NURBS (Redbook ch12)

31 March 2009
CMPT370
Dr. Sean Ho
Trinity Western University

IBiblio e-notes

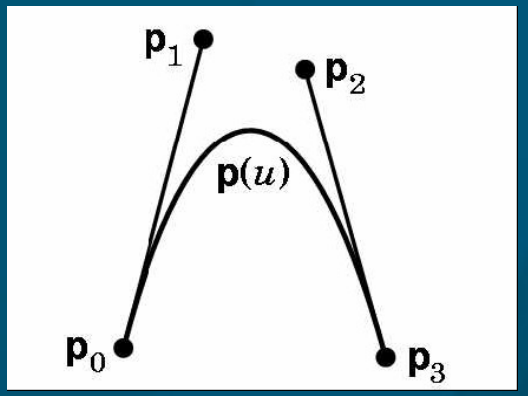
Cambridge notes

TRINITY WESTERN UNIVERSITY

Review last time: cubic curves

- Polynomial curves and surfaces
- Cubic polynomial curves:
  - Interpolating
  - Hermite
  - Bezier
- Solving for the geometry matrix to get coefficients
- Blending functions
- Types of continuity

Bezier curves

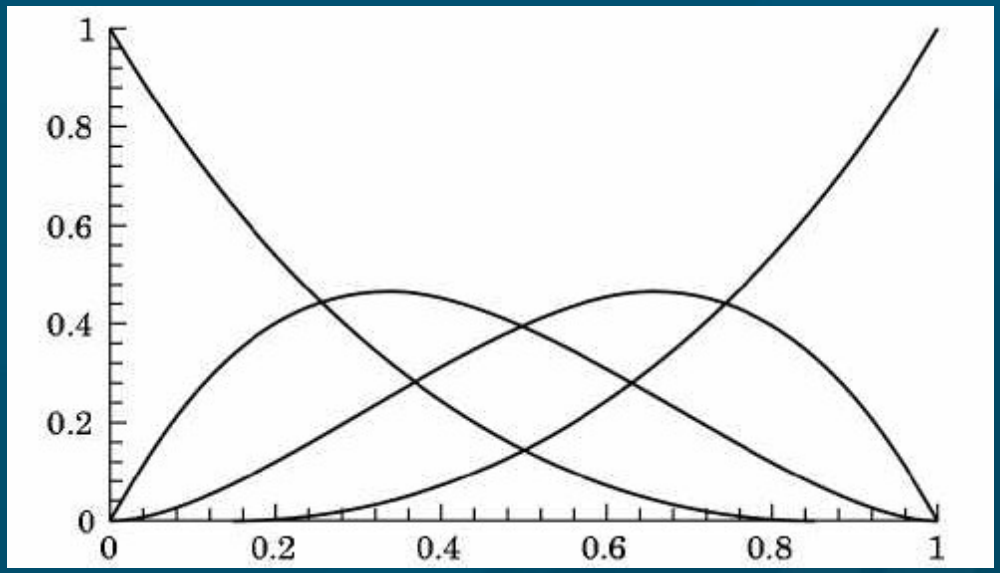


- Widely used, provided in OpenGL
- Use control points to indicate tangent vectors
  - Does not interpolate middle control points!
  - $p'(0) = 3(p_1 - p_0)$, $p'(1) = 3(p_3 - p_2)$
- p_0, p_3 specify start+end position
- start+end velocity derived from control points
- Use Hermite form
- C^0 but not C^1

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

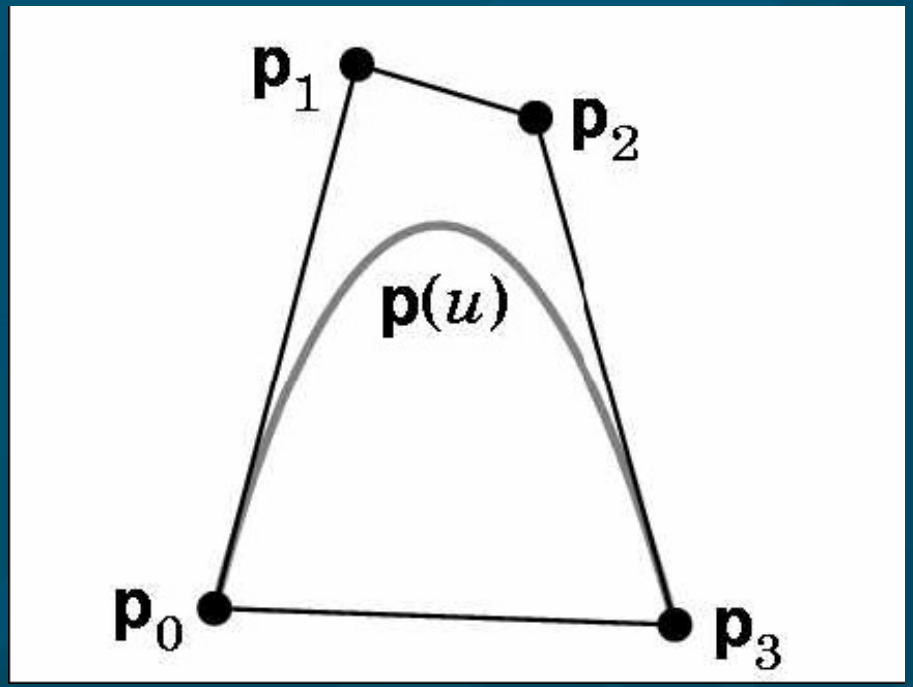
Bezier blending functions

- Blending functions are smooth polynomials
  - $b_0(u) = (1-u)^3$
  - $b_1(u) = 3u(1-u)^2$
  - $b_2(u) = 3u^2(1-u)$
  - $b_3(u) = 3u^3$



Convex hull property

- Why the factor of 3 in the definition of Bezier curves?
  - $p'(0) = 3(p_1 - p_0)$
  - $p'(1) = 3(p_3 - p_2)$
- Ensures that the curve is contained within the convex hull of the four control points



Bezier evaluators in OpenGL

- Specify array (1D or 2D) of control points:
  - GLfloat ctrlpoints[4][3] = { {-4.0, -4.0, 0.0}, ...
- Create a Bezier evaluator: *(type=GL_MAP1_VERTEX_3)*
  - glMap1f(*type*, u_{min}, u_{max}, *stride*, *order*, *points*);
- Enable the evaluator:
  - glEnable(*type*);
- Evaluate the Bezier at a particular u/v:
  - glEvalCoord1f((GLfloat) u);
    - Use this instead of glVertex(), e.g., within glBegin(GL_LINE_STRIP)

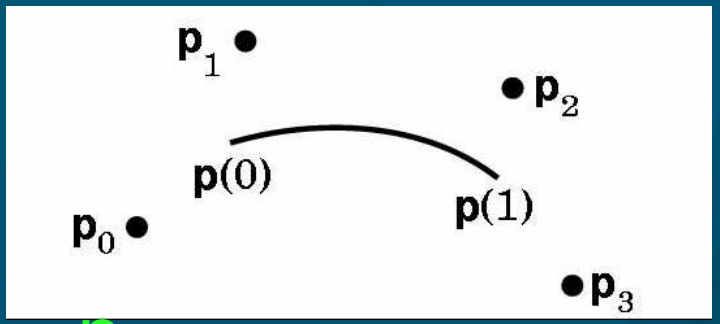
How OpenGL computes Beziers

- de Casteljau's algorithm: opengl/bezierdemo/
- 4 control points:
  - Plot a point u of the way from p_0 to p_1
  - Similarly between (p_1, p_2), and (p_2, p_3)
  - Get 3 points (q_0, q_1, q_2)
- Plot points u of the way between (q_0, q_1), (q_1, q_2)
  - Get 2 points (r_0, r_1)
- Plot a point u of the way between (r_0, r_1)
  - This is our point on the Bezier curve

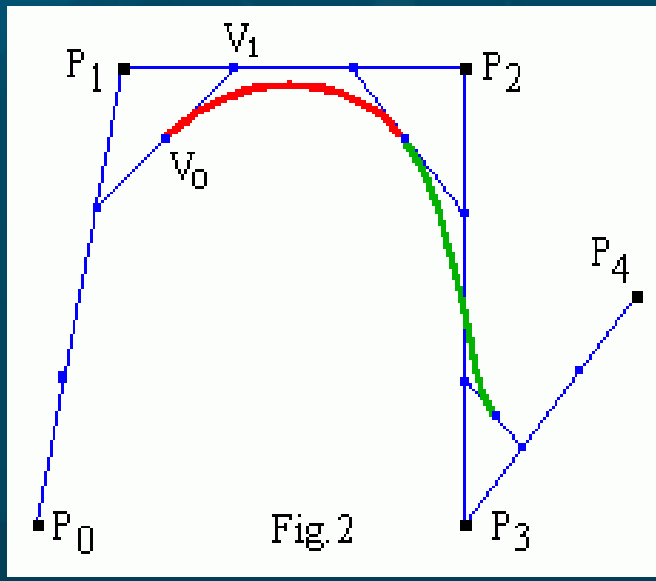
Splines

- Draftsman's tool for drawing smooth curves:
  - Flexible wood/plastic strip
  - Bent to pass through knots (control points)
- A spline is any sort of smooth curve that has a series of control points
  - Interpolating splines
    - Interpolating cubic spline
    - Interpolating Catmull-Rom spline
  - Cubic Bezier is a spline
  - B-splines: basis splines

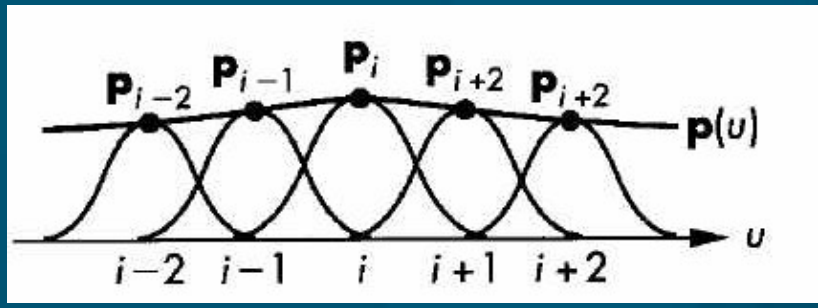
Cubic B-splines



- n+3 deBoor control points $p_0, \ldots, p_{n+2}$.
- Make n Bezier curve segments
  - Want C^2 at the joins; sacrifice interpolation
  - Derive Bezier control points (v_0, v_1, v_2, v_3) from deBoor points (p_0, p_1, p_2, p_3):
  - v_1 is (1/3)-way btw p_1 and p_2
  - v_0 is halfway between v_1 and $(1/3)p_0 + (2/3)p_1$
- Cubic B-spline: *order* == 4



Knot spacing



- Region of influence for each deBoor control point p_i is 4 Bezier segments
- Knots $(u_0, ..., u_{n+4})$ specify where the joins are in parameter space: e.g., {.0, .25, .50, .75, 1.0}
  - # knots == n+5
- Open-spacing: duplicate end knots to get interpolation:
  - {.0, .0, .0, .25, .50, .75, 1.0, 1.0, 1.0} (4 Beziers)
  - Some systems require extra duplicate @start/end
- Uniform B-spline: uniform spacing of knots

NURBS

- Spline:
  - Smoothish curve defined by control points
- B-spline:
  - Joined Bezier curves with C^2 continuity
- Non-uniform B-spline:
  - Non-uniform spacing of knots (e.g., can use multiplicity to get interpolation of endpoints)
- Rational B-spline:
  - Add weights to each control point
  - Leverages perspective division hardware

Properties of NURBS

- More computationally expensive than Bezier curves/patches
- C^2 continuity makes shading look much better
- Local control: moving a control point only affects 4 Bezier segments
- Convex hull property: each point on the spline is within the convex hull of the four control points it's affected by
- Affine-invariant (including perspective):
  - Transforming control points == transforming the curve

Using NURBS with GLU

- See Redbook example: surface.c
- Create pointer to new NURBS object:
  - #include <GL/glu.h>
  - GLUnurbsObj *n = gluNewNurbsRenderer();
- Enable auto-generation of normals:
  - glEnable(GL_AUTO_NORMAL);
- Set rendering options and register for errors:
  - gluNurbsProperty(n, GLU_SAMPLING_TOLERANCE, 25.0);
  - gluNurbsProperty(n, GLU_DISPLAY_MODE, GLU_FILL);
  - gluNurbsCallback(n, GLU_ERROR, nurbsError);

GLU NURBS, cont.

- In the display() callback: start a curve/surface:
  - gluBeginSurface(n);
- Specify the NURBS:
  - gluNurbsSurface(n, ...);
    - # knots and list of knots in each dim u,v
    - array of control pts (give stride in u,v)
    - polynomial order (cubic=4) in u,v
  - surface type: GL_MAP2_VERTEX_3 for pts, GL_MAP2_TEXTURE_COORD_2 for texcoords
- Finish with gluEndSurface(n);

Trimming NURBS surfaces

- You can trim a NURBS surface by cutting out part of the parameter space ((0,1) in u,v)
- Specify trimming curves before gluEndSurface:
  - gluBeginTrim(n) / gluEndTrim(n);
- Piecewise linear trim: gluPwlCurve()
- Use a NURBS to trim: gluNurbsCurve()
- Orientation matters:
  include left side, exclude right side
- Can cut out islands
- See Redbook trim.c