Stanford CS248: Interactive Computer Graphics
Participation Exercise 6

Problem 1: Defining a Quadratic (Not-Cubic) Spline

In class we talked about how **cubic splines** are common in computer graphics, but there's no reason we can't build a system based on lower degree curves instead. In this problem we are going to implement quadratic splines of the form $f(t) = at^2 + bt + c$. Consider a version of the spline where the artist specifies three control points: P_0 (the curve's position at $t = 0$), P_1 (the curve's position at $t = 0.5$), and P_2 (the curve's position at $t = 1$).

A. Please write down the **three constraint equations** for this spline (in terms of the three control point values P_0, P_1, P_2 and spline coefficients a, b and c). Remember, a constraint equation is an equation representing a constraint like "When $t = 0$ then $p(t)$ should equal the value P_0."

B. If you solve the system of three equations above from part A above for the unknowns a,b, and c (don't worry we've done it for you), you will find that:

$$a = 2P_0 - 4P_1 + 2P_2$$
$$b = -3P_0 + 4P_1 - P_2$$

We don't give you c since you should be able to figure that out from the previous part. Given this information, please write down the three basis functions $B_i(t)$ for this spline, such that:
$f(t) = P_0B_0(t) + P_1B_1(t) + P_2B_2(t)$

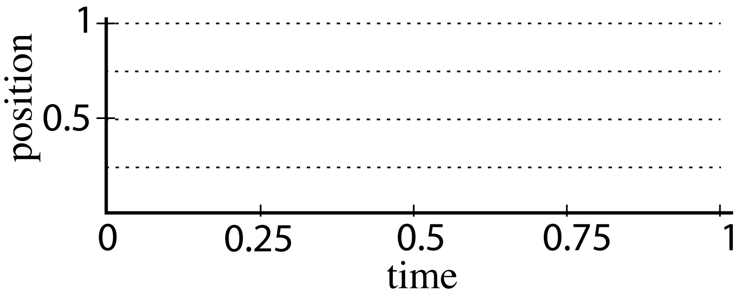
C. Using your formulation from the previous sections, now consider creating a N-segment spline out of $3N$ control points P_i. (You don't need to have answered the prior questions correctly to answer this one.) Segment 0 is defined by P_0, P_1, P_2, segment 1 defined by P_3, P_4, P_5 etc. To make sure the segments are connected, we ensure that $P_i = P_{i-1}$ for $i = 3j$ where $1 \leq j < N$. The spline is C0 contiguous by definition (the segments are connected). But does the spline have a continuous first derivative? (is it C1 continuous?)

Problem 2: More Practice with Splines

This problem is more challenging that problem 1, so make sure you "warm up" with problem 1 first.

A. In class we talked about the animation principle of ease-in, ease out, where a motion starts slowly, accelerates, then decelerates to a stop. Consider specifying the spline control points for an animation of a ball, where the spline's value at time t determines the ball's Z coordinate in the scene. Assume that you are to supply control point values at times $t_0 = 0$, $t_1 = 0.25$, $t_2 = 0.75$, $t_3 = 1.0$ (let's call these values C_i's). The spline *interpolates* the control points at t_0 and t_3, and the difference between C_1 and C_0 as well as C_3 and C_2 is used to compute curve tangents at the spline's endpoints.

   Please draw control points on the figure below that result in an ease-in, ease-out animation where the ball begins at Z=0 and ends at Z=0.75. (Hint: you want "flat" tangents.)



B. Now assume that the interior control points C_1 and C_2 can be placed at any t (not $t = 0.25$ and $t = 0.75$ like in part A). In other words, there are interior control points at (t_1, C_1) and (t_2, C_2). Write down the four constraint equations for the spline $at^3 + bt^2 + ct + d$ in terms of the (t_i, C_i)'s. Keep in mind that the spline interpolates (t_0, C_0) and (t_3, C_3) and the spline tangent at t_0 is the slope of the line between (t_1, C_1) and (t_0, C_0) (similarly for the tangent at t_3).

C. Now turn your four constraint equations into a linear system in matrix form. (Fill out the coefficients for the matrix $\mathbf{M}$, as well as the vector on the right-hand side of the equation below.)

$$\mathbf{M}\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} \qquad\qquad\qquad\qquad\qquad \\ \\ \\ \\ \end{bmatrix}\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} \qquad\qquad \\ \\ \\ \\ \end{bmatrix}$$

D. Imagine that instead of ease-in, ease-out, you instead want to make sure the position of the ball begins at Z=0.25, is located at Z=0.5 at t_1, at Z=0.75 at t_2 and ends at Z=0.5. What is the name of a cubic spline that provides a good representation for this problem? (Hermite? B-spline? Catmull-Rom? Bezier?) Why? You might need to refer back to the lecture slides for the definitions of these splines.

E. Let's return to the spline you derived in part C, which can be written in terms of the input control vector $\mathbf{c}$ (your right hand side in part C) and matrix $\mathbf{M}$, which you provided in part C as well. **(NOTE: a correct answer to part C is not necessary to do this problem.)** Here's a definition of the spline $f(t)$ as a matrix project.

$$f(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \mathbf{M}^{-1}\mathbf{c}$$

Now consider a different spline representation, such as a Bezier spline, which can represent the same spline curve $f(t)$, but using a **different basis matrix** and **different input control points** ($\mathbf{c}_{\text{bez}}$)

$$f(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \mathbf{B}\mathbf{c}_{\text{bez}}$$

Imagine you only had an implementation of Bezier splines, but you wanted to implement the same curve that was specified in part B. How could you convert your spline control points $\mathbf{c}$ from part B to a vector of Bezier control points that result in the same curve? (Hint: you should think of this question as just asking you to make a change of basis!)

F. A bit off topic, but potentially interesting... One of the benefits of a Bezier curve is that the control points forms *a convex hull* for the actual spline curve. Consider the challenge of determining if two spline curves intersect. How might you use the convex hull property to accelerate Bezier spline-Bezier spline intersection tests?

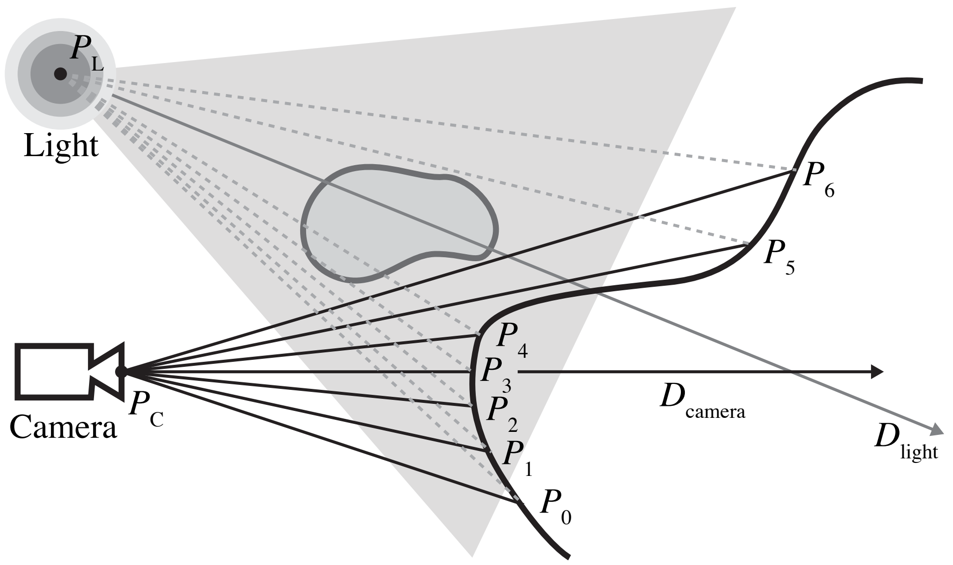
Problem 3: A Highly Irregular Rasterizer (THIS PROBLEM IS OPTIONAL AND NOT REQUIRED FOR PARTICIPATION CREDIT)

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 2D coverage sample points: in [-1,1]^2
vector<Triangle> geometry;  // list of scene triangles in WORLD SPACE
Transform worldToCam;       // 4x4 world space to camera space transform
Transform worldToLight;     // 4x4 world space to light space transform
Transform perspProj;        // 4x4 perspective projection transform


// this call returns the distance to the closest scene element from the camera
// for all points in myPoints (assume infinity if no coverage)
vector<float> depths = fancyRasterize(geometry, myPoints, worldToCam, 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. Assume you use a traditional rasterizer to compute the depth of the closest scene element 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 `worldToLight`. (Light space is the coordinate space whose origin in world space is P_L and whose -Z axis is in the direction D_{light}.) Describe an algorithm that computes, for each point P_i, if the point is in shadow from a point light source located at P_L. Your algorithm accepts as input an array of world space points P_i, world space points P_C, P_l, and has access to all variables listed in the example code. The algorithm should call `fancyRasterize` only once. (No, you are not allowed to just implement a ray tracer from scratch.)

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 describe how it converts points in world space to a list of 2D sample points in this plane. **This involves transformation, perspective projection via `perspProj`, then convert from a homogeneous 3D representation to 2D.**

B. Does the algorithm you gave above generate “hard” or “soft” shadows? Why? (You can answer this question even if you did not correctly answer part A—just assume a solution that does what was asked in part A exists.)

C. Prof. Kayvon quickly looks at the algorithm you devised above and waves his hand dismissively. He says, “remember I told you in class that shadow mapping is such a hack”, it only yields an approximation to ray traced shadows. Haotian jumps in and says, “Wait a minute here, this algorithm seems to compute the same solution a ray tracer would to me!” Who is correct? Why?