Picking and Curves
Week 6

David Breen
Department of Computer Science
Drexel University

Based on material from Ed Angel, University of New Mexico

Objectives

• Picking
  – Select objects from the display
• Introduce types of curves and surfaces
  – Explicit
  – Implicit
  – Parametric
  – Strengths and weaknesses
• Discuss Modeling and Approximations
  – Conditions
  – Stability

Picking

• Identify a user-defined object on the display
• In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds
• Practical difficulties
  – Pipeline architecture is feed forward, hard to go from screen back to world
  – Complicated by screen being 2D, world is 3D
  – How close do we have to come to object to say we selected it?

Three Approaches

• Hit list
  – Most general approach but most difficult to implement
• Use back or some other buffer to store object ids as the objects are rendered
• Rectangular maps
  – Easy to implement for many applications
  – Divide screen into rectangular regions

Using another buffer and colors for picking

• For a small number of objects, we can assign a unique color (often in color index mode) to each object
• We then render the scene to a color buffer other than the front buffer so the results of the rendering are not visible
• We then get the mouse position and use glReadPixels() to read the color in the buffer we just wrote at the position of the mouse
• The returned color gives the id of the object

Using Regions of the Screen

• Many applications use a simple rectangular arrangement of the screen
  – Example: paint/CAD program
  - mouse
drawing area
menus

• Easier to look at mouse position and determine which area of screen it is in that using selection mode picking
Rendering Modes

- OpenGL can render in one of three modes selected by `glRenderMode(mode)`
  - GL_RENDER: normal rendering to the frame buffer (default)
  - GL_FEEDBACK: provides list of primitives rendered but no output to the frame buffer
  - GL_SELECTION: Each primitive in the view volume generates a hit record that is placed in a name stack which can be examined later

Hit Record

- Number of names
- Minimum Z
- Maximum Z
- Name 0
- Name 1

Using Selection Mode

- Initialize name buffer
- Enter selection mode (using mouse)
- Render scene with user-defined names (id#)
  - Every object in view volume generates a hit
  - Name stack processing always done
- Reenter normal render mode
  - This operation returns number of hits
- Examine contents of name buffer (hit records)
  - Hit records include number of ids, depth information and ids on stack at moment of rendering

Selection Mode Functions

- `glSelectBuffer()`: specifies name buffer
- `glInitNames()`: initializes name buffer
- `glPushName(id)`: push id on name buffer
- `glPopName()`: pop top of name buffer
- `glLoadName(id)`: replace top name on buffer

- id is set by application to identify objects
- Can’t be called inside `glBegin/glEnd`

Selection Mode and Picking

- In general, selection mode won’t work for picking because every primitive in the view volume will generate a hit
- Change the viewing parameters so that only those primitives near the cursor are in the altered view volume
  - Use `gluPickMatrix()`
    - Creates a projection matrix for picking that restricts drawing to a w x h area centered at (x, y) in the window coordinates within the viewport `vp`
Escaping Flatland

- Until now we have worked with flat entities such as lines and flat polygons
  - Fit well with graphics hardware
  - Mathematically simple
- But the world is not composed of flat entities
  - Need curves and curved surfaces
  - May only have need at the application level
  - Implementation can render them approximately with flat primitives

What Makes a Good Representation?

- There are many ways to represent curves and surfaces
- Want a representation that is
  - Stable
  - Smooth
  - Easy to evaluate
  - Must we interpolate or can we just come close to data?
  - Do we need derivatives?

Explicit Representation

- Most familiar form of curve in 2D
  \( y = f(x) \)
- Cannot represent all curves
  - Vertical lines
  - Circles
- Extension to 3D
  - \( y = f(x), z = g(x) \)
  - The form \( z = f(x,y) \) defines a surface
Implicit Representation

- Two dimensional curve(s) \( g(x,y)=0 \)
- Much more robust
  - All lines \( ax+by+c=0 \)
  - Circles \( x^2+y^2=r^2=0 \)
- Three dimensions \( g(x,y,z)=0 \) defines a surface
  - Intersect two surfaces to get a curve
- In general, we cannot solve for points that satisfy the equation

Parametric Curves

- Separate equation for each spatial variable
  \[
  x = x(u) \\
  y = y(u) \\
  z = z(u)
  \]
- For \( u_{\text{max}} \geq u \geq u_{\text{min}} \) we trace out a curve in two or three dimensions

Parametric Polynomial Curves

\[
\begin{align*}
  x(u) &= \sum_{i=0}^{N} c_i u^i \\
  y(u) &= \sum_{j=0}^{M} c_j u^j \\
  z(u) &= \sum_{k=0}^{K} c_k u^k
\end{align*}
\]
- If \( N=M=K \), we need to determine \( 3(N+1) \) coefficients
- Equivalently we need \( 3(N+1) \) independent conditions
- Noting that the curves for \( x, y \) and \( z \) are independent, we can define each independently in an identical manner
- We will use the form
  \[
  p(u) = \sum_{p} c_p u^p
  \]
  where \( p \) can be any of \( x, y, z \)
Why Polynomials

- Easy to evaluate
- Continuous and differentiable everywhere
  - Must worry about continuity at join points including continuity of derivatives

\[ p(u) \quad q(u) \quad \text{join point } p(1) = q(0) \]

but \( p'(1) \neq q'(0) \)

Cubic Parametric Polynomials

- \( N=M=K=3 \), gives balance between ease of evaluation and flexibility in design
  \[ p(u) = \sum_{i=0}^{3} c_i u^i \]

- Four coefficients to determine for each of \( x, y \) and \( z \)
- Seek four independent conditions for various values of \( u \) resulting in 4 equations in 4 unknowns for each of \( x, y \) and \( z \)
  - Conditions are a mixture of continuity requirements at the join points and conditions for fitting the data

Designing Parametric Cubic Curves

Objectives

- Introduce the types of curves
  - Interpolating
  - Hermite
  - Bezier
  - B-spline
- Analyze their performance

Matrix-Vector Form

\[ p(u) = \sum_{i=0}^{3} c_i u^i \]

define \( c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \quad u = \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix} \)

then \( p(u) = u^T c = c^T u \)

Interpolating Curve

Given four data (control) points \( p_0, p_1, p_2, p_3 \)
determine cubic \( p(u) \) which passes through them

Must find \( c_0, c_1, c_2, c_3 \)
Interpolating Multiple Segments

use \( p = [p_0 \ p_1 \ p_2 \ p_3] \)^T

Get continuity at join points but not continuity of derivatives

Interpolation Equations

apply the interpolating conditions at \( u=0, 1/3, 2/3, 1 \)

\[ p_0 = p(0) = c_0 \]
\[ p_1 = p(1/3) = c_0 + (1/3)c_1 + (1/3)c_2 + (1/3)c_3 \]
\[ p_2 = p(2/3) = c_0 + (2/3)c_1 + (2/3)c_2 + (2/3)c_3 \]
\[ p_3 = p(1) = c_0 + c_1 + c_2 + c_3 \]

or in matrix form with \( p = [p_0 \ p_1 \ p_2 \ p_3] \)^T

\[ p = Ac \]

Blending Functions

These functions are not monotonic
   - Hence the interpolation polynomial will wiggle

Blending Functions

Rewriting the equation for \( p(u) \)

\[ p(u) = u^T \mathbf{c} = u^T \mathbf{M} \mathbf{p} = b(u)^T \mathbf{p} \]

where \( b(u) = [b_0(u) \ b_1(u) \ b_2(u) \ b_3(u)]^T \) is
an array of blending polynomials such that
\[ p(u) = b_0(u)p_0 + b_1(u)p_1 + b_2(u)p_2 + b_3(u)p_3 \]

- \( b_0(u) = -4.5(u-1/3)(u-2/3)(u-1) \)
- \( b_1(u) = 13.5u (u-2/3)(u-1) \)
- \( b_2(u) = -13.5u (u-1/3)(u-1) \)
- \( b_3(u) = 4.5u (u-1/3)(u-2/3) \)

Other Types of Curves and Surfaces

- How can we get around the limitations of the interpolating form
  - Lack of smoothness
  - Discontinuous derivatives at join points
- We have four conditions (for cubics) that we can apply to each segment
  - Use them other than for interpolation
  - Need only come close to the data
Hermite Form

- Use two interpolating conditions and two derivative (tangent) conditions per segment.
- Ensures continuity and first derivative continuity between segments.

Equations

Interpolating conditions are the same at ends:

\[ p(0) = p_0 = c_0 \]
\[ p(1) = p_3 = c_0 + c_1 + c_2 + c_3 \]

Differentiating we find:

\[ p'(u) = c_1 + 2uc_2 + 3u^2c_3 \]

Evaluating at end points:

\[ p'(0) = p'_0 = c_1 \]
\[ p'(1) = p'_3 = c_1 + 2c_2 + 3c_3 \]

Matrix Form

\[
\begin{bmatrix}
 p_0 \\
 p_1 \\
 p_0' \\
 p_1'
\end{bmatrix} = \begin{bmatrix}
 1 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 \\
 0 & 1 & 0 & 1 \\
 0 & 1 & 2 & 3
\end{bmatrix} \begin{bmatrix}
 c_0 \\
 c_1 \\
 c_2 \\
 c_3
\end{bmatrix}
\]

Solving, we find \( c = M_{hp} q \) where \( M_{hp} \) is the Hermite matrix:

\[
M_{hp} = \begin{bmatrix}
 1 & 0 & 0 & 0 \\
 0 & 1 & 0 & 1 \\
 -3 & 3 & -2 & -1 \\
 2 & -2 & 1 & 1
\end{bmatrix}
\]

Blending Polynomials

\[
p(u) = b(u)^T q
\]

\[
b(u) = \begin{bmatrix}
 2u^3 - 3u^2 + 1 \\
 -2u^2 + 3u - u
\end{bmatrix}
\]

Although these functions are smooth, the Hermite form is not used directly in Computer Graphics and CAD because we usually have control points but not derivatives.

However, the Hermite form is the basis of the Bezier form.

Parametric and Geometric Continuity

- We can require the derivatives of \( x, y, \) and \( z \) to each be continuous at join points (parametric continuity).
- Alternately, we can only require that the tangents of the resulting curve be continuous (geometry continuity).
- The latter gives more flexibility as we have need satisfy only two conditions rather than three at each join point.

Example

- Here the \( p \) and \( q \) have the same tangents at the ends of the segment but different derivatives.
- Generate different Hermite curves.
- This technique is used in drawing applications.
Beziers and Spline Curves

Objectives

- Introduce the Bezier curves
- Derive the required matrices
- Introduce the B-spline and compare it to the standard cubic Bezier

Bezier's Idea

- In graphics and CAD, we do not usually have derivative data
- Bezier suggested using the same 4 data points as with the cubic interpolating curve to approximate the derivatives in the Hermite form

Approximating Derivatives

\[ p'(0) = \frac{p_1 - p_0}{1/3} \]
\[ p'(1) = \frac{p_3 - p_2}{1/3} \]

Equations

Interpolating conditions are the same
\[ p(0) = p_0 = c_0 \]
\[ p(1) = p_3 = c_0 + c_1 + c_2 + c_3 \]
Approximating derivative conditions
\[ p'(0) = 3(p_1 - p_0) = c_0 \]
\[ p'(1) = 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3 \]
Solve four linear equations for \( c \equiv M^{-1}p \)

Bezier Matrix

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
-3 & 3 & 0 & 0 \\
3 & -6 & 3 & 0 \\
-1 & 3 & -3 & 1 \\
\end{bmatrix}
\]

\[ p(u) = u^3M_b p = b(u)^3 p \]
Blending Functions

\[ b(u) = \begin{bmatrix} (1-u)^3 \\ 3u(1-u)^2 \\ 3u^2(1-u) \\ u^3 \end{bmatrix} \]

Note that all zeros are at 0 and 1 which forces the functions to be smooth over (0,1).

Bernstein Polynomials

- The blending functions are a special case of the Bernstein polynomials
  \[ b_{kn}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k} \]
- These polynomials give the blending polynomials for any degree Bezier form
  - All zeros at 0 and 1
  - For any degree they all sum to 1
  - They are all between 0 and 1 inside (0,1)

Convex Hull Property

- The properties of the Bernstein polynomials ensure that all Bezier curves lie in the convex hull of their control points
- Hence, even though we do not interpolate all the data, we cannot be too far away

Analysis

- Although the Bezier form is much better than the interpolating form, we have the derivatives are not continuous at join points
- Can we do better?
  - Go to higher order Bezier
    - More work
    - Derivative continuity still only approximate
    - Supported by OpenGL
  - Apply different conditions
    - Tricky without letting order increase

B-Splines

- Basis splines: use the data at \( p = [p_0, p_1, \ldots, p_n] \) to define curve only between \( p_{i-1} \) and \( p_i \)
- Allows us to apply more continuity conditions to each segment
  - For cubics, we can have continuity of function, first and second derivatives at join points
  - Cost is 3 times as much work for curves
    - Add one new point each time rather than three
  - For surfaces, we do 9 times as much work

Cubic B-spline

\[ p(u) = u^3 M_b p = b(u)^3 p \]

\[ M_b = \begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \]

\[ p_0 \rightarrow p_1 \rightarrow p_2 \]
Blending Functions

\[ b(u) = \begin{cases} 
\frac{1}{6} & \text{if } u \leq 0 \\
\frac{(1-u)^3}{4} - \frac{2}{3}u^3 + \frac{3}{4}u^2 & \text{if } 0 < u < 1 \\
\frac{2}{3}u^3 - \frac{3}{4}u^2 + \frac{3}{4} & \text{if } 1 \leq u \leq 2 \\
0 & \text{if } u > 2 
\end{cases} \]

convex hull property

Splines and Basis

- If we examine the cubic B-spline from the perspective of each control (data) point, each interior point contributes (through the blending functions) to four segments.
- We can rewrite \( p(u) \) in terms of the data points as

\[ p(u) = \sum B_i(u) p_i \]

defining the basis functions \( \{B_i(u)\} \).

Basis Functions

In terms of the blending polynomials

\[ B_i(u) = \begin{cases} 
0 & u < i - 2 \\
b_i(u + 2) & i - 2 \leq u < i - 1 \\
b_i(u + 1) & i - 1 \leq u < i \\
b_i(u) & i \leq u < i + 1 \\
b_i(u - 1) & i + 1 \leq u < i + 2 \\
0 & u \geq i + 2 
\end{cases} \]

Generalizing Splines

- We can extend to splines of any degree.
- Data and conditions to not have to be given at equally spaced values (the knots):
  - Nonuniform and uniform splines
  - Can have repeated knots
  - Can force spline to interpolate points
  - Cox-deBoor recursion gives method of evaluation

NURBS

- Nonuniform Rational B-Spline curves and surfaces add a fourth variable \( w \) to \( x,y,z \):
  - Can interpret it as weight to give more importance to some control data
  - Can also interpret as moving to homogeneous coordinate
- Requires a perspective division:
  - NURBS act correctly for perspective viewing
- Quadrics are a special case of NURBS

Every Curve is a Bezier Curve

- We can render a given polynomial using the recursive method if we find control points for its representation as a Bezier curve.
- Suppose that \( p(u) \) is given as an interpolating curve with control points \( q \):

\[ p(u) = \sum B_i(u) q_i \]

- There exist Bezier control points \( p \) such that

\[ p(u) = \sum B_i(u) p_i \]

- Equating and solving, we find \( p = M_i^T p_i \).
Matrices

Interpolating to Bezier: \[ M_I \cdot M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{5}{3} & 3 & -\frac{1}{2} & 1 \\ \frac{1}{3} & -\frac{3}{2} & 3 & -\frac{5}{3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

B-Spline to Bezier: \[ M_S \cdot M_B = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 4 & 2 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix} \]

Example

These three curves were all generated from the same original data using Bezier recursion by converting all control point data to Bezier control points.

Basic Procedure

- Enable an evaluator (glEnable)
  - For vertices, normals, and colors
- Define Bezier parameters (glMap1f)
- Evaluate Bezier curve (glEvalCoord1f, or glMapGrid1f and glEvalMesh1)

Example

```c
GLfloat data[4][3];
glEnable(GL_MAP1_VERTEX3);
glMap1f(GL_MAP1_VERTEX3, 0.0, 1.0, 3, 4, data);
glBegin(GL_LINE_STRIP);
for (i = 0; i <= 20; i++)
  glEvalCoord1f(i / 20.0);
glEnd();
```

1D Evaluator: `glMap1f()`

- `glMap1f(entity, TYPE u0, TYPE ul, GLint stride, GLint order, TYPE* data)`
  - `entity` describes type entity
  - `GL_MAP1_VERTEX_3`, `GL_MAP1_VERTEX_4`, `GL_MAP1_COLOR_4`, `GL_MAP1_NORMAL`
  - `u0` and `ul` is parameter range
  - `stride` is number of variables between data points
  - `order` of Bernstein polynomial (1 more than degree)
Polynomial Order Has a Limit

- Each OpenGL implementation has a maximum Bernstein polynomial order

  `glGetIntegeriv(GL_MAX_EVAL_ORDER, &max_order);`
  - Gets maximum polynomial order
  - e.g. 8, 10, 15, 30

Defining partitions:

`glMapGrid1f()`

- Defines \( n \) equally spaced partitions between parameters \( u_0 \) and \( u_1 \)
- Produces \( n+1 \) samples

Evaluate Bezier at samples:

`glEvalMesh1()`

- Renders in mode (\( GL_{\text{LINE}} \), \( GL_{\text{POINT}} \)) all enabled evaluators from the first to last values of \( u \) defined by `glMapGrid()`

Example

```c
GLfloat data[4][3];

glEnable(GL_MAP1_VERTEX3);
glMap1f(GL_MAP1_VERTEX3, u0, u1, 3, 4, data);
glMapGrid1f(20, u0, u1);
glEvalMesh1(GL_LINE, 0, 20);
```

Go to `bezier.c`