Full Name: _____

Andrew ID: _____

15-462/662, Spring 2020

Final Exam

May 8, 2020

Instructions:

- This exam is **closed book, closed notes, closed neighbor, closed cell phone, closed telepathy, closed internet**.
- You may however use a single 3in x 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		24
2		30
3		21
4		25
Total		XXX

1 (24 points) Getting Things Started

a. (4 points) The James Webb Space Telescope (JWST) will be launched in 2021, and is 100x more powerful than Hubble—allowing us to see some of the very first galaxies in the universe. Suppose you point the JWST at a small flat asteriod 1m in diameter, and get back a 10x10 pixel image with perfectly vertical brown and beige stripes 1 pixel across. Without any further data, what can you conclude about the appearance of the asteroid's surface?

- **A.** It has vertical stripes that are *at least* 10cm wide
- **B.** It has vertical stripes that are *exactly* 10cm wide
- **C.** It has vertical stripes that are *at most* 10cm wide
- **D.** It has horizontal stripes that appear vertical due to aliasing.
- **E.** The resolution is too low to say anything about the actual signal.

SOLUTION.

Solution: Stripes larger than 10cm would be directly resolved by the image, and would appear more than 1 pixel wide. The stripes could be exactly 10cm wide, but they could also be even more narrow—appearing as 1cm stripes only due to aliasing. Horizontal stripes could not appear vertical due to aliasing; at worst they would be aliased to a constant signal. Finally, the image does contain some information about the signal: it puts a lower bound on the frequency of oscillations in the horizontal direction.

b. (4 points) Suppose you want to take a picture of a beautiful sunrise and reproduce it as accurately as possible with your inkjet printer. However, when you print it out, it doesn't look nearly as good as it did in person. Which of the following is NOT responsible for degrading the perceived quality of the color reproduction?

- **A.** The fact that your digital camera stores gamma-encoded values.
- **B.** The fact that your camera records values in the sRGB color space.
- **C.** The color of the lights used to view the final image.
- **D.** The color of the paper used by the printer.
- **E.** The fact that inkjet printer inks are not spectrally accurate.

SOLUTION.

Solution: Gamma encoding of color values (*i.e.*, applying a nonlinear remapping before storing) doesn't cause any harm; if anything it will yield better reproduction because the color intensities will be better spread out over the available numerical range.

c. (4 points) Suppose you are trying to estimate the integral of a function $f : [0,1] \to \mathbb{R}$ via Monte Carlo integration. Let $f_i := f(X_i)$, where X_i are independent random samples drawn uniformly from the interval $[0,1]$. Suppose after taking 1024 samples you compute the variance

$$\frac{1}{1024} \sum_{i=1}^{1024} \left(f_i - \tfrac{1}{1024} \sum_{j=1}^{1024} f_j \right)^2,$$

and discover that it is still a very large value. What does that tell you about the accuracy of your estimator?

A. Floating point calculations are unreliable for this problem.

B. This is simply a tough function to integrate, and you should keep sampling.

C. Nothing, this is not the correct expression for variance.

D. Nothing, this is the variance of the function f.

E. Nothing, variance in general has nothing to do with accuracy.

SOLUTION.

Solution: The expression gives the (correct) variance for the integrand f itself; this quantity is fixed and cannot be increased or decreased according to the number of Monte Carlo samples taken. For measuring accuracy, the quantity of interest is the variance of the *estimator*, not the integrand.

d. (4 points) Suppose you are integrating an ODE for a double pendulum. You notice that after just a short amount of time, the animation "blows up," *i.e.*, it rapidly gets faster, and produces ridiculous values. What can you do to make the animation more stable?

A. Simply increase the time step until it stabilizes.

B. Nothing: the double pendulum is chaotic, hence any integrator will blow up.

C. Use stratified sampling to reduce the variance of the chaotic motion.

D. Evaluate the speed function at the current (known) time step.

E. Evaluate the speed function at the next (unknown) time step.

SOLUTION.

Solution: Increasing the time step generally makes ODE integrators *less* stable. Chaotic motion cannot be resolved *accurately* over long time scales, but it can still be integrated in a way that doesn't cause the *energy* to blow up (*e.g.*, using implicit or symplectic Euler). Stratified sampling is not a relevant quantity for ODE integrators; there are no random variables involved. Evaluating the speed function at the current step yields forward Euler integration, which is unconditionally unstable. Evaluating it at the next time step yields backward Euler integration, which is unconditionally stable.

e. (4 points) Suppose you're trying to minimize a function $f(x_1, \ldots, x_n)$ to discover the most realistic parameters for a chocolate cake BRDF (yummmmm...). You start with a random set of parameters $\mathbf{x}_0 \in \mathbb{R}^n$, and apply a very simple gradient descent scheme $\mathbf{x}_{k+1} = \mathbf{x}_k - \tau \nabla f(\mathbf{x}_k)$, for some fixed step size $\tau > 0$. Your stopping condition is that the norm of the gradient $|\nabla f(\mathbf{x})|$ must fall below some threshold $\varepsilon > 0$. However, after 1,000,000 gradient descent iterations, your code is still running. Which of the following could NOT be the source of the problem? You may assume that your implementation uses fixed-precision floating point (say, all 64-bit/double precision).

A. The step size τ is too big.

B. The tolerance ε is too small.

C. The objective is not bounded from below.

D. The objective is not coercive.

E. Any of them could be the problem.

SOLUTION.

Solution: If the step size is too big, you could bounce around forever and never find a local minimum. If the tolerance is too small (say, floating point zero) then in finite precision there may not be any points where the gradient is sufficiently small. If the objective is not bounded from below, you could keep decreasing the objective forever without ever finding a local minimum; likewise if the objective is not coercive. Hence, any of them could be the problem (and which one cannot be determined purely by the fact that you don't find a local minimum).

f. (4 points) During quarantine you decided to try your hand at baking sourdough bread. To make a sourdough starter, you usually have to wait several days for yeasts and lactic acid bacteria (LAB) to overtake other competing microbes. But you had a brilliant idea, based on your knowledge of radiometry from 15-462: if you can expose the starter to *just* enough UV radiation, you can keep the yeasts and LAB alive, while killing off some of their competitors. In particular, suppose that the maximum dose of radiation they can withstand is 25 J/cm^2. Assume that you have a 25W area light source of total area 50cm^2. What's the maximum amount of time you can shine this light on your sourdough starter without killing off the "good microbes" (*i.e.*, the yeast and LAB)? You can assume the light is shining directly down on the starter (*i.e.*, the light has a Dirac delta-like distribution in the downward direction).

A. 12.5 seconds

B. 20 seconds

C. 50 seconds

D. 100 seconds

E. Cannot be determined without knowing the shape of the light source.

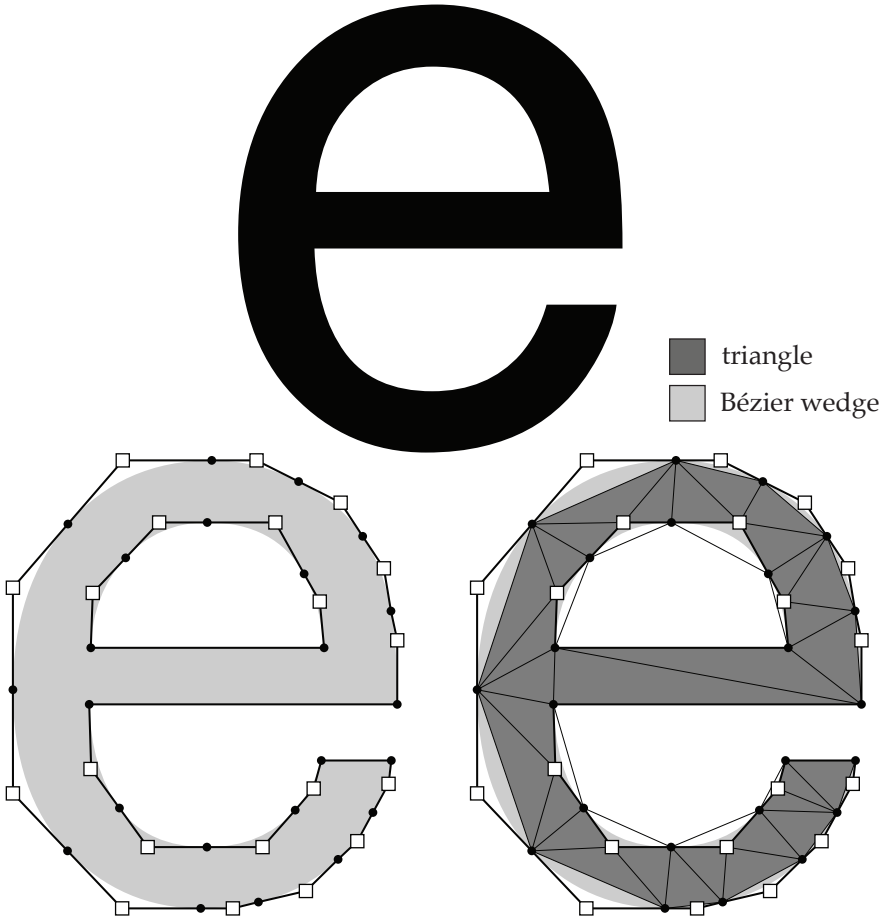
SOLUTION.

Solution: 25W of power spread over 50cm^2 has an irradiance of 0.5W/cm^2. Recall that one Watt is equal to one Joule per second. So if you run the light for 50s, you get $50\text{s} \times 0.5\,\text{J s}^{-1}\,\text{cm}^{-2} = 25\,\text{J/cm}^2$, which is just enough to start killing the good microbes.

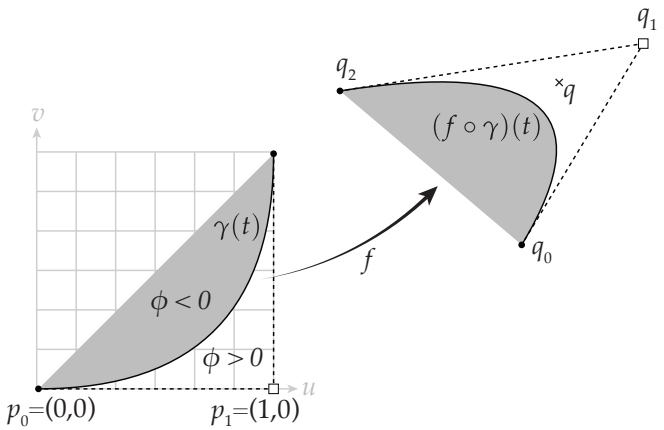
2 (30 points) Modern Font Rendering

Something we take for granted is how modern computers display beautiful, high-quality typography at blazing fast speeds. How do they do it? An old-fashioned approach is to use an incremental algorithm somewhat like Bresenham's line algorithm, but this approach isn't well-matched to real graphics architectures like GPUs. In this question, we'll compare and contrast several more modern approaches, and you'll implement (part of) one that provides a good speed/quality tradeoff.

Many modern fonts, such as *TrueType fonts*, are made up of quadratic Bézier curves (whereas others, like *OpenType fonts* are made of cubic Béziers). Consider the letter 'e' below, from the Helvetica typeface. Its boundary consists of a sequence of either quadratic Bézier curves, or straight line segments.



Our basic strategy for rendering this glyph is to break up the shape into two kinds of triangles: ordinary triangles, that fill in the interior, and what we'll call "Bézier wedges," which are used to draw the nice smooth outline. In particular, the three vertices of a triangle will be interpreted as three control points for a quadratic Bézier curve, and we'll want to fill in the region of the triangle to the "left" of the curve (assuming all curves have a consistent, counter-clockwise orientation).



To fill in an individual Bézier wedge, we'll do something similar to texture mapping:

- First, we'll rasterize the triangle as usual, *i.e.*, for each fragment, we'll check if it's inside or outside the triangle made by the three control points $q_0, q_1, q_2 \in \mathbb{R}^2$.
- If it's inside, we'll compute barycentric coordinates $(b_0, b_1, b_2) \in \mathbb{R}^3$.
- We'll use these barycentric coordinates to interpolate texture coordinates $(u_i, v_i) \in \mathbb{R}^2$ stored at the three vertices.
- Finally, we'll use these texture coordinates to fill in the Bézier wedge. But with a twist! Rather than looking up into a texture image, we'll evaluate an implicit function $\phi(u, v)$ that has a closed-form definition.
- The final color and alpha value for the fragment will be determined by whether the implicit function is positive or negative.

a. (5 points) Let f be the map that takes any point q in the triangle to its corresponding barycentric coordinates (b_0, b_1, b_2), and let g be the map that uses these barycentric coordinates to find the interpolated texture coordinates (u, v). What kind of map is $h := g \circ f$? Why is this fact important for ensuring that our rasterization scheme actually draws the correct Bézier curve?

SOLUTION.

The map h is affine, which is important because it means that testing whether points are inside/outside the reference curve is equivalent to testing whether a point is inside/outside the actual curve. In particular, a Bézier curve is an affine combination of control points, hence it doesn't matter whether we evaluate a Bézier curve in the plane and then apply an affine map, or apply the same affine map to the control points and then evaluate the Bézier curve.

b. (6 points) Consider a quadratic Bézier curve $\gamma(t) : [0, 1] \to \mathbb{R}^2$ in texture space with control points $p_0 = (0, 0)$, $p_1 = (1, 0)$, and $p_2 = (1, 1)$. Give an explicit expression for γ involving just t and no other variables.

SOLUTION.

In general, a quadratic Bézier curve has the form

$$\gamma(t) = \sum_{i=0^2} B_{2,i}(t) p_i, \tag{1}$$

where $B_{2,i}$ are the quadratic Bernstein polynomials. In particular,

$$\begin{aligned} B_{2,0}(t) &= (1-t)^2, \\ B_{2,1}(t) &= 2t(1-t), \\ B_{2,2}(t) &= t^2. \end{aligned}$$

Expanding Equation 1 yields

$$\gamma(t) = (1-t)^2(0,0) + 2t(1-t)(1,0) + t^2(1,1) = (2t(1-t) + t^2, t^2),$$

which simplifies to

$$\gamma(t) = (t(2-t), t^2).$$

c. (6 points) So far, we have an *explicit* description of the curve. To be able to evaluate whether we're inside/outside a Bézier wedge, we'll need an *implicit* expression that can be evaluated at any point. Give an expression for a function $\phi(u,v)$ that is negative inside the Bézier curve, and positive outside. "Inside" means in the grey shaded region, as depicted above; you do not need to worry about what happens for (u,v) values outside the triangle (p_0, p_1, p_2). *[Hint: in general, how would you test if a point (u,v) is above the graph of a function $v = f(u)$?]*

SOLUTION.

In general, to test if a point (u,v) is above the graph of a function $v = f(u)$, you can just check if $v > f(u)$, or equivalently, you can check if $v - f(u) > 0$. To rewrite $\gamma(t) = (t(2-t), t^2)$ in the form $(u, f(u))$, we can first solve the equation $u = t(2-t)$ for t. Using the quadratic formula, we get $t = 1 \pm \sqrt{1-u}$, and since in our case $u \in [0,1]$, we want the root $t = 1 - \sqrt{1-u}$, which also has values in $[0,1]$ (*i.e.*, in the domain of our curve γ). Substituting this expression into our original expression for $\gamma(t)$ gives $\gamma(u) = (u, (1-\sqrt{1-u})^2)$. Hence, the function

$$\phi(u,v) = (1-\sqrt{1-u})^2 - v$$

will be negative when (u,v) is above the curve.

d. (7 points) Let's put it all together, and write a routine that determines whether a given sample point q is covered by a given primitive, which can be either a triangle or a Bézier wedge. You can assume that you are given a function

```
double phi(Vec2 uv);
```

which returns the correct value for the function $\phi(u,v)$ (whether or not you got the correct answer to the previous question). You can also assume you are given a function

```
Vec3D barycentric_coordinates(Vec2 q, Vec2 q0, Vec2 q1, Vec2 q2);
```

which returns the barycentric coordinates of q relative to a triangle with vertices q_0, q_1, q_2; more precisely, it gives the normalized signed triangle areas (whether or not q is in the triangle). You should implement the routine `fragment_color` below, which takes the sample point and the three vertices as input, as well as a flag `isBezier` which is true if the triangle should be drawn as a Bezier wedge, and is false if the triangle should be drawn as just an ordinary triangle. The return value `color` of

any sample point q covered by the primitive should be bright orange (using an RGB color model with values in the range $[0,1]$), and have an alpha value of 1. If q is not covered by the primitive, it should have an alpha value of 0 (and the color does not matter).

```
void fragment_color( Vec2D q, // sample point
                     Vec2D q0, Vec2D q1, Vec2D q2, // vertices
                     bool isBezier, // true if it's a Bezier wedge
                     Vec4D& color ) output color and alpha
{
   // By default, set the output color to orange, 100% opaque
   color[0] = 1.0;
   color[1] = 0.5;
   color[2] = 0.0;
   color[3] = 1.0;

   // Get the barycentric coordinates at the sample point
   Vec3D b = barycentric_coordinates( q, q0, q1, q2 );

   // First check if we're inside the triangle; if not, then
   // we don't need to draw the triangle or the Bezier wedge,
   // and can just set the alpha to zero.
   for( int k = 0; k < 3; k++ ) {
      if( b[k] < 0. || b[k] > 1. ) {
         color[4] = 0.;
         return;
      }
   }

   // If we're drawing an ordinary triangle, we can just stop
   // here since we know all fragments inside the triangle
   // should be shaded.
   if( !isBezier ) return;

   // To rasterize a Bezier wedge, we first interpolate the
   // texture coordinates. Then we check if the point where
   // we land in texture space is covered by the standard
   // Bezier wedge.
   Vec2D uv = b[0] + Vec2D(0,0) +
              b[1] + Vec2D(1,0) +
              b[2] + Vec2D(1,1) ;
   if( phi( uv ) > 0. ) color[3] = 0.;

   return;
}
```

e. (6 points) There are other ways we could shade in a Bézier wedge. For instance, rather than evaluating an implicit function (as above), we could precompute a texture that is negative only inside the shaded region, and just look up into this texture. Or, we could simply approximate the shaded region by tessellating it into a collection of ordinary triangles. Give at least one pro and one con for each approach.

implicit function

SOLUTION.

PRO: You'll get exactly the right (analytical) solution at any resolution.

CON: You can't apply prefiltering; instead, you have to anti-alias using supersampling (or some specialized technique for the implicit function).

texture map

SOLUTION.

PRO: You can apply prefiltering to avoid aliasing.

CON: You incur significant use of texture bandwidth, which is probably a much bigger bottleneck than the arithmetic needed to evaluate the implicit function.

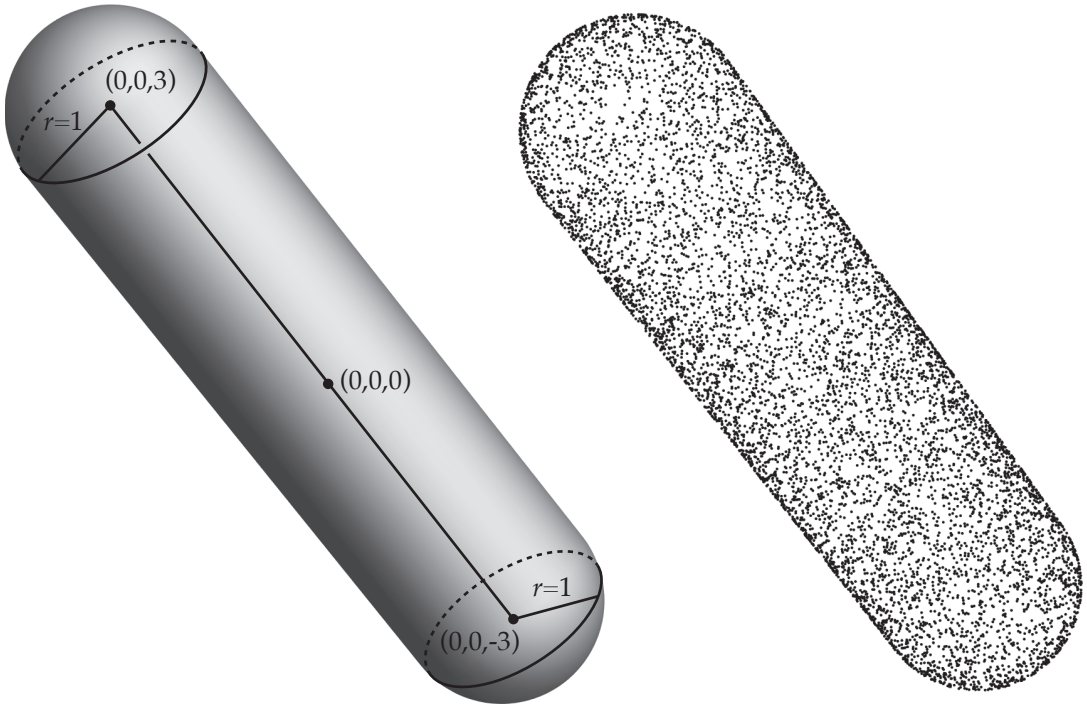
tessellation

SOLUTION.

PRO: Rasterization is simple, because you can just use existing routines.

CON: You get geometric aliasing, or else have to use a huge number of triangles adapted to the screen resolution.

3 (21 points) Sampling a Capsule



In your renderer, you want to add a capsule-shaped area light source. To do this, you'll need to be able to sample the surface of the capsule uniformly at random. We can break this down into a few easier pieces.

Throughout, you can assume that the light source is made of a cylinder of radius 1 and length 6, centered around the origin and lying along the z-axis; the ends of the cylinder are therefore at $(0, 0, \pm 3)$, as depicted in the illustration above. At each end of this cylinder there is a hemispherical cap, also of radius 1.

For the code, you can assume that you have any standard math functions (sine, cosine, *etc.*), and any of the usual operations on 3D vectors (dot, cross, *etc.*); basically anything you had available in the Scotty3D skeleton code. You may assume that you have a function `random(a,b)` that returns a random value sampled uniformly from the range $[a, b]$.

a. (7 points) Let's start out easy: just write a little piece of code that takes a uniform random sample from the cylindrical piece. Your implementation must use an *explicit* (*i.e.*, parametric) representation of the cylinder. The return value should be a point in $\mathbb{R}^3$, sitting somewhere on the surface of the cylinder.

```
Vec3D sampleCylinder()
{
   double theta = random(0,2*pi);
   double h = random(-3,3);

   return Vec3D( cos(theta), sin(theta), h );
}
```

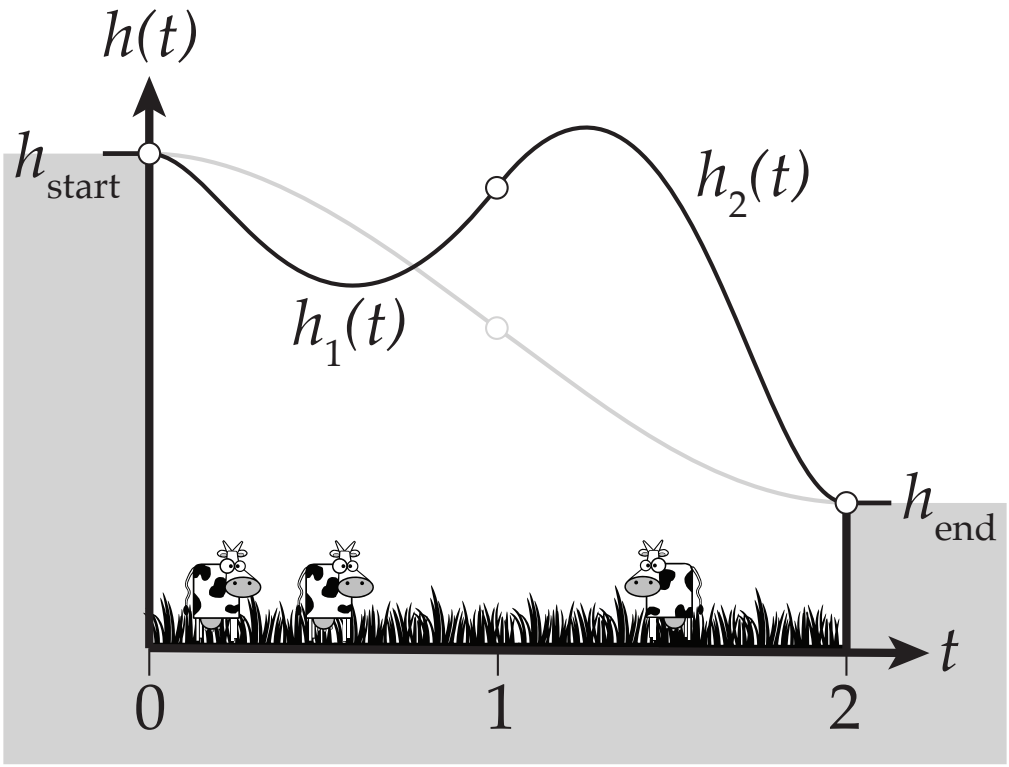
b. (7 points) Next, write a piece of code that samples *one* of the two hemispherical caps. It doesn't matter which one, but the points should land on one of the two caps (not just an arbitrary hemisphere somewhere in space). Here, your implementation must use rejection sampling.

```
Vec3D sampleCap()
{
   Vec3D p;

   do {
      p = Vec3D( random(-1,1),
                 random(-1,1),
                 random(-1,1) );
   }
   while( p.norm() > 1 )

   p /= p.norm();
   p.z = 3 + fabs(p.z);
   return p;
}
```

c. (7 points) Now let's put the pieces together into a single routine that samples uniformly from the whole surface. Your implementation should basically be a wrapper around your two previous routines, with a little logic and (biased) coin flipping to decide which routine to call. Be careful here: you don't want to sample the cylindrical piece too much relative to the hemispherical caps, and vice versa. *[Hint: what's the area of each piece?]*

```
Vec3D sampleCapsule()
{
   if( random(0.,1.) > 4./16. )
      return sampleCylinder();

   Vec3D p = sampleCap();
   if( random(0.,1) < .5 )
      p.z = -p.z;
   return p;
}
```

4 (25 points) Spline Canyon



You have been tasked with building a bridge across Spline Canyon. The challenge is that the two sides of the canyon are at *very* different heights. So, to make the ride as smooth as possible, you will design a spline that has some nice geometry. However, you only get to work with two cubic segments—so you really need to be sure to pick your constraints carefully! In particular, you will design a piecewise cubic height function

$$h(t) = \begin{cases} h_1(t), & 0 \le t < 1 \\ h_2(t-1), & 1 \le t \le 2, \end{cases}$$

where both h_1 and h_2 are cubic splines expressed in the cubic Bézier basis. In particular, we'll say that

$$h_1(t) = a_0 B_{3,0}(t) + a_1 B_{3,1}(t) + a_2 B_{3,2}(t) + a_3 B_{3,3}(t)$$

and

$$h_2(t) = a_4 B_{3,0}(t) + a_5 B_{3,1}(t) + a_6 B_{3,2}(t) + a_7 B_{3,3}(t),$$

where $B_{3,i}$ is the ith cubic Bernstein polynomial (which is defined over the interval $[0, 1]$), and $a_1, \ldots, a_7 \in \mathbb{R}$ are as-of-yet undetermined coefficients that determine the shape of the spline.

a. (6 points) First, we need to figure out what constraints will actually work. Which of the following conditions determine the spline?

A. Fix the two endpoints $h(0)$ and $h(2)$ to h_{start} and h_{end} (respectively), set the 1st derivatives at these same two endpoints to zero, and make sure the heights as well as the 1st, 2nd, and 3rd derivatives of the two segments agree at the midpoint $t = 1$.

B. Fix the two endpoints as in (A), set the 1st and 2nd derivatives at these same two endpoints to zero, and make sure the heights as well as the 1st and 2nd derivatives of the two segments agree at the midpoint $t = 1$.

C. Fix the two endpoints as in (A), set the 1st and 2nd derivatives at these same two endpoints to zero, and make sure the heights agree at the midpoint $t = 1$.

D. Let the heights of the two endpoints be free, but require that the 1st and 2nd derivatives at endpoints be equal to zero, and the heights as well as the 1st, 2nd, and 3rd derivatives of the two segments agree at the midpoint.

E. A and C.

SOLUTION.

Only A works, because the number of constraints (8) is equal to the number of degrees of freedom (8). B has one too many constraints; C has one too few constraints (7) and hence does not determine the curve. Likewise, although there are 8 constraints in D, they don't uniquely determine the curve: any curve satisfying these constraints can be shifted up and down by a constant without violating the constraints (which just means that the constraints are not linearly independent).

b. (7 points) Suppose you choose the set of constraints given in answer A from the previous problem. Write these constraints out explicitly, *i.e.*, write the equations that would have to be satisfied in terms of the variables a_i, and the constants h_{start}, h_{end}.

SOLUTION.

$$\begin{aligned}
a_0 &= h_{\text{start}} \\
a_7 &= h_{\text{end}} \\
-3a_0 + 3a_1 &= 0 \\
-3a_6 + 3a_7 &= 0 \\
a_3 &= a_4 \\
-3a_2 + 3a_3 &= -3a_4 + 3a_5 \\
6a_1 - 12a_2 + 6a_3 &= 6a_4 - 12a_5 + 6a_6 \\
-6a_0 + 18a_1 - 18a_2 + 6a_3 &= -6a_4 + 18a_5 - 18a_6 + 6a_7
\end{aligned}$$

c. (6 points) Now write the equations from the previous question as a matrix equation. Is this system of linear equations overdetermined, underdetermined, or uniquely determined? (You should assume that h_{start} and h_{end} are known, constant values.)

SOLUTION.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -3 & 3 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -3 & 3 & -3 & 3 & 0 & 0 \\ 0 & 6 & -12 & 6 & -6 & 12 & -6 & 0 \\ -6 & 18 & -18 & 6 & 6 & -18 & 18 & -6 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} = \begin{bmatrix} h_{\text{start}} \\ h_{\text{end}} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The system is uniquely determined, which can be discovered by inspecting the matrix, or by simply noting that the geometric problem is well-posed: there are 8 constraints for 8 degrees of freedom; fixing the endpoints avoids a freedom by a constant shift, and none of the constraints are in conflict (since they pertain to different derivatives).

d. (6 points) Suppose you now want to solve the linear system you setup in the previous problem, but don't have a linear solver library available. A very simple-to-implement (but slow!) algorithm for solving it is to apply gradient descent to the norm of the residual. More explicitly, suppose that the equation you want to solve is

$$\mathsf{Ma} = \mathsf{b},$$

where M and b are the matrix and right-hand side you setup in the previous problem, and a is the vector of unknown coefficients $a_1, \ldots, a_n$. The *residual* r is the difference between the left- and right-hand side:

$$\mathsf{r} = \mathsf{Ma} - \mathsf{b}.$$

The linear system will therefore be satisfied when the residual is zero ($\mathsf{r} = 0$). So, to find a solution, we can just minimize the function $f(\mathsf{a}) = ||\mathsf{r}||^2$ with respect to a, via gradient descent. Letting $\tau > 0$ be a fixed time step, and using a_k to denote the guess at step k of gradient descent, write down the update rule for gradient descent on f. I.e., write down what you would compute each time to take a gradient step.

SOLUTION.

The gradient of f with respect to a is

$$\begin{aligned} \nabla_{\mathsf{a}} f &= \nabla_{\mathsf{a}} \mathsf{r}^T \mathsf{r} \\ &= \nabla_{\mathsf{a}} (\mathsf{a}^T \mathsf{M}^T \mathsf{Ma} - 2\mathsf{a}^T + \mathsf{b}^T \mathsf{Ma} + \mathsf{b}^T \mathsf{b}) \\ &= 2\mathsf{M}^T \mathsf{Ma} - 2\mathsf{M}^T \mathsf{b}. \end{aligned}$$

So, to move in the direction opposite the gradient by a distance τ, we'd just have an update

$$\mathsf{a}_{k+1} = \mathsf{a}_k - 2\tau \mathsf{M}^T (\mathsf{Ma} - \mathsf{b}),$$

or equivalently,

$$\mathsf{a}_{k+1} = \mathsf{a}_k - 2\tau \mathsf{M}^T \mathsf{r}.$$