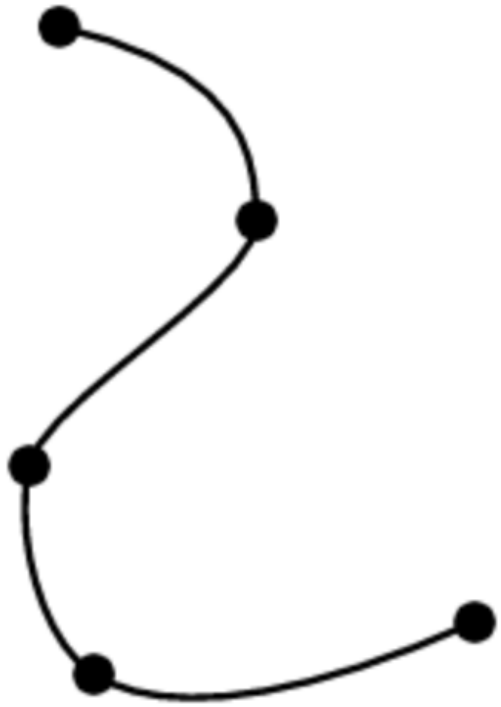
Notes for November 13 class -- Spline curves

Splines

There are many reasons we might want to create smooth controllable curves in computer graphics. Perhaps we might want to create an organic shape, or animate something along a continuous path.



As you can see on the right, we can do this by breaking down our smooth curve into simpler pieces.

If we think of the spline curve as a path of motion, each of these pieces must match its neighbors in both position and rate, which means that for each coordinate x and y, we need four values: a position at both the start and the end, and a rate (or derivative) at both the start and the end.

The smallest order polynomial that can satisfy four constraints is a cubic. So we describe both the x and y coordinates of each piece using *parametric cubic* polynomials in parameter t:

$$x = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y = a_y t^3 + b_y t^2 + c_y t + d_y$$

where (a_x, b_x, c_x, d_x) and (a_y, b_y, c_y, d_y) are constant valued polynomial coefficients, and t varies from t = 0 to t = 1 along the curve.

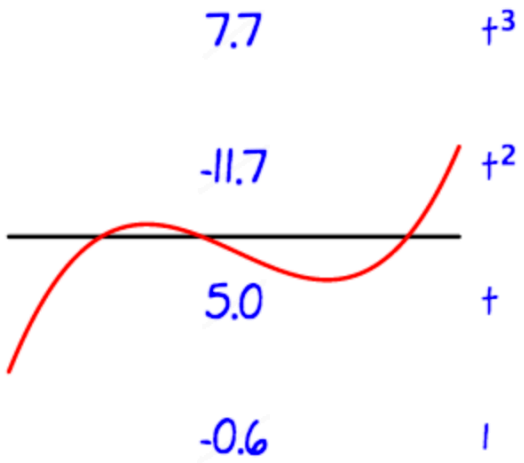
Cubic splines

Although it is technically true that cubic splines can be designed by tweaking their polynial coefficients, in practice that doesn't usually work out very well.

As you can see in the example on the right, there is no intuitive connection between the shape of a cubic polynomial and the values of its four coefficients -- in this case 7.7, -11.7, 5.0 and -0.6, respectively, for the t^3, t^2, t and constant term.

For that reason, we need a better way to specify cubic spline curves. Rather than use t^3, t^2, t and 1 as our four *basis functions*, we can use four different basis functions that have a more intuitive geometric meaning.

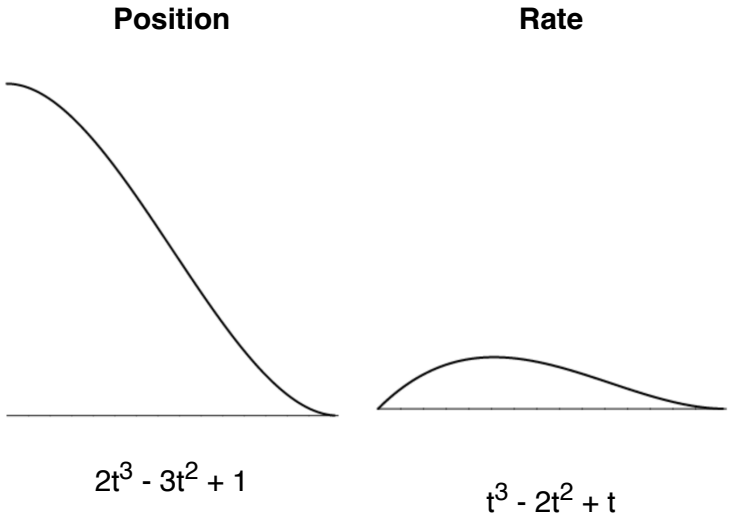
In the following sections we will look at various alternate basis functions.

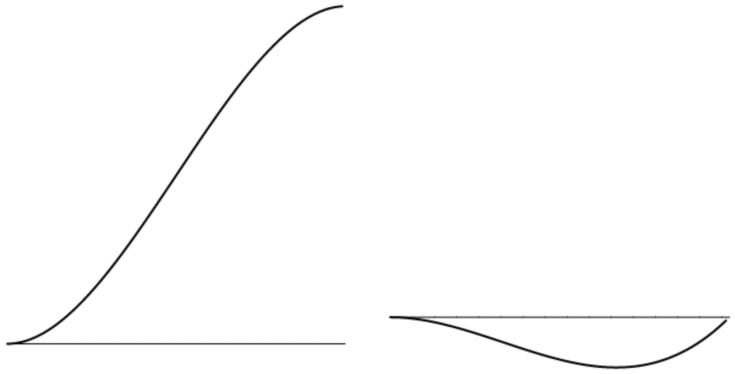


Hermite splines, part 1

We can choose four basis functions that give us independent control over the position at t = 0 and t = 1, as well as the rate of change at t = 0 and t = 1. This is called the *Hermite* basis, after the french mathematician who devised it.

If we want a curve with position P_0 at t = 0, P_1 at t = 1, rate R_0 at t = 0, and rate R_1 at t = 1, we can use the four functions to the right to compute the cubic polynomial we are looking for.





Hermite splines, part 2

Because these four hermite basis polynomials never change, and the cubic polynomial we want is just a weighted sum of those four polynomials, we can express this weighted sum as a multiplication of the weights by a matrix, which we call the *Hermite matrix*.

In other words, the expression:

$$P_0 (2t^3 - 3t^2 + 1) + P_1 (-2t^3 + 3t^2) + R_0 (t^3 - 3t^2 + t) + R_1 (t^3 - t^2)$$

can be expressed as a matrix vector product:

$$\begin{matrix} a \\ b \\ c \\ d \end{matrix} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} P_0 \\ P_1 \\ R_0 \\ R_1 \end{matrix}$$

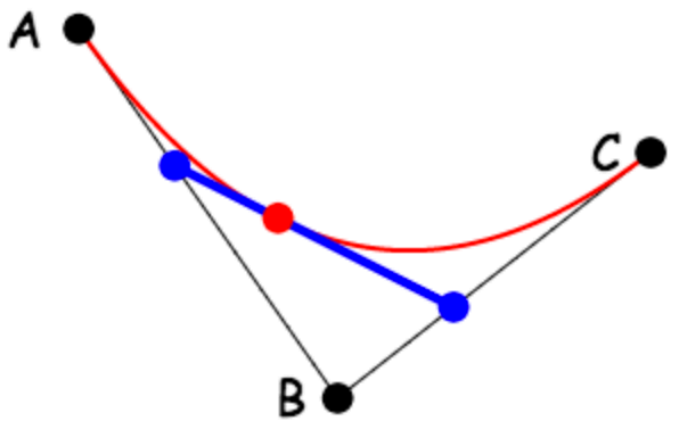
to convert positions and rates at the two ends into the desired cubic polynomial:

$$at^3 + bt^2 + ct + d.$$

Bezier splines, part 1

Artists and designers often find it more convenient to create splines by moving points around, rather than needing to deal with derivatives. The *Bezier* spline enables designers of spline curves to work this way.

Bezier splines work by repeated linear interpolation. For example, the image to the right shows a simplified version of a Bezier spline, using three key points to specify a parametric quadratic spline.



Note that points along the curve are found as a linear interpolation of linear interpolations. We first find points along the edges AB and BC by linear interpolations (to get the points represented as blue dots):

$$(1-t)\,A + t\,B$$

$$(1-t)\,B + t\,C$$

and then we interpolate again (to get the point represented as a red dot):

$$P = (1-t)\,(\,(1-t)\,A + t\,B\,) + t\,(\,(1-t)\,B + t\,C\,)$$

If we multiply out all the terms, we get:

$$A\,(1-t)^2 + 2\,B\,(1-t)\,t + C\,t^2$$

Note that the weights of the coefficients (1 2 1) follow Pascal's triangle.

Bezier splines, part 2

Now it becomes easier to see what is going on with the full parametric cubic Bezier spline, which uses four key points: The basic set-up is a linear interpololation of linear interpolations of linear interpolations.

So we start with the points in blue:

$P = (1-t)\ A + t\ B$

$Q = (1-t)\ B + t\ C$

$R = (1-t)\ C + t\ D$

When the first and second terms are linearly interpolated, we get the two dots in violet:

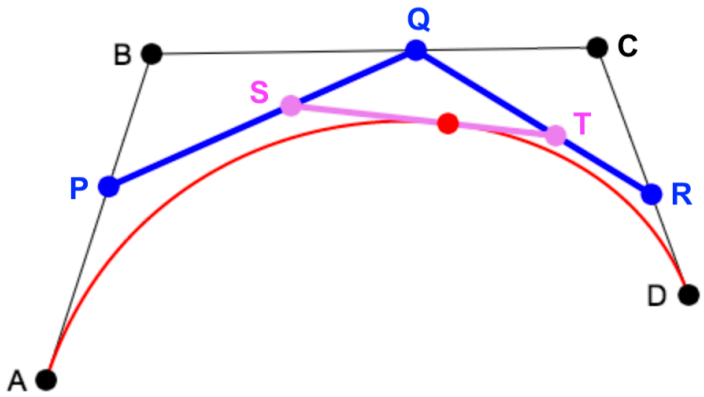
$S = (1-t)\ P + t\ Q$

$T = (1-t)\ Q + t\ R$

Finally we linearly interpolate these two points: $(1-t)\ S + t\ T$

When we multiply everything out, writing the equation in terms of our original four key points A,B,C and D, the weights form the next level (1 3 3 1) of Pascal's triangle:

$$A\ (1-t)^3 + 3\ B\ (1-t)^2\ t + 3\ C\ (1-t)\ t^2 + D\ t^3$$



Bezier splines, part 3

We can multiply out the terms of the above polynomial, and regroup by powers of t, to get:

$$(-A + 3B - 3C + D)\ t^3 + (3A - 6B + 3C)\ t^2 + (-3A + 3B)\ t + D$$

This makes it easy to see that, as was the case for Hermite splines, the Bezier spline has a characteristic matrix, which can be used to translate the above polynomial until the standard cubic polynomial, with coefficients, (a,b,c,d):

a		-1	3	-3	1	A
b	=	3	-6	3	0	B
c		-3	3	0	0	C
d		1	0	0	0	D

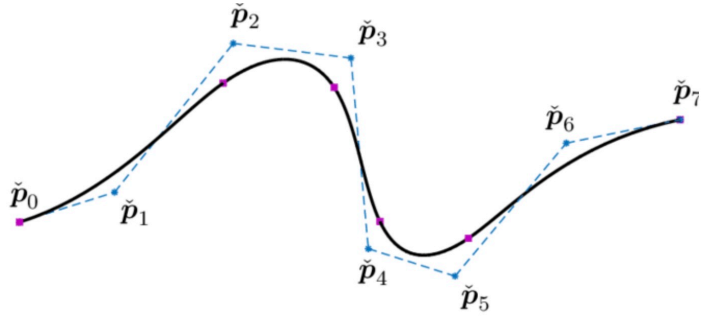
One powerful property of Bezier splines is that the direction between A and B determines the direction of the spline curve at t=0, and the direction between C and D determines the direction of the spline curve at t=1.

This makes it easy to match up splines end-to-end, so that the resulting composite curve has a continuous derivative.

B-splines, part 1

B-splines are a way to create a very smooth non-interpolating spline. The curve does not go through the keypoints, but the resulting curve has second order continuity (continuity not just of derivative, but also of curvature).

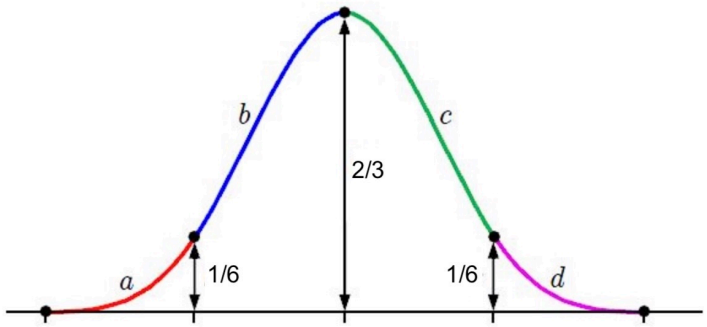
Note that in the example to the right, control points p1 through p6 are not on the spline curve itself.



B-splines, part 2

The characteristic matrix for a B-spline looks like this:

a	=	1/6	**-3**	**3**	**-3**	**1**	B_0
b			**3**	**-6**	**3**	**0**	B_1
c			**-3**	**0**	**3**	**0**	B_2
d			**1**	**4**	**1**	**0**	



B_3

to convert positions and rates at the two ends into the desired cubic polynomial:

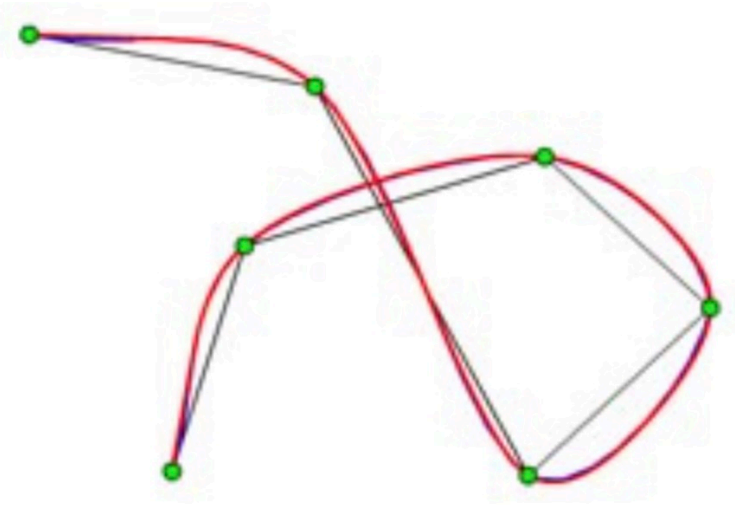
$at^3 + bt^2 + ct + d.$

Catmull-Rom splines, part 1

Catmull-Rom splines are a way to create interpolating splines. The curve goes through every keypoint.

Note that in the example to the right, the spline goes through every key point.

Also, note that the resulting curve is not as smooth as a B-spline curve.



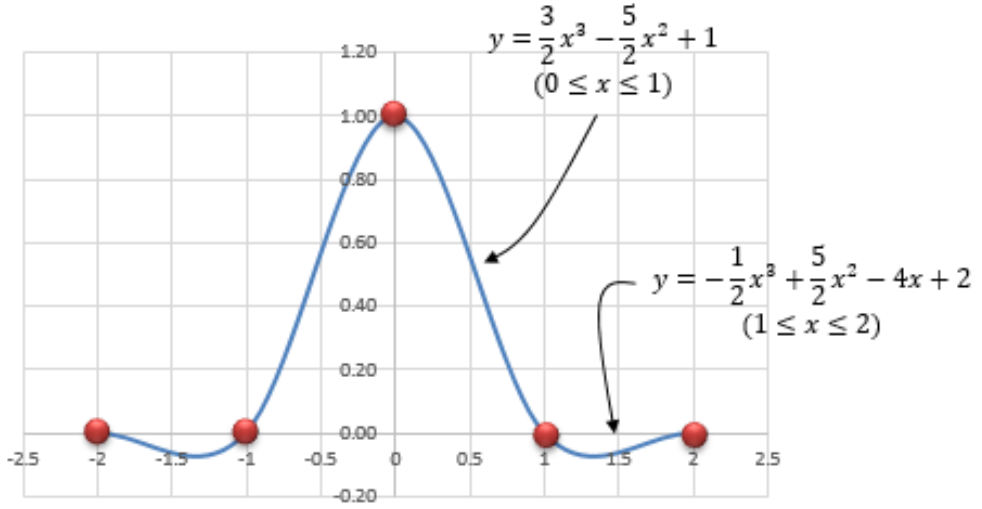
Catmull-Rom splines, part 2

The characteristic matrix for a Catmull-Rom spline looks like this:

a		-.5	1.5	-1.5	.5	A
b	=	1	-2.5	2	-.5	B
c		-.5	0	.5	0	C
d		0	1	0	0	D

to convert positions and rates at the two ends into the desired cubic polynomial:

$at^3 + bt^2 + ct + d.$




```
let catmullRom = (K, t) => {
   let n = K.length - 1, i = floor(n * t), f = (n * t) % 1;
   let A = K[max(0, i-1)], B = K[i], C = K[i+1], D = K[min(n, i+2)];
   return ((((-A+3*B-3*C+D) * f + (2*A-5*B+4*C-D)) * f + (-A+C)) * f + 2*B) / 2;
}
```

Homework, due by start of class on Wednesday November 20

1. Animate the shapes you have already made by using time-varying spline curves, using any of the types of splines we have discussed. You can use the values of the splines to feed into the parameters of the translate, rotate and scale primitives that you have already implemented.

   Use your spline-based animation software to make interesting and compelling animations.

2. Create your own curve editor, to create Hermite, Bezier, B-spline or Catmull-Rom based curves. Your editor should allow users to add, move, and delete key points, and should allow users to specify whether two adjoining spline curves have matching derivatives.

   Use your editor to create interesting shapes, such as outlines of animals, letters from fonts, or something that you think would be cool and fun.