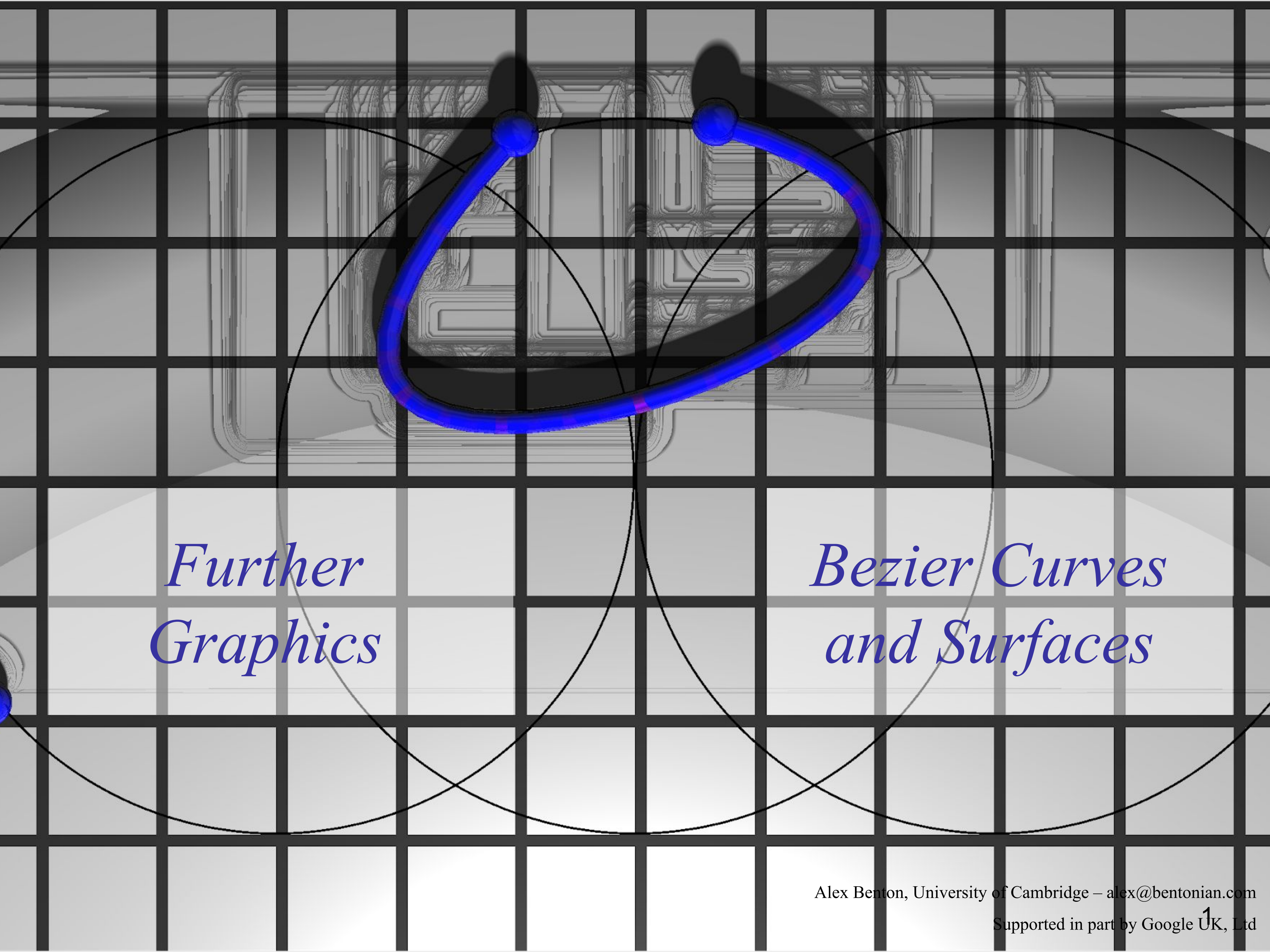


Further Graphics

Bezier Curves and Surfaces

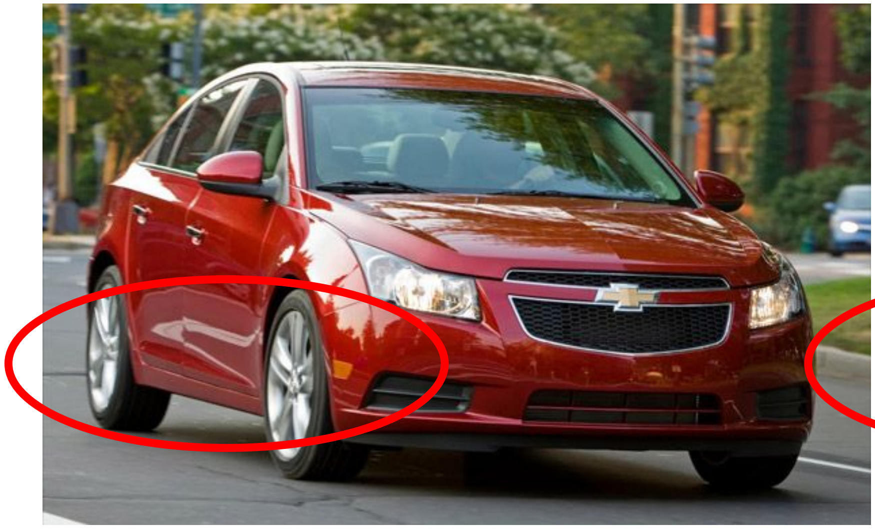
Alex Benton, University of Cambridge – alex@bentonian.com

Supported in part by Google UK, Ltd

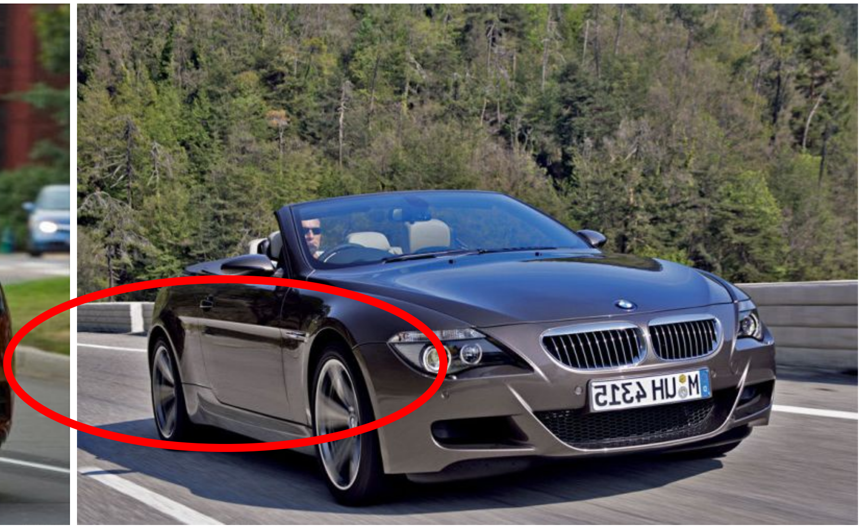
CAD, CAM, and a new motivation: *shiny things*

Expensive products are sleek and smooth.

→ Expensive products are C2 continuous.



Shiny, but reflections are warped



Shiny, and reflections are perfect

The drive for smooth CAD/CAM

- *Continuity* (smooth curves) can be essential to the perception of *quality*.
- The automotive industry wanted to design cars which were aerodynamic, but also visibly of high quality.
- Bezier (Renault) and de Casteljau (Citroen) invented Bezier curves in the 1960s. de Boor (GM) generalized them to B-splines.

History

The term *spline* comes from the shipbuilding industry: long, thin strips of wood or metal would be bent and held in place by heavy 'ducks', lead weights which acted as control points of the curve.

Wooden splines can be described by C_n-continuous Hermite polynomials which interpolate n+1 control points.

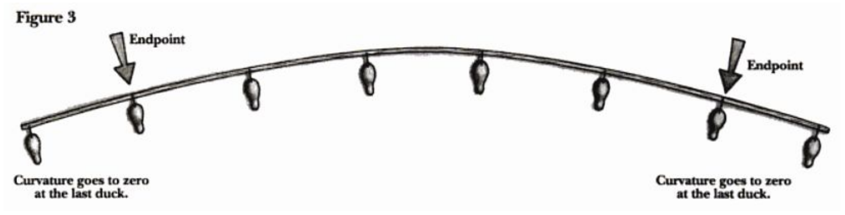
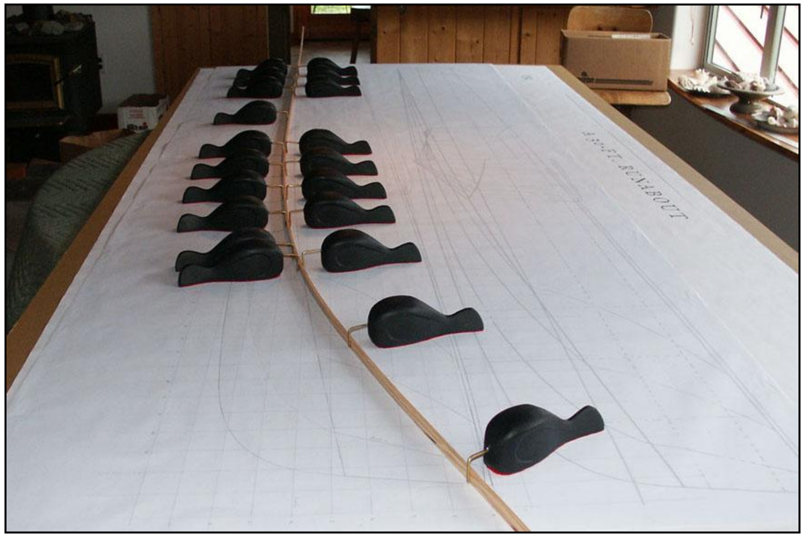


Figure 3
Endpoint
Endpoint
Curvature goes to zero at the last duck.
Curvature goes to zero at the last duck.



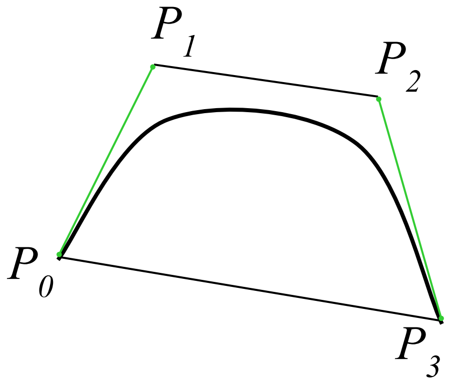
Top: Fig 3, P.7, Bray and Spectre, *Planking and Fastening*, Wooden Boat Pub (1996)

Bottom: http://www.pranos.com/boatsofwood/lofting%20ducks/lofting_ducks.htm

Bezier cubic

- A *Bezier cubic* is a function P(t) defined by four control points:

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



- P_0 and P_3 are the endpoints of the curve
- P_1 and P_2 define the other two corners of the bounding polygon.
- The curve fits entirely within the convex hull of $P_0 \ldots P_3$.

Beziers

Cubics are just one example of Bezier splines:

- Linear: $P(t) = (1-t)P_0 + tP_1$
- Quadratic: $P(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2$
- Cubic: $P(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3$

…

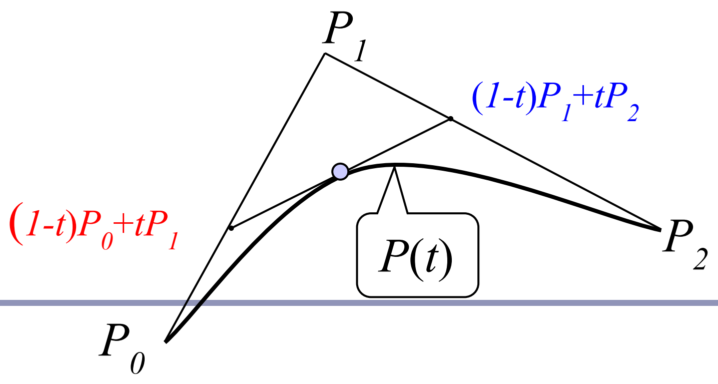
"n choose i" = n! / i!(n-i)!

General:

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

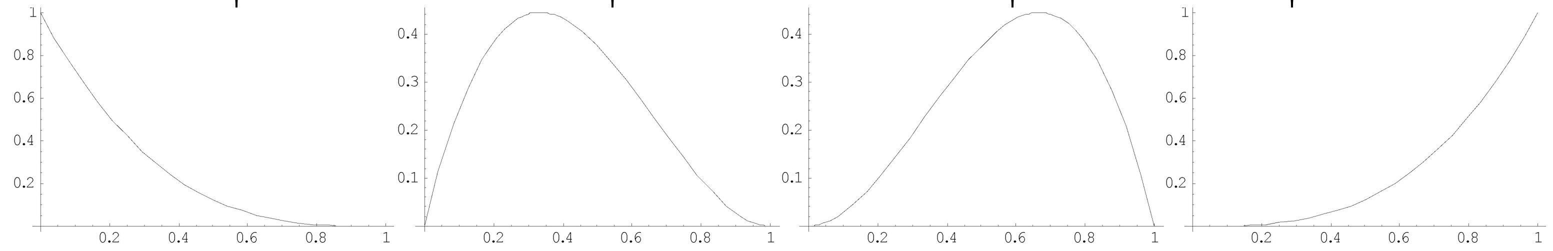
Beziers

- You can describe Beziers as *nested linear interpolations:*
  - The linear Bezier is a linear interpolation between two points: $P(t) = (1\text{-}t)\,(P_0) + (t)\,(P_1)$
  - The quadratic Bezier is a linear interpolation between two lines: $P(t) = (1\text{-}t)\,((1\text{-}t)P_0+tP_1) + (t)\,((1\text{-}t)P_1+tP_2)$
  - The cubic is a linear interpolation between linear interpolations between linear interpolations… etc.
- Another way to see Beziers is as a *weighted average* between the control points.



Bernstein polynomials

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



- The four control functions are the four *Bernstein polynomials* for *n=3*.
  - General form: $b_{v,n}(t) = \binom{n}{v} t^v (1-t)^{n-v}$
  - Bernstein polynomials in $0 \leq t \leq 1$ always sum to 1:

$$\sum_{v=1}^{n} \binom{n}{v} t^v (1-t)^{n-v} = (t + (1-t))^n = 1$$

Drawing a Bezier cubic: Iterative method

Fixed-step iteration:

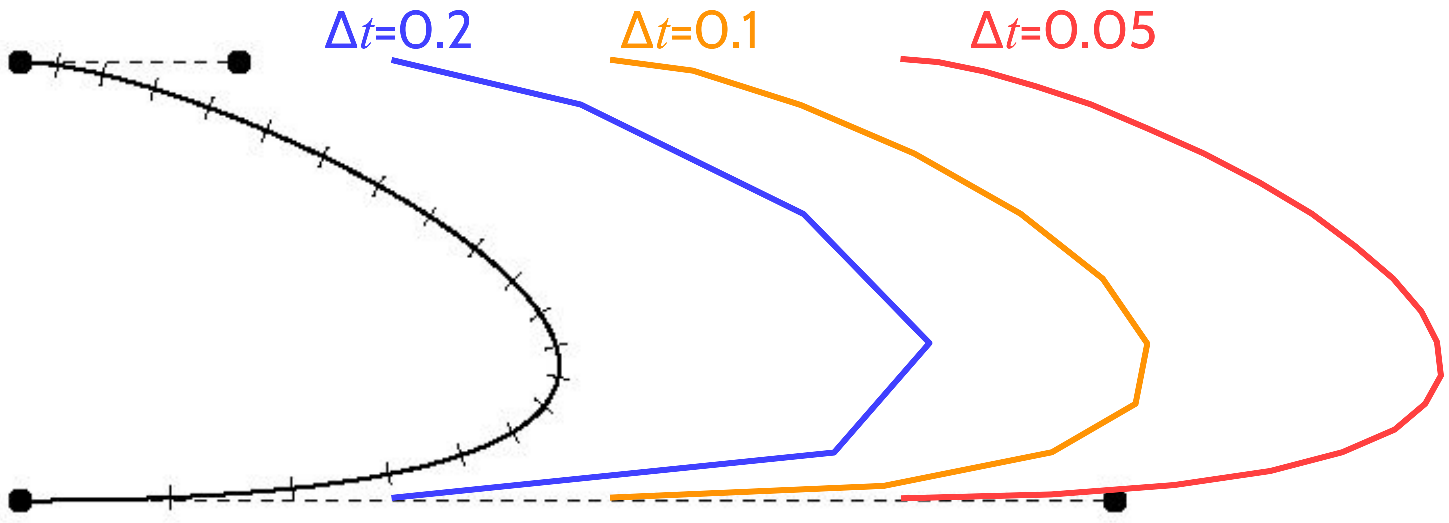
- Draw as a set of short line segments equispaced in parameter space, *t*:

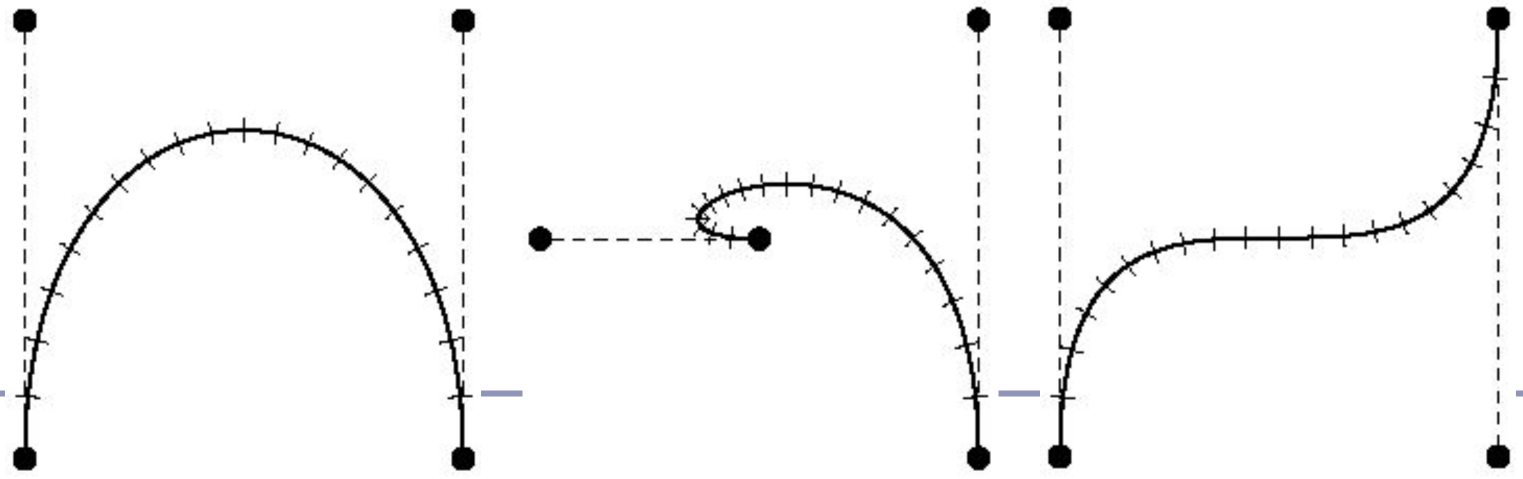
```
(x0,y0) = Bezier(0)
FOR t = 0.05 TO 1 STEP 0.05 DO
    (x1,y1) = Bezier(t)
    DrawLine( (x0,y0), (x1,y1) )
    (x0,y0) = (x1,y1)
END FOR
```

- Problems:
  - Cannot fix a number of segments that is appropriate for all possible Beziers: too many or too few segments
  - distance in real space, *(x,y)*, is not linearly related to distance in parameter space, *t*

Drawing a Bezier cubic

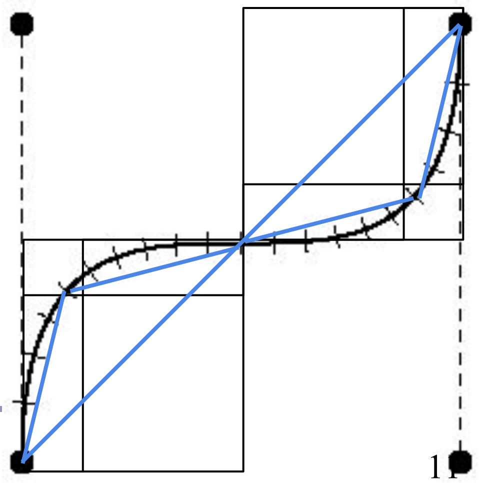
...but not very well





Drawing a Bezier cubic: Adaptive method

- Subdivision:
  - check if a straight line between P_0 and P_3 is an adequate approximation to the Bezier
  - if so: draw the straight line
  - if not: divide the Bezier into two halves, each a Bezier, and repeat for the two new Beziers
- Need to specify some tolerance for when a straight line is an adequate approximation
  - when the Bezier lies within half a pixel width of the straight line along its entire length



Drawing a Bezier cubic: Adaptive method

```
Procedure DrawCurve( Bezier curve )
VAR Bezier left, right
BEGIN DrawCurve
  IF Flat(curve) THEN
     DrawLine(curve)
  ELSE
     SubdivideCurve(curve, left, right)
     DrawCurve(left)
     DrawCurve(right)
  END IF
END DrawCurve
```

e.g. if P_1 and P_2 both lie within half a pixel width of the line joining P_0 to P_3, then...

...draw a line from P_0 to P_3; otherwise,

...split the curve into two Beziers covering the first and second halves of the original and draw recursively

Checking for flatness

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

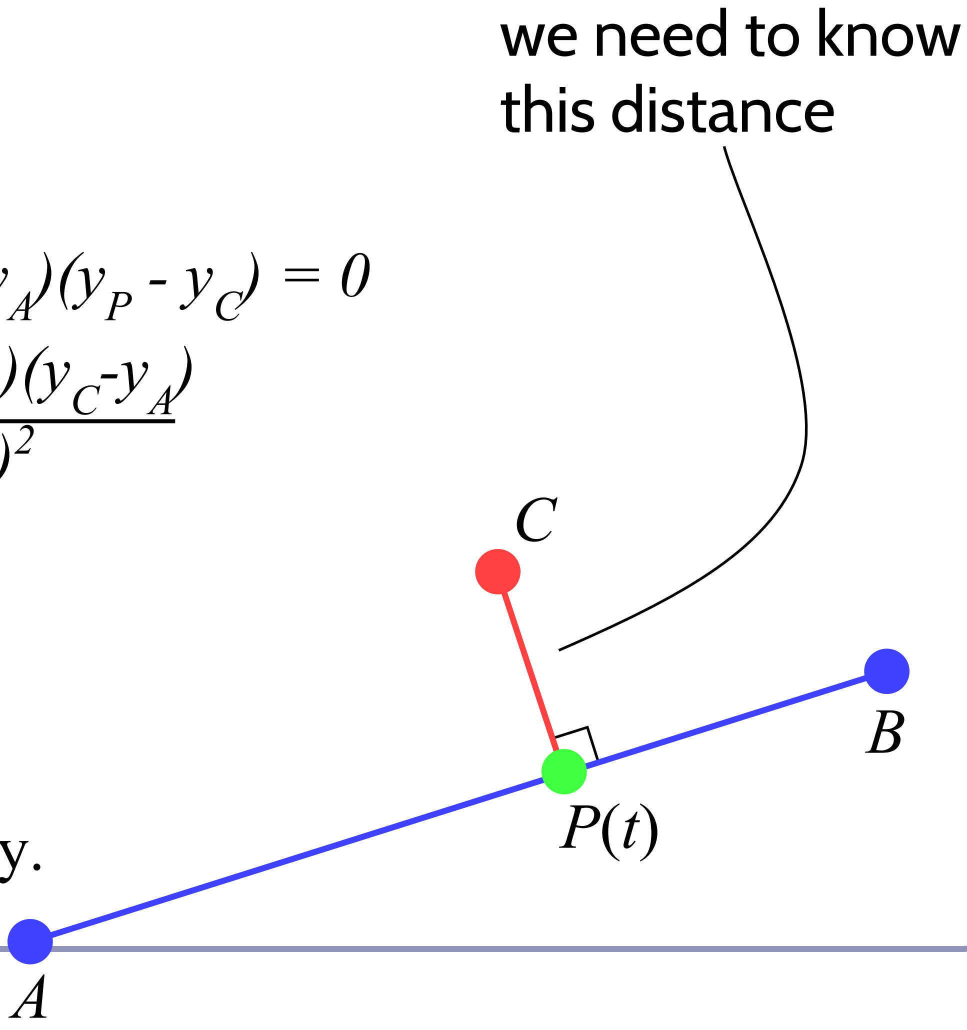
$AB \cdot CP(t) = 0$

$\rightarrow (x_B - x_A)(x_P - x_C) + (y_B - y_A)(y_P - y_C) = 0$

$\rightarrow t = \dfrac{(x_B-x_A)(x_C-x_A)+(y_B-y_A)(y_C-y_A)}{(x_B-x_A)^2+(y_B-y_A)^2}$

$\rightarrow t = \dfrac{AB \cdot AC}{|AB|^2}$

Careful! If $t < 0$ or $t > 1$,
use $|AC|$ or $|BC|$ respectively.



Subdividing a Bezier cubic in two

To split a Bezier cubic into two smaller Bezier cubics:

$Q_0 = P_0$

$Q_1 = \frac{1}{2} P_0 + \frac{1}{2} P_1$

$Q_2 = \frac{1}{4} P_0 + \frac{1}{2} P_1 + \frac{1}{4} P_2$

$Q_3 = \frac{1}{8} P_0 + \frac{3}{8} P_1 + \frac{3}{8} P_2 + \frac{1}{8} P_3$

$R_3 = \frac{1}{8} P_0 + \frac{3}{8} P_1 + \frac{3}{8} P_2 + \frac{1}{8} P_3$

$R_2 = \frac{1}{4} P_1 + \frac{1}{2} P_2 + \frac{1}{4} P_3$

$R_1 = \frac{1}{2} P_2 + \frac{1}{2} P_3$

$R_0 = P_3$

These cubics will lie atop the halves of their parent exactly, so rendering them = rendering the parent.

Drawing a Bezier cubic: Signed Distance Fields

1. Iterative implementation

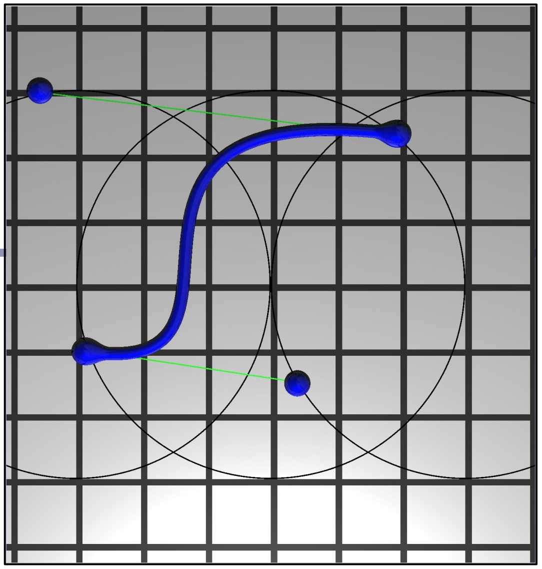
   SDF(P) = *min*(distance from *P* to each of *n* line segments)

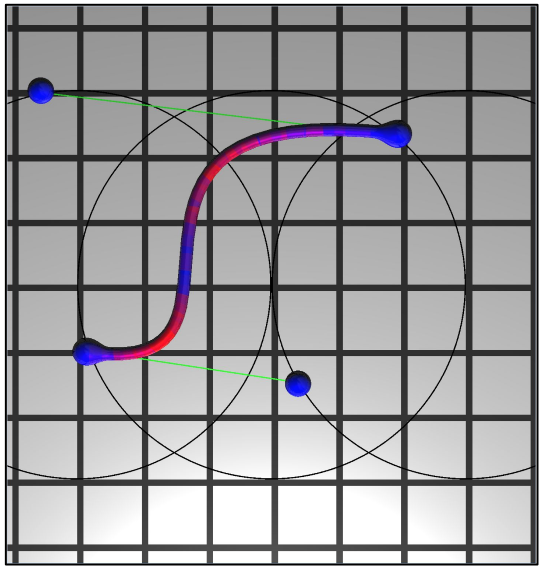
   - In the demo, 50 steps suffices

2. Adaptive implementation

   SDF(P) = *min*(distance to each sub-curve whose bounding box contains *P*)

   - Can fast-discard sub-curves whose bbox doesn't contain *P*
   - In the demo, 25 subdivisions suffices



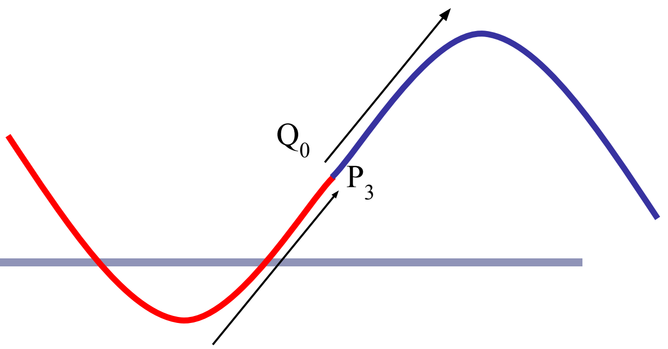


Overhauser's cubic

Overhauser's cubic: a Bezier cubic which passes through four target data points

- Calculate the appropriate Bezier control point locations from the given data points
  - e.g. given points A, B, C, D, the Bezier control points are:
  - P0 = B   P1 = B + (C-A)/6
  - P3 = C   P2 = C - (D-B)/6
- Overhauser's cubic *interpolates* its controlling points
  - good for animation, movies; less for CAD/CAM
  - moving a single point modifies four adjacent curve segments
  - compare with Bezier, where moving a single point modifies just the two segments connected to that point

Types of curve join

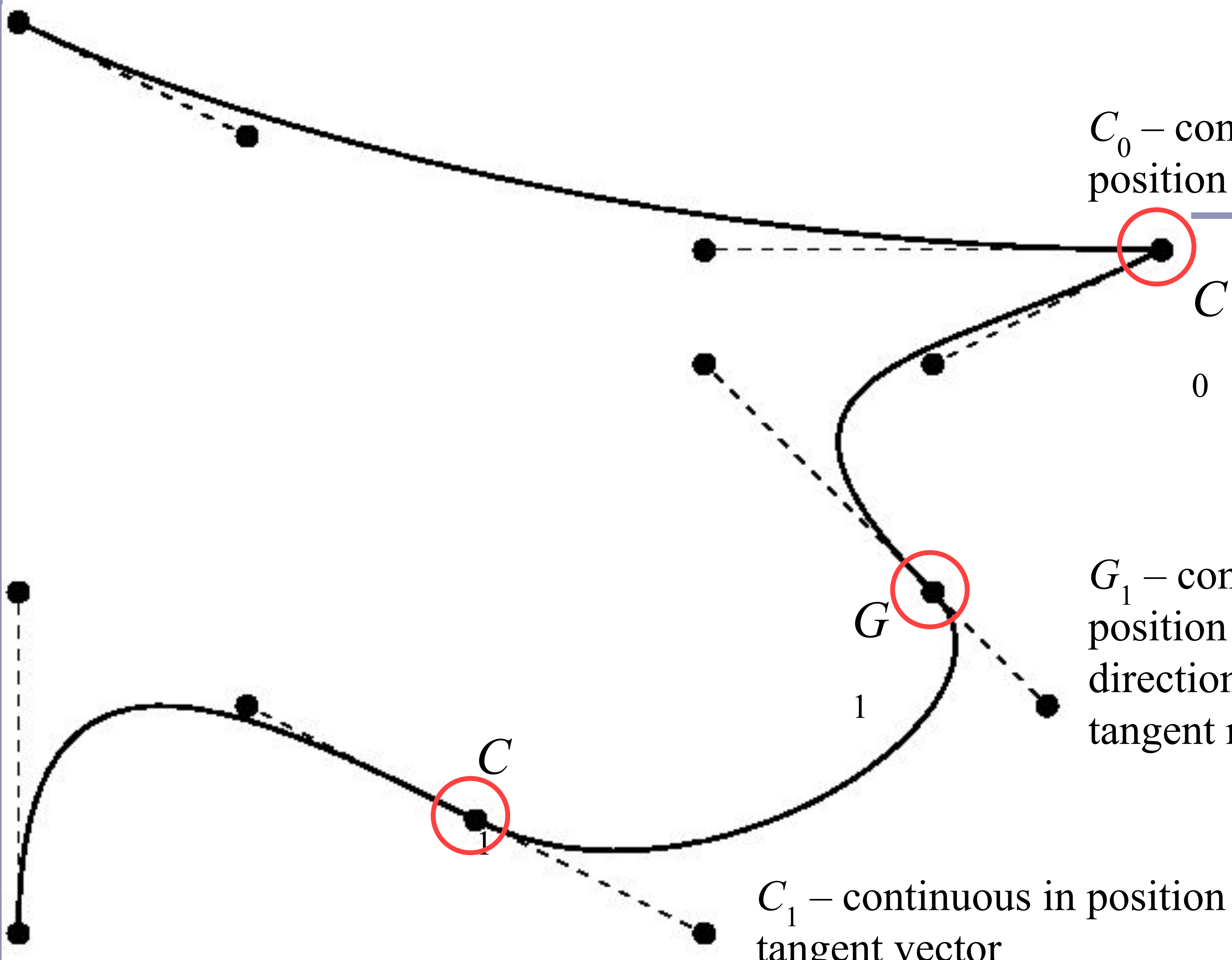


- each curve is smooth within itself
- joins at endpoints can be:
  - C_1 – continuous in both position and tangent vector
    - smooth join in a mathematical sense
  - G_1 – continuous in position, tangent vector in same direction
    - smooth join in a geometric sense
  - C_0 – continuous in position only
    - “corner”
  - discontinuous in position

C_n (mathematical continuity): continuous in all derivatives up to the n^{th} derivative

G_n (geometric continuity): each derivative up to the n^{th} has the same “direction” to its vector on either side of the join

$C_n \Rightarrow G_n$



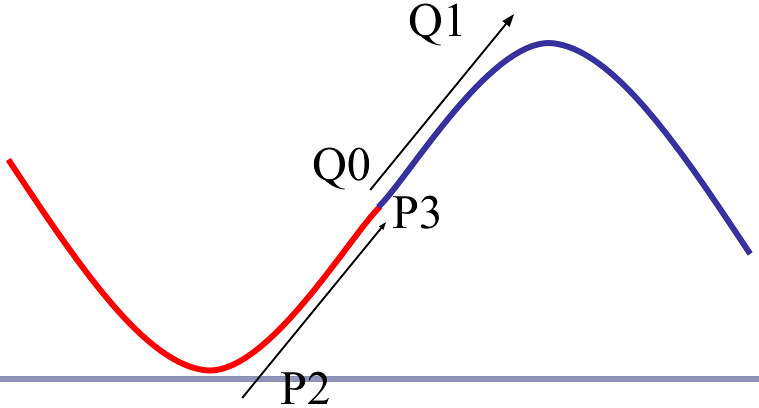
C_0 – continuous in position only

G_1 – continuous in position & tangent direction, but not tangent magnitude

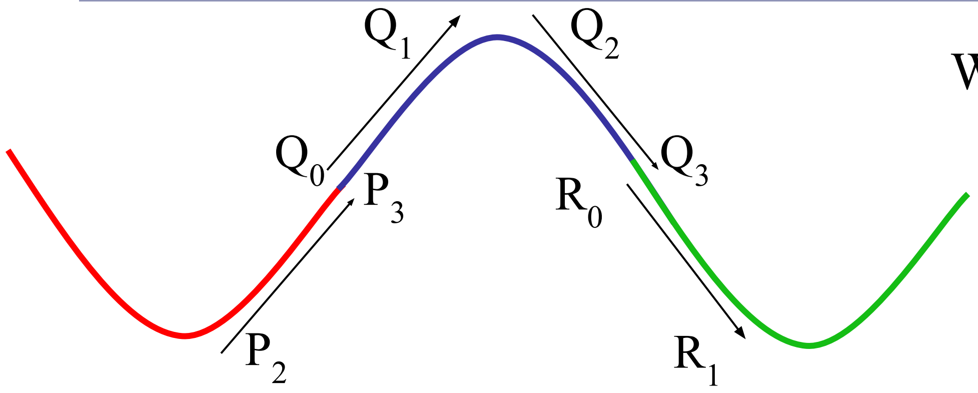
C_1 – continuous in position & tangent vector

Joining Bezier splines

- To join two Bezier splines with C0 continuity, set $P_3=Q_0$.
- To join two Bezier splines with C1 continuity, require C0 and make the tangent vectors equal: set $P_3=Q_0$ and $P_3-P_2=Q_1-Q_0$.



What if we want to chain Beziers together?



Consider a chain of splines with many control points…

$P = \{P_0, P_1, P_2, P_3\}$
$Q = \{Q_0, Q_1, Q_2, Q_3\}$
$R = \{R_0, R_1, R_2, R_3\}$

…with C1 continuity…

$P3=Q_0,\ P_2\text{-}P_3=Q_0\text{-}Q_1$
$Q3=R_0,\ Q_2\text{-}Q_3=R_0\text{-}R_1$

We can parameterize this chain over *t* by saying that instead of going from 0 to 1, *t* moves smoothly through the intervals [0,1,2,3]

The curve *C*(*t*) would be:

$$C(t) = P(t) \bullet ((0 \le t < 1)\ ?\ 1 : 0) + Q(t-1) \bullet ((1 \le t < 2)\ ?\ 1 : 0) + R(t-2) \bullet ((2 \le t < 3)\ ?\ 1 : 0)$$

[0,1,2,3] is a type of *knot vector*. 0, 1, 2, and 3 are the *knots*.

Tensor product

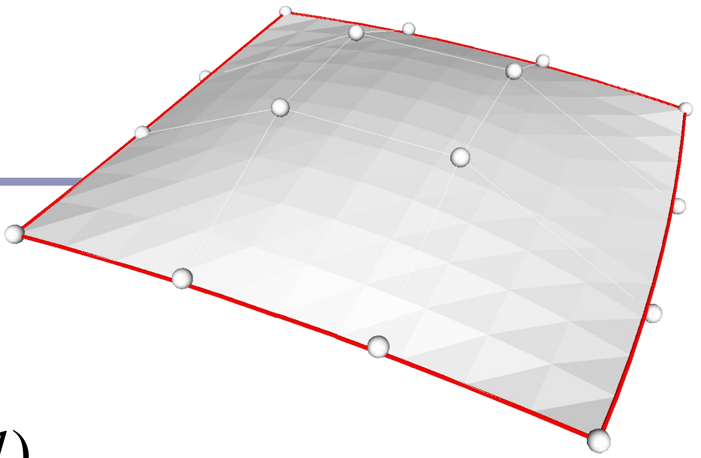
- The *tensor product* of two vectors is a matrix.

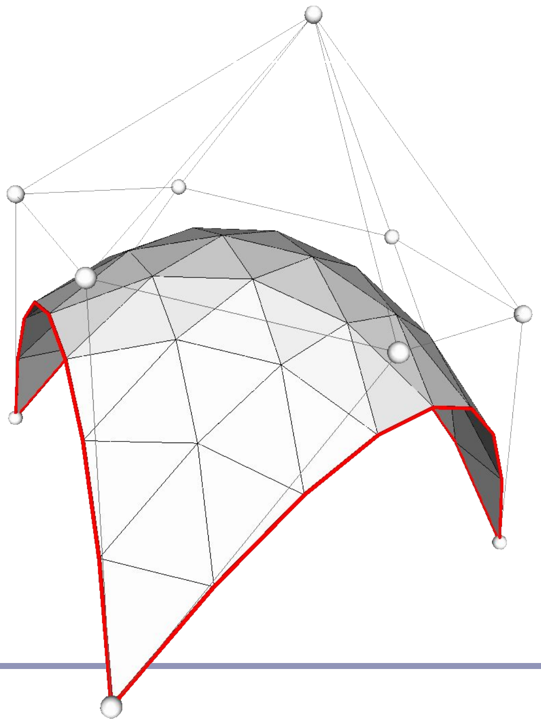
$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \otimes \begin{bmatrix} d \\ e \\ f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

- Can take the tensor of two polynomials.
  - Each coefficient represents a piece of each of the two original expressions, so the cumulative polynomial represents both original polynomials completely.

Bezier patches

- If curve A has *n* control points and curve B has *m* control points then A⊗B is an (*n*)X(*m*) matrix of polynomials of degree *max*(*n-1, m-1*).
  - ⊗ = *tensor product*
- Multiply this matrix against an (*n*)X(*m*) matrix of control points and sum them all up and you've got a bivariate expression for a rectangular surface patch, in 3D
- This approach generalizes to triangles and arbitrary *n*-gons.





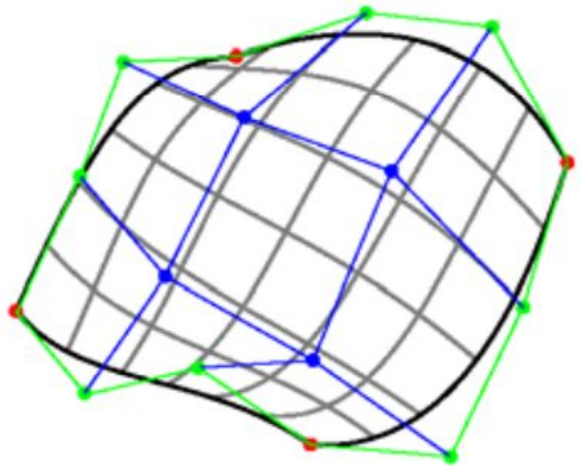
Bezier patch definition

The Bezier patch defined by sixteen control points,

$$\begin{matrix} P_{0,0} & \dots & P_{0,3} \\ \vdots & & \vdots \\ P_{3,0} & \dots & P_{3,3} \end{matrix}$$

is:

$$P(s,t) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(s) b_j(t) P_{i,j}$$



Compare this to the 2D version:

$$P(t) = \sum_{i=0}^{3} b_i(t) P_i$$

Continuity between Bezier patches

Ensuring continuity in 3D:

- C0 – continuous in position
  - the four edge control points must match
- C1 – continuous in position and tangent vector
  - the four edge control points must match
  - the two control points on either side of each of the four edge control points must be co-linear with both the edge point, and each other, *and* be equidistant from the edge point
- G1 – continuous in position and tangent direction the four edge control points must match the relevant control points must be co-linear

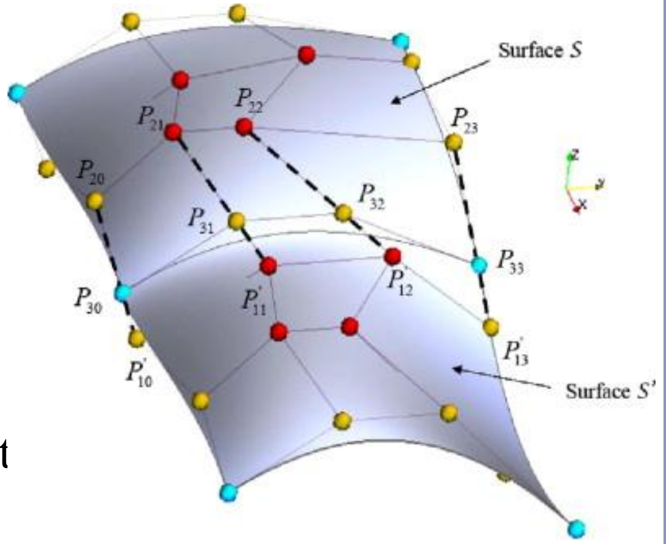


Image credit: Olivier Czarny, Guido Huysmans. *Bézier surfaces and finite elements for MHD simulations*. Journal of Computational Physics Volume 227, Issue 16, 10 August 2008

References

- Les Piegl and Wayne Tiller, *The NURBS Book*, Springer (1997)
- Alan Watt, *3D Computer Graphics*, Addison Wesley (2000)
- G. Farin, J. Hoschek, M.-S. Kim, *Handbook of Computer Aided Geometric Design*, North-Holland (2002)