Curve Details

Reading

Required

- Foley, 11.2
- Hearn & Baker, 10.6 -10.9

Optional

- Bartels, Beatty, and Barsky. *An Introduction to Splines for use in Computer Graphics and Geometric Modeling*. 1987.
- Farin. *Curves and Surfaces CAGD: A Practical Guide*. 4th ed. 1997.

Alternative Bezier Formulation

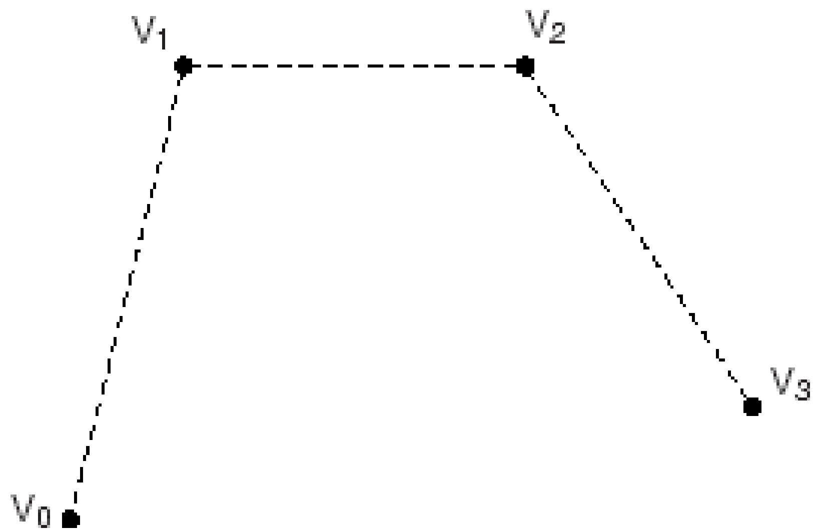
$$Q(t) = \sum_{i=0}^{3} P_i \underbrace{\binom{3}{i}}_{{}_3C_i} t^i (1-t)^{3-i}$$

$$\mathbf{Q}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \\ \mathbf{P}_4 \end{bmatrix}$$

$$Q(t) = \sum_{i=0}^{n} P_i \binom{n}{i} t^i (1-t)^{n-i}$$

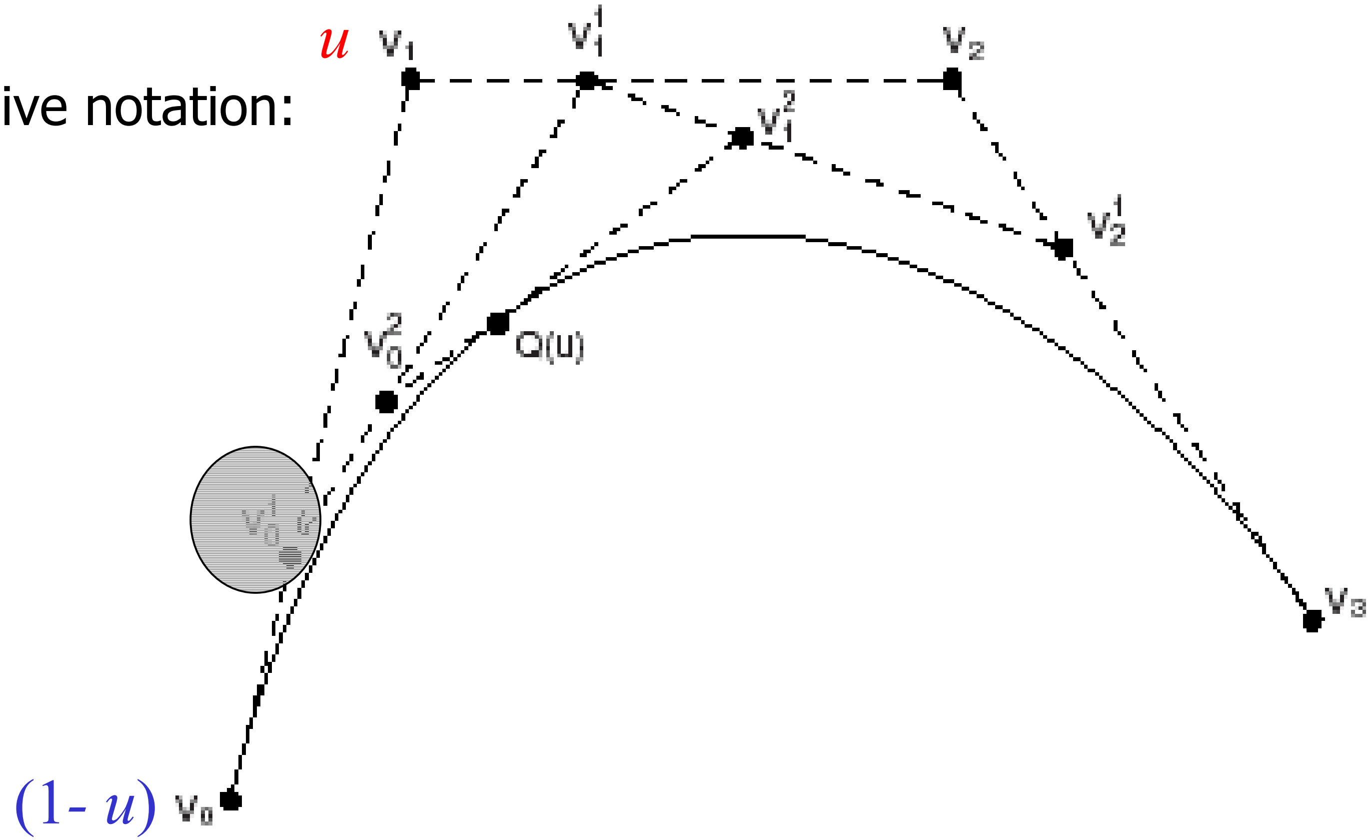
de Castaljau's algorithm

- Recursive interpolation



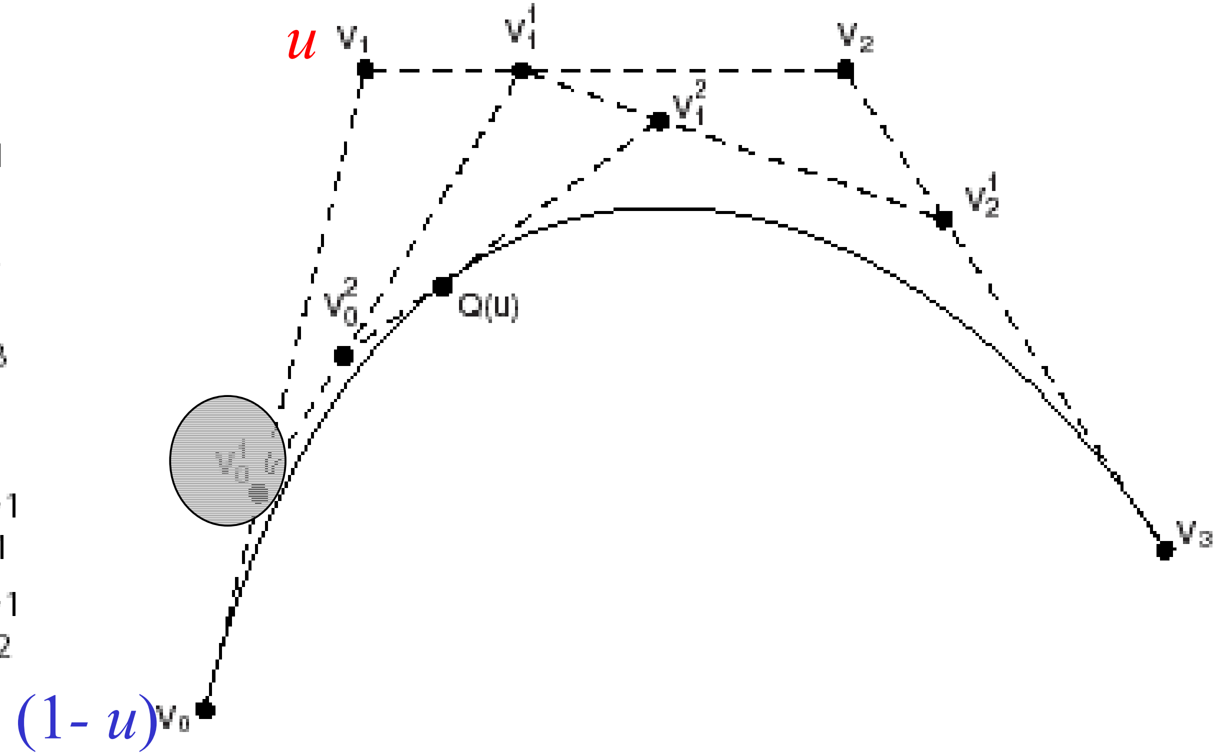
de Castaljau's algorithm

- Recursive notation:



- What is the equation for V_0^1 ?

Finding Q(u)



$$V_0^1 = (1-u)V_0 + uV_1$$

$$V_1^1 = (1-u)V_1 + uV_2$$

$$V_2^1 = (1-u)V_2 + uV_3$$

$$V_0^2 = (1-u)V_0^1 + uV_1^1$$

$$V_1^2 = (1-u)V_1^1 + uV_2^1$$

$$\begin{aligned}
Q(u) &= (1-u)V_0^2 + uV_1^2 \\
&= (1-u)[(1-u)V_0^1 + uV_1^1] + u[(1-u)V_1^1 + uV_2^1] \\
&= (1-u)[(1-u)\{(1-u)V_0 + uV_1\} + u\{(1-u)V_1 + uV_2\}] + \ldots \\
&= (1-u)^3 V_0 + 3u(1-u)^2 V_1 + 3u^2(1-u)V_2 + u^3 V_3
\end{aligned}$$

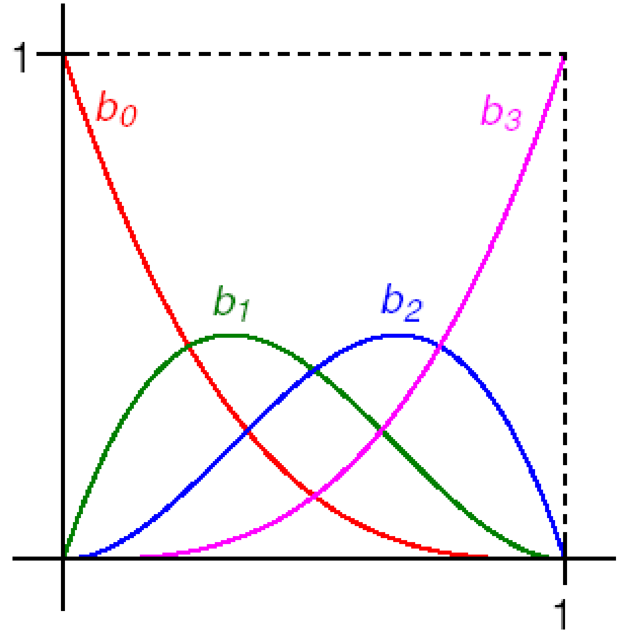
Bernstein polynomials

- The coefficients of the control points are a set of functions called the **Bernstein polynomials**.

$$Q(u) = \sum_{i=0}^{n} b_i(u) V_i$$

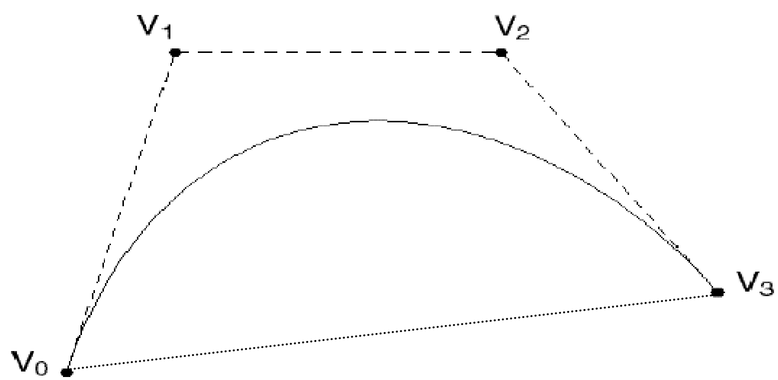
- For degree 3, we have:

$$b_0(u) = (1-u)^3$$
$$b_1(u) = 3u(1-u)^2$$
$$b_2(u) = 3u^2(1-u)$$
$$b_3(u) = u^3$$



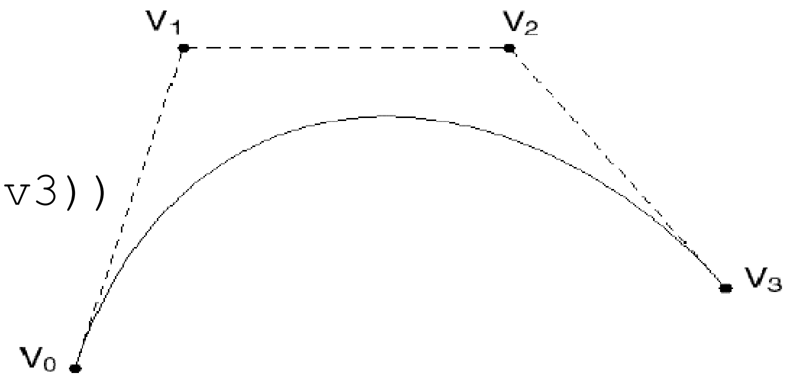
Useful properties

- Useful properties on the interval [0,1]:
  - each Bernstein coefficient is between 0 and 1
  - sum of all four is exactly 1 (a.k.a. , a “partition of unity”)
- These together implies that the curve lies within the **convex hull** of its control points. (convex hull is the smallest convex polygon that contains the control points)



Displaying Bezier Curves

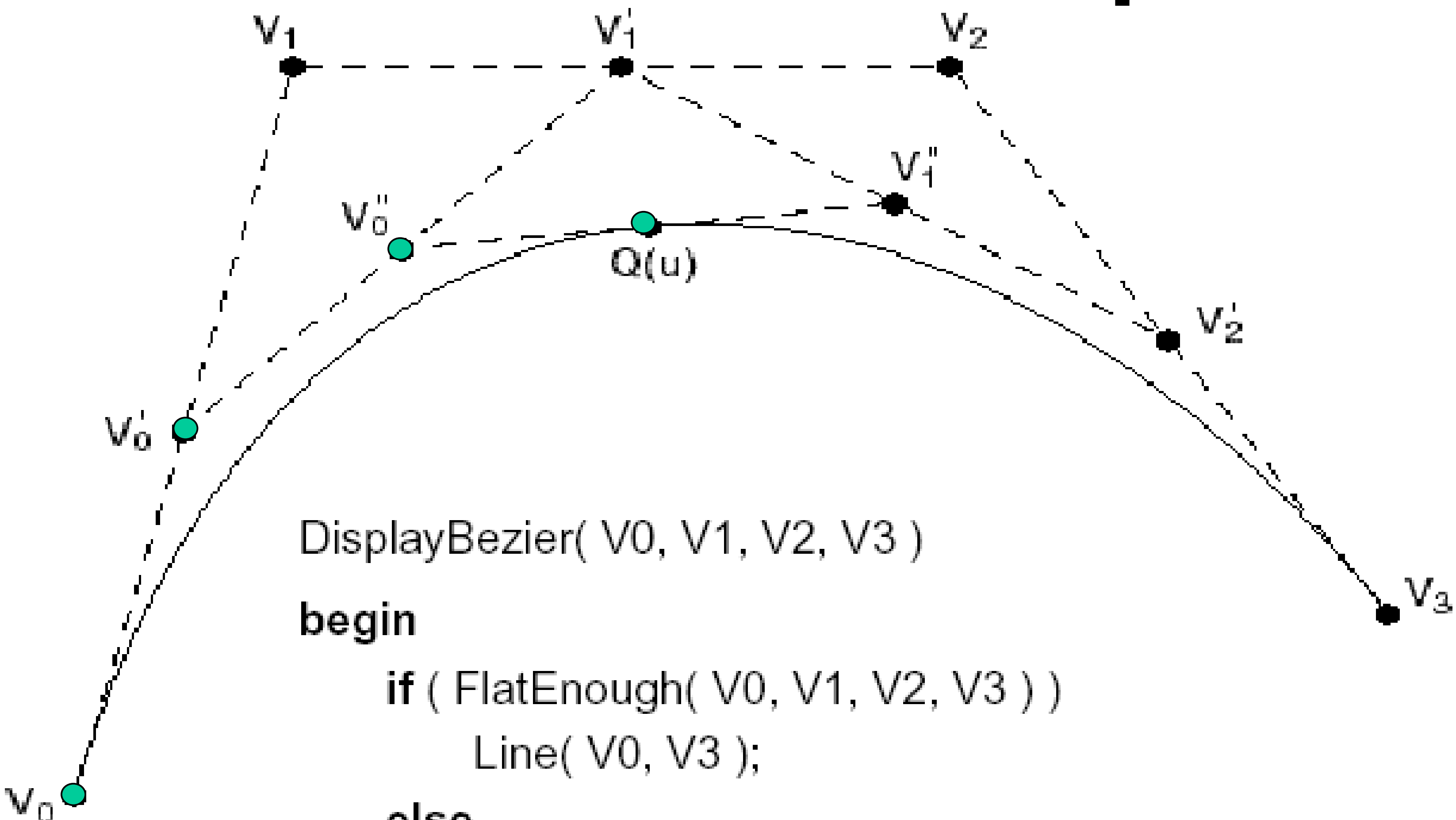
- Recall that most graphics board can only display lines and polygons.
- How can we display Bezier curves?




```
DisplayBezier (vo,v1,v2,v3)
    if (FlatEnough(v0,v1,v2,v3))
        Line(v0,v3)
    else
        do something smart
```

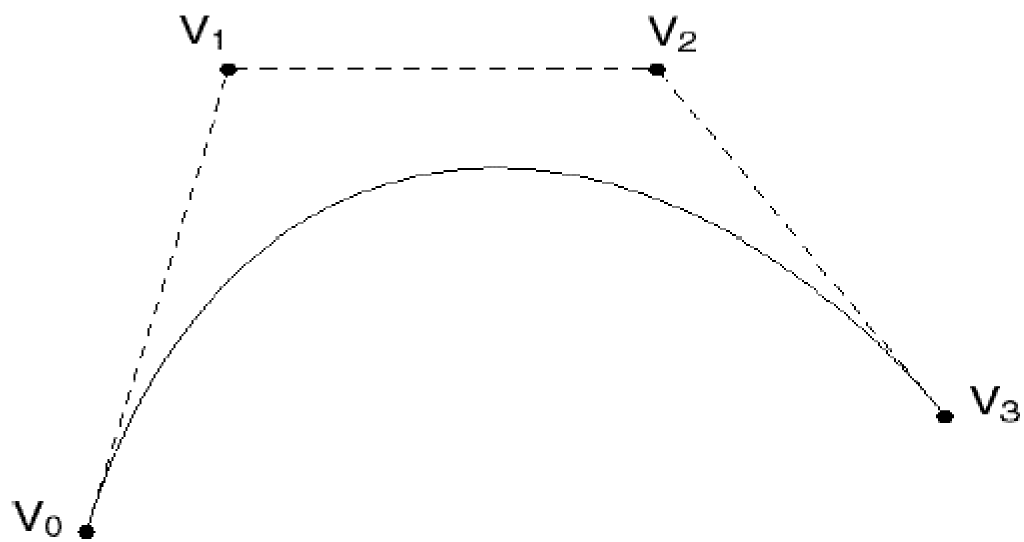
- It would be nice to have an *adaptive* algorithm that takes flatness into account.

Subdivide and Conquer




```
DisplayBezier( V0, V1, V2, V3 )
begin
    if ( FlatEnough( V0, V1, V2, V3 ) )
        Line( V0, V3 );
    else
        Subdivide(V) ⇒ L, R
        DisplayBezier( L0, L1, L2, L3 );
        DisplayBezier( R0, R1, R2, R3 );
end;
```

Testing for Flatness



- Compare the total length of control polygon to the length connecting the endpoints:

$$\frac{|V_0 - V_1| + |V_1 - V_2| + |V_2 - V_3|}{|V_0 - V_3|} < 1 + \varepsilon$$

More Complex Curves

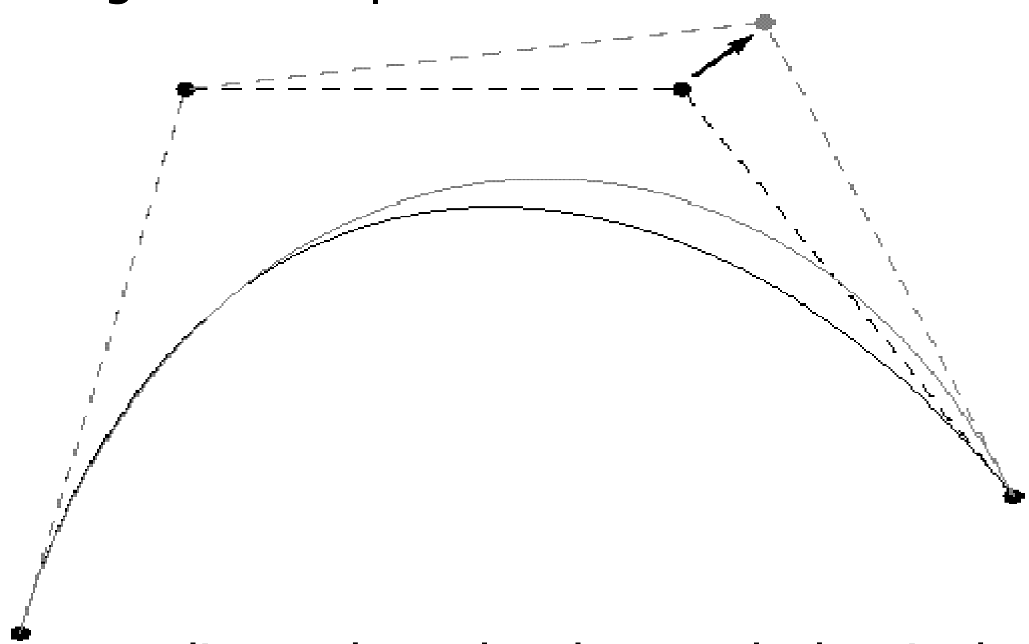
- Suppose we want to draw a more complex curve.
- Why not use a high-order Bezier?

- Instead, we'll connect together individual curve segments that are cubic Beziers to form a longer curve.

- There are three properties that we'd like to have in our newly constructed splines (curves)…

Local Control

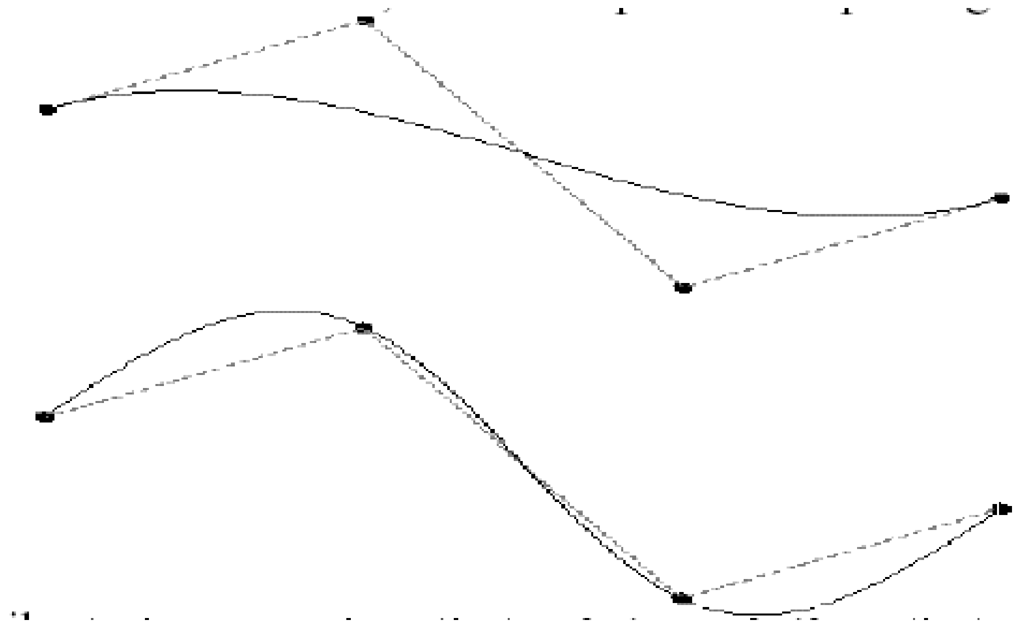
- One problem with Beziers is that every control point affects every point on the curve (except the endpoints)
- Moving a single control point affects the whole curve!



- We'd like our spline to have local control, that is, have each control point affect some well-defined neighborhood around that point.

Interpolation

- Bezier curves are **approximating**. The curve does not (necessarily) pass through all the control points. Each point pulls the curve toward it, but other points are pulling as well.



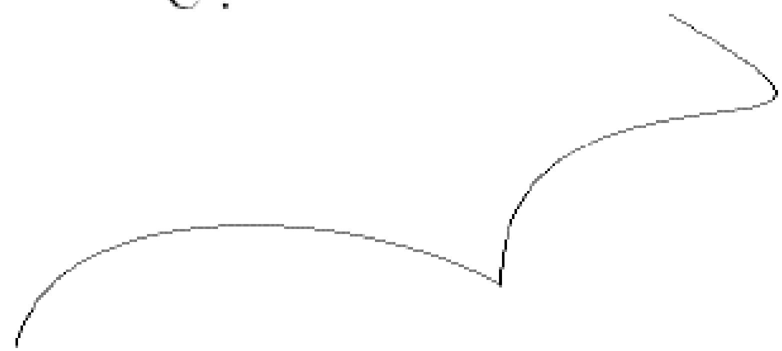
- We'd like to have a spline that is **interpolating**, that is, it always passes through every control point.

Continuity

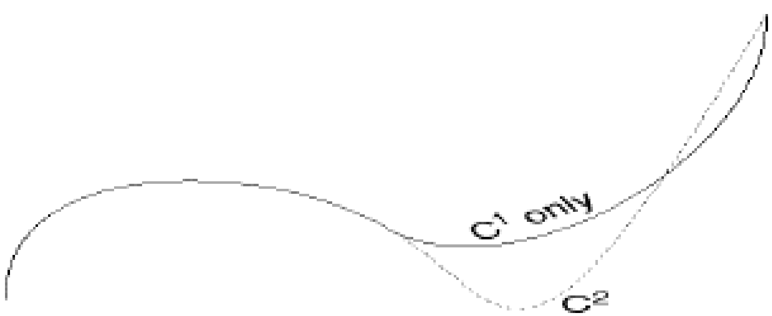
- We want our curve to have **continuity**. There shouldn't be an abrupt change when we move from one segment to the next.
- There are nested degree of continuity:

C^0:

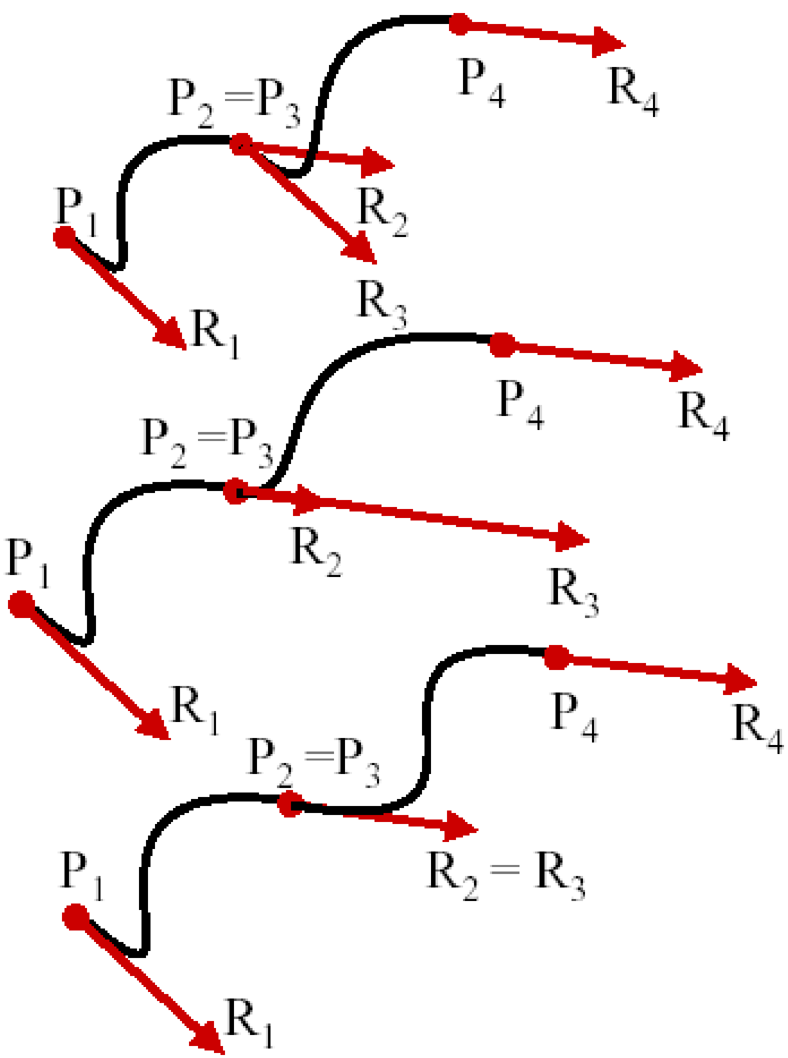
C^1:



C^2:



Continuity of Splines



- Splines: 2 or more curves are concatentated together
- C^0: points coincide, velocity don't
- G^1: points coincide, velocities have the same direction.
- C^1: points and velocities coincide.
- **Q**: What's C^2?

Ensuring Continuity

- Let's look at continuity first.
- Since the functions defining a Bezier curve are polynomials, all their derivatives exist and are continuous.
- Therefore, we only need to worry about the derivatives at the endpoints of the each cubic Bezier (joints).
- First, we'll rewrite our equation for Q(t) in matrix form:

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

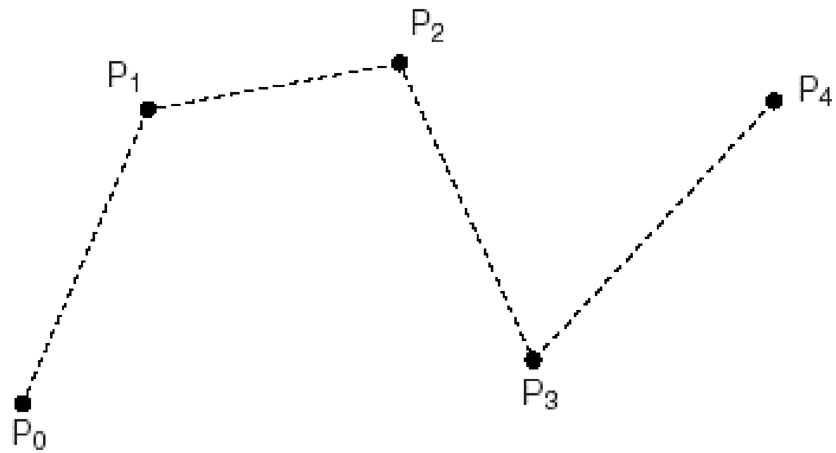
Ensuring C^0 continuity

- Suppose we have a cubic Bezier defined by (V_0, V_1, V_2, V_3) and we want to attach another curve defined by (W_0, W_1, W_2, W_3) to it, so that there is C^0 continuity at the joint.

$$C^0 : Q_V(1) = Q_W(0)$$

The C^0 Bezier spline

- How then could we construct a curve passing through a set of points $P_1 \dots P_k$?



- We call this curve a **spline**. The endpoints of the Bezier segments are called **joints**.

Ensuring C^1 continuity

- Suppose we have a cubic Bezier defined by (V_0,V_1,V_2,V_3) and we want to attach another curve defined by (W_0,W_1,W_2,W_3) to it, so that there is C^1 continuity at the joint.

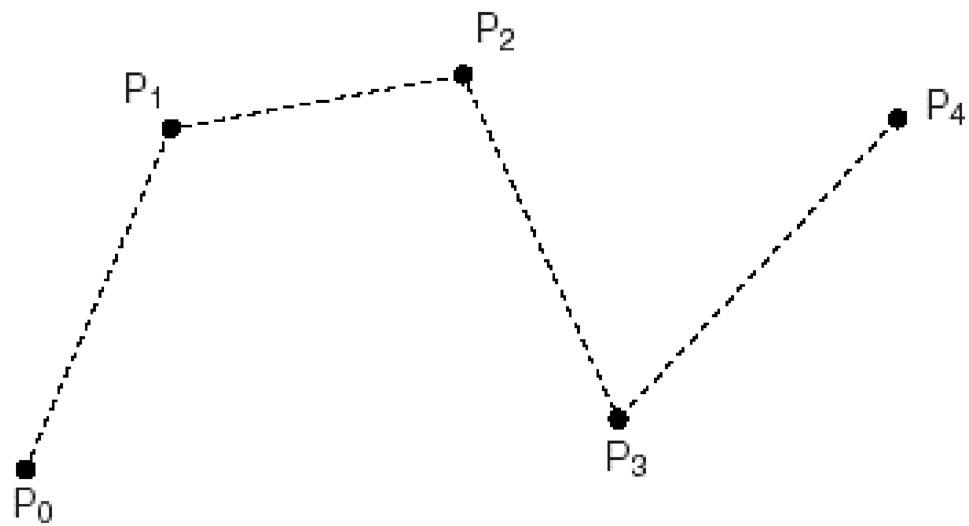
$$C^0 : Q_V(1) = Q_W(0)$$

$$C^1 : Q_V'(1) = Q_W'(0)$$

- What constraint(s) does this place on (W_0,W_1,W_2,W_3)?

The C^1 Bezier spline

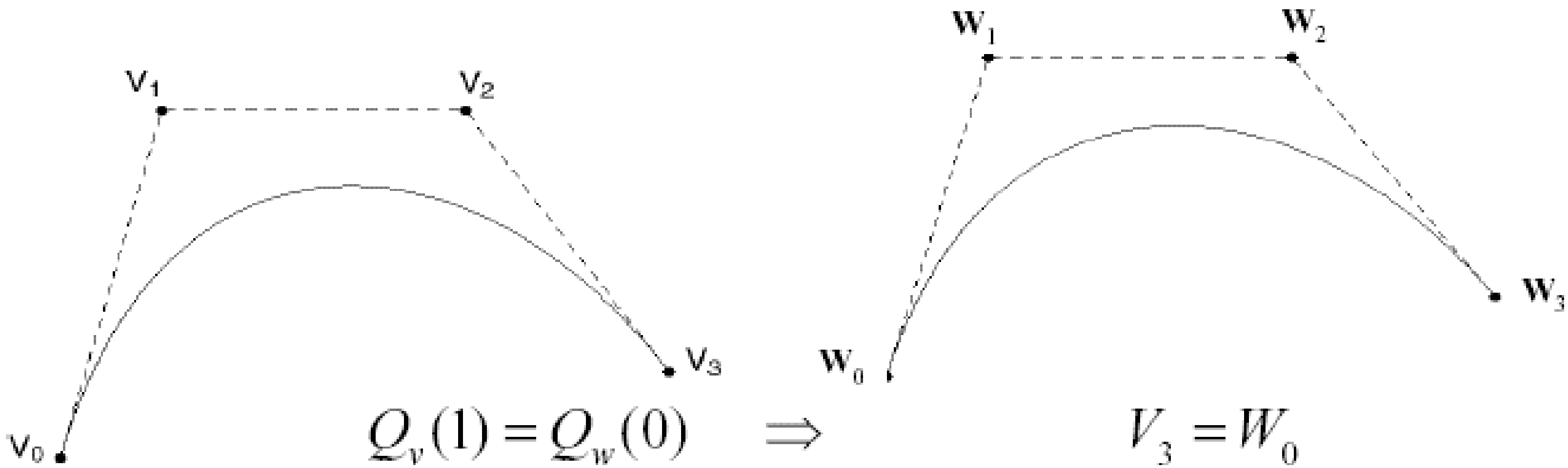
- How then could we construct a curve passing through a set of points $P_1 \ldots P_k$?



- We can specify the Bezier control points directly, or we can devise a scheme for placing them automatically.

Ensuring C^2 Continuity

- Suppose we want to join two cubic Bezier curves (V_0,V_1,V_2,V_3) and (W_0,W_1,W_2,W_3) so that there is C^2 continuity at the joint.



$$Q_v(1) = Q_w(0) \Rightarrow V_3 = W_0$$

$$Q_v'(1) = Q_w'(0) \Rightarrow V_3 - V_2 = W_1 - W_0$$

$$Q_v''(1) = Q_w''(0) \Rightarrow V_1 - 2V_2 + V_3 = W_0 - 2W_1 + W_2$$

$$\Downarrow$$

$$\boxed{W_2 = V_1 + 4V_3 - 4V_2}$$

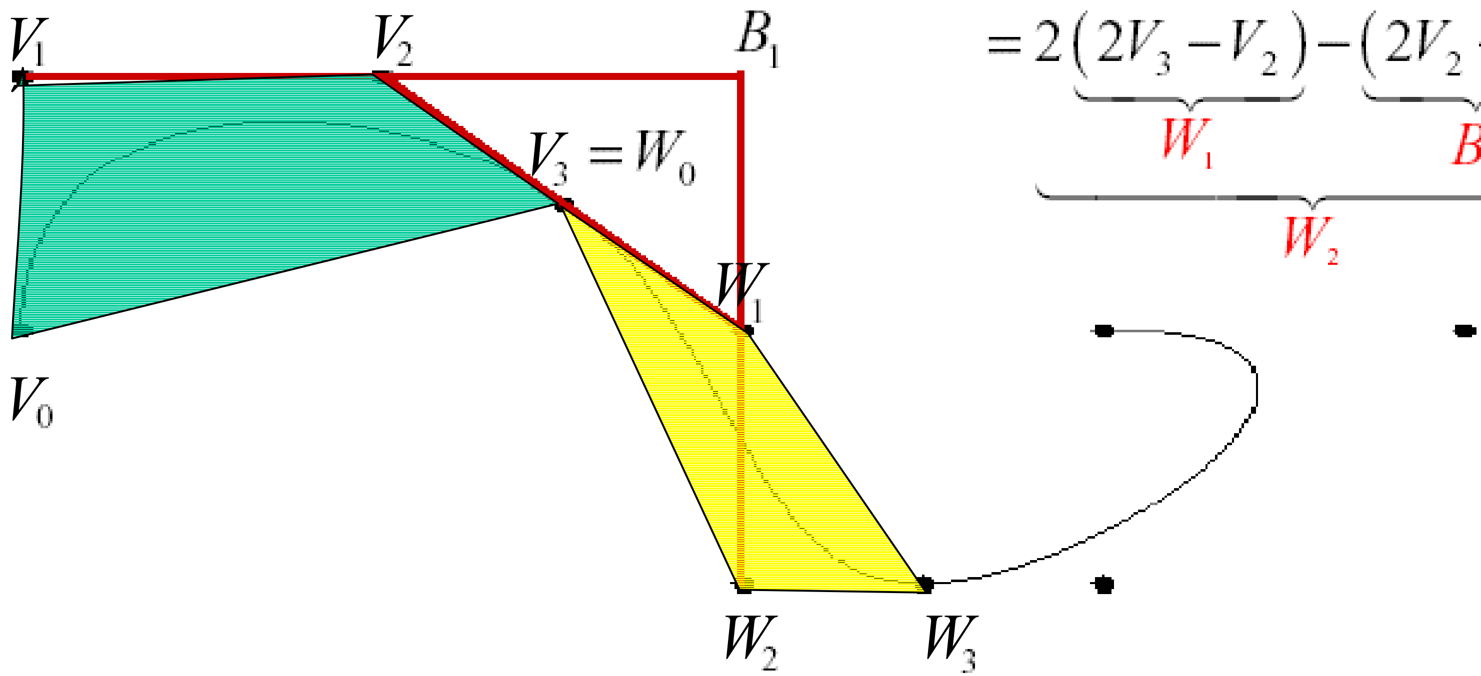
$$\boxed{\begin{aligned} Q'(0) &= 3(V_1 - V_0) \\ Q'(1) &= 3(V_3 - V_2) \end{aligned}}$$

$$\begin{aligned} Q''(0) &= 6(V_0 - 2V_1 + V_2) \\ Q''(1) &= 6(V_1 - 2V_2 + V_3) \end{aligned}$$

A-frames and Continuity

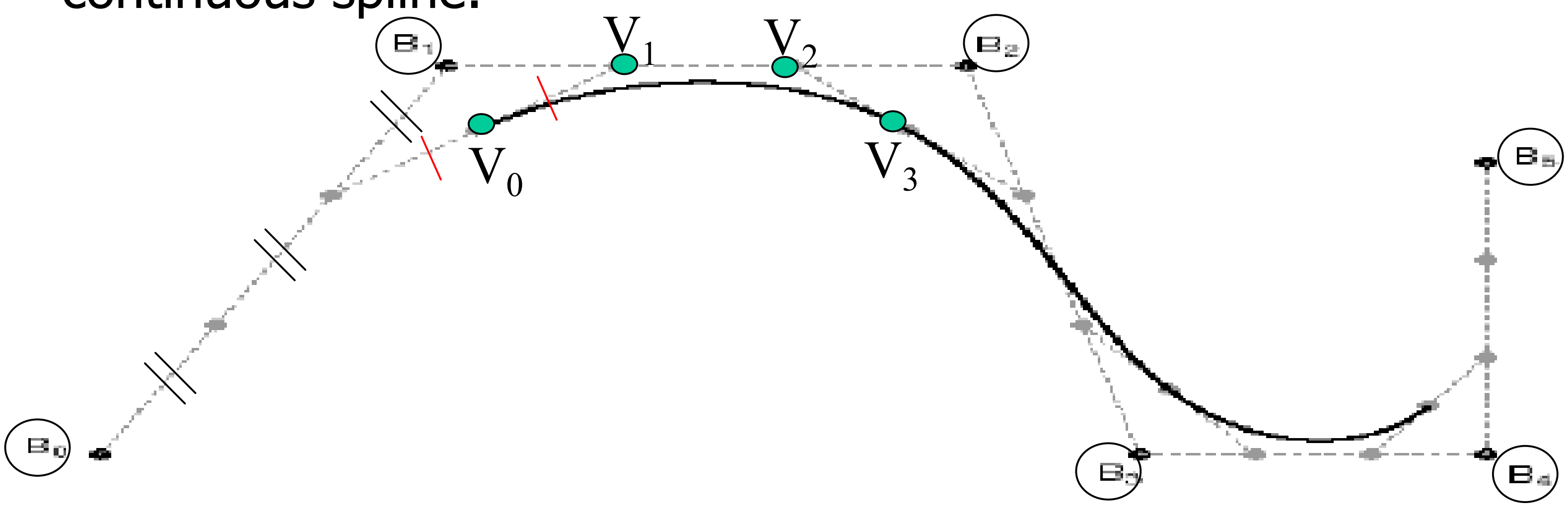
- Let's try to get some geometric intuition about what this last continuity equation means.
- If a and b are points, what is the significance of the point 2a-b?

$$
\begin{aligned}
W_2 &= V_1 + 4V_3 - 4V_2 \\
&= \underbrace{2\underbrace{\left(2V_3 - V_2\right)}_{W_1} - \underbrace{\left(2V_2 - V_1\right)}_{B_1}}_{W_2}
\end{aligned}
$$



Building a Complex Spline

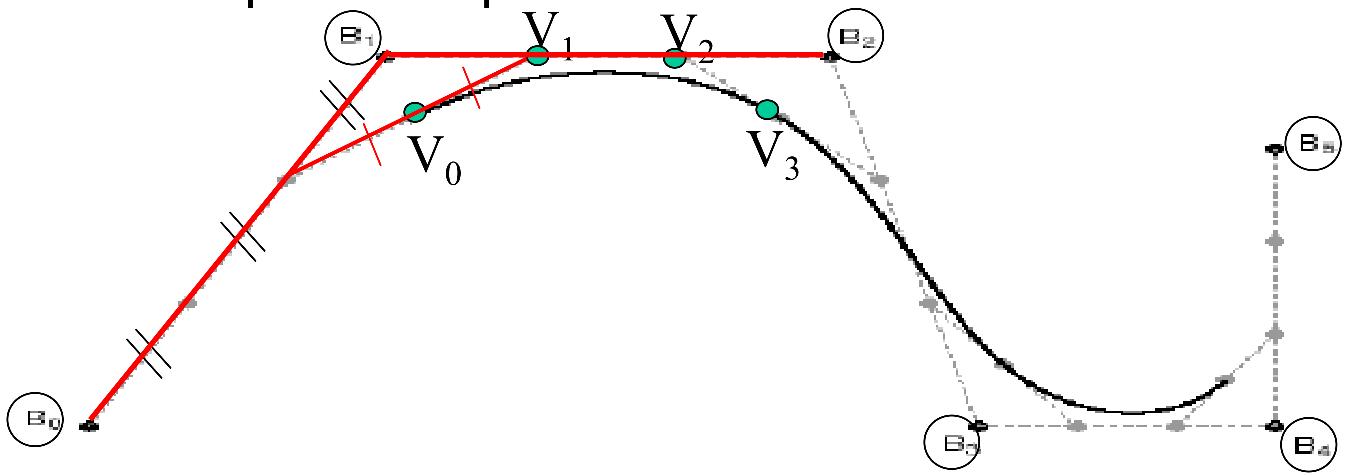
- Instead of specifying the Bezier control points V's themselves, let's specify the corners of the A-frames in order to build a C^2 continuous spline.



- These are called **B-splines**. The starting set of points are called de Boor points.

Constructing B-splines

- Here is the completed B-spline:



$$V_1 = \_\_\_\_\_\_\_ B_1 + \_\_\_\_\_\_\_ B_2$$

$$V_2 = \_\_\_\_\_\_\_ B_1 + \_\_\_\_\_\_\_ B_2$$

$$V_0 = \_\_\_\_\_\_\_ [\_\_\_\_\_\_\_ B_0 + \_\_\_\_\_\_\_ B_1] + \_\_\_\_\_\_\_ [\_\_\_\_\_\_\_ B_1 + \_\_\_\_\_\_\_ B_2]$$

$$= \_\_\_\_\_\_\_ B_0 + \_\_\_\_\_\_\_ B_1 + \_\_\_\_\_\_\_ B_2$$

$$V_3 = \_\_\_\_\_\_\_ B_1 + \_\_\_\_\_\_\_ B_2 + \_\_\_\_\_\_\_ B_3$$

- What are the Bezier control (V) points, in terms of the de Boor points (B)?

Constructing B-splines

- Once again, the construction of Bezier points from de Boor points can be expressed in terms of a matrix:

$$\begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{pmatrix}\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix}$$

$$Q(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & \\ -3 & 3 & & \\ 1 & & & \end{bmatrix}\begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{pmatrix}$$

B-spline Basis Matrix

$$\mathbf{Q}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{B}_0 \\ \mathbf{B}_1 \\ \mathbf{B}_2 \\ \mathbf{B}_3 \end{bmatrix}$$

Displaying B-splines

- Drawing B-splines is therefore very simple:

DisplayBSpline (B_0,B_1,…,B_n)

    for i = 0 to n-3

        Convert B_i,…,B_{i+3} into Bezier control points V_0, …, V_3

        DisplayBezier(V_0,V_1,V_2,V_3)

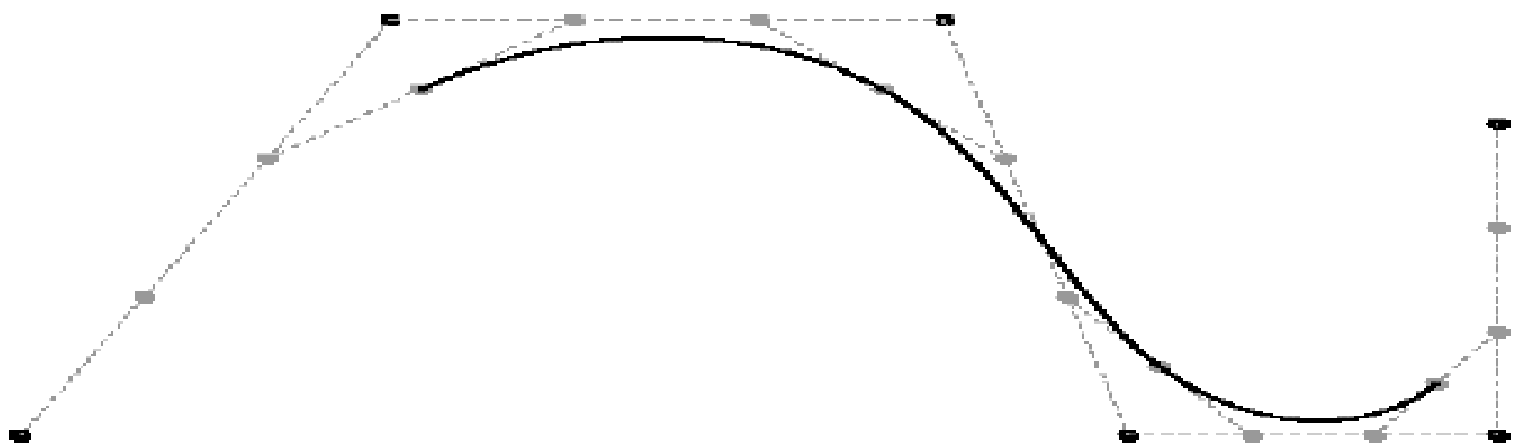
    endfor

B-spline Properties

- C^2 continuity
- Approximating
  - Does not interpolate de Boor points
- Locality
  - Each segment determined by 4 de Boor points (B).
  - Each de Boor point determines 4 segments.
- Convex hull
  - Curve lies inside the convex hull of de Boor points (the smallest convex polygon that contains the 4 de Boor points).

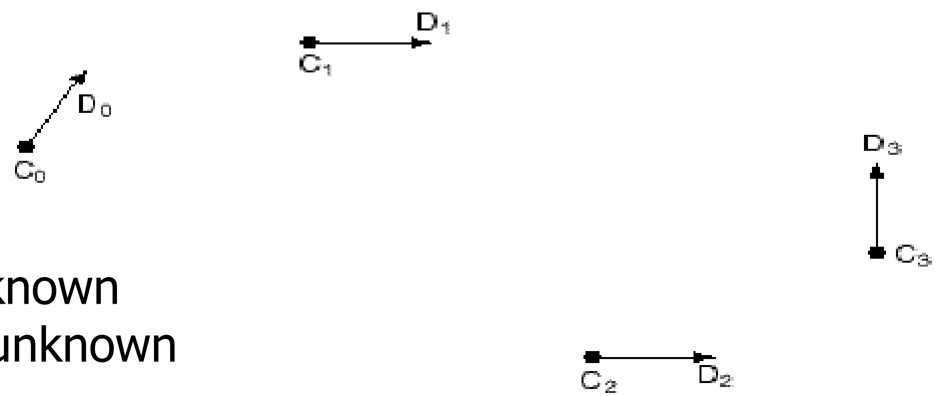
Endpoints of B-splines

- We can see that B-splines don't interpolate the de Boor points.
- It would be nice if we could at least control the *endpoints* of the splines explicitly.
- There's a hack to make the spline begin and end at control points, by repeating them.



C^2 Interpolating Splines

- Interpolation is a really handy property to have.
- How can we keep the C^2 continuity we get with B-splines but get the interpolation too?
- Here's the idea behind **C^2 interpolating splines**. Suppose we had cubic Beziers connecting our controls points C_0, C_1, C_2, …, and that we somehow knew the first derivative of the spline at each point.

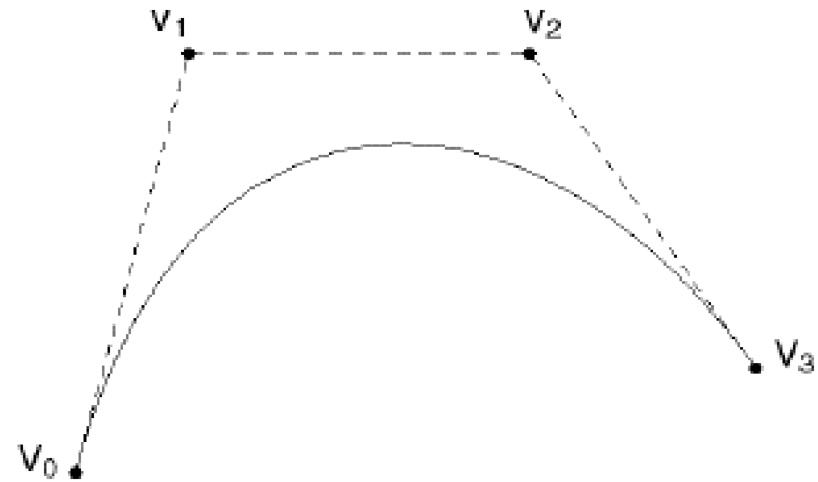


C_i's are known
D_i's are unknown

- What are the *V* and *W* (bezier control points) in terms of *C*s and *D*s?

Derivatives at the endpoints

$$Q'(0) = 3(V_1 - V_0)$$
$$Q'(1\ ) = 3(V_3 - V_2)$$
$$Q''(0) = 6(V_0 - 2V_1 + V_2)$$
$$Q''(1\ ) = 6(V_1 - 2V_2 + V_3)$$

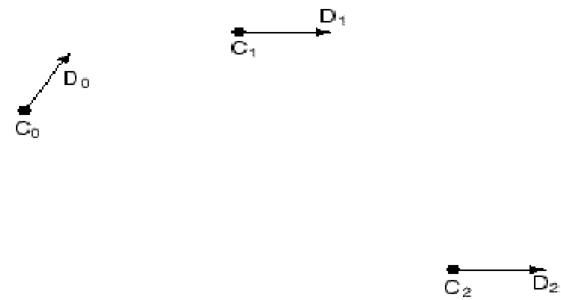


$$Q''(t) = \begin{bmatrix} 6t & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & \\ -3 & 3 & & \\ 1 & & & \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{bmatrix}$$

- In general, the nth derivative at an endpoint depends only on the (n+1)th nearest that endpoint.

Finding the Derivatives

- Now what we need to do is to solve for the derivatives D's. To do this, we'll use the C^2 continuity requirement.



$$V_0 = C_0 \qquad W_0 = C_1$$

$$V_1 = C_0 + \tfrac{1}{3} D_0 \qquad W_1 = C_1 + \tfrac{1}{3} D_1$$

$$V_2 = C_1 - \tfrac{1}{3} D_1 \qquad W_2 = C_2 - \tfrac{1}{3} D_2$$

$$V_3 = C_1 \qquad W_3 = C_2$$

$$6(V_1 - 2V_2 + V_3) = 6(W_0 - 2W_1 + W_2)$$

$$\Downarrow$$

$$D_0 + 4D_1 + D_2 = 3(C_2 - C_0)$$

Finding the Derivatives

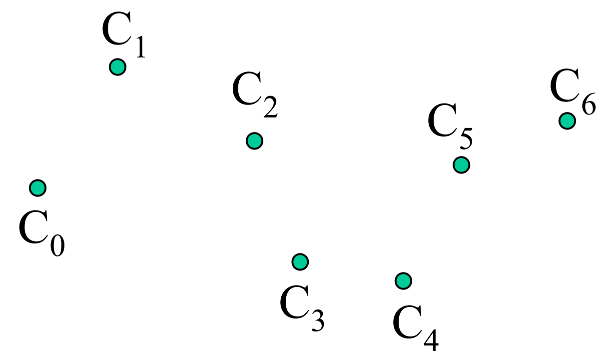
- Here's what we've got so far:

$$D_0 + 4D_1 + D_2 = 3(C_2 - C_0)$$

$$D_1 + 4D_2 + D_3 = 3(C_3 - C_1)$$

$$\vdots$$

$$D_{m-2} + 4D_{m-1} + D_m = 3(C_m - C_{m-2})$$



- How many equations are there?

- How many unknowns are we solving for?

Not quite done yet!

- We have two additional degrees of freedom, which we can nail down by imposing more conditions on the curve.
- There are various ways to do this. We'll use the variant called natural C^2 interpolating splines, which requires that second derivative to be zero at the endpoints.
- This condition gives us the two additional equations we need. At the C_0 endpoint, it is:

$$6(V_0 - 2V_1 + V_2) = 0$$

Solving for the Derivatives

- Let's collect our m+1 equations into a single linear system:

$$\begin{bmatrix} 2 & 1 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & & \ddots & & \\ & & & 1 & 4 & 1 \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} D_0 \\ D_1 \\ D_2 \\ \vdots \\ D_{m-1} \\ D_m \end{bmatrix} = \begin{bmatrix} 3(C_1 - C_0) \\ 3(C_2 - C_0) \\ 3(C_3 - C_1) \\ \vdots \\ 3(C_m - C_{m-2}) \\ 3(C_m - C_{m-1}) \end{bmatrix}$$

$$6(V_0 - 2V_1 + V_2) = 0$$

- It's easier to solve than it looks.
- We can use **forward elimination** to zero out everything below the diagonal, then **back substitution** to compute each D value.

Forward Elimination

- First, we eliminate the elements below the diagonal.

$$\begin{bmatrix} 2 & 1 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & & \ddots & & \\ & & & 1 & 4 & 1 \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} \mathbf{E}_0 \\ \mathbf{E}_1 \\ \mathbf{E}_2 \\ \vdots \\ \mathbf{E}_{m-1} \\ \mathbf{E}_m \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 & & & & \\ 0 & 7/2 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & & \ddots & & \\ & & & 1 & 4 & 1 \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} \mathbf{F}_0 = \mathbf{E}_0 \\ \mathbf{F}_1 = \mathbf{E}_1 - (1/2)\mathbf{E}_0 \\ \mathbf{E}_2 \\ \vdots \\ \mathbf{E}_{m-1} \\ \mathbf{E}_m \end{bmatrix}$$

Back Substitution

- The resulting matrix is **upper triangular**:

$$\mathbf{U}\mathbf{D} = \mathbf{F}$$

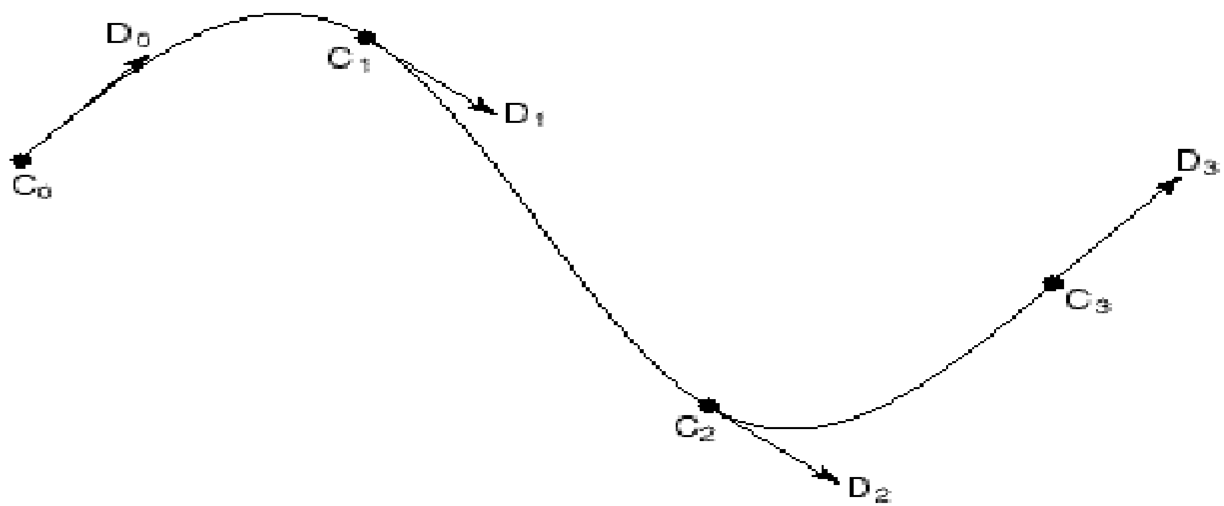
$$\begin{bmatrix} u_{11} & \cdots & u_{1m} \\ & & \\ & & \\ & \ddots & \vdots \\ & & \\ & & u_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_{m-1} \\ \mathbf{D}_m \end{bmatrix} = \begin{bmatrix} \mathbf{F}_0 \\ \mathbf{F}_1 \\ \mathbf{F}_2 \\ \vdots \\ \mathbf{F}_{m-1} \\ \mathbf{F}_m \end{bmatrix}$$

- We can now solve for the unknowns by back substitution:

$$\begin{aligned} u_{mm}\mathbf{D}_m &= \mathbf{F}_m \\ u_{m-1m-1}\mathbf{D}_{m-1} + u_{m-1m}\mathbf{D}_m &= \mathbf{F}_{m-1} \end{aligned}$$

C^2 Interpolating Spline

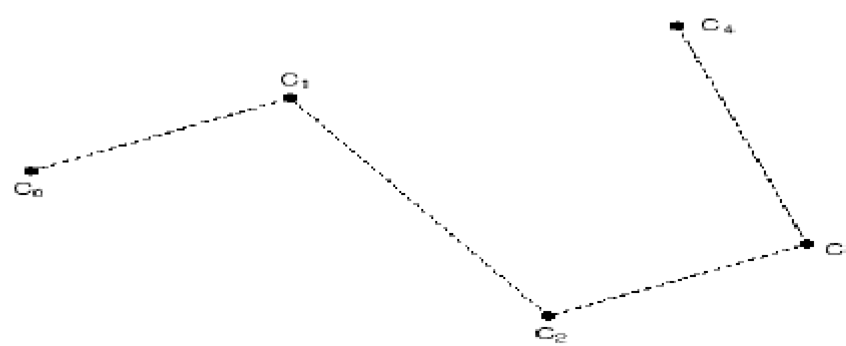
- Once we've solved for the real D's, we can plug them in to find our Bezier control points and draw the final spline:



- Have we lost anything?

A Third Option

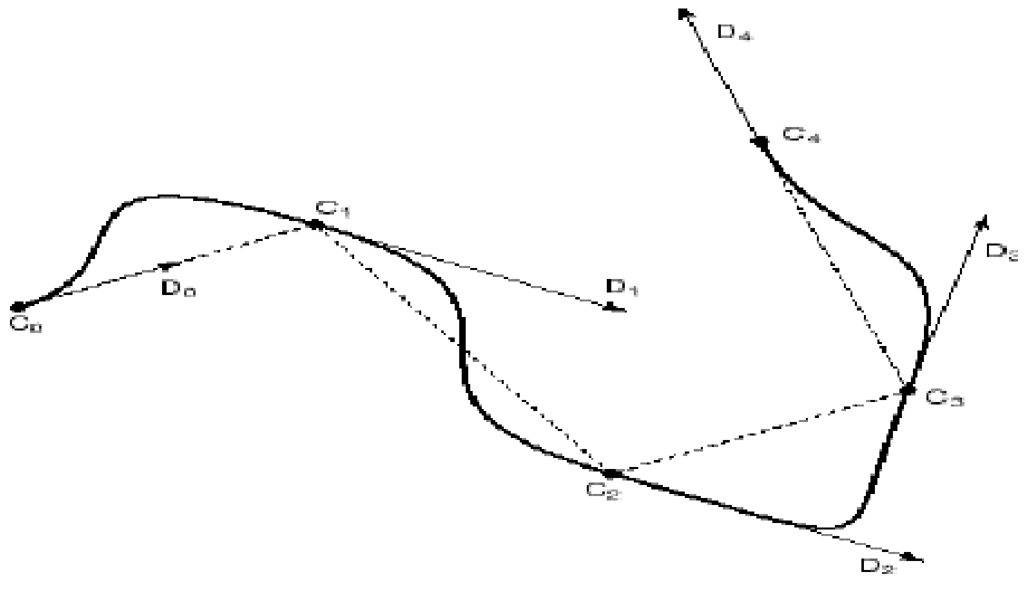
- If we're willing to sacrifice C^2 continuity, we can get interpolation *and* local control.
- Instead of finding the derivatives by solving a system of continuity equations, we'll just pick something arbitrary but local.
- If we set each derivative to be constant multiple of the vector between the previous and the next controls, we can a **Catmull-Rom spline**.



Catmull-Rom Splines

- The math for Catmull-Rom splines is pretty simple:

$$
\begin{aligned}
D_0 &= C_1 - C_0 \\
D_1 &= \tfrac{1}{2}(C_2 - C_0) \\
D_2 &= \tfrac{1}{2}(C_3 - C_1) \\
&\vdots \\
D_n &= C_n - C_{n-1}
\end{aligned}
$$



Catmull-Rom Basis Matrix

$$\mathbf{Q}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \\ \mathbf{P}_4 \end{bmatrix}$$

Tension control

- We can give more control by expressing the derivative scale factor as a parameter:

$$V_0 = P_1$$

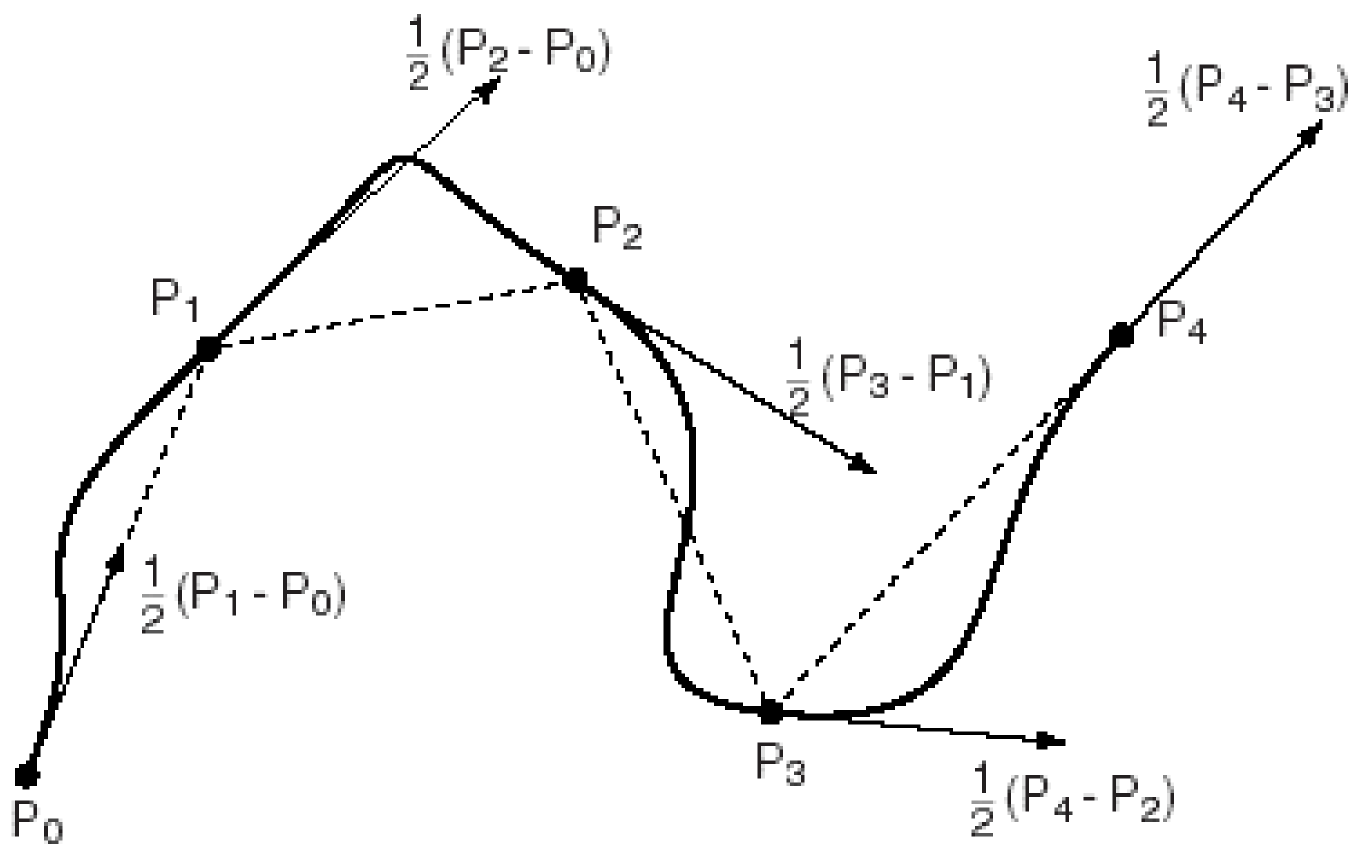
$$V_1 = P_1 + \tfrac{\tau}{3}(P_2 - P_0)$$

$$V_2 = P_2 - \tfrac{\tau}{3}(P_3 - P_1)$$

$$V_3 = P_2$$

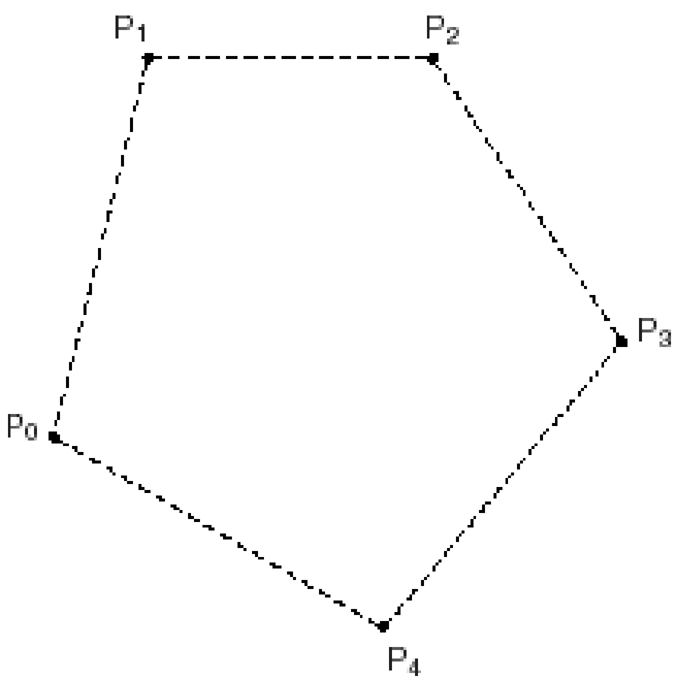
- The parameter τ controls the tension. Catmull-Rom uses τ=1/2. Here's an example with τ =3/2.

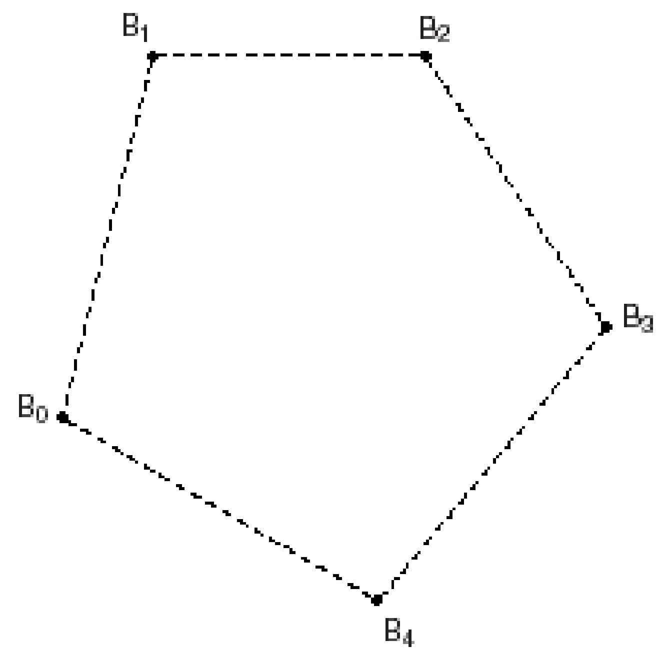
Tension control



Closing the loop

- What if we want a closed curve, i.e., a loop?
- With Catmull-Rom and B-spline curves, this is easy:





Curves in the animator project

- In the animator project, you will draw a curve on the screen:

$$\mathbf{Q}(u) = (x(u), y(u))$$

- You will actually treat this curve as:

$$\theta(u) = y(u)$$

$$t(u) = x(u)$$

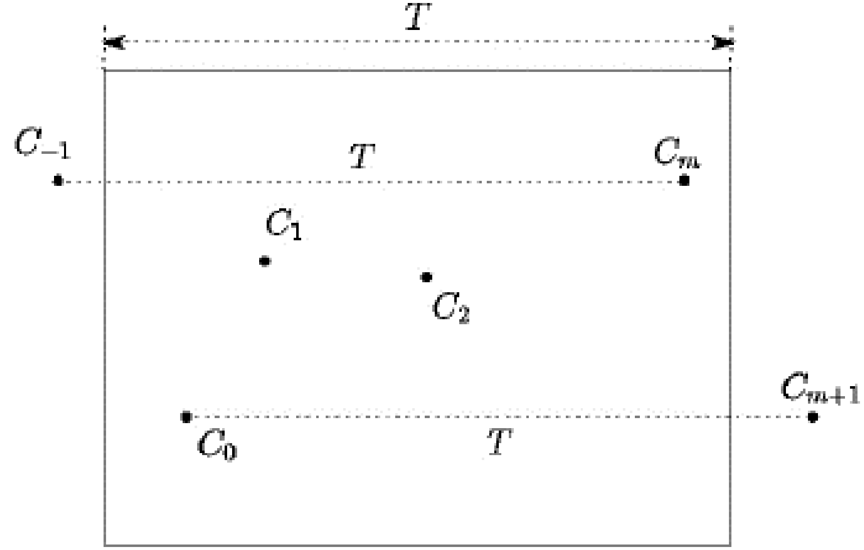
- When η is a variable you want to animate. We can think of the result as a function:

$$\theta(t)$$

- You have to apply some constraints to make sure $\eta(t)$ is a *function*.

"Wrapping"

- One of the requirements is to implement "wrapping" so that the animation restarts smoothly when looping back to the beginning.
- This is a lot like making a closed curve: the calculations for the η-coordinate are exactly the same.
- The t-coordinate is a little trickier: you need to create "phantom" t-coordinates before and after the first and last coordinates.



Summary

What to take home from this lecture:

- How to display Bézier curves with line segments.
- Meanings of C^k continuities.
- Geometric conditions for continuity of cubic splines.
- Properties of C^2 interpolating splines, B-splines.
- Construction of B-splines and Catmull-Rom splines.

$$Q(u) = \sum_{i=0}^{3} V_i \binom{3}{i} u^i (1-u)^{3-i}$$

$$= (1 - u)^3 V_0 + 3u\,(1 - u)^2 V_1 + 3u^2 (1 - u)\, V_2 + u^3 V_3$$

$$= \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{pmatrix}$$

$$= \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \; M_{Bezier} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \\ V_3 \end{pmatrix}$$