Week 3 : Curves & Surfaces

Goal : To introduce various methods of creating and using curves

Topics: parametric curves, Bezier curves, Hermite curves, b-splines/NURBS, parametric surfaces

Parametric curves

A “parametric” curve is a curve that is parameterized by a function with a dependent variable.

That is, for a function f(t, **P**),
providing a number t results in a unique position vector P

ex: Line
the equation y = mx + b can be parameterized by t like so:

$\mathbf{P}_x = x_0 + v_x t$
$\mathbf{P}_y = y_0 + v_y t$

where
x_0 and y_0 are the intial values of x and y
v is a vector that is parallel to the line

Thus using these equations we can determine the position of a point on the line at any specific t value.

Parametric line

example, if you have the line equation:

$y = 2x + 2$

you can parameterize it:

A vector parallel to this line would be $v = (1, 2)$

And we can set the initial x value to 0

and the initial y value to the y-intercept = 2;

$\mathbf{P}_x = 0 + 1t$

$\mathbf{P}_y = 2 + 2t$

so for the values $t = \{0, .25, .5, .75, 1\}$

we get the points $\mathbf{P} = \{(0, 2), (.25, 2.5), (.5, 3), (.75, 3.5), (1, 4)\}$

(draw the functions $f(t, \mathbf{P}_x)$ and $f(t, \mathbf{P}_y)$)

Parametric curves

ex: circle

$(x - a)^2 + (y - b)^2 = r^2$
where (a,b) = the center of the circle and r = the radius

can be parameterized by t using sine and cosine like so:

$x = a + r\cos(t)$
$y = b + r\sin(t)$

If we step through values of t between 0 and 2pi we will approximate a circle.

If we add another parametric equation for the 3rd dimension, we get a helix

$x = a + r\cos(t)$
$y = b + r\sin(t)$
$z = rt$

(draw $f(t, \mathbf{P}_x)$ and $f(t, \mathbf{P}_y)$ for circle and $f(t, \mathbf{P}_x)$, $f(t, \mathbf{P}_y)$ and $f(t, \mathbf{P}_z)$ for helix)

Bezier curves

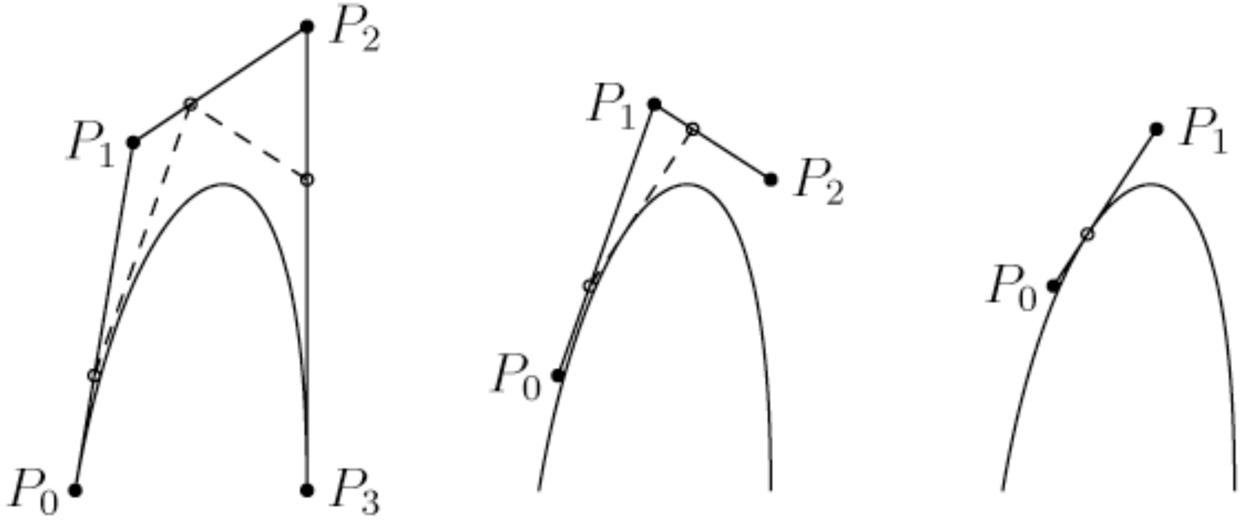
Bezier curves are simple parametric curves that are defined by two endpoints and some number of control points.

They are usually found in one of two flavors: quadratic (with a single control point) or cubic (with two control points), although there is no reason you couldn't have more control points.

The control point(s) define the position function of the curve as the variable t changes.

An algorithm created by DeCasteljau provides a simple mechanism for creating these curves parametrically.

DeCasteljau's algorithm



Bezier curves

Bezier curves with 2 control points are called *cubic* curves, because the mathematical "blending" function that defines the curve from the control points is a cubic polynomial.

$(1 – t)^3 P_0 + 3t(1 – t)^2 P_1 + 3t^2(1 – t) P_2 + t^3 P_3$

Where t is between 0 and 1.

That is, this equation defines the amount each point contributes to the curve as t moves from 0 to 1.

At $t = 0$, P_0 contributes 100%

At $t = 1$, P_1 contributes 100%

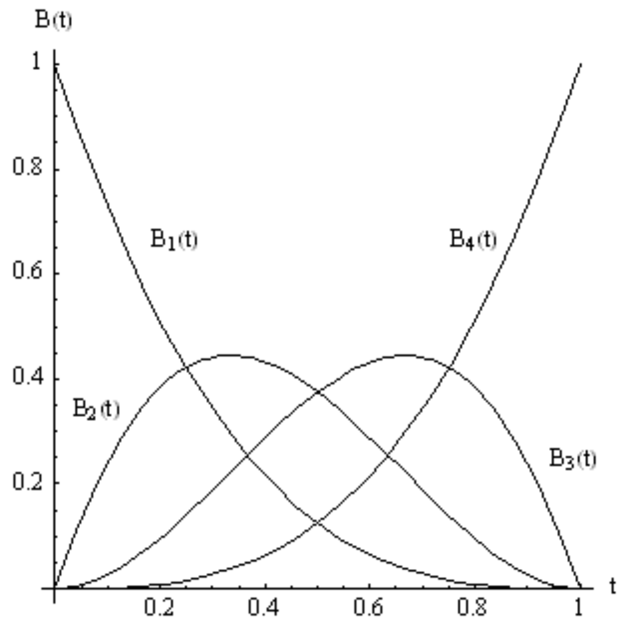
At $t = .5$, P_0 contributes 12.5%, P_1 cs 37.5%, P_2 cs 37.5%, P_3 cs 12.5%

etc.

Bezier curves

And it turns out that at each point the total sum of the contribution is 100%.

Moreover, at time 0 and time 1 a single point account for the full contribution of the curve, which means that the curve begins and end on a control point.



Also, the blending is symmetrical about t = .5

Bezier curves

Since we know the blending functions we can shorten our equation to:

$P(t) = B_0(t)\ P_0 + B_1(t)\ P_1 + B_2(t)\ P_2 + B_3(t)\ P_3$

or just

$P(t) = \Sigma B_i P_i$

In practice, you can calculate each dimension of the curve separately. I.e. :

$P_x(t) = (1 - t)^3\ P_{x,0} + 3t(1 - t)^2\ P_{x,1} + 3t^2(1 - t)\ P_{x,2} + t^3\ P_{x,3}$
$P_y(t) = (1 - t)^3\ P_{y,0} + 3t(1 - t)^2\ P_{y,1} + 3t^2(1 - t)\ P_{y,2} + t^3\ P_{y,3}$
$P_z(t) = (1 - t)^3\ P_{z,0} + 3t(1 - t)^2\ P_{z,1} + 3t^2(1 - t)\ P_{z,2} + t^3\ P_{z,3}$

Hermite curves

A similar curve that also lets you create a curve around set endpoints is called a Hermite curve. In the Bezier curve you move the two control points to influence the curve. In a Hermite curve you instead adjust the *tangents* at the endpoints to control the curve.

Hermite curves are also known as “the pen tool” in Illustrator.

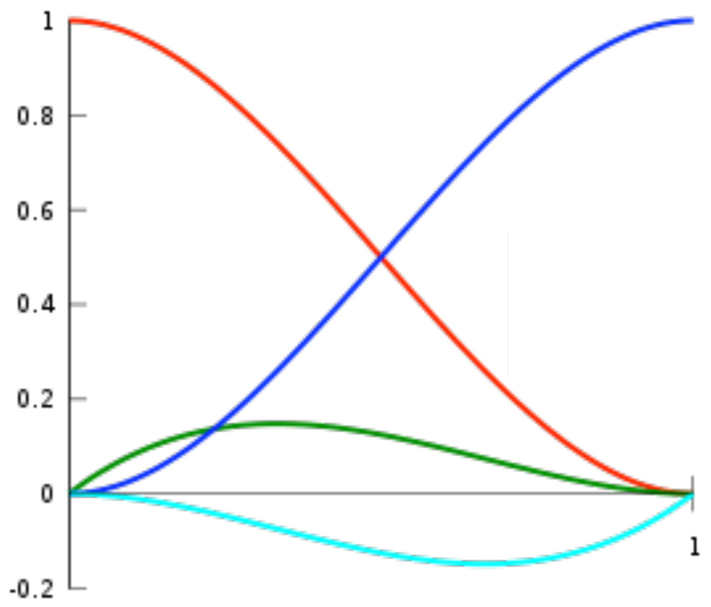
The curve is parametrically defined by the following equation:

$$P(t) = (2t^3 – 3t^2 + 1)P_0 + (t^3 – 2t^2 + t)M_0 + (t^3 – t^2)M_1 + (-2t^3 + 3t^2)P_1$$

where P0 and P1 are the endpoints of the curve, and M0 and M1 are tangent vectors originating at P0 and P1 respectively.

Hermite curves

Here is the graph of the blending function for the two endpoints. (The red and blue indicate the blending for the points and the green and turquoise for the tangents)



Catmull-Rom splines

Nice because the curve actually goes through the points that define the curve! (often used for camera animations)

Continuity

Obviously both these simple cubic parametric curves suffer from the fact that you are limited to the kinds of curves you can generate. That is, they have at most one inflection point.

To increase the range / loopiness of your curve you can either increase the order/complexity of your curve by adding more control points.

Or you can stitch together a series of simpler curves.

Continuity

When stitching together curves there is generally a trade-off between ease of use and flexibility. A simple naming scheme is used to describe the power of the different techinques to join together curve pieces;

C^0 = the curves share an end point, but the end point may look discontinuous, sharp

G^1 = the curves share an end point and those end points have the same tangent.

C^1 = same as G1 except that the tangent vector is also required to have the same magnitude

C^2 = the curves share an end point, the tangent is the same, and also the second derivative (representing curvature or acceleration) is the same.

C^∞ = all derivates of the curve are the same.

Generally strive for C^2 as it looks good and if we are using the curve for animation it guarantees that both the velocity and the acceleration are the same.

B-splines

B-splines are a more general way to think about curves. The “B” stands for the *basis* which defines the blending functions. The phrase “b-spline” is often used to describe a certain category of continuously connected curve pieces.

B-splines guarantee C2 continuity, but at the price of a increased complexity and the loss of some control (none of the points go through the control points!)

B-splines come in three flavors: Uniform, Nonuniform, and Nonuniform Rational.

The latter is also called NURBS (NonUniform Rational B-Splines)

Uniform b-splines

The basic b-spline is defined like so:

Given a set of n+1 control points, the b-spline curve is composed of n-2 cubic curves pieces.

Each piece is defined again by a blending of 4 points, where for each piece Q_i,

$Q_i(t = 1) = Q_{i+1}(t = 0)$

$Q'_i(t = 1) = Q'_{i+1}(t = 0)$

$Q''_i(t = 1) = Q''_{i+1}(t = 0)$

That is, the joins are C^2 continuous. These joins are called *internal knots*.

Uniform b-splines

For each curve piece Q_i,

$Q_i(t) = B_0(t) + B_1(t) + B_2(t) + B_3(t)$

where the blending functions are the following:

$B_0(t) = [\ (1 – t)^3 / 6\ ] * P_0$
$B_1(t) = [\ (4 – 6t^2 + 3t^3) / 6\ ] * P_1$
$B_2(t) = [\ (1 + 3t + 3t^2 + 3t^3) / 6\ ] * P_2$
$B_3(t) = [\ t^3 / 6\ ] * P_3$

Uniform b-splines

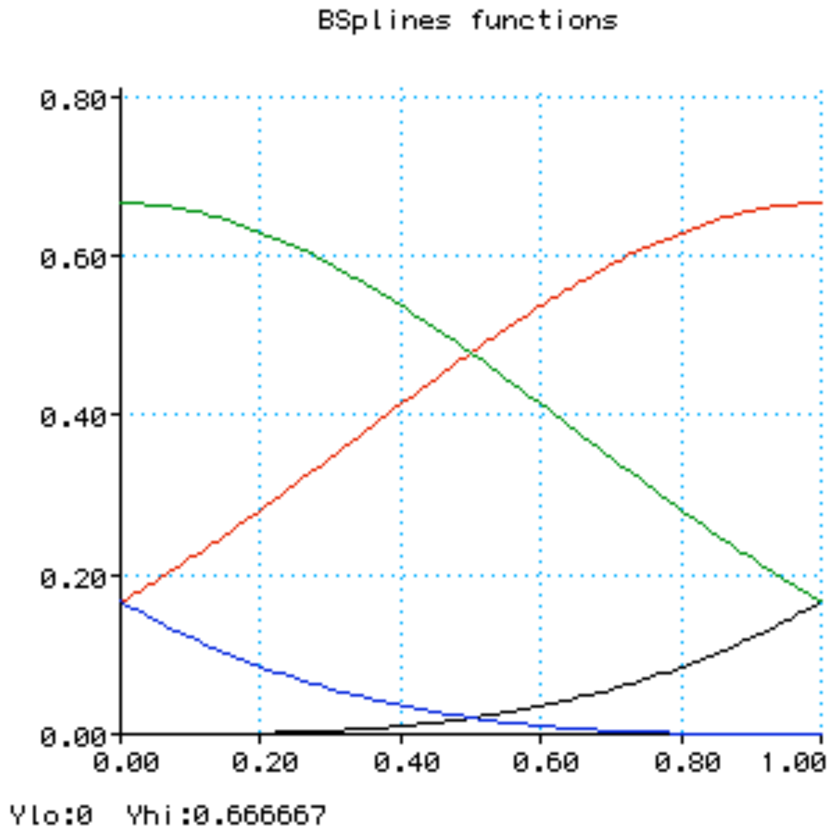
$B_0(t) = [\ (1 - t)^3 / 6\ ] * P_0$

$B_1(t) = [\ (4 - 6t^2 + 3t^3) / 6\ ] * P_1$

$B_2(t) = [\ (1 + 3t + 3t^2 + 3t^3) / 6\ ] * P_2$

$B_3(t) = [\ t^3 / 6\ ] * P_3$

which looks like this:



Nonuniform b-splines

In uniform b-splines, the knots are automatically positioned at equal distances along the curve.

In a nonuniform b-spline, we can space the knots at nonuniform locations along the curve, which can change the curve in various ways. For instance, we can create looped curves, and perfect circle arcs.

To create a nonuniform cubic b-spline we need to define a knot-vector to describe the knot spacing.

If our curve uses 7 control points, then we will need

(# of control points + degree of curve + 1) knots = 7 + 3 + 1 = 11 knots

If the knot vector is (0,1,2,3,4,5,6,7,8,9,10) then the curve will reduce to the uniform b-spline.

Cox-deBoor algorithm

Each piece is defined like so:

$Q_i(u) = \Sigma B_{i+k-1,d}(u)P_{i+k-1}$ (sum as k goes from 0 to 3)

Where
the range of u is defined by the knot vector (the val at t_0 -> val at $t_{\#knots}$)
d = the degree parameter (which is the degree + 1, or 3 + 1 for cubic b-spline)

The blending functions are recursively defined like so:

$B_{i,0}(u) = 1$ if $t_{i-2} <= u <= t_{i-1}$, otherwise 0

$$B_{i,k}(u) = (u - t_{i-2})\ [N_{i,k-1}(u) / t_{i+k-2} - t_{i-2}] +$$
$$(t_{i-+k-1} - u)\ [N_{i+1,k-1}(u) / t_{i+k-1} - t_{i-1}]$$

(ie, input $B_{i+k-1,d}(u)$ from curve algo above and calc from reduced dimensionality)

NURBS

Similar to nonunifrom b-splines, except that you can specify a weight to every control point.

$Q_i(u) = \Sigma B_{i+k-1,d}(u) P_{i+k-1} W_{i+k-1}$

The weight is thought of as 4th component to the point, making it a point in homogeneous coordinates.

The weights have the effect of shifting the curve in the direction of the weighted control point

(whereas moving knots close together has the effect of causing the curve to converge upon a control point.)

NURBS

With a nonuniform b-spline you can insert new control points as desired and update the knot-vector appropriately. This will let you have more control over a specfic range, while leaving the curve controlled by points further away the same.

Also, you can repeat the beginning and end values of your knot vector (order) times which will force your curve to line up with the control point endpoints

NURBS surfaces

defined by a mesh of control points, which define two sets of curves – a column of curves and a row of curves...

(demo)

Curves in OpenGL / GLU

1. create a NURBS object

```
 gluNewNurbsRenderer()
```

2. indicate that you are using the NURBS renderer

```
gluBeginCurve(nurbsObj)
```

3. draw the curve

```
gluNurbsCurve(
    lengthOfKnotVector, //how many knots
    knotVectorArray, //the knots
    stride, //the stride through the controlArray, either 3 or 4 generally
    contolPointsArray, //the control points
    curveOrder, //cubic = 3, can be higher but requires more knots
    GL_MAP1_VERTEX_3 //use _3 for unweighted points, _4 if you weight
);
```