12. Parametric curves

Reading

Required:

- Watt, 3 (intro), 3.1, 3.2 (intro), 3.2.1, 3.2.2

Optional

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

Curves before computers

The "loftsman's spline":

- long, narrow strip of wood or metal
- shaped by lead weights called "ducks"
- gives curves with second-order continuity, usually

Used for designing cars, ships, airplanes, etc.

But curves based on physical artifacts can't be replicated well, since there's no exact definition of what the curve is.

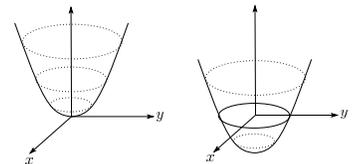Around 1960, a lot of industrial designers were working on this problem.

Today, curves are easy to manipulate on a computer and are used for CAD, art, animation, …

Mathematical curve representation

- Explicit $y=f(x)$
  - what if the curve isn't a function, e.g., a circle?

- Implicit $g(x,y) = 0$

- Parametric $(x(u),y(u))$
  - For the circle:
    $x(u) = \cos 2\pi u$
    $y(u) = \sin 2\pi u$

Parametric polynomial curves

We'll use parametric curves, $Q(u)=(x(u),y(u))$, where the functions are all polynomials in the parameter.

$$x(u) = \sum_{k=0}^{n} a_k u^k$$
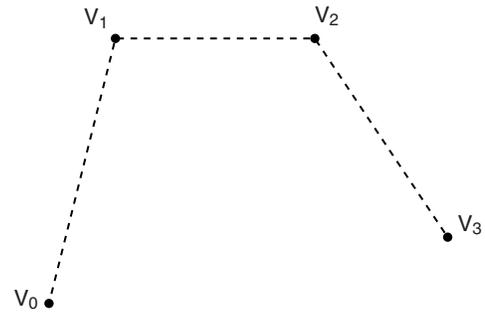
$$y(u) = \sum_{k=0}^{n} b_k u^k$$

Advantages:

- ◆ easy (and efficient) to compute
- ◆ infinitely differentiable

We'll also assume that u varies from 0 to 1.

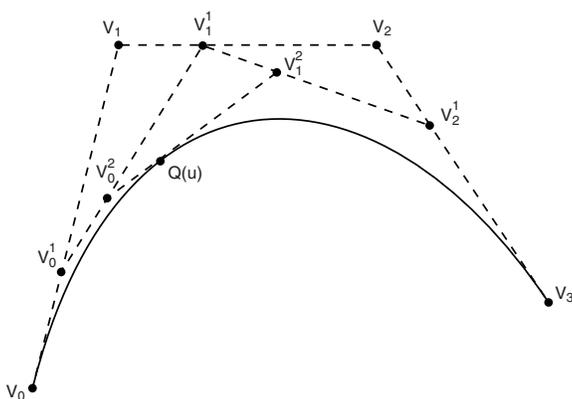## de Casteljau's algorithm

Recursive interpolation:

What if $u=0$?

What if $u=1$?

de Casteljau's algorithm, cont'd

Recursive notation:

What is the equation for V_0^1?

Finding Q(u)

Let's solve for $\mathbf{Q}(u)$:

$$V_0^1 = (1-u)V_0 + uV_1$$
$$V_1^1 = (1-u)V_1 + uV_2$$
$$V_2^1 = (1-u)V_2 + uV_3$$

$$V_0^2 = (1-u)V_0^1 + uV_1^1$$
$$V_1^2 = (1-u)V_1^1 + uV_2^1$$

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

Finding Q(u)   (cont'd)

In general,

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

where "*n* choose *i*" is:
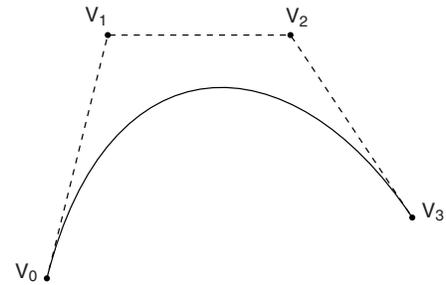
$$\binom{n}{i} = \frac{n!}{(n-i)!i!}$$

This defines a class of curves called **Bézier curves**.

What's the relationship between the number of control points and the degree of the polynomials?

Displaying Bézier curves
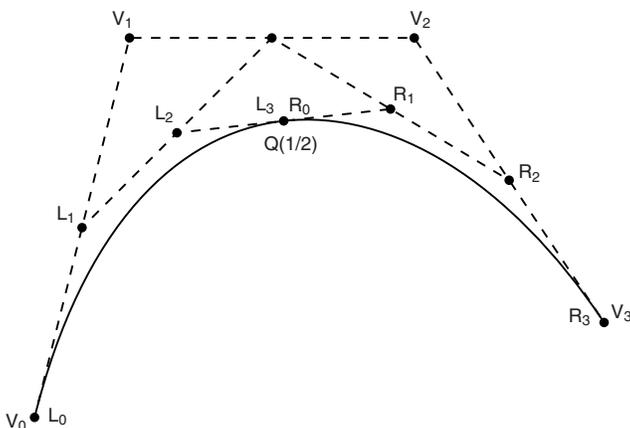
How could we draw one of these things?

It would be nice if we had an *adaptive* algorithm, that would take into account flatness.

```
DisplayBezier( V0, V1, V2, V3 )
begin
    if ( FlatEnough( V0, V1, V2, V3 ) )
        Line( V0, V3 );
    else
        something;
end;
```
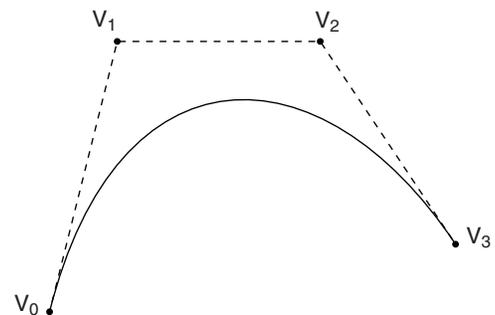
Subdivide and conquer


```
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 total length of control polygon to length of line connecting 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 Bézier?

Instead, we'll splice together a curve from individual segments that are cubic Béziers.
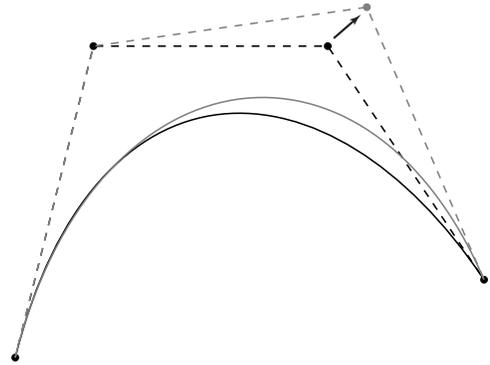
Why cubic?

There are three properties we'd like to have in our newly constructed splines…

Local control

One problem with Béziers 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.
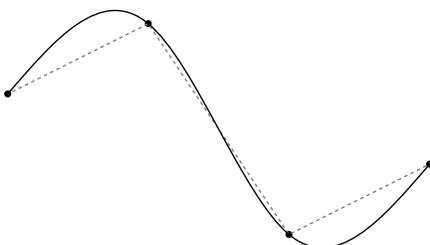
Interpolation

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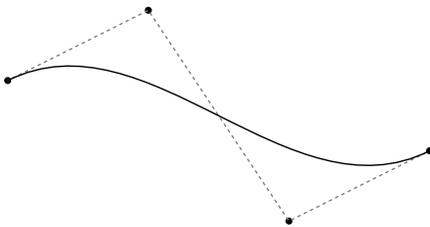
We'd like to have a spline that is **interpolating**, that is, that always passes through every control point.
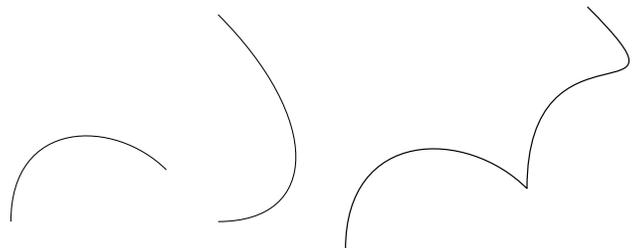
Continuity

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

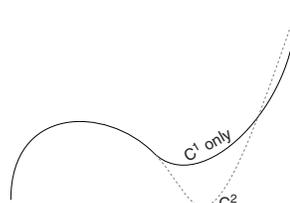There are *nested* degrees of continuity:

C^{-1}: C^{0}:

C^{1}, C^{2} : $C^{3}, C^{4}, …$:

C^{1} only

C^{2}

Ensuring continuity

Let's look at continuity first.

Since the functions defining a Bézier curve are polynomial, all their derivatives exist and are continuous on the interior of the curve.

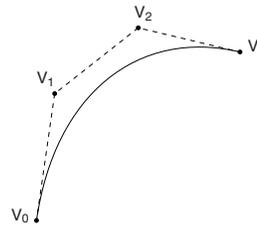Therefore, we only need to worry about the derivatives at the endpoints of the curve.

Ensuring C^0 continuity

Suppose we have a cubic Bézier defined by (V_0, V_1, V_2, V_3), and we want to attach another curve (W_0, W_1, W_2, W_3) to it, so that there is C^0 continuity at the joint.
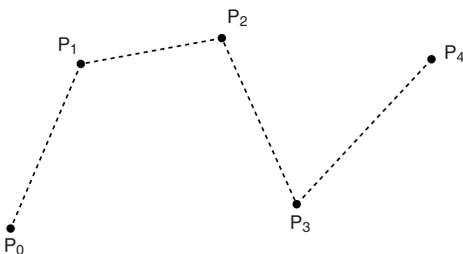
$$C^0 : Q_V(1) = Q_W(0)$$

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

The C^0 Bezier spline

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

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

In the animator project, you will construct such a curve by specifying all the Bezier control points directly.

1st derivatives at the endpoints

For degree 3 (cubic) curves, we have already shown that we get:

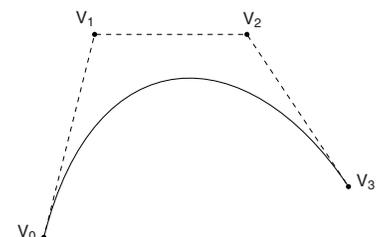$$Q(u) = (1-u)^3 V_0 + 3u(1-u)^2 V_1 + 3u^2(1-u)V_2 + u^3 V_3$$

We can expand the terms in u and rearrange to get:

$$Q(u) = \ (-V_0 + 3V_1 - 3V_2 + V_3)u^3 \ + $$
$$(3V_0 - 6V_1 + 3V_2)u^2 \ + \ (-3V_0 + 3V_1)u \ + \ V_0$$

What then is the first derivative when evaluated at each endpoint, $u=0$ and $u=1$?
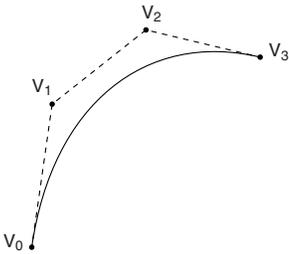
$$Q'(0) =$$

$$Q'(1) =$$

Ensuring C^1 continuity

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

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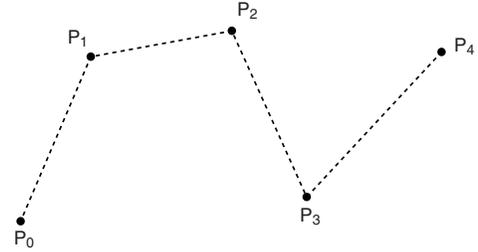$$C^1 : Q_V^{'}(1) = Q_W^{'}(0)$$

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 $P_1 \ldots P_n$?

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

Catmull-Rom splines

If we set each derivative to be one half of the vector between the previous and next controls, we get a **Catmull-Rom spline**.
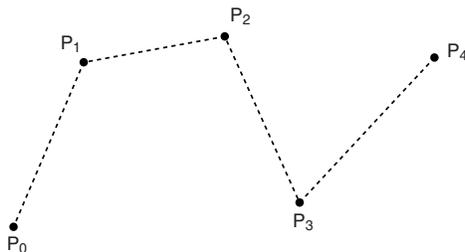
This leads to:

$$V_0 = P_1$$
$$V_1 = P_1 + \tfrac{1}{6}(P_2 - P_0)$$
$$V_2 = P_2 - \tfrac{1}{6}(P_3 - P_1)$$
$$V_3 = P_2$$

Tension control

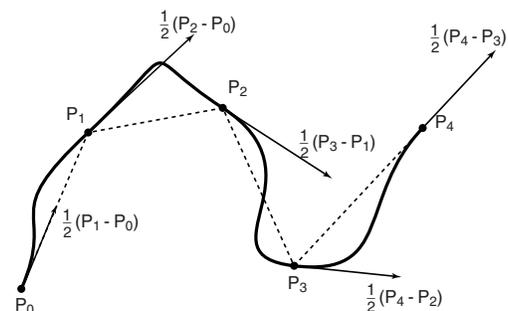We can give more control by exposing the derivative scale factor as a parameter:

$$V_0 = P_1$$
$$V_1 = P_1 + \tfrac{\tau}{3}(P_2 - P_0)$$
$$V_2 = P_2 - \tfrac{\tau}{3}(P_3 - P_1)$$
$$V_3 = P_2$$

The parameter τ controls the tension. Catmull-Rom uses $\tau = 1/2$. Here's an example with $\tau = 3/2$.
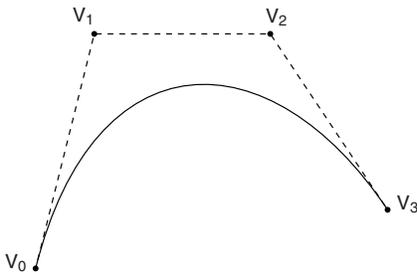
2nd derivatives at the endpoints

Finally, we'll want to develop C^2 splines. To do this, we'll need second derivatives of Bezier curves.

Taking the second derivative of $Q(u)$ yields:

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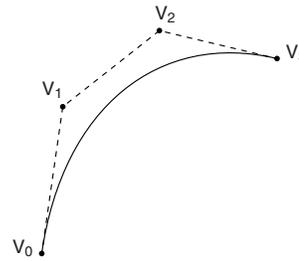$$= -6[(V_2 - V_3) + (V_2 - V_1)]$$

Ensuring C^2 continuity

Suppose we have a cubic Bézier defined by (V_0, V_1, V_2, V_3), and we want to attach another curve (W_0, W_1, W_2, W_3) to it, so that there is C^2 continuity at the joint.

$$C^0 : Q_V(1) = Q_W(0)$$
$$C^1 : Q'_V(1) = Q'_W(0)$$
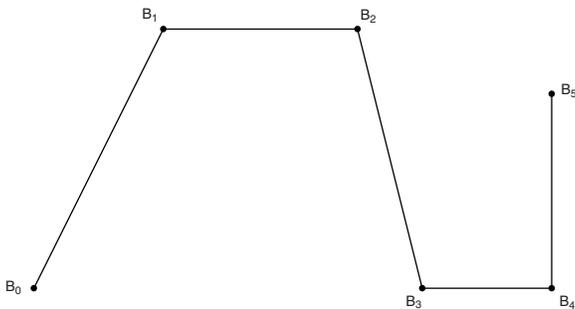$$C^2 : Q''_V(1) = Q''_W(0)$$

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

Building a complex spline

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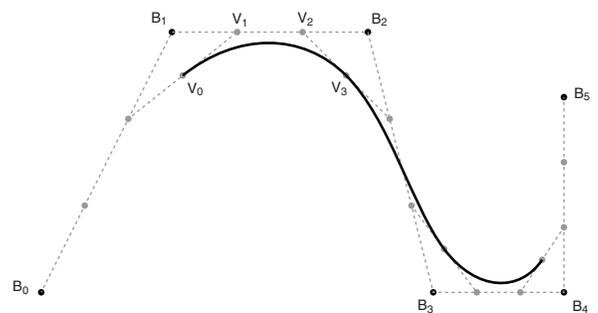

These are called **B-splines**. The starting set of points are called **de Boor points**.

B-splines

Here is the completed B-spline.

What are the Bézier control points, in terms of the de Boor points?

$$V_0 = \underline{\quad}[\underline{\quad}B_0 + \underline{\quad}B_1]$$
$$+ \underline{\quad}[\underline{\quad}B_1 + \underline{\quad}B_2]$$
$$= \underline{\quad}B_0 + \underline{\quad}B_1 + \underline{\quad}B_2$$
$$V_1 = \underline{\quad}B_1 + \underline{\quad}B_2$$
$$V_2 = \underline{\quad}B_1 + \underline{\quad}B_2$$
$$V_3 = \underline{\quad}B_1 + \underline{\quad}B_2 + \underline{\quad}B_3$$

B-splines and Beziers

We can write the B-spline to Bezier transformation as:

$$\begin{bmatrix} V_0^T \\ V_1^T \\ V_2^T \\ V_3^T \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \begin{bmatrix} B_0^T \\ B_1^T \\ B_2^T \\ B_3^T \end{bmatrix}$$
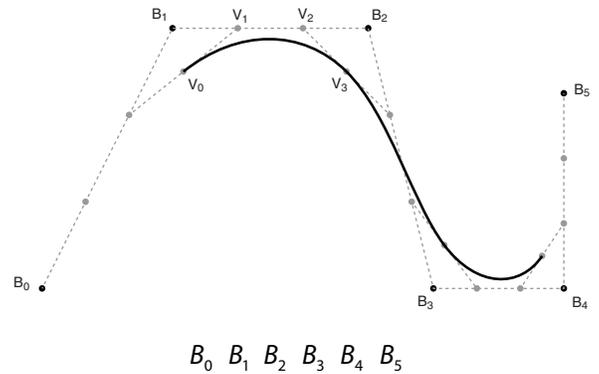
$$\mathbf{V} = \mathbf{M_{B\text{-spline}}B}$$

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 trick to make the spline begin and end at control points by repeating them.

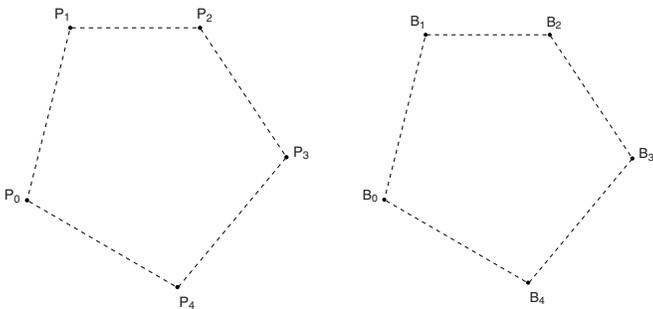In the example below, let's force interpolation of the last endpoint:

$$B_0 \;\; B_1 \;\; B_2 \;\; B_3 \;\; B_4 \;\; B_5$$

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:

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

You will actually treat this curve as:

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

Where θ is a variable you want to animate. We can think of the result as a function:

$$\theta(t)$$

In general, you have to apply some constraints to make sure that $\theta(t)$ actually is a *function*.

"Wrapping"

One of the extra credit options in the animator project 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 θ-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.

33