CSCI 480 - Spline Notes

We've talked about how to model (mathematically) and render (i.e., draw) straight lines. We now turn our attention to how we can model and render curved lines. Before we get to curved lines, however, we're going to build some mathematical machinery that uses straight lines, but will generalize nicely to curved lines later.

One popular tool for modeling curves is called a **spline**; supposedly, this term comes from shipbuilding tools used back in the day to create curved surfaces such as the hulls of ships. A number of pegs would be set in position, then a thin strip of metal woudl be bent around the pegs to create a smoothly curving line. Here, the curve's overall trajectory is **controlled** by the placement of the pegs, while the "smoothness" of the curve is determined by the physical properties of the metal. You could imagine stiffer metal creating a curve with different properties than a more flexible or thinner metal.

We will use analogous mathematical tools to model these curves in computer graphics. In particular, we would like to give a modeler control over the curve by allowing them to specify **control points**. We will then decide on some smooth mathematical form that determines the shape of the curve as it travels among control points; in our case, we're going to use low-degree polynomials to ensure that the path is smooth.

Not all desired curves can be modeled with low-degree polynomials; but higher-degree polynomials are quite twitchy to work with, and may not end up particularly smooth; for this reason, we will model more complex curves by stringing together segments that are made of low-degree polynomials.

Preliminaries: Linear Interpolation

We've seen several representations of straight lines so far, and a few of them will be useful to think about here, so let's review. For now, let's assume that we have two points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$ that define our line segment.

Implicit equations for a line include slope-intercept form:

$$y = mx + b \tag{1}$$

and

$$ax + by + c = 0 \tag{2}$$

Meanwhile, we have the parametric form

$$r(t) = \mathbf{p} + t\mathbf{d} \tag{3}$$

Let's build some intuition by deriving some relationships among these, using our specific points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$.

In the case where we want to talk about the line between two points, we can write a parametric form that begins at $\mathbf{p}_0$ and travels along the direction vector between the two points $(\mathbf{p}_1 - \mathbf{p}_0)$:

$$\mathbf{r}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0) \tag{4}$$

Parametric to Linear Interpolation

From here, we can move towards the standard way that we write linear interpolation:

$$\begin{aligned}\mathbf{r}(t) &= \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0)\\ &= \mathbf{p}_0 + t\mathbf{p}_1 - t\mathbf{p}_0\\ &= (1-t)\mathbf{p}_0 + (t)\mathbf{p}_1\end{aligned}$$

This form makes it clear that this function is $\mathbf{p}_0$ at $t=0$, $\mathbf{p}_1$ at $t=1$, and a linear combination of the two points at values of t in between. In fact, for situations like this where we really just want to talk about the segment between the two points, we have a convention to use the parameter variable u instead of t - this (by convention only) signifies that we only expect to work with values of u within the range 0 to 1:

$$\mathbf{f}(u) = (1-u)\mathbf{p}_0 + u\mathbf{p}_1 \tag{5}$$

Degree-1 Polynomials: Piecewise Linear "Curves"

To model curves that are more complicated than lines, we could use increasingly complicated mathematical functions. Take the example of polynomials: a line is a degree-1 polynomial, and it is flexible enough only to model a straight line. A degree-2 polynomial can model any parabola (including a straight line if the x^2 coefficient is 0), and the curves only get more expressive as the polynomial degree increases. However, they also get more unstable and unweidly; a degree-100 polynomial is extremely hard to work with and its shape is extremely sensitive to its coefficients.

An easier way to model complicated curves is to use **piecewise** polynomial segments. In the case of degree-1 polynomials, this looks like a linestring: a sequence of points, where each consecutive pair is connected by a line segment. If we want well-behaved curved segments that are more flexible, we can increase the polynomial degree (e.g., to 2 or 3); chaining many of these together gives us a lot of expressiveness without the fiddliness of high-degree polynomials.

Representations: Control Points vs Polynomial Coefficients

If you were looking to model a linestring, I'd argue that the above linear interpolation representation is one of the most intuitive ways to do it. You supply a sequence of points, and each consecutive pair has a line segment running between them. Using the above interpolation equation, the coordinates of any point along each line segment can be computed by an appropriate choice of u between 0 and 1.

If you were rendering that linestring, this wouldn't be a bad representation either. However, it's not obvious how we would change this equation to interpolate a degree-1 polynomial into one that interpolates a degree-2 polynomial (or higher). So we're going to add what seems like a lot of unnecessary structure to the straight line case, but the payoff is that we can straightforwardly generalize to other types of polynomials later.

Let's write down a parametric equation for a line in a way that makes it glarily obvious that it's a degree-1 polynomial in u:

$$\mathbf{f}(u) = u^0\mathbf{a}_0 + u^1\mathbf{a}_1 \tag{6}$$

Let me sell you on this polynomial representation first. Notice that the above equation is equivalent to the following:

$$\mathbf{f}(u) = [u^0 \quad u^1] \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a_1} \end{bmatrix} = \vec{u}^T \vec{\mathbf{a}} \tag{7}$$

This means that if we know the $\mathbf{a}$'s, then computing the coordinates of the point at a chosen u reduces to a dot product. That's nice! The polynomial coefficients give a nice and convenient form for evaluating the points on the line, and that's a handy thing for actually drawing the line.

However, for *modeling* purposes, the $\mathbf{p}$'s are much more convenient to work with. You might notice that, since u^0 is 1, the polynomial form reduces to $\mathbf{f}(u) = \mathbf{a}_0 + u\mathbf{a}_1$, which is a change of variable names away from our original parametric ray, and based on that it's pretty easy to figure out how the polynomial coefficients $(\mathbf{a}_0, \mathbf{a}_1)$ relate to the control points $(\mathbf{p}_0, \mathbf{p}_1)$: $\mathbf{a}_0$ is equal to $\mathbf{p}_0$, while $\mathbf{a_1}$ is $\mathbf{p}_1 - \mathbf{p}_0$. So if you were asked to specify the polynomial coefficients, you'd be giving two different kinds of things - a point, and the vector from that point to the other one; that's not as user-friendly as just giving the two points themselves. As you'll see, this difference in user-friendliness gets much more pronounced as we move away from lines.

So to summarize, **polynomial coefficients** (the $\mathbf{a}$'s') are convenient for rendering, while **control points** (the $\mathbf{p}$'s) are convenient for modeling. The next thing we need is a general strategy for converting between the two.

From Control Points to Polynomial Coefficients

What I know about the control points for a line segment is that they appear at the beginning and end of the line segment. In other words,

$$\begin{aligned} \mathbf{p}_0 &= \mathbf{f}(0) \\ \mathbf{p}_1 &= \mathbf{f}(1) \end{aligned} \tag{8}$$

Plugging these values of u into the dot product equation above, we get:

$$\begin{align} \mathbf{p}_0 &= \mathbf{f}(0) \tag{9} \\ &= [0^0 \quad 0^1] \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a_1} \end{bmatrix} \tag{10} \\ &= [1 \quad 0] \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a_1} \end{bmatrix} \tag{11} \\ &= \mathbf{a_1} \tag{12} \end{align}$$

I can do that with $\mathbf{p_1}$ as well:

$$\begin{align} \mathbf{p}_1 &= \mathbf{f}(1) \tag{13} \\ &= [1^0 \quad 1^1] \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a_1} \end{bmatrix} \tag{14} \\ &= [1 \quad 1] \begin{bmatrix} \mathbf{a}_0 \\ \mathbf{a_1} \end{bmatrix} \tag{15} \\ &= \mathbf{a_1} + \mathbf{a_2} \tag{16} \end{align}$$

That gives me the control point if I know the coefficients; usually, I'm given the control points and I want the coefficients, so the useful thing would be

$$\mathbf{p}_1 = \mathbf{a}_0 + \mathbf{a}_1 \tag{17}$$
$$\mathbf{a_1} = \mathbf{p}_1 - \mathbf{a}_0 \tag{18}$$
$$\mathbf{a_1} = \mathbf{p}_1 - \mathbf{p}_0 \tag{19}$$

We've now arrived at what we knew already. But let's just add a tiny little bit *more* seemingly unnecessary structure to this. Notice that I plugged two values of u in above to generate two $\vec{u}$ vectors. If I stack these into a matrix, I can compute both control points at once:

$$\begin{bmatrix}\mathbf{p}_0\\ \mathbf{p}_1\end{bmatrix} = \begin{bmatrix}\mathbf{f}(0)\\ \mathbf{f}(1)\end{bmatrix} = \begin{bmatrix}1 & 0\\ 1 & 1\end{bmatrix}\begin{bmatrix}\mathbf{a}_0\\ \mathbf{a}_1\end{bmatrix} \tag{20}$$

The extra-nice thing about putting this into a matrix is that if we want to go the other way (get polynomial coefficients from control points), we can do this by inverting the matrix! We'll write the above in matrix notation as:

$$\vec{\mathbf{p}} = C\vec{\mathbf{a}} \tag{21}$$

so that

$$C^{-1}\vec{\mathbf{p}} = \vec{\mathbf{a}} \tag{22}$$

The matrix C is important enough to have a name - it's called the control matrix; its inverse is called the **basis** matrix, and we'll call it B:

$$\vec{\mathbf{a}} = B\vec{\mathbf{p}} \tag{23}$$

If I did the matrix inversion and calculated this out, I would get

$$\begin{bmatrix}\mathbf{a}_0\\ \mathbf{a}_1\end{bmatrix} = \begin{bmatrix}\mathbf{p}_0\\ \mathbf{p}_1 - \mathbf{p}_0\end{bmatrix} \tag{24}$$

which is exactly what we would expect based on the non-matrix version above, and in fact based on the ray form of the line outline all the way back in the Preliminaries section.

Evaluating Points on the Curve

Now that we have the polynomial coefficients $\vec{\mathbf{a}}$, we can evaluate any point on the curve with the same dot product as we saw above:

$$\mathbf{f}(u) = \vec{u}^T\vec{\mathbf{a}} \tag{25}$$

where $\vec{u}^T = [u^0 \quad u^1]$. We can also put these together to get an expression that goes all the way from control points to any point on the curve:

$$\mathbf{f}(u) = \vec{u}^T B\vec{\mathbf{p}} \tag{26}$$

Degree-2 Polynomials: Quadratics

This was all a lot of machinery in order to draw line segments, but now it's a little easier to see how we could generalize it to higher-degree polynomials. Let's model a quadratic. Our first design decision is: what should I use as control points? For a line segment, the endpoints are a natural choice. For a parabola, we could make various choices; for now, let's give the user control over the start, middle, and end of the curve. In other words, $\mathbf{p}_0$ is the point on the parabolic segment at $u = 0$, $\mathbf{p}_1$ is the point at $u = 0.5$, and $\mathbf{p}_2$ is the point at $u = 1$. Let's write this out just like we did above:

$$\begin{bmatrix}\mathbf{p}_0\\\mathbf{p}_1\\\mathbf{p}_2\end{bmatrix} = \begin{bmatrix}\mathbf{f}(0)\\\mathbf{f}(0.5)\\\mathbf{f}(1)\end{bmatrix} = \begin{bmatrix}0^0 & 0^1 & 0^2\\0.5^0 & 0.5^1 & 0.5^2\\1^0 & 1^1 & 1^2\end{bmatrix}\begin{bmatrix}\mathbf{a}_0\\\mathbf{a}_1\\\mathbf{a}_2\end{bmatrix} \tag{27}$$

This yields the control matrix C:

$$C = \begin{bmatrix}1 & 0 & 0\\1 & 0.5 & 0.25\\1 & 1 & 1\end{bmatrix} \tag{28}$$

whose inverse is the basis matrix B:

$$C^{-1} = B = \begin{bmatrix}1 & 0 & 0\\-3 & 4 & -1\\2 & -4 & 2\end{bmatrix} \tag{29}$$

So just as above, if we have the control points $\vec{\mathbf{p}}$ and a value of u, the polynomial coefficients are $\vec{\mathbf{a}} = B\vec{\mathbf{p}}$, and the position of the point on the curve at u is directly computable as $\mathbf{f}(u) = \vec{u}^T B\vec{\mathbf{p}}$.