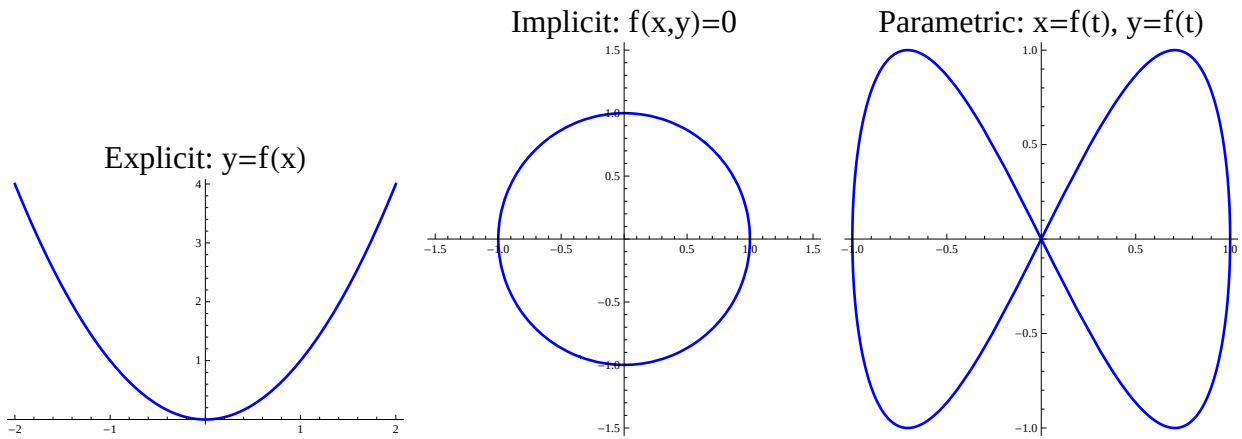
Computer Graphics: Bezier curves and surfaces

Table of contents

Curve and surface representation

- Explicit representation
- Implicit representation
- Parametric representation



Parametric representation

- Widely used in the design industry
- Can be used to fit curve or surface to sampled data (interpolation or approximation)
- Can be used for curve or surface design
- Importance of the choice of a suitable basis

Curve design problem

- Given control points $\mathbf{p}_1, \mathbf{p}_2, ... \in \mathbb{R}^n$
- Given a selected basis $b_1(t), b_2(t), ...$
- Define a parametric curve as a function $\mathbf{p} : \mathbb{R} \to \mathbb{R}^n$ such that $\mathbf{p}(t) = b_1(t)\mathbf{p}_1 + b_2(t)\mathbf{p}_2 + ...$
- The basis is selected such that the control points have intuitive meaning
- The control points are edited by the user to modify the curve

Properties for a spline basis

- Smooth basis functions (smoothness of the curve)
- Functions with compact support (local control)
- Affine invariance (affine transformation of the control points corresponds to an affine transformation of the curve)
- Convex hull property: the curve is contained in the convex hull of its control points (prevent weird oscillations)

Piecewise polynomials

Choice of degree for the basis

- piecewise constant: not smooth enough
- piecewise linear: not smooth enough
- piecewise quadratic: constant second derivative, too inflexible (can only represent planar curves in 3D)
- piecewise cubic: degree of choice in computer graphics and applications
- higher degrees: difficult to control

Cubic splines

- Can get C^2 continuity without fixing the second derivative throughout the curve
- A piecewise cubic curve has the least square acceleration among all C^2 interpolating curve (smoothest curve)
- C^2 continuity is perceptually important

Spline with local control

- For algorithmic curve control (fitting, optimization) it is not a problem if the shape of the entire curve depends on all control points
- But our goal is interactive geometric modeling
- We prefer local control i.e. editing one control point has only a local effect

Examples of spline bases

- Hermite (specify position and derivatives at two consecutive control points; no convex hull property)
- Bezier (this lesson)
- B-Splines
- NURBS

History

- Independently developed by de Casteljau at Citroen and Bezier at Renault for automative design applications (in the late 50s early 60s)
- Standard for 2D curve editing (design industry, vector fonts, ...)
- Standard in 3D curve and surface modeling (design industry, automobile and aeronautic industry, ...)

Applications

- Animation: interpolation of parameters
- Vector font design (e.g. TEX/Metafont)
- Hair/fur modeling
- Surface modeling in mechanical engineering, automobile and aeronautic industry

Applications: Hair/fur modeling

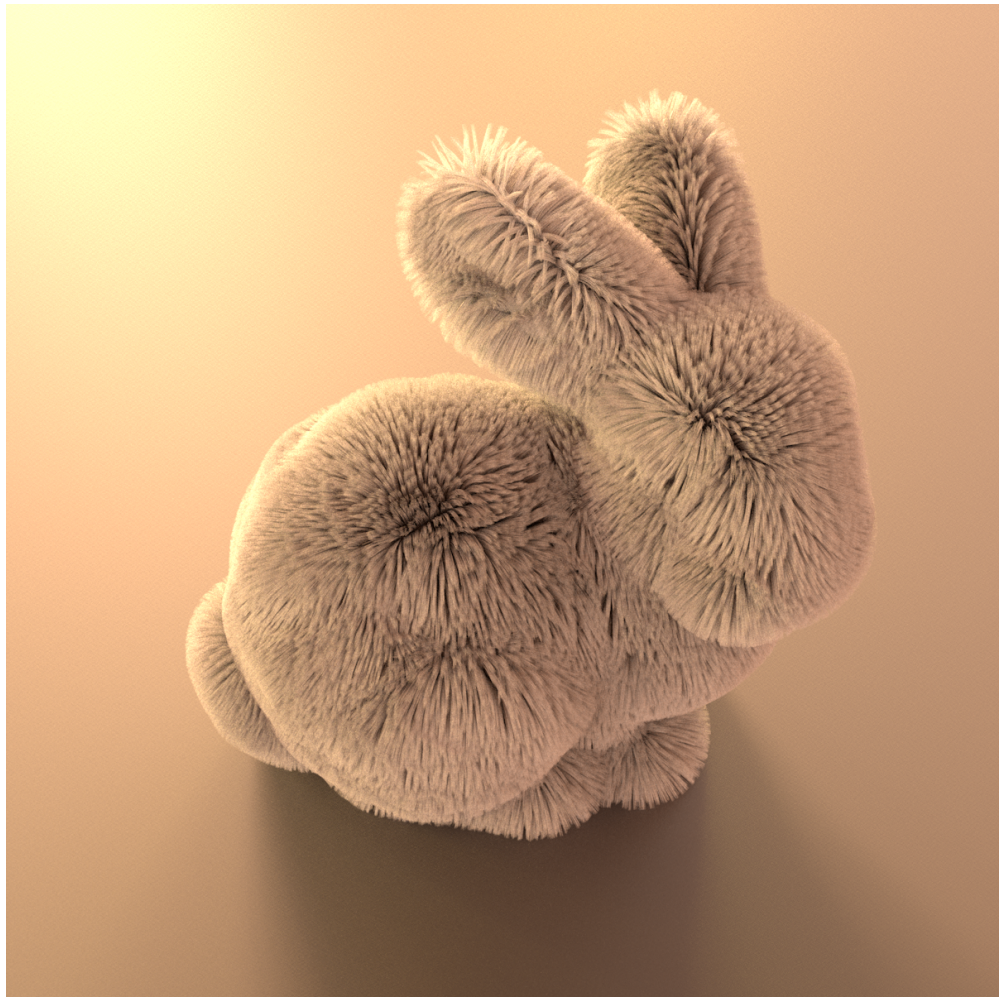


Figure: The fur of the bunny are made of one million Bézier curves. Rendered with PBRT. Courtesy of M. Pharr, W. Jacob and G. Humphreys.

Linear algebra approach: Bernstein basis

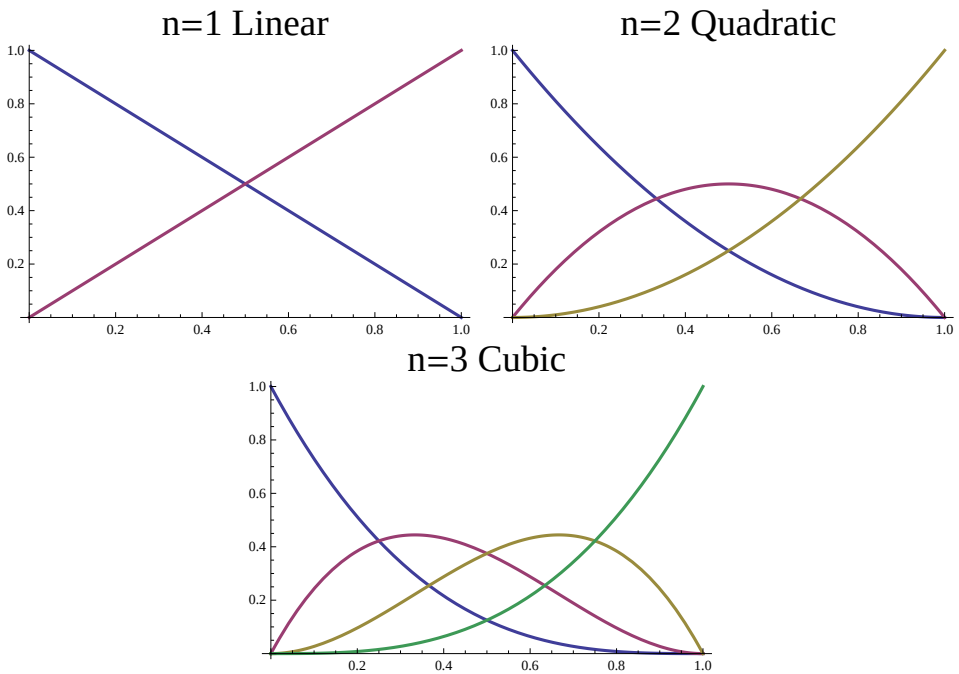
- A point $\mathbf{p} \in \mathbb{R}^2$ (or $\mathbb{R}^3$ for three-dimensional curves) on a Bezier curve of degree n is given by $\mathbf{p}(t) = \sum_{i=0}^{i=n} B_i^{(n)}(t)\mathbf{p}_i$ where $B_i^{(n)}$ are called the Bernstein basis functions
- $B_i^{(n)}(t)$ is the ith Bernstein basis function of degree n defined by: $B_i^{(n)}(t) = \binom{n}{i} t^i (1-t)^{(n-i)}$ where $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ is the binomial coefficient

Bernstein basis properties

- The basis functions form a partition of unity: $\sum_{i=0}^{n} B_i^{(n)}(t) = 1$
- They are positive: for $t \in [0,1]$, $B_i^{(n)}(t) \geq 0$
- Recursive computation: $B_i^{(n)}(t) = (1-t)B_i^{(n-1)}(t) + tB_{i-1}^{(n-1)}(t)$ with $B_0^{(0)}(t) = 1$ and $B_i^{(n)}(t) = 0$ for $i \notin \{0..n\}$
- Symmetry: $B_i^{(n)}(t) = B_{n-i}^{(n)}(1-t)$

Examples of Bernstein basis

Bernstein basis functions for $n = 1, 2, 3$

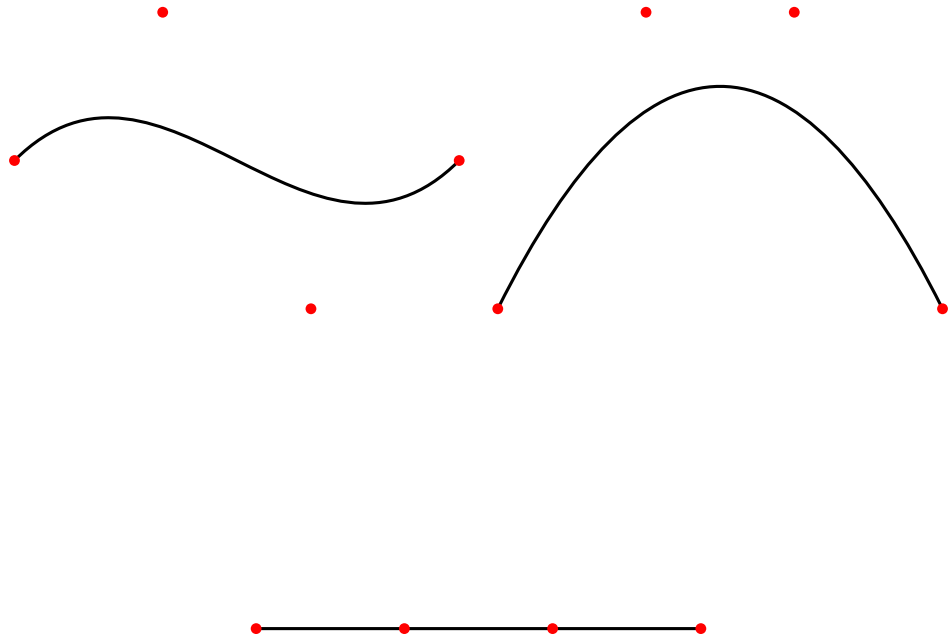


Bezier curve properties

- Bezier curve of degree n: for $t \in [0,1]$, $\mathbf{p}(t) = \sum_{i=0}^{i=n} B_i^{(n)}(t)\mathbf{p}_i$
- Affine invariant
- Contained in the convex hull of the control points
- Interpolate $\mathbf{p}_0$ and $\mathbf{p}_n$
- Tangent at $\mathbf{p}_0$ colinear to vector $\mathbf{p}_1 - \mathbf{p}_0$, and tangent at $\mathbf{p}_n$ colinear to vector $\mathbf{p}_n - \mathbf{p}_{n-1}$
- Linear precision: if the control points are colinear, the Bezier curve is linear

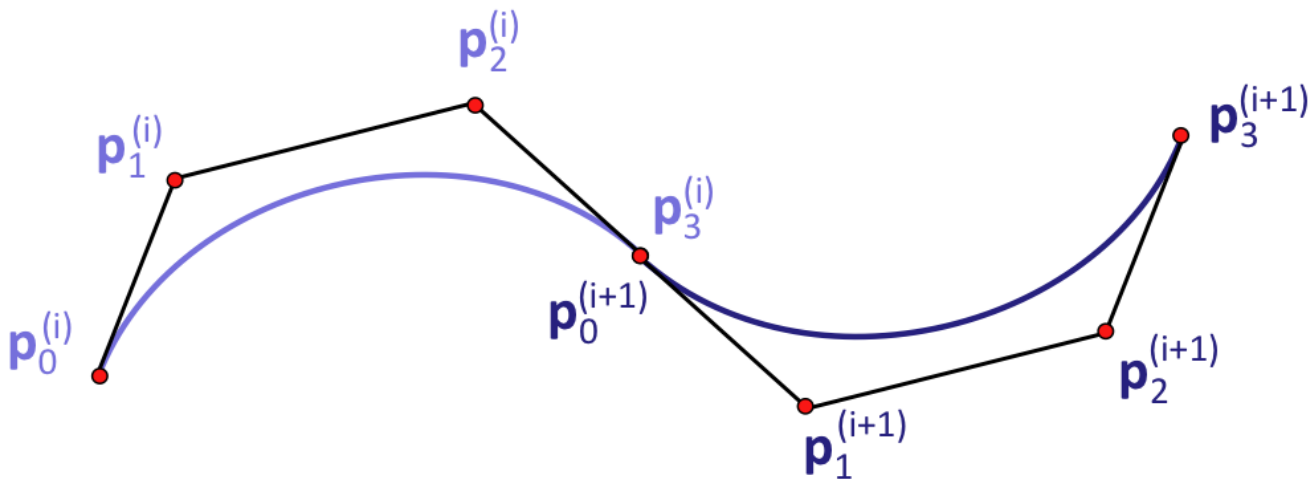
Examples of cubic Bezier curves

Example of cubic Bezier curves for various configurations of the control points:



From curve segments to splines

- Bezier splines are obtained by concatenating several Bezier curves
- Continuity at the junction of two Bezier curves is controlled by the control points

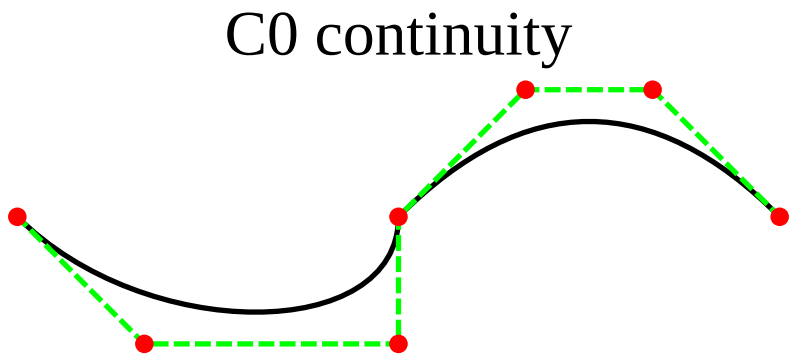


Constraints for continuous concatenation

- Continuity at the junction is controlled by the value of the functions and derivatives at the parameters $t = 0$ and $t = 1$
- Function and derivative values at $t = 0$ and $t = 1$ of Bezier curve:
  - Function value: $\mathbf{p}(0) = \mathbf{p}_0$, $\mathbf{p}(1) = \mathbf{p}_n$
  - First derivative: $\mathbf{p}'(0) = n(\mathbf{p}_1 - \mathbf{p}_0)$, $\mathbf{p}'(1) = n(\mathbf{p}_n - \mathbf{p}_{n-1})$
  - Second derivative: $\mathbf{p}''(0) = n(n-1)(\mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2)$, $\mathbf{p}''(1) = n(n-1)(\mathbf{p}_n - 2\mathbf{p}_{n-1} + \mathbf{p}_{n-2})$

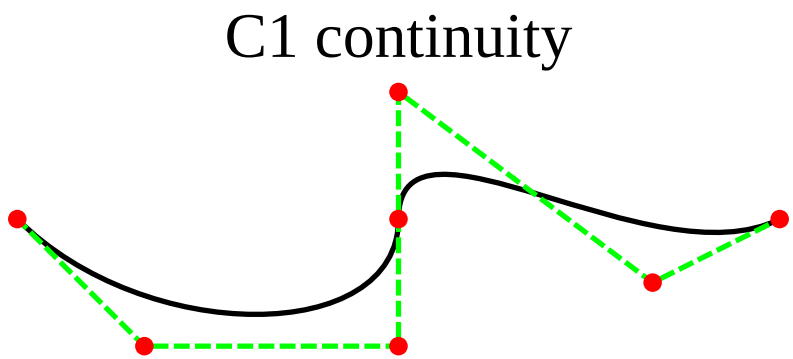
C^0 continuity

C^0 continuity is obtained when the last control point of the first curve coincides with the first control point of the next curve



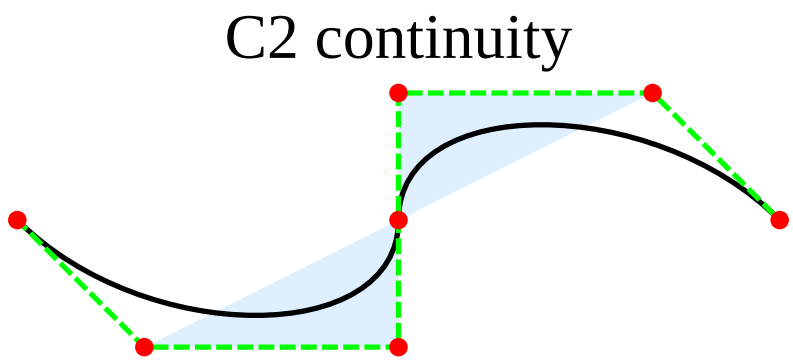
C^1 continuity

C^1 continuity happens when the derivative of the ith curve segment at $t = 1$ coincides with the derivative of the $(i + 1)$th curve segment at $t = 0$, i.e. when the vector $(\mathbf{p}_n^{(i)} - \mathbf{p}_{n-1}^{(i)})$ and $(\mathbf{p}_1^{(i+1)} - \mathbf{p}_0^{(i+1)})$ are identical.



C^2 continuity

C^2 continuity happens when the second derivative of the ith curve segment at $t = 1$ and the $(i + 1)$th curve segment at $t = 0$ coincides, i.e when the triangles made by the points $\mathbf{p}_{n-2}^{(i)}$, $\mathbf{p}_{n-1}^{(i)}$ and $\mathbf{p}_n^{(i)}$ and $\mathbf{p}_0^{(i+1)}$, $\mathbf{p}_1^{(i+1)}$ and $\mathbf{p}_2^{(i+1)}$ are identical.



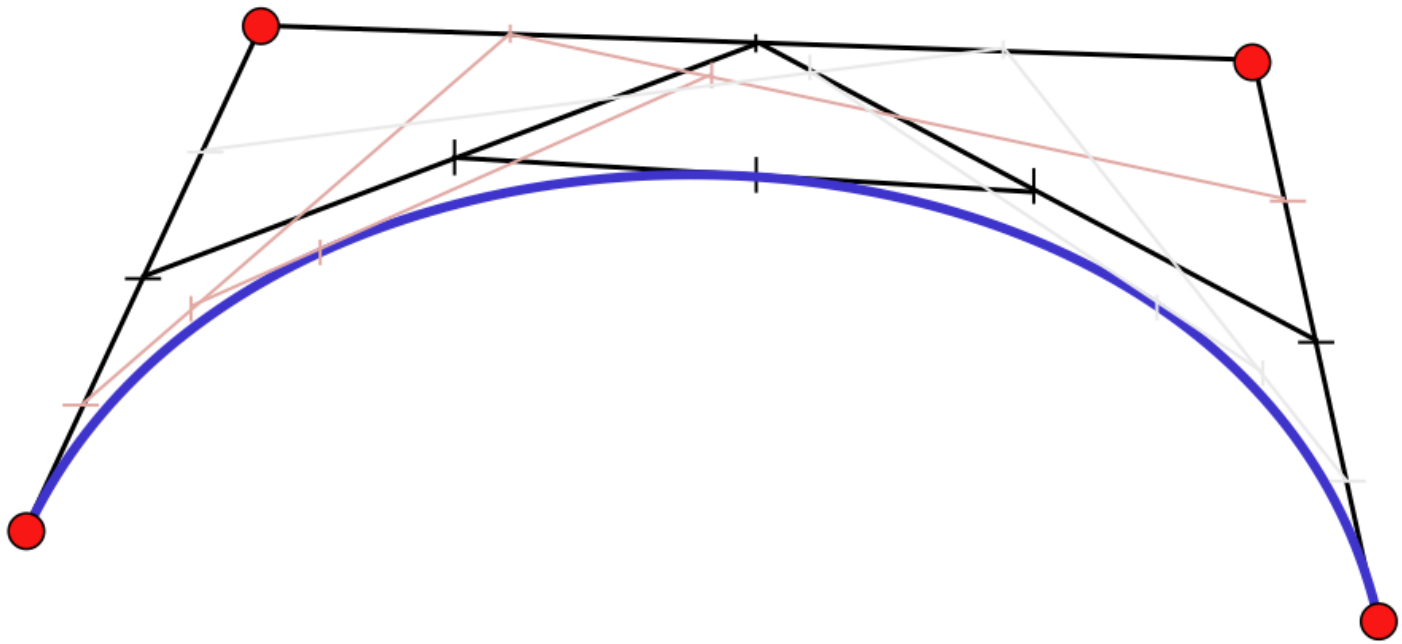
Geometric approach: The de Casteljau algorithm

- The idea behind the Bezier curves can be formulated geometrically by repeated linear interpolation between successive control points
- The degree depends on the number of cascaded interpolations and therefore on the number of control points ($n+1$ control points implies n cascaded interpolations, which implies a degree n)
- Other properties of the Bezier curves can also be interpreted geometrically
- The de Casteljau algorithm gives the same result as an evaluation of the representation in the Bernstein basis but is numerically more stable

The de Casteljau algorithm

In order to compute the $\mathbf{p}(t)$ for a given $t \in [0, 1]$:

- Bissect each adjacent segment of the control polygon with the ratio t and $(1 - t)$
- Connect by lines the adjacent points generated in the previous step
- Iterate the previous steps until there is only one point left. This point is $\mathbf{p}(t)$



The de Casteljau algorithm

```
// nc is the number of control points
// Initially p(0,0) .. p(nc-1,0) contains
// the control points
function EvalDeCasteljau(p, t)
 for j = 1 to nc - 1:
   for i = 0 to nc - j - 1:
    p(i,j) = (1-t) p(i, j-1) + t p(i+1, j-1)
   endfor
 endfor
 return p(0, nc-1)
```

Bezier curves in OpenGL

In order to draw Bezier curves and splines in OpenGL:

- Define control points
- Define an evaluator using glMap1()
- Enable the evaluator with glEnable()
- Generate the coordinates of vertices on the curve with glEvalCoord1()

Example: control points and evaluator

```
// Define control points
static GLfloat cpoints[4][3] = {
    {0.0, 2.0, 0.0}, {-4.0, -2.0, 0.0},
    {4.0, -4.0, 0.0}, {4.0, 4.0, 0.0} };

void init (void) {
  glClearColor (0.0, 0.0, 0.0, 1.0);
  // Define the evaluator
  glMap1f (GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4,
    &cpoints[0][0]);
  // Enable
  glEnable (GL_MAP1_VERTEX_3);
  glShadeModel (GL_FLAT);
}
```

One dimensional evaluator

```
glMap1{fd}(GLenum target, TYPE u1, TYPE u2,
  GLint stride, GLint order, const TYPE* points);
```

- target: specifies what the control point represents
- u1, u2: range for the parameter (t in the precedent equations)
- stride: specifies how the control points are stored
- order: number of control points (degree of the polynomial plus one)
- points: pointer to the control points

Example continued: draw all

```
void display (void) {
  glClear(GL_COLOR_BUFFER_BIT);

  // display the bezier curve
  display_curve();

  // display the control points
  display_control_points();

  // display the control polygon
  display_control_polygon();

  glFlush();
}
```

Example continued: draw the cubic Bezier curve

```
void display_curve(void) {
  int i;
  glColor3f (1.0, 1.0, 1.0);
  glBegin (GL_LINE_STRIP);
  for (i = 0; i <= 20; i++)
    glEvalCoord1f((GLfloat)i/20.0);
  glEnd();
}
```

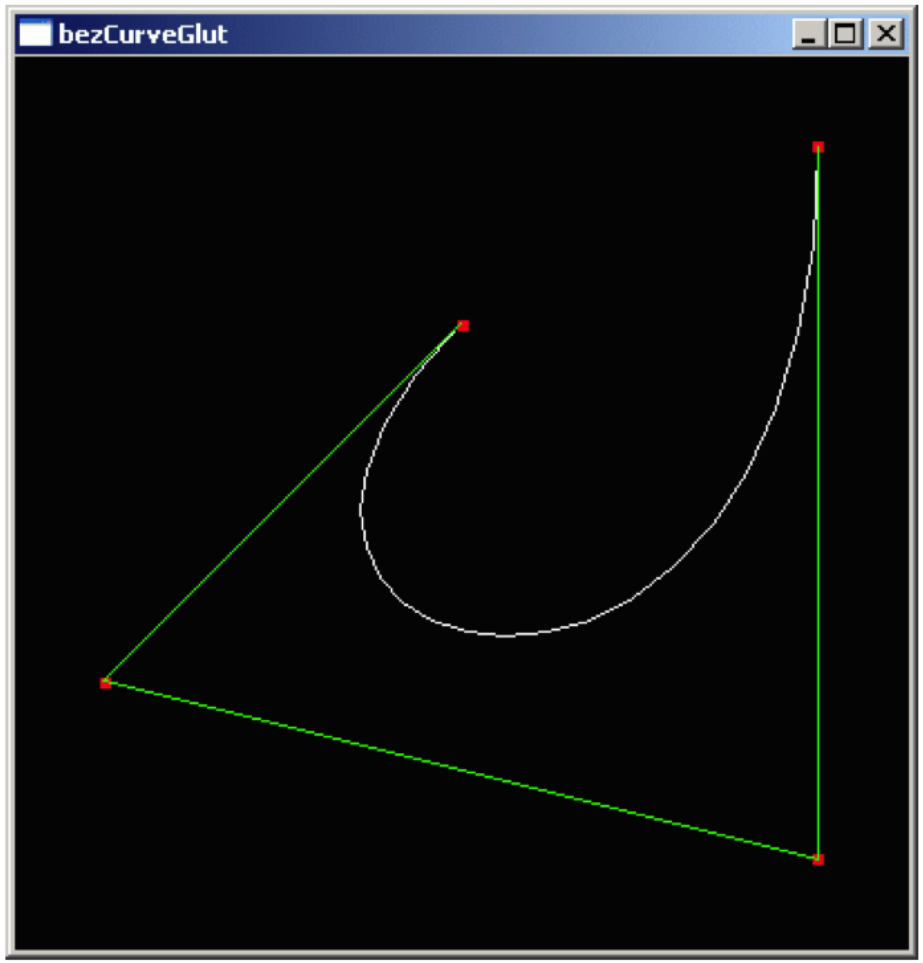
Example continued: draw the control points and polygon

```
void display_control_points(void) {
  int i;
  glPointSize (5.0);
  glColor3f (1.0, 0.0, 0.0);
  glBegin (GL_POINTS);
  for (i = 0; i < 4; i++)
    glVertex3fv (&cpoints[i][0]);
  glEnd ();
}

void display_control_polygon(void) {
  int i;
  glColor3f (0.0, 1.0, 0.0);
  glBegin (GL_LINE_STRIP);
  for (i = 0; i < 4; i++)
    glVertex3fv (&cpoints[i][0]);
  glEnd ();
}
```

Result

Result of the cubic Bezier curve drawn by the previous program:



Spline surfaces

- Parametric vector function of two parameters u, v
- Obtained by blending control points with piecewise bivariate polynomials
- A spline surface is made by assembling together multiple pieces (called spline patches)
- Generally obtained from curves by tensor product

Tensor product surface

- General idea:
  - Given a basis for a one dimensional function space: $B = \{b_1(t), .., b_n(t)\}$
  - Build a basis for a two dimensional function space by taking all possible products: $B = \{b_1(u)b_1(v), b_1(u)b_2(v), .., b_n(u)b_n(v)\}$
- A point p on the tensor product surface is given by $\mathbf{p}(u, v) = \sum_{i=1}^{i=n} \sum_{j=1}^{j=n} b_i(u)b_j(v)\mathbf{p}_{i,j}$
- Properties of the surface are induced from the properties of the curve basis

Bezier patches

Similar to Bezier curves, using the Bernstein basis. A point p on a Bezier patch specified by $(n+1)^2$ control points is defined by:
$\mathbf{p}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{n} B_i^{(n)}(u) B_j^{(n)}(v) \mathbf{p}_{i,j}$
Properties of Bezier patches are obtained from the properties of the Bezier curves:

- Affine invariance
- Convex hull property
- First derivatives at the boundary points are proportional to the differences of the control points

Bezier surfaces

- Bezier surfaces are obtained by assembling together Bezier patches (similar to the curves)
- C^0 continuity is obtained by making the boundary control points match
- C^1 continuity is obtained by making the difference vectors match at the boundary

Bezier patches in OpenGL

There are two methods to draw Bezier patches in OpenGL. The first method is a simple modification of the method used for drawing Bezier curves:

- Define control points
- Define an evaluator using glMap2()
- Enable the evaluator with glEnable()
- Generate the coordinates of vertices on the surface with glEvalCoord2()

Example: Specify control points

```
// Define control points
static GLfloat cpoints[4][4][3] = {
  { {-3.0, -3.0, -1.0}, {-1.0, -3.0, 0.0},
    { 1.0, -3.0,  0.0}, {3.0, -3.0, -2.0} },

  { {-3.0, -1.0,  0.0}, {-1.0, -1.0, 2.0},
    { 1.0, -1.0, 2.0}, {3.0, -1.0, 0.0} },

  { {-3.0, 1.0, 0.0}, {-1.0, 1.0, 2.0},
    {1.0, 1.0, 2.0}, {3.0, 1.0, 0.0} },

  { {-3.0, 3.0, -1.0}, {-1.0, 3.0, 0.0},
    { 1.0, 3.0, 0.0}, {3.0, 3.0, -1.0} },
};
```

Example: control points and evaluator

```
void init (void) {
  glClearColor (0.0, 0.0, 0.0, 1.0);
  // Define the evaluator
  glMap2f (GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
    0.0, 1.0, 12, 4, &cpoints[0][0][0]);
  // Enable
  glEnable (GL_MAP2_VERTEX_3);
}
```

Two dimensional evaluator

```
glMap2{fd}(GLenum target,
    TYPE u1, TYPE u2, GLint ustride, GLint uorder,
    TYPE v1, TYPE v2, GLint vstride, GLint vorder,
    const TYPE* points);
```

- target: specifies what the control point represents
- u1, u2: range for the parameter (u in the precedent equations for Bezier patches)
- ustride: specifies how the control points (in the u direction) are stored
- uorder: number of control points (in the u direction)
- v1, v2, vstride, vorder: same as above but for v
- points: pointer to the control points

Example continued: draw all

```
void display (void) {
  glClear(GL_COLOR_BUFFER_BIT);

  // display the bezier patch (in wireframe)
  display_patch_wireframe();

  // display the control points
  display_control_points();

  // display the control polygon
  display_control_polygon();

  glFlush();
}
```

Example continued: draw the cubic Bezier patch

```
void display_patch_wireframe(void) {
  int i, j;
  glColor3f (1.0, 1.0, 1.0);
  // draw 7 vertical/horizontal curves on the surface
  for (i=0; i<=6; ++i) {
    // draw the vertical curves
    glBegin(GL_LINE_STRIP);
    for(j=0; j<=20; ++j)
      glEvalCoord2f((GLfloat)i/6.0f, (GLfloat)j/20.0f)
    glEnd();

    // draw the horizontal curves
    glBegin(GL_LINE_STRIP);
    for(j=0; j<=20; ++j)
      glEvalCoord2f((GLfloat)j/20.0f, (GLfloat)i/6.0f);
    glEnd();
  }
```

Example continued: draw the control points

```
void display_control_points(void) {
  int i, j;
  glPointSize (5.0);
  glColor3f (1.0, 0.0, 0.0);
  glBegin (GL_POINTS);
  for (i=0; i<4; i++) {
    for (j=0; j<4; ++j) {
      glVertex3fv (&cpoints[i][j][0]);
    }
  }
  glEnd ();
}
```

Example continued: draw the control polygon

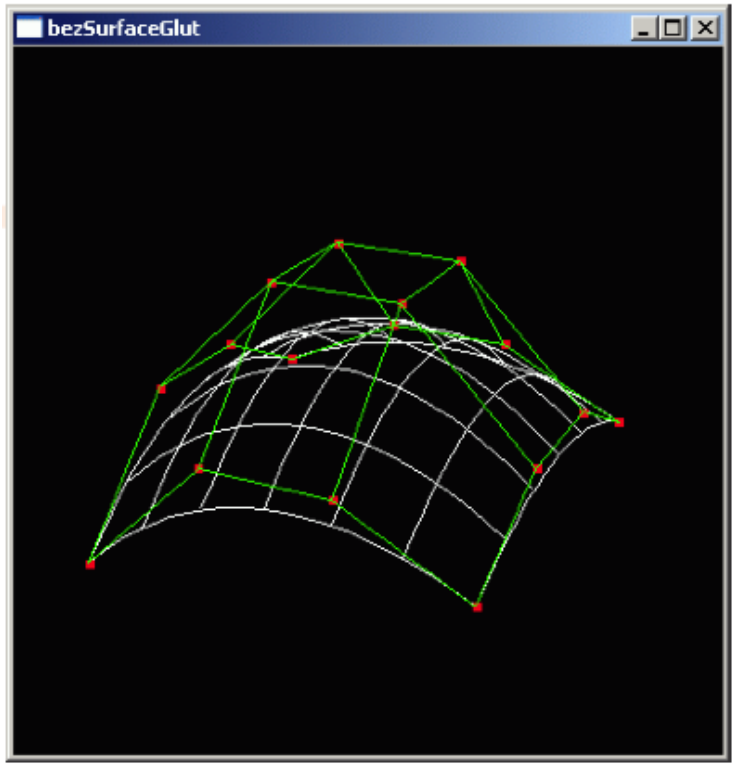
```
void display_control_polygon(void) {
  int i, j;
  glColor3f (0.0, 1.0, 0.0);

  for(i=0; i<4; ++i) {
    glBegin (GL_LINE_STRIP);
    for (j=0; j<4; j++)
      glVertex3fv (&cpoints[i][j][0]);
    glEnd ();

    glBegin (GL_LINE_STRIP);
    for (j=0; j<4; j++)
      glVertex3fv (&cpoints[j][i][0]);
    glEnd ();
  }
}
```

Result

Cubic Bezier patch drawn in wireframe mode with the input control points and the control polygon:



Bezier patches in OpenGL

By reordering the calls to glEvalCoord2f and using the mode GL_QUAD_STRIP we can draw a shaded surface instead of a wireframe surface. We can use a simpler method to reach the same result:

- Define control points
- Define an evaluator using glMap2()
- Define a grid using glMapGrid2()
- Enable the evaluator with glEnable()
- Generate the coordinates of the mesh vertices with glEvalMesh2()

Example: initialization

```
void init (void) {
  glClearColor(0.0, 0.0, 0.0, 1.0);
  // Define the evaluator
  glMap2f(GL_MAP2_VERTEX_3, 0.0, 1.0, 3, 4,
    0.0, 1.0, 12, 4, &cpoints[0][0][0]);
  // Enable it
  glEnable(GL_MAP2_VERTEX_3);
  // normals are needed for lighting
  // make OpenGL compute them
  glEnable(GL_AUTO_NORMAL);
  glMapGrid2f(10, 0.0f, 1.0f, 10, 0.0f, 1.0f);
  glShadeModel(GL_SMOOTH);
}
```

Example: draw the patch

```
void display (void) {
  glClear(GL_COLOR_BUFFER_BIT);

  // display the bezier patch
  glEvalMesh2(GL_FILL, 0, 10, 0, 10);

  glFlush();
}
```

Evaluation of evenly spaced coordinates

```
glMapGrid2{fd}(GLint nu, TYPE u1, TYPE u2,
    GLint nv, TYPE v1, TYPE v2)
```

where:

- nu: number of steps along u direction
- u1, u2: bounds on parameter u
- nv, v1, v2: similar but for the parameter v

```
glEvalMesh2(GLenum mode, GLint nu1, GLint nu2,
    GLint nv1, nv2)
```

where:

- mode: specifies whether to compute a two-dimensional mesh of points, lines or polygons. Possible values are: GL_POINT, GL_LINE or GL_FILL
- nu1, nu2: first and last integer value for grid domain variable u
- nv1. nv2: same for v

Result

Bezier patch with Gouraud shading obtained from the previous program: