CS3388B: Lecture 7

February 2, 2023

7 Drawing in 2D using Open GL: Part 3

The end of 2D drawing has two main, and related, topics: tweens and splines.

7.1 Lerps and Tweens

A **linear interpolation** is a simple operation. Given two vectors A and B (positions, sizes, attributes, etc.) we can define a function which smoothly *blends* one vector into the other. The **lerp** defines a simple function $P(t)$ where $P(0) = A$ and $P(1) = B$:

$$P(t) = A(1 - t) + Bt$$

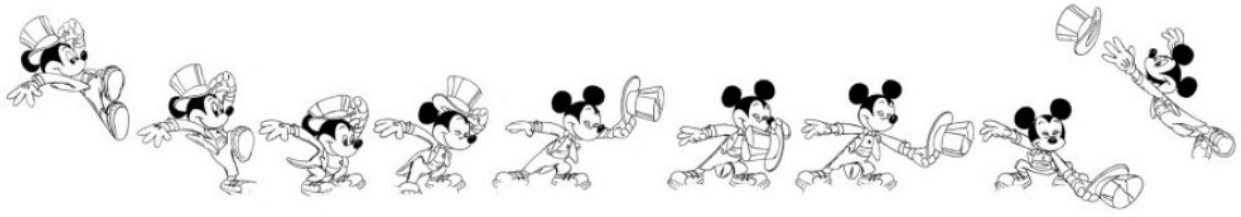
i

Another way to think of this blending is that linear interpolation is providing a value that is some weighted average of A and B. At $t = 0$ the value is 100% A. At $t = 1$ the value is 100$ B. At $t = 0.5$ the value is exactly half A and half B.

When a lerp is applied to *positions*, it is called **tween** for *in-between*.

Tweens are particularly useful for animation. For a particular object with N vertices, we can define tweens for each vertex independently. Applying each vertex's tween simultaneously gives the impression of the whole figure being animated.

Of course, we must define many **key frames** which are the "end points" of each tween. We tween from frame i to $i + 1$ for $t = 0...1$. Then, we tween from frame $i + 1$ to $i + 2$ for $t = 0...1$ again. Repeat forever and you have a **key frame animation**.



7.2 Tweens in OpenGL

Consider two polyline drawings (or any set of primitives) with the same number of vertices. We can define a tween between them by creating a bijection (a one-to-one mapping between the vertices. Then, we lerp each vertex independently.

Let's give it a try.

```
std::vector<float> star = {0.0f, 10.0f, 2.5f, 2.5f, 10.0f, 2.5f, 4.0f, -2.
    5f, 7.0f, -10.0f, 0.0f, -5.0f, -7.0f, -10.0f, -4.0f, -2.5f, -10.0f, 2.5
    f, -2.5f, 2.5f, 0.0f, 10.0f};

float v1x = 10* 0.5, v1y = 10*1.5388, v2x = 10*1.30902,
      v2y = 10*0.951056, v3x = 10*1.618034, v3y = 0;
std::vector<float> poly={v1x, v1y, v2x, v2y, v3x, v3y, v2x, -v2y, v1x, -v1
    y, -v1x, -v1y, -v2x, -v2y, -v3x, v3y, -v2x, v2y, -v1x, v1y, v1x, v1y};

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-20, 20, -20, 20, -1, 1);

float deltaT = 0.005, t = 0, vx, vy;

/* Loop until the user closes the window */
while (!glfwWindowShouldClose(window))
{
    //...
    glBegin(GL_LINE_STRIP);
    for (int i = 0; i < star.size(); i +=2) {
        vx = lerp(star[i], poly[i], t);
        vy = lerp(star[i+1], poly[i+1], t);
        glVertex2f(vx, vy);
    }
    glEnd();

    if (t > 1.0f || t < 0.0f) {
        deltaT *= -1.0f;
    }
    t += deltaT;
    //...
}
```

7.3 Non-linear tweens

The classic lerp equation $A(1-t)+B(t)$ can be viewed as a *split of unity*.

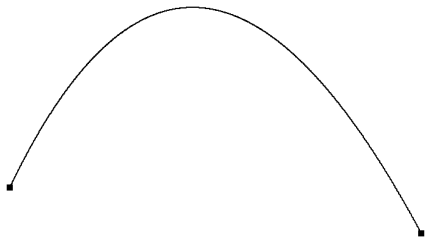
For any t between 0 and 1, $(1-t)+t=1$. This is a two-way partition. What if we partition 1 in three ways?

$$1 = ((1-t)+t)^2$$
$$1 = (1-t)^2 + 2(1-t)t + t^2$$

Now, instead of a linear interpolation we can create a smooth **quadratic approximation** between three points, A, B, C. This smooth interpolation is a **Bezier curve** of degree 2 and can be described by a parametric formula $P(t)$.

$$P(t) = (1-t)^2 A + 2t(1-t)B + t^2 C$$

In this case, we again have $P(0) = A$ and $P(1) = C$. Moreover, there is no value t' such that $P(t') = B$. Rather, B acts as a *control point* which influences the interpolation between A and C. One can feel that B acts as a sort of "gravity" for the interpolation.



A most common and practical tween is **cubic approximation**, producing a Bezier curve of degree 3. As before, we will split unity. This time, into 4.

$$1 = ((1-t)+t)^3$$
$$1 = (1-t)^3 + 3(1-t)^2 t + 3(1-t)t^2 + t^3$$

Cubic interpolation thus occurs between four points, A, B, C, D. Again, the idea is to define a function so that $P(0) = A$ and $P(1) = D$, while B and C act as control points.

$$P(t) = (1-t)^3 A + 3(1-t)^2 tB + 3(1-t)t^2 C + t^3 D$$

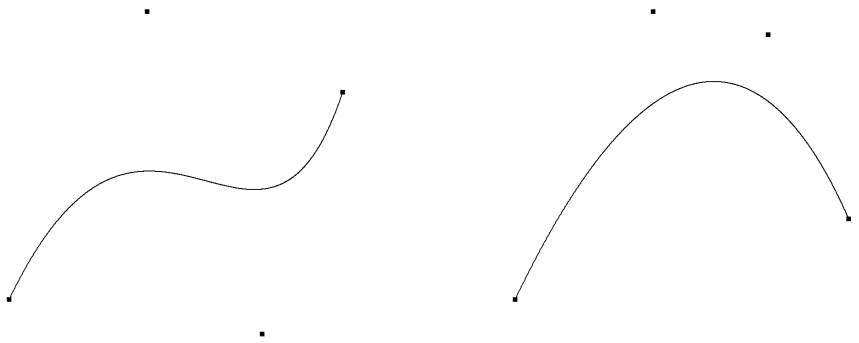


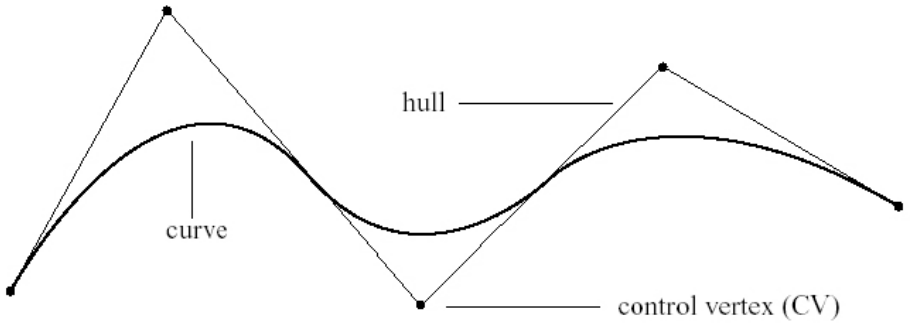
Figure 1: Two different degree 3 Bezier curves.

7.4 Splines

Consider the points in space: A, B, C, D, E where we want each pair to be connected. As polylines, this is easy, just draw the line segments (A, B), (B, C), (C, D), (D, E). But what if we want to use Bezier curves to connect each pair of points?

This problem needs to be solved in areas like computer-aided design, different typefaces, and every image/video manipulation program. The *pen tool* of photoshop, for example, draws splines.

In graphics, the goal of splines is to create a curve in space which connects many dots in sequence. Basically a polyline, but multiple curves instead of multiple line segments.



I'll try not to go too deep into the math on this one. Splines are really just functions defined *piecewise* by polynomials. A **Bezier Spline** is a spline where each piece is a Bezier curve.

Consider a slightly formal definition. We look to define a *continuous* function $S : [a, b] \mapsto \mathbb{R}$.[1] Formally, we say that $S \in C^0$, the set of continuous functions. The interval $[a, b]$ will be split

[1] In practice, with two dimensions, we actually define two functions: one controlling the x coordinate and one controlling the y coordinate.

into many sub-intervals, and each interval defined by a particular polynomial. Since S should be continuous, the intermediate end points of the polynomials must be equal.

Let's say we have k such uniform subintervals. Thus, we have $[t_0, t_1), [t_1, t_2), \ldots, [t_{k-1}, t_k]$ with $a = t_0$ and $b = t_k$. We will define a Bezier curve $P_i(t)$ for each interval.

$$S(t) = \begin{cases} P_1(t), & a \leq t < t_1 \\ P_2(t), & t_1 \leq t < t_2 \\ P_3(t), & t_1 \leq t < t_3 \\ \vdots & \\ P_k(t), & t_{k-1} \leq t < b \end{cases}$$

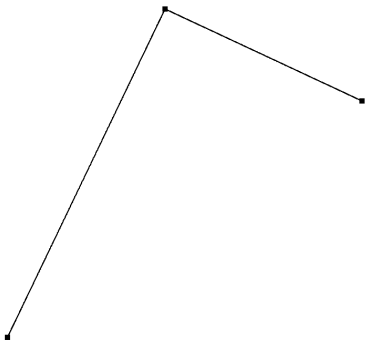
Since each Bezier curve is usually defined on the interval [0, 1], we have to do a little more work. We must *scale* the range $[t_i, t_{i+1}]$ to [0, 1]. That's quite easy. For $P_i(t)$ let $t' = \frac{t-t_i}{t_{i+1}-t_i}$ and then compute $P_i(t')$.

But, in practice, particularly in computer graphics graphics where we just use Bezier splines to draw curves, we can just let the interval for each P_i be [0, 1].

Let's do an example.

Consider the points A, B, C that we want connect using cubic Bezier curves. One from A to B, and another from B to C. Without any control points, we would just use straight lines to connect them. This is the same result as letting each control point equalling the closer endpoint. We can draw each piece of the spline independently and simultaneously using t in the range [0, 1].

$$P_1(t) = (1-t)^3 A + 3(1-t)^2 t c_1 + 3(1-t)t^2 c_2 + t^3 B$$
$$P_2(t) = (1-t)^3 B + 3(1-t)^2 t c_3 + 3(1-t)t^2 c_4 + t^3 C$$



Now we want curves and non-trivial control points. If we specify those control points ahead of time, it's easy:

```
float Ax = -15, Ay = -10;
float Bx = -3, By = 15;
float Cx = 12, Cy = 8;

float c1x = -12, c1y = 9;
float c2x = -7, c2y = 17;
float c3x = 1, c3y = 3;
float c4x = 9, c4y = -5;

glPointSize(10.0f);
glBegin(GL_POINTS);
    glVertex2f(Ax, Ay);
    glVertex2f(Bx, By);
    glVertex2f(Cx, Cy);
glEnd();
glPointSize(2.0f);
glBegin(GL_POINTS);
    for (int i = 0; i < N; ++i) {
        float t = (float) i;
        t /= N;
        glVertex2f(bezier4(Ax,c1x,c2x,Bx,t), bezier4(Ay,c1y,c2y,By,t));
        glVertex2f(bezier4(Bx,c3x,c4x,Cx,t), bezier4(By,c3y,c4y,Cy,t));
    }
glEnd();
```

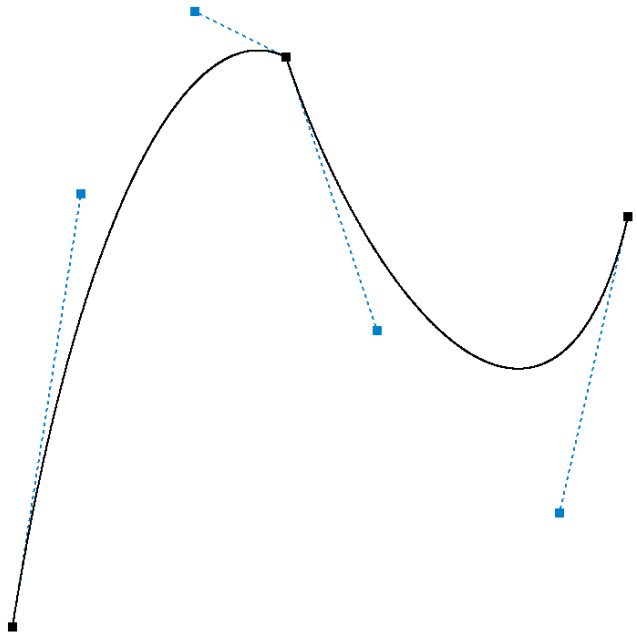


Figure 2: My second spline.

While this spline is continuous, it is no particularly *smooth*. Look at what happens right before the middle point B and right after. There's a sharp crease in the curve. Typically, we like to avoid that in splines. Mathematically, this is caused by the derivative of the spline $S(t)$ not being continuous, there is a sharp jump in the slope of $S(t)$.

We could get into the mathematics of it, or, we could just give the intuition. For an interior point on a Bezier spline: if the control point just before it, the interior point itself, and the control point just after it all fall on a straight line, then the spline will have a continuous first derivative and the curve will appear more "smooth". Clearly this is not the case in Figure 2.

Let's calculate the line connecting control point 2 and B. Recall that the formula for the line between two points is:

$$y - y_1 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_1)$$

$$c_2 = (-7, 17) \quad B = (-3, 15)$$

$$y = \frac{15 - 17}{-3 + 7}(x + 7) + 17 = \frac{-x}{2} + \frac{27}{2}$$

So, we should move c_3 to also be on this line. Let's keep its x-coordinate the same and compute its correct y coordinate as:

$$c_{3,y} = \frac{-1}{2} + \frac{27}{2} = 13$$

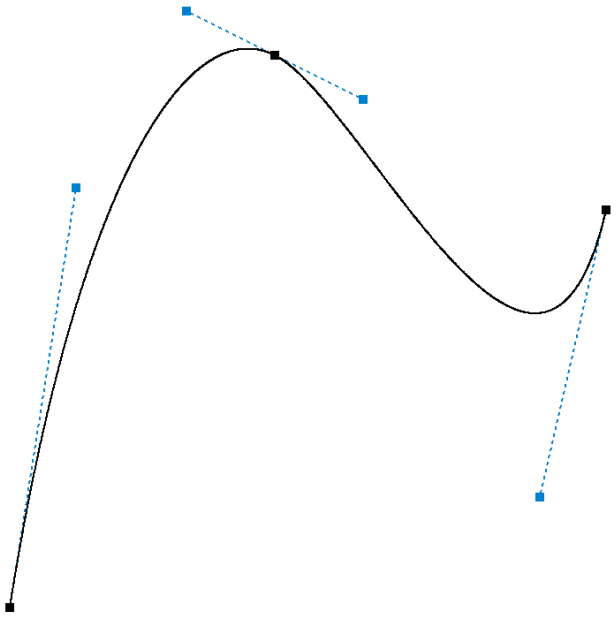


Figure 3: My smooth third spline.