weight the CVT energy by the image intensity, which would encourage smaller cells in brighter regions. To modify our algorithm, we could replace the area A_i with the total brightness covered by the cell Ω_i.*

F. (2.5 pts) The idea of concentrating samples in bright regions is one way to reduce variance, since each sample will tend to contribute more to the integrand. What's a very different strategy suggested by our discussion of stratified sampling in the first part of this question? What should the region covered by each cell of our Voronoi diagram look like, and how might this strategy be turned into CVT-like algorithm for generating a sample pattern? Consider again the example of a 2D area light source coming from a known image.

Solution: *Instead of concentrating samples in regions of high brightness, we could try to make the variance in each cell as small as possible. In the ideal case, where the image is constant over each cell, we would need only one sample per cell. We could try doing this by penalizing the variance in the image values, rather than the variance in position.* [Note to the graders: this is a high-level conceptual question, and any good effort should receive full points.]