Curves

Specified with:

1) many closely spaced points

or 2) small set of control points

Mtd (2) is easier to specify and modify: important to iterative curve design in CAD (relieves user from having to deal with many points on curve).

Parametric Cubic Curves

Functions of higher degree than linear fcts allow us to approximate smooth curves with less storage + easier interactive manipulation.

3 methods (reps.): explicit, implicit, parametric

Explicit Rep.

$y = f(x)$, $z = g(x)$ $\Rightarrow$ Problems:

1) impossible to get multiple values of y or z for a single value of x
2) not rotationally invariant
3) ∞ slope (vertical tangents) cannot be represented

Implicit Rep.:

$f(x,y,z)=0 \Rightarrow$ This is best suited for answering question: "Is (x,y,z) on curve?"

Problems:

1) the equation may have more solutions than we want. For ex, how do we model a semi-circle (constraints such as $x \geq 0$ can't be constrained within the implicit eq.)
2) difficult to determine whether tangent directions agree at join point between two curve segments

Parametric Rep.:

$x = X(t)$, $y = Y(t)$, $z = Z(t) \Rightarrow$ Similar to explicit rep. except y and z now become independent of x. All are fcts of t.

Curves are approximated by piecewise polynomial curves instead of piecewise linear curves.

Cubic polynomials are most often used because:

1) lower-degree polynomials give too little flexibility in shape control
2) higher-degree polynomials introduce unwanted wiggles
3) no lower-degree rep. allows a curve to pass (interpolate) through 2 endpoints with specified derivatives at each endpoint.
4) cubics are lowest-degree curves that are nonplanar in 3D.
   Reason: cubic is specified by 4 points. Since 3 pts define a plane, 4th pt may lie off plane

curve segment $Q(t)$

$$Q(t)\begin{cases} x(t) = a_x t^3 + b_x t^2 + c_x t + d_x \\ y(t) = a_y t^3 + b_y t^2 + c_y t + d_y \\ z(t) = a_z t^3 + b_z t^2 + c_z t + d_z \end{cases}, \quad 0 \le t \le 1$$

We can rewrite this more compactly:

$$Q(t) = [x(t) \;\; y(t) \;\; z(t)] = T \cdot C$$

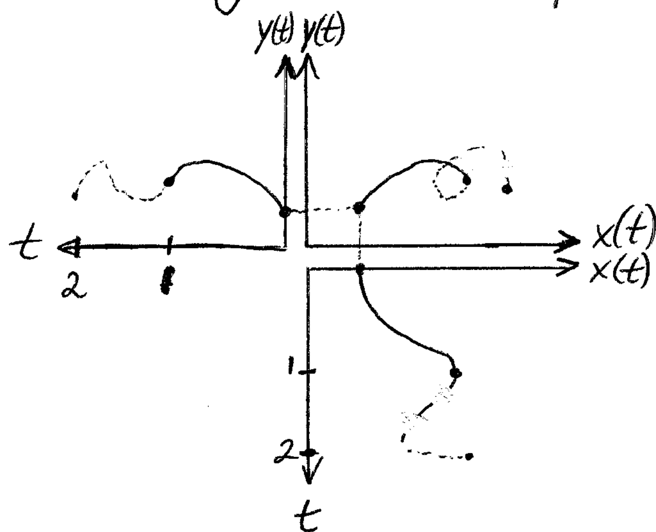
where

$$T = [t^3 \;\; t^2 \;\; t \;\; 1]$$

and

$$C = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$$

Ex: 2 joined 2D parametric curve segments



Note: the x(t) and y(t) plots for the 2nd segment have been translated to begin at t=1, rather than t=0, to show the continuity of the curves at their join point.

The derivative of Q(t) is:

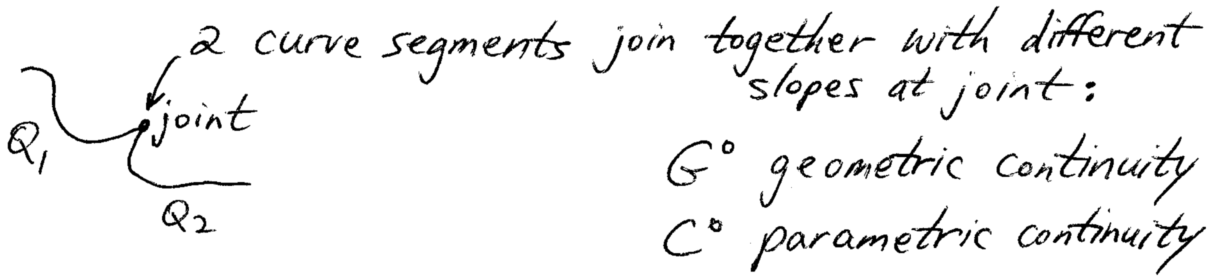
$$\frac{dQ(t)}{dt} = Q'(t) = \begin{bmatrix} \frac{dx(t)}{dt} & \frac{dy(t)}{dt} & \frac{d\,z(t)}{dt} \end{bmatrix}$$

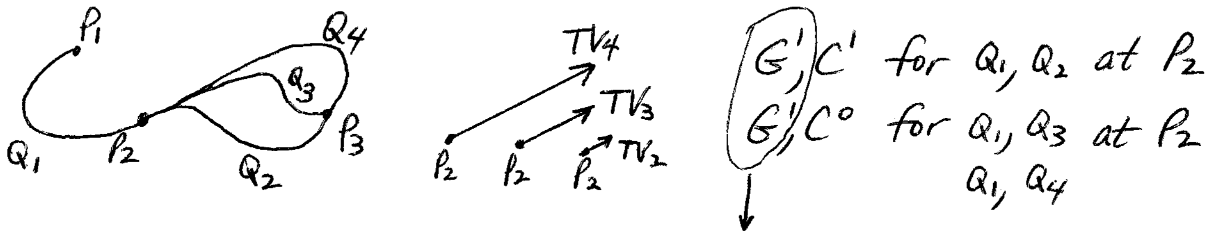
$$= \frac{d}{dt}T \cdot C$$

$$= \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \cdot C$$

$$= \begin{bmatrix} 3a_x t^2 + 2b_x t + c_x & 3a_y t^2 + 2b_y t + c_y & 3a_z t^2 + 2b_z t + c_z \end{bmatrix}$$

Geometric + Parametric Continuity





tangent vectors share same dir (but not necessarily same mag.)

* For 2 tangent vectors to have the same direction, it is necessary that one be a scalar multiple of the other:

$$TV_1 = k \cdot TV_2 \quad , \quad k > 0$$

If $k=1$, the curve has 1st-degree parametric continuity and is said to be C^1 continuous (continuous 1st derivative).

If dir. and magnitude of $\frac{d^n}{dt^n}[Q(t)]$ are equal at join point $\Rightarrow$ C^n continuous curve

Important for animation: $Q'(t)$ is velocity
$Q''(t)$ is acceleration

Camera vel. and accel. at join pts should be continuous to avoid jerky movements. Although G^1 and C^1 look smooth, C^1 is more restrictive because it was also drawn more uniformly in t.

Constraints

Cubics have 4 coefficients ⇒ 4 constraints will be needed to determine 4 coefficients.

The constraints can be pts, tangent vectors, ...

Rewrite $Q(t)$ to show this effect:

$$Q(t) = [x(t)\ y(t)\ z(t)] = \underbrace{[t^3\ t^2\ t\ 1]}_{T} \underbrace{\begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}}_{C}$$

$$= [t^3\ t^2\ t\ 1] \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix} \begin{bmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{bmatrix}$$

$C = M \cdot G$

M: basis matrix; G: geometry matrix

Ex: $x(t) = (T \cdot M) \cdot G_x$

$$\begin{aligned} &= (t^3 m_{11} + t^2 m_{21} + t m_{31} + m_{41})\, g_{1x} \\ &+ (t^3 m_{12} + t^2 m_{22} + t m_{32} + m_{42})\, g_{2x} \\ &+ (t^3 m_{13} + t^2 m_{23} + t m_{33} + m_{43})\, g_{3x} \\ &+ (t^3 m_{14} + t^2 m_{24} + t m_{34} + m_{44})\, g_{4x} \end{aligned}$$

} curve is weighted sum of the elements of the geometry matrix. $(T \cdot M)$ are the blending fcts (cubic polynomials)

This is similar to piecewise linear case: $x(t) = (1-t) g_{1x} + t g_{2x}$

Another way to look at this:

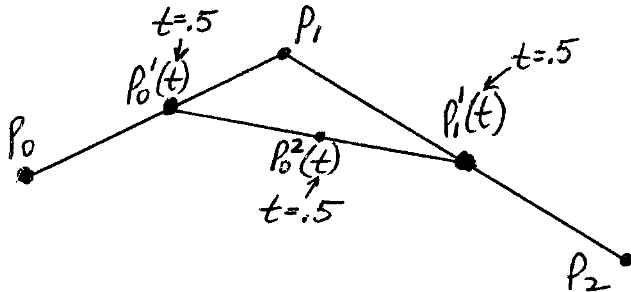
More useful for drawing since we evaluate $Q(t)$ at successive steps in t → $x(t) = T \cdot (M \cdot G_x)$

$$\begin{aligned} &= (m_{11} g_{1x} + m_{12} g_{2x} + m_{13} g_{3x} + m_{14} g_{4x})\, t^3 \\ &+ (m_{21} g_{1x} + m_{22} g_{2x} + m_{23} g_{3x} + m_{24} g_{4x})\, t^2 \\ &+ (m_{31} g_{1x} + m_{32} g_{2x} + m_{33} g_{3x} + m_{34} g_{4x})\, t \\ &+ (m_{41} g_{1x} + m_{42} g_{2x} + m_{43} g_{3x} + m_{44} g_{4x}) \end{aligned}$$

} ordinary cubic polynomial

Building a Curve Out of Control Points

de Casteljau Algorithm (1959, Citroen)



$P_0'(t) = (1-t)P_0 + tP_1$

$P_1'(t) = (1-t)P_1 + tP_2$

$P(t) = P_0^2(t)$

$= (1-t)P_0'(t) + tP_1'(t)$

By direct substitution of $P_0'(t)$ and $P_1'(t)$ in $P(t)$:

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \leftarrow \text{quadratic in } t$$

parabola

Parabola passes thru P_0 and P_2 at $t=0$ and $t=1$, respectively.

The deCasteljau algorithm generalizes to $L+1$ pts ($L \geq 2$).

We build up r^{th} generation from $(r-1)^{th}$ generation:

$$P_i^r(t) = (1-t)P_i^{r-1}(t) + tP_{i+1}^{r-1}(t) \quad \text{for } r = 1, \ldots, L; \; i = 0, \ldots, L-r$$

The process starts by using control points P_i for P_i^0.

The final generation $P_0^L(t)$ is called the Bezier curve for pts $\underbrace{P_0, P_1, \ldots P_L}_{\text{Control pts}}$

The polygon formed by the control pts is known as the control polygon (open).

<u>Note</u>: de Casteljau (1959) + Bezier (1962) independently developed (Citroen) what are now (Renault) known as Bezier curves.

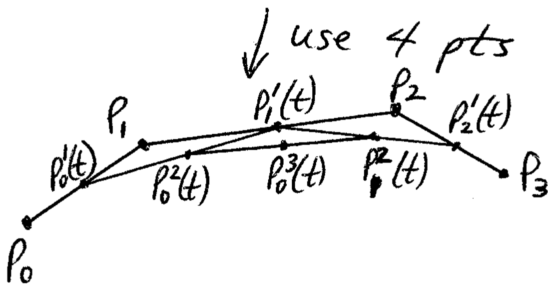
$L+1$ pts $\rightarrow$ curve of order $L+1$ and degree L

Recall: L^{th} degree polynomial in t is a fct given by $a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \ldots + a_L t^L$, $a_L \neq 0$

The <u>degree</u> is the highest power to which t is raised.

The <u>order</u> is the number of coefficients in the polynomial and is always one greater than the degree ($L+1$ here)

$\therefore$ parabola is of degree 2, order 3
cubic is of degree 3, order 4



$$P_0'(t) = (1-t)P_0 + tP_1$$
$$P_1'(t) = (1-t)P_1 + tP_2$$
$$P_2'(t) = (1-t)P_2 + tP_3$$
$$\begin{aligned} P_0^2(t) &= (1-t)P_0'(t) + tP_1'(t) \\ &= (1-t)[(1-t)P_0 + tP_1] + t[(1-t)P_1 + tP_2] \\ &= (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \end{aligned}$$

$$\begin{aligned} P_1^2(t) &= (1-t)P_1'(t) + tP_2'(t) \\ &= (1-t)[(1-t)P_1 + tP_2] + t[(1-t)P_2 + tP_3] \\ &= (1-t)^2 P_1 + 2(1-t)tP_2 + t^2 P_3 \end{aligned}$$

$$\begin{aligned} P(t) = P_0^3(t) &= (1-t)P_0^2(t) + tP_1^2(t) \\ &= (1-t)^3 P_0 + 2(1-t)^2 tP_1 + (1-t)t^2 P_2 \\ &\quad + (1-t)^2 tP_1 + 2(1-t)t^2 P_2 + t^3 P_3 \end{aligned}$$

cubic Bezier curve (4-pt curve) $\searrow$

$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

Bernstein Form for Bezier Curves

The preceding was an algorithmic development of Bezier curves. We now consider the analytical development that restates its functional form.

$$p(t) = \sum_{K=0}^{L} P_K B_K^L(t)$$

↑ Bernstein polynomials; weights of P_K point

where

$$B_K^L(t) = \binom{L}{K} (1-t)^{L-K} t^K$$

↑ coef. — all combinations of $(1-t)$ and t products in an Lth degree poly. in t

and

$$\binom{L}{K} = \frac{L!}{K!\,(L-K)!} \quad \text{for } L \geq k$$

↑ binomial coefficient fct: # ways of choosing K items from amongst L items

Note: The Bernstein polynomials are the terms in $((1-t)+t)^L$. Their sum is always 1 for any value of t and each is of degree L.

For $L=3$ (cubic):

$$B_0^3(t) = \binom{3}{0}(1-t)^{3-0}t^0 = (1-t)^3$$

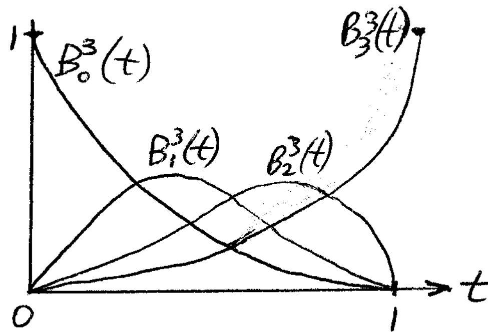
$$B_1^3(t) = \binom{3}{1}(1-t)^{3-1}t^1 = 3(1-t)^2 t$$

$$B_2^3(t) = \binom{3}{2}(1-t)^{3-2}t^2 = 3(1-t)t^2$$

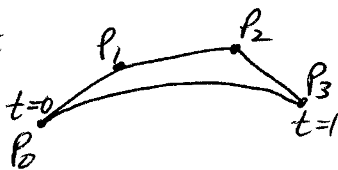
$$B_3^3(t) = \binom{3}{3}(1-t)^{3-3}t^3 = t^3$$

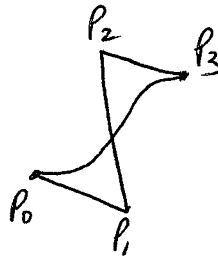
Identical to previous result:

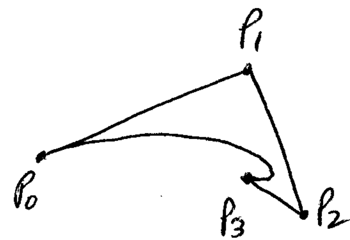
$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$



Exs:







Properties of Bezier Curves

1) Endpoint interpolation:

$B_0^L(t) = (1-t)^L = 1$ for $t=0$, 0 for $t=L$

$B_L^L(t) = t^L = 1$ for $t=1$, 0 for $t=0$

all others $= 0$ for $t \neq 0,1$

2) Affine invariance: to apply an affine transformation to the curve, we must only apply it to the control points, and redisplay.

Let $M_A + t_r$ = affine transf. matrix and offset vector

$$p'(t) = \sum_{K=0}^{L} (M_A P_K + t_r) B_K^L(t)$$

$$= \sum_{K=0}^{L} M_A P_K B_K^L(t) + \underbrace{\sum_{K=0}^{L} t_r B_K^L(t)}_{= t_r \text{ because } \sum_{K=0}^{L} B_K^L(t) = 1}$$

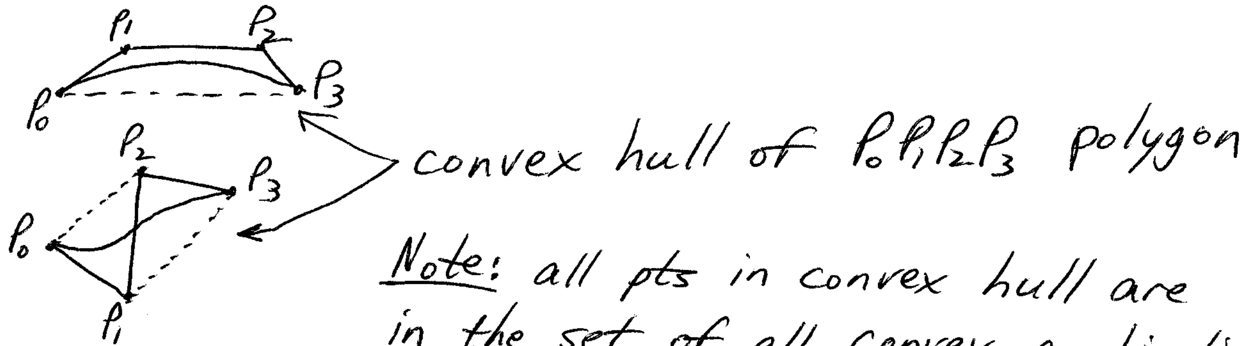
$$p'(t) = \sum_{K=0}^{L} P_K' B_K^L(t) + t_r$$

↓

Simply apply affine transf. to P_K to get P_K' and re-apply Bernstein polynomials to generate curve.

Intuition: since curve can be generated by de Casteljau alg. using linear interpolation, and lines are preserved under affine transformations, then so are Bezier curves.

3) The curve is contained within the convex hull of the control polygon. The convex hull is the largest convex polygon obtainable with the control pts.



Note: all pts in convex hull are in the set of all convex combinations of the control points:

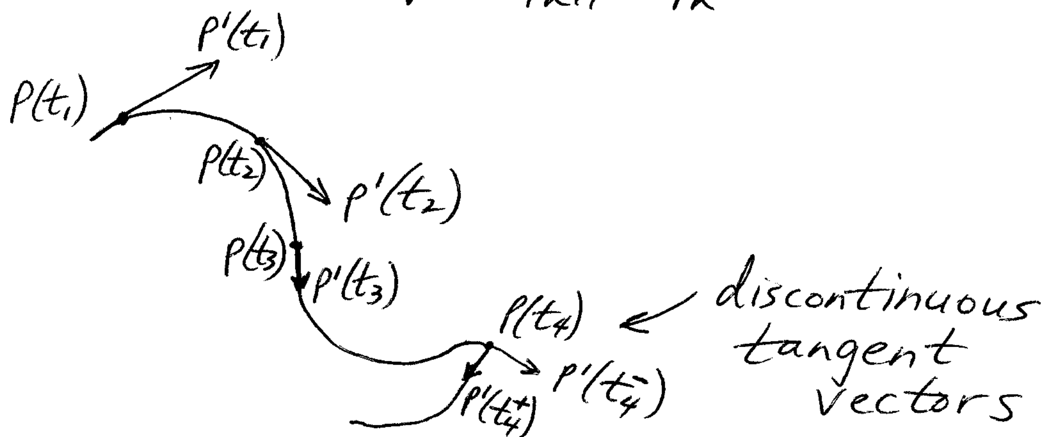
$$\sum_{k=0}^{L} \alpha_k P_k \,, \quad \underbrace{\alpha_k \geq 0 \text{ and } \sum_{k=0}^{L} \alpha_k = 1}_{\text{satisfied by } B_k^L(t)}$$

4) Linear precision: Bezier curve can be a line if ctrl. pts. are collinear

5) Variation diminishing: curves can't wiggle more than control polygon. No straight line can have more intersections with the Bezier curve than it has with the control polygon.

6) Derivative of $p(t)$ is $p'(t) = L\sum_{k=0}^{L-1} \Delta p_k B_k^{L-1}(t)$

where $\Delta p_k = p_{k+1} - p_k$



Note: $p'(0) = L\Delta p_0 = L(p_1 - p_0) = 3(p_1 - p_0)$

$p'(1) = L\Delta p_{L-1} = L(p_L - p_{L-1}) = 3(p_L - p_{L-1})$

↑ for cubic Bezier curves

The tangent vectors at the curve endpoints have the same direction as the initial and final polygon spans.

Matrix Rep. of Bezier Curves

$$p(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

Isolate t, t^2, t^3 terms:

$$p(t) = [1 - 3t + 3t^2 - t^3] P_0 + [3t - 6t^2 + 3t^3] P_1 + [3t^2 - 3t^3] P_2 + t^3 P_3$$

$$p(t) = P_0 + t[-3P_0 + 3P_1] + t^2[3P_0 - 6P_1 + 3P_2] + t^3[-P_0 + 3P_1 - 3P_2 + P_3]$$

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$\therefore$

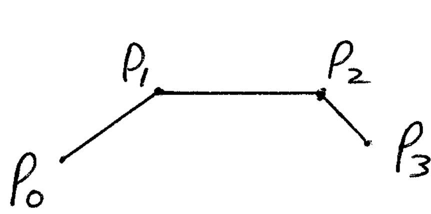
$$p(t) = T \cdot M_B \cdot G_B$$

← advantageous if hardware matrix multiplication is available. Otherwise, $p(t) = \sum_{k=0}^{L} P_k B_k^L(t)$ is more straightforward to compute for arbitrary L.

- T: powers of t for cubic polynomial
- M_B: Bezier basis matrix
- G_B: Bezier geometry vector (geometric constraints such as endpts, ctrl pts, tangent vectors, ...)

<u>Note</u>: $T \cdot M_B$ is expansion of Bernstein polynomials. They are weights (blending fcts) for the geometry matrix. Also realize that $M_B \cdot G_B$ are coeffs for t^3, t^2, t, and 1 terms in cubic polynomial for curve $a_0 + a_1 t + a_2 t^2 + a_3 t^3$.

Ex:



$P_0(0,0)$
$P_1(1,2)$
$P_2(4,2)$
$P_3(5,1)$

$$P(t) = [x(t)\ y(t)] = [t^3\ t^2\ t\ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 1 & 2 \\ 4 & 2 \\ 5 & 1 \end{bmatrix}$$

$\Downarrow$

$$x(t) = -4t^3 + 6t^2 + 3t$$
$$y(t) = t^3 - 6t^2 + 6t$$

Note: $x(0), y(0) \overset{\checkmark}{=} 0,0$
$x(1), y(1) \overset{\checkmark}{=} 5,1$

Display curve:

```
drawCubic(cx, cy, steps)
double  cx[4], cy[4];
int     steps;
{
        int i, x1, y1, x2, y2;
        double t, t2, t3, stepsize;

        t=0.;
        stepsize = 1.0/steps;
        x1 = cx[3];     /* t=0: start at x(0) */
        y1 = cy[3];     /* t=0: start at y(0) */
        for(i=1; i<steps; i++) {
                t += stepsize;
                t2 = t*t;
                t3 = t*t2;
                x2 = cx[0]*t3 + cx[1]*t2 + cx[2]*t + cx[3];
                y2 = cy[0]*t3 + cy[1]*t2 + cy[2]*t + cy[3];
                drawLine(x1, y1, x2, y2);
                x1 = x2;
                y1 = y2;
        }
}
```

evaluation of cubic polynomial can be optimized using forward differences → (x2, y2 lines)

Appendix 3

FORWARD DIFFERENCE METHOD

The method of forward differences is used to simplify the computation of polynomials. It basically extends the incremental evaluation, as used in scanline algorithms, to higher-order functions, e.g., quadratic and cubic polynomials. We find use for this in Chapter 7 where, for example, perspective mappings are approximated without costly division operations. In this appendix, we derive the forward differences for quadratic and cubic polynomials. The method is then generalized to polynomials of arbitrary degree.

The (first) forward difference of a function $f(x)$ is defined as

$$\Delta f(x) = f(x+\delta) - f(x), \qquad \delta > 0 \tag{A3.1}$$

It is used together with $f(x)$, the value at the current position, to determine $f(x+\delta)$, the value at the next position. In our application, $\delta = 1$, denoting a single pixel step size. We shall assume this value for δ in the discussion that follows.

For simplicity, we begin by considering the forward difference method for linear interpolation. In this case, the first forward difference is simply the slope of the line passing through two supplied function values. That is, $\Delta f(x) = a_1$ for the function $f(x) = a_1 x + a_0$. We have already seen it used in Section 7.2 for Gouraud shading, whereby the intensity value at position $x+1$ along a scanline is computed by adding $\Delta f(x)$ to $f(x)$. Surprisingly, this approach readily lends itself to higher-order interpolants. The only difference, however, is that $\Delta f(x)$ is itelf subject to update. That update is driven by a second increment, known as the second forward difference. The extent to which these increments are updated is based on the degree of the polynomial being evaluated. In general, a polynomial of degree N requires N forward differences.

We now describe forward differencing for evaluating quadratic polynomials of the form

$$f(x) = a_2 x^2 + a_1 x + a_0 \tag{A3.2}$$

The first forward difference for $f(x)$ is expressed as

128.1

$$\Delta f(x) = f(x+1) - f(x) = \left[a_2(x+1)^2 + a_1(x+1) + a_0 \right] - \left[a_2x^2 + a_1x + a_0 \right] = a_2(2x+1) + a_1 \quad \text{(A3.3)}$$

Thus, $\Delta f(x)$ is a linear expression. If we apply forward differences to $\Delta f(x)$, we get

$$\Delta^2 f(x) = \Delta(\Delta f(x)) = \Delta f(x+1) - \Delta f(x) = \left[a_2(2[x+1]+1)+a_1 \right] - \left[a_2(2x+1)+a_1 \right] = 2a_2 \quad \text{(A3.4)}$$

Since $\Delta^2 f(x)$ is a constant, there is no need for further terms. The second forward difference is used at each iteration to update the first forward difference which, in turn, is added to the latest result to compute the new value. Each loop in the iteration can be rewritten as

$$f(x+1) = f(x) + \Delta f(x) \qquad \Delta f(x+1) = \Delta f(x) + \Delta^2 f(x) \quad \text{(A3.5)}$$

If computation begins at $x=0$, then the basis for the iteration is given by f, Δf, and $\Delta^2 f$ evaluated at $x=0$. Given these three values, the second-degree polynomial can be evaluated from 0 to *lastx* using the following C code.

```
for(x = 0; x < lastx; x++) {
        f[x+1] = f[x] + Δf;           /* compute next point */
        Δf += Δ²f;                    /* update 1st forward difference */
}
```

Notice that Δf is subject to update by $\Delta^2 f$, but the latter term remains constant throughout the iteration.

A similar derivation is given for cubic polynomials. However, an additional forward difference constant must be incorporated due to the additional polynomial degree. For a third-degree polynomial of the form

$$f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \quad \text{(A3.6)}$$

the first forward difference is

128.2

$$\begin{aligned}\Delta f(x) &= f(x+1) - f(x) \qquad \text{(A3.7)}\\ &= \left[a_3(t+1)^3 + a_2(t+1)^2 + a_1(t+1) + a_0 \right] - \left[a_3x^3 + a_2x^2 + a_1x + a_0 \right] \\ &= 3a_3x^2 + (3a_3 + 2a_2)x + a_3 + a_2 + a_1\end{aligned}$$

Since $\Delta f(x)$ is a second-degree polynomial, two more forward difference terms are derived. They are

$$\begin{aligned}\Delta^2 f(x) &= \Delta(\Delta f(x)) = 6a_3x + 6a_3 + 2a_2 \qquad \text{(A3.8)}\\ \Delta^3 f(x) &= \Delta(\Delta^2 f(x)) = 6a_3\end{aligned}$$

The use of forward differences to evaluate a cubic polynomial between 0 and *lastx* is demonstrated in the following C code.

```
for(x = 0; x < lastx; x++) {
        f[x+1] = f[x] + Δf;          /* compute next point */
        Δf   += Δ^2f;                /* update 1st forward difference */
        Δ^2f += Δ^3f;                /* update 2nd forward difference */
}
```

In contrast to the earlier example, this case has an additional forward difference term that must be updated, i.e., $\Delta^2 f$. Even so, this method offers the benefit of computing a third-degree polynomial with only three additions per value. An alternate approach, using Horner's rule for factoring polynomials, requires three additions *and* three multiplications. This makes forward differences the method of choice for evaluating polynomials.

The forward difference approach for cubic polynomials is depicted in Fig. A3.1. The basis of the entire iteration is shown in the top row. For consistency with our discussion of this method in Chapter 7, texture coordinates are used as the function values. Thus, we begin with u_0, Δu_0, and $\Delta^2 u_0$ defined for position x_0, where the subscripts refer to the position along the scanline.

In order to compute our next texture coordinate at $x = 1$, we add Δu_0 to u_0. This is denoted by the arrows that are in contact with u_0. Note that diagonal arrows denote addition, while vertical arrows refer to the computed sum. Therefore, u_1 is the result of adding Δu_0 to u_0. The following coordinate, u_2, is the sum of u_1 and Δu_1. The latter term is derived from Δu_0 and $\Delta^2 u_0$. This regular structure collapses nicely into a compact iteration, as demonstrated by the short programs given earlier.

Higher-order polynomials are handled by adding more forward difference terms. This corresponds to augmenting Fig. A3.1 with additional columns to the right. The order of computation is from the left to right. That is, the summations corresponding to the diagonal arrows are executed beginning from the left column. This gives rise to the

128.3

adjacent elements directly below. Those elements are then combined in similar fashion. This cycle continues until the last diagonal is reached, denoting that the entire span of points has been evaluated.

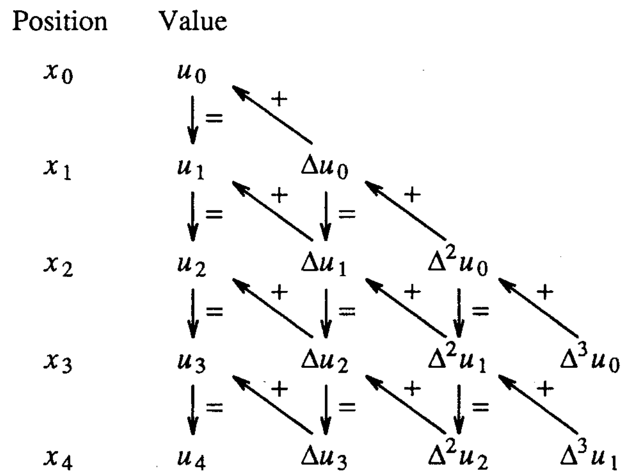


Figure A3.1: Forward difference method.

128.4

Disadvantages of Bezier Curves

No local control
Global $P(t)$
Changing one control point affects entire curve due to full span of all blending functions over $[0,1]$

Solution: use blending functions that have support only over a part of $[0,1]$ range for t.

"Support" is the interval over which the function is nonzero

We seek a collection of piecewise polynomials on adjacent intervals: splines.