Computer Graphics

Surfaces

Based on slides by Dianna Xu, Bryn Mawr College
Parametric Surfaces

- Generalizing from curves to surfaces by using two parameters $u$ and $v$

- Parametric surfaces can be either rectangular or triangular, depending on how the parameter plane is divided
Parametric Surfaces

- Parametric surface:

\[
p(u,v) = \begin{bmatrix} f_x(u,v) \\ f_y(u,v) \\ f_z(u,v) \end{bmatrix} = \sum_{i=0}^{n} \sum_{j=0}^{m} C_{i,j} u^i v^j
\]

- Cubic interpolating patch:

\[
p(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)p_{ij}
\]
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 Patch

Need 16 conditions to determine the 16 coefficients $c_{ij}$
Choose at $u,v = 0, 1/3, 2/3, 1$
Approximating Derivatives

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

slope \( p'(0) \)

\[ p'(1) \approx \frac{p_3 - p_2}{1/3} \]

slope \( p'(1) \)

\( p_1 \) located at \( u=1/3 \)

\( p_2 \) located at \( u=2/3 \)
Bezier Matrix

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

\[ p(u) = u^T M_B P = b(u)^T P \]

blending functions
Blending Functions

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

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

Using same data array $\mathbf{P} = [p_{ij}]$ as with interpolating form

$$p(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)p_{ij} = u^T M_B P M_B^T v$$

Patch lies in convex hull
Bézier Surfaces

- Defined in terms of a two dimensional control net
B-spline Surfaces: local flexibility

- Local flexibility is one of the most desirable properties of B-splines
- Modification of a control point only affects a small neighborhood
B-Spline Patches

\[ p(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)p_{ij} = u^T M_s P M_s^T v \]

defined over only 1/9 of region
Basis Functions

In terms of the blending polynomials

\[ B_i(u) = \begin{cases} 
0 & u < i - 2 \\
b_0(u + 2) & i - 2 \leq u \leq i - 1 \\
b_1(u + 1) & i - 1 \leq u \leq i \\
b_2(u) & i \leq u \leq i + 1 \\
b_3(u - 1) & i + 1 \leq u \leq i + 2 \\
0 & u \geq i + 2 
\]
Evaluating Polynomials

- Simplest method to render a polynomial curve is to evaluate the polynomial at many points and form an approximating polyline.
- For surfaces we can form an approximating mesh of triangles or quadrilaterals.
- Use Horner’s method to evaluate polynomials:
  \[ p(u) = c_0 + u(c_1 + u(c_2 + uc_3)) \]
  - 3 multiplications/evaluation for cubic.
Finite Differences

For equally spaced \( \{u_k\} \) we define finite differences

\[
\Lambda^{(0)} p(u_k) = p(u_k)
\]

\[
\Lambda^{(1)} p(u_k) = p(u_{k+1}) - p(u_k)
\]

\[
\Lambda^{(m+1)} p(u_k) = \Lambda^{(m)} p(u_{k+1}) - \Lambda^{(m)} p(u_k)
\]

For a polynomial of degree \( n \), the \( n \)th finite difference is constant
Building a Finite Difference Table

\[ p(u) = 1 + 3u + 2u^2 + u^3 \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>1</td>
<td>7</td>
<td>23</td>
<td>55</td>
<td>109</td>
<td>191</td>
</tr>
<tr>
<td>( \Delta^{(1)} p )</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>( \Delta^{(2)} p )</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Delta^{(3)} p )</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finding the Next Values

Starting at the bottom, we can work up generating new values for the polynomial

<table>
<thead>
<tr>
<th>t</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>1</td>
<td>7</td>
<td>23</td>
<td>55</td>
<td>109</td>
<td>191</td>
</tr>
<tr>
<td>$\Delta^{(1)}P$</td>
<td>6</td>
<td>16</td>
<td>32</td>
<td>54</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>$\Delta^{(2)}P$</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Delta^{(3)}P$</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
de Casteljau Recursion

- We can use the convex hull property of Bezier curves to obtain an efficient recursive method that does not require any function evaluations.
- Uses only the values at the control points.
- Repeatedly refine the control polygon until point on curve is reached.
Splitting a Cubic Bezier

$p_0, p_1, p_2, p_3$ determine a cubic Bezier polynomial and its convex hull

Consider left half $l(u)$ and right half $r(u)$
Efficient Form

\[ l_0 = p_0 \]
\[ r_3 = p_3 \]
\[ l_1 = \frac{1}{2}(p_0 + p_1) \]
\[ r_2 = \frac{1}{2}(p_2 + p_3) \]
\[ l_2 = \frac{1}{2}(l_1 + \frac{1}{2}(p_1 + p_2)) \]
\[ r_1 = \frac{1}{2}(r_2 + \frac{1}{2}(p_1 + p_2)) \]
\[ l_3 = r_0 = \frac{1}{2}(l_2 + r_1) \]

Requires only shifts and adds!
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 \( Q \).
  \[
p(u) = u^T M_s Q
\]
- There exist Bezier control points \( P \) such that
  \[
p(u) = u^T M_b P
\]
- Equating and solving, we find
  \[
P = M_b^{-1} M_s Q
\]
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.
Surfaces

- Can apply the recursive method to surfaces if we recall that for a Bezier patch curves of constant $u$ (or $v$) are Bezier curves in $u$ (or $v$)
- First subdivide in $u$
  - Process creates new points
  - Some of the original points are discarded

![Diagram of surfaces with original and discarded points highlighted](image)
Second Subdivision

- New points created by subdivision
- Old points discarded after subdivision
- Old points retained after subdivision

16 final points for 1 of 4 patches created
Utah Teapot

- Most famous data set in computer graphics
- Widely available as a list of 306 3D vertices and the indices that define 32 Bezier patches
What Does OpenGL Support?

- Evaluators: a general mechanism for working with the Bernstein polynomials
  - Can use any degree polynomials
  - Can use in 1-4 dimensions
  - Automatic generation of normals and texture coordinates
- NURBS supported in GLU
- Quadrics
  - GLU and GLUT contain polynomial approximations of quadrics
One-Dimensional Evaluators

- Evaluate a Bernstein polynomial of any degree at a set of specified values
- Can evaluate a variety of variables
  - Points along a 2, 3 or 4 dimensional curve
  - Colors
  - Normals
  - Texture Coordinates
- We can set up multiple evaluators that are all evaluated for the same value
Setting Up an Evaluator

what we want to evaluate

max and min of \( u \)

\texttt{glMap1f(type, u\_min, u\_max, stride, order, pointer\_to\_array)}

1+degree of polynomial

separation between data points

pointer to control data

Each type must be enabled by \texttt{glEnable(type)}
Example

Consider an evaluator for a cubic Bezier curve over (0,1)

Point cpoints[]={............}; * /3d data /*

`glMap1f(GL_MAP_VERTEX_3, 0.0, 1.0, 3, 4, cpoints);`

data are 3D vertices

data are arranged as x,y,z,x,y,z......
three floats between data points in array

`glEnable(GL_MAP_VERTEX_3);`
Evaluating

- The function `glEvalCoord1f(u)` causes all enabled evaluators to be evaluated for the specified \( u \)
  - Can replace `glVertex`, `glNormal`, `glTexCoord`
  - The values of \( u \) need not be equally spaced
Example

- Consider the previous evaluator that was set up for a cubic Bezier over (0,1)
- Suppose that we want to approximate the curve with a 100 point polyline

```c
glBegin(GL_LINE_STRIP)
    for(i=0; i<100; i++)
        glEvalCoord1f( (float) i/100.0);
glEnd();
```
Equally Spaced Points

Rather than using a loop, we can set up an equally spaced mesh (grid) and then evaluate it with one function call

```c
glMapGrid(100, 0.0, 1.0);
```

sets up 100 equally-spaced points on (0,1)

```c
glEvalMesh1(GL_LINE, 0, 99);
```

renders lines between adjacent evaluated points from point 0 to point 99
Bezier Surfaces

- Similar procedure to 1D but use 2D evaluators in \( u \) and \( v \)

  \[
glMap2f(type, \ u_{\text{min}}, \ u_{\text{max}}, \ u_{\text{stride}}, \ u_{\text{order}}, \ v_{\text{min}}, \ v_{\text{max}}, \ v_{\text{stride}}, \ v_{\text{order}}, \ \text{pointer\_to\_data})
\]

- Evaluate with \( glEvalCoord2f(u,v) \)
Example

bicubic over \((0,1) \times (0,1)\)

Point cpoints[4][4] = {………};

`glMap2f(GL_MAP_VERTEX_3, 0.0, 1.0, 3, 4, 0.0, 1.0, 12, 4, cpoints);`

Note that in v direction data points are separated by 12 floats since array data is stored by rows
must draw in both directions

```c
for(j=0;j<100;j++) {
    glBegin(GL_LINE_STRIP);
    for(i=0;i<100;i++)
        glEvalCoord2f((float) i/100.0, (float) j/100.0);
    glEnd();

    glBegin(GL_LINE_STRIP);
    for(i=0;i<100;i++)
        glEvalCoord2f((float) j/100.0, (float) i/100.0);
    glEnd();
}
```
for (j=0; j<99; j++) {
    glBegin(GL_QUAD_STRIP);
    for (i=0; i<100; i++) {
        glEvalCoord2f ((float) i/100.0, (float) j/100.0);
        glEvalCoord2f ((float)(i+1)/100.0, (float)j/100.0);
    }
    glEnd();
}
Uniform Meshes

- We can form a 2D mesh (grid) in a similar manner to 1D for uniform spacing
  
  \[
  \text{glMapGrid2}(u\_num, u\_min, u\_max, v\_num, v\_min, v\_max)
  \]

- Can evaluate as before with lines or if want filled polygons
  
  \[
  \text{glEvalMesh2}(\text{GL\_FILL}, u\_start, u\_num, v\_start, v\_num)
  \]
Rendering with Lighting

- If we use filled polygons, we have to shade or we will see solid color uniform rendering.
- Can specify lights and materials but we need normals
  - Let OpenGL find them
    ```
    glEnable(GL_AUTO_NORMAL);
    ```
NURBS

- OpenGL supports NURBS surfaces through the GLU library

- Why GLU?
  - Can use evaluators in 4D with standard OpenGL library
  - Many complexities with NURBS that need a lot of code
  - There are five NURBS surface functions plus functions for trimming curves that can remove pieces of a NURBS surface
Quadrics

- Quadrics are in both the GLU and GLUT libraries
  - Both use polygonal approximations where the application specifies the resolution
  - Sphere: lines of longitude and latitude
- GLU: disks, cylinders, spheres
  - Can apply transformations to scale, orient, and position
- GLUT: Platonic solids, torus, Utah teapot, cone
GLUT Objects

- `glutWireCone()`
- `glutWireTorus()`
- `glutWireTeapot()`
GLUT Platonic Solids

- glutWireTetrahedron()
- glutWireDodecahedron()
- glutWireOctahedron()
- glutWireIcosahedron()
Quadric Objects in GLU

- GLU can automatically generate normals and texture coordinates
- Quadrics are objects that include properties such as how we would like the object to be rendered

disk
partial disk
sphere
Defining a Cylinder

GLUquadricOBJ *p;
P = gluNewQuadric(); /*set up object */
gluQuadricDrawStyle(GLU_LINE); /*render style*/
gluCylinder(p, BASE_RADIUS, TOP_RADIUS,
            BASE_HEIGHT, sections, slices);