Objectives

- Introduce types of curves and surfaces
  - Explicit
  - Implicit
  - Parametric
  - Strengths and weaknesses
- Discuss Modeling and Approximations
  - Conditions
  - Stability

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

Modeling with Curves

- 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 exactly solve for points that satisfy the equation.

Algebraic Surface

\[ \sum_{i,j,k} a_{ij}x^iy^jz^k \]

- Quadric surface \( 2 \geq i+j+k \)
- At most 10 terms
- Can solve intersection with a ray by reducing problem to solving quadratic equation.

Parametric Curves

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

Selecting Functions

- Usually we can select "good" functions
  - not unique for a given spatial curve
  - Approximate or interpolate known data
  - Want functions which are easy to evaluate
  - Want functions which are easy to differentiate
    - Computation of normals
    - Connecting pieces (segments)
  - Want functions which are smooth

Parametric Lines

We can normalize \( u \) to be over the interval \((0,1)\)

- Line connecting two points \( p_0 \) and \( p_1 \)
  \[ p(u) = (1-u)p_0 + up_1 \]
  \[ p(0) = p_0 \]
  \[ p(1) = p_1 \]
- Ray from \( p_0 \) in the direction \( d \)
  \[ p(u) = p_0 + ud \]
  \[ p(0) = p_0 \]
  \[ p(1) = p_0 + d \]

Parametric Surfaces

- Surfaces require 2 parameters
  \[ \begin{align*}
  x &= x(u,v) \\
  y &= y(u,v) \\
  z &= z(u,v)
  \end{align*} \]
  \[ p(u,v) = [x(u,v), y(u,v), z(u,v)]^T \]
- Want same properties as curves:
  - Smoothness
  - Differentiability
  - Ease of evaluation
Normals

We can differentiate with respect to u and v to obtain the normal at any point \( \mathbf{p} \):

\[
\frac{\partial \mathbf{p}(u, v)}{\partial u} = \frac{\partial^2 \mathbf{p}(u, v)}{\partial u^2} \quad \text{and} \quad \frac{\partial \mathbf{p}(u, v)}{\partial v} = \frac{\partial^2 \mathbf{p}(u, v)}{\partial v^2}
\]

Tangents in u and v directions

\[
\mathbf{n} = \frac{\partial \mathbf{p}(u, v)}{\partial u} \times \frac{\partial \mathbf{p}(u, v)}{\partial v}
\]

Parametric Planes

point-vector form

\[
\mathbf{p}(u, v) = \mathbf{p}_0 + u \mathbf{q} + v \mathbf{r}
\]

three-point form

\[
\mathbf{q} = \mathbf{p}_1 - \mathbf{p}_0 \\
\mathbf{r} = \mathbf{p}_2 - \mathbf{p}_0
\]

Parametric Sphere

\[
x(u, v) = r \cos \theta \sin \phi \\
y(u, v) = r \sin \theta \sin \phi \\
z(u, v) = r \cos \phi
\]

\[
360 \geq \theta \geq 0 \\
180 \geq \phi \geq 0
\]

\( \theta \) constant: circles of constant longitude
\( \phi \) constant: circles of constant latitude

differentiate to show \( \mathbf{n} = \mathbf{p} \)

Parametric Polynomial Curves

\[
x(u) = \sum_{i=0}^{N} c_{x,i} u^i \\
y(u) = \sum_{j=0}^{M} c_{y,j} u^j \\
z(u) = \sum_{k=0}^{K} c_{z,k} u^k
\]

-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 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
Cubic Parametric Polynomials

- \( N=M=L=3 \), gives balance between ease of evaluation and flexibility in design

\[
p(u) = \sum_{k=0}^{3} c_k u^k
\]

- 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

Cubic Polynomial Surfaces

\[
p(u,v) = [x(u,v), y(u,v), z(u,v)]^T
\]

where

\[
p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{ij} u^i v^j
\]

\( p \) is any of \( x, y \) or \( z \)

Need 48 coefficients (3 independent sets of 16) to determine a surface patch

Designing Parametric Cubic Curves

Objectives

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

Matrix-Vector Form

\[
p(u) = \sum_{k=0}^{3} c_k u^k
\]

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 \( c_0, c_1, c_2, c_3 \)
Interpolation Equations

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

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

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

\[
p = Ac
\]

Interpolation Matrix

Solving for \( c \) we find the interpolation matrix

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
9 & -18.5 & 9 & -4.5 \\
22.5 & -45 & 22.5 & 4.5 \\
\end{bmatrix}
\]

\[
c = M p
\]

Note that \( M \) does not depend on input data and can be used for each segment in \( x, y, \) and \( z \)

Blending Functions

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

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

where \( b(u) = [b_0(u) \ b_1(u) \ b_2(u) \ b_3(u)]^T \) is an array of blending polynomials such that

\[
p(u) = b_0(u)p_0 + b_1(u)p_1 + b_2(u)p_2 + b_3(u)p_3
\]

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

Blending Functions

- These functions are not smooth
  - Hence the interpolation polynomial is not smooth

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 since we need to satisfy only two conditions rather than three at each join point

Parametric Continuity

- **Continuity** (recall from the calculus):
  - Two curves are \( C^i \) continuous at a point \( p \) iff the \( i \)-th derivatives of the curves are equal at \( p \)
Interpolating Multiple Segments

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

Get continuity at join points but not continuity of derivatives

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 conditions per segment

Ensures continuity and first derivative continuity between segments

Equations

Interpolating conditions are the same at ends

\[
\begin{align*}
\text{p}(0) &= p_0 = c_0 \\
\text{p}(1) &= p_3 = c_0 + c_1 + c_2 + c_3 \\
\text{p}(u) &= c_0 + c_1 u + c_2 u^2 + c_3 u^3 \\
\text{p}'(0) &= p'_0 = c_1 \\
\text{p}'(1) &= p'_3 = c_1 + 2c_2 + 3c_3
\end{align*}
\]

Differentiating we find \( \text{p}'(u) = c_1 + 2c_2 + 3u c_3 \)

Evaluating at end points

\[
\begin{align*}
\text{p}'(0) &= p'_0 = c_1 \\
\text{p}'(1) &= p'_3 = c_1 + 2c_2 + 3c_3
\end{align*}
\]

Matrix Form

\[
\begin{bmatrix}
q_0 \\
q_1 \\
q'_0 \\
q''_0 \\
q'''_0
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 2 & 3
\end{bmatrix}
\begin{bmatrix}
p_0 \\
p_1 \\
p'_0 \\
p''_0
\end{bmatrix}
\]

Solving, we find \( q = M H q \) where \( M_H \) is the Hermite matrix

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

Blending Polynomials

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

\[
\begin{bmatrix}
2u^3 - 3u^2 + 1 \\
-2u^3 + 3u^2 \\
2u^2 + u \\
u - u^2
\end{bmatrix}
\]

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
Example

Here the p and q have the same tangents at the ends of the segment but different derivatives
Generate different Hermite curves
This techniques is used in drawing applications

Bezier and Spline Curves

Objectives

Introduce 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

Computing Derivatives

p_1 located at u=1/3
p'(0) = \frac{p_2 - p_1}{1/3}
Slope p'(0) = \frac{p}{1/3}

p_2 located at u=2/3
p'(1) = \frac{p_3 - p_2}{1/3}
Slope p'(1) = \frac{p}{1/3}

Equations

Interpolating conditions are the same
\begin{align*}
p(0) &= p_0 = c_0 \\
p(1) &= p_3 = c_0 + c_1 + c_2 + c_3
\end{align*}

Approximating derivative conditions
\begin{align*}
p'(0) &= 3(p_1 - p_0) = c_1 \\
p'(1) &= 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3
\end{align*}

Solve four linear equations for \( c = M a p \)
**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**

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

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

**Cubic Bezier Curve**

- Multiplying it all out gives

\[p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u) p_2 + u^3 p_3\]

**Bernstein Polynomials**

- The blending functions are a special case of the Bernstein polynomials

\[b_{kd}(u) = \frac{d!}{k!(d-k)!} u^k (1-u)^{d-k}\]

- 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)

**General Form of Bezier Curve**

\[\bar{p}(u) = \sum_{i=0}^{k} \bar{p}_{i+1} \begin{pmatrix} k \\ i \end{pmatrix} (1-u)^{k-i} u^i\]

**Convex Hull Property**

- The properties of the Bernstein polynomials ensure that all Bezier curves lie within 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, its 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 \( p=[p_{-2}, p_{-1}, p_0, p_1]^T \) to define curve only between \( p_{-1} \) and \( p_0 \)
• 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

\[
p(u) = u^TM_p = b(u)^Tp
\]

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

B-splines: Knot Selection

• Instead of working with the parameter space \( 0 \leq t \leq 1 \), use \( t_{\text{min}} \leq t_0 \leq t_1 \leq t_2 \cdots \leq t_{m-1} \leq t_{\text{max}} \)
• The knot points
  - Joint points between curve segments, \( Q_i \)
  - Each has a knot value
  - \( m-1 \) knots for \( m+1 \) points

Blending Functions

\[
b(u) = \frac{1}{6} \begin{bmatrix}
(1-u)^3 \\
4 - 6u^2 + 3u^3 \\
1 + 3u + 3u^2 - 3u^3
\end{bmatrix}
\]

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
  \[
p(u) = \sum_{i=0}^{m-1} B_i(u)p_i
\]
  defining the basis functions \( \{B_i(u)\} \)
### Basis Functions

In terms of the blending polynomials

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

### B-splines: Cox-deBoor Recursion

- Cox-deBoor Algorithm: defines the blending functions for spline curves (not limited to deg 3)
- Curves are weighted avgs of lower degree curves
- Let \( B_{i,d}(t) \) denote the \( i \)-th blending function for a B-spline of degree \( d \), then:

\[
B_{i,d}(t) = \begin{cases} 
1, & \text{if } t_i \leq t < t_{i+1} \\
0, & \text{otherwise}
\end{cases}
\]

\[
B_{i,d}(t) = \frac{t - t_i}{t_{i+d} - t_i} B_{i,d-1}(t) + \frac{t_{i+d+1} - t}{t_{i+d+1} - t_{i+1}} B_{i+1,d-1}(t)
\]

### B-spline Blending Functions

- \( B_{i,0}(t) \) is a step function that is 1 in the interval
- \( B_{i,1}(t) \) spans two intervals and is a piecewise linear function that goes from 0 to 1 (and back)
- \( B_{i,2}(t) \) spans three intervals and is a piecewise quadratic that grows from 0 to 1/4, then up to 3/4 in the middle of the second interval, back to 1/4, and back to 0
- \( B_{i,3}(t) \) is a cubic that spans four intervals growing from 0 to 1/6 to 2/3, then back to 1/6 and to 0

### Generalizing Splines

- We can extend to splines of any degree
- Data and conditions do 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 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

---

Pics/Math courtesy of Dave Mount @ UMD
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 \( q \).
- There exist Bezier control points \( p \) such that \( p(u) = u^T M_I q \).
- Equating and solving, we find \( p = M_B^{-1} M_I \).

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.

Rendering Other Polynomials

- Every polynomial is a Bezier polynomial for some set of control data.
- We can use a Bezier renderer if we first convert the given control data to Bezier control data.
  - Equivalent to converting between matrices.
- Example: Interpolating to Bezier
  \( M_B = M_I M_B^{-1} \).

Objectives

- Introduce methods to draw curves
  - Approximate with lines
  - Subdivision
- Derive the recursive method for evaluation of Bezier curves
- Learn how to convert all polynomial data to data for Bezier polynomials.
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.

\[ p(u) = c_0 + u(c_1 + u(c_2 + u c_3)) \]

- 3 multiplications/evaluation for cubic.

The de Casteljau Algorithm

- Basic case, with two points:
  - Plotting a curve via repeated linear interpolation.
  - Given \( p_0, p_1, \ldots \)
  - a sequence of control points.
  - Simple case: Mapping a parameter \( u \) to the line \( p_0, p_1 \).

\[ p(u) = (1-u)p_0 + up_1 \quad \text{for } 0 \leq u \leq 1. \]

The de Casteljau Algorithm

- The complete solution from the algorithm for three iterations:

\[
\begin{align*}
\mathbf{p}_0 &= (1-u)\mathbf{p}_0 + u\mathbf{p}_1 \\
\mathbf{p}_1 &= (1-u)\mathbf{p}_1 + u\mathbf{p}_2 \\
\mathbf{p}_2 &= (1-u)\mathbf{p}_2 + u\mathbf{p}_3 \\
\mathbf{p}(u) &= (1-u)\mathbf{p}_0 + up_1 \\
\end{align*}
\]

The de Casteljau Algorithm

- The solution after four iterations:

\[
\begin{align*}
\mathbf{p}_0 &= (1-u)\mathbf{p}_0 + u\mathbf{p}_1 \\
\mathbf{p}_1 &= (1-u)\mathbf{p}_1 + u\mathbf{p}_2 \\
\mathbf{p}_2 &= (1-u)\mathbf{p}_2 + u\mathbf{p}_3 \\
\mathbf{p}_3 &= (1-u)\mathbf{p}_3 + u\mathbf{p}_4 \\
\mathbf{p}(u) &= (1-u)\mathbf{p}_0 + up_1 \\
\end{align*}
\]

The de Casteljau Algorithm

- Input: \( \mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n \in \mathbb{R}^3 \), \( t \in \mathbb{R} \)
- Iteratively set:

\[
p_{0,i}(t) = (1-t)p_{i,i+1}(t) + t p_{i,i+1}(1-t) \\
\text{for } i = 0, \ldots, n - 1 \quad \text{and} \quad p_{i,n}(t) = p_i
\]

Then \( p_{0,i}(t) \) is the point with parameter value \( t \) on the Bézier curve defined by the \( p_i \)'s.
De Casteljau: Cubic Curve Animation

Subdivision

- Common in many areas of graphics, CAD, CAGD, vision
- Basic idea
  - primitives defined by control polygons
  - set of control points is not unique
    - more than one way to compute a curve
  - subdivision refines representation of an object by introducing more control points
- Allows for local modification
- Subdivide to pixel resolution

Bézier Curve Subdivision

- Subdivision allows display of curves at different/adaptive levels of resolution
- Rendering systems (OpenGL, ActiveX, etc) only display polygons or lines
- Subdivision generates the lines/facets that approximate the curve/surface
  - output of subdivision sent to renderer

deCasteljau 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
- Based on the idea that “any polynomial and any part of a polynomial is a Bezier polynomial for properly chosen control data”

Splitting a Cubic Bezier

Consider left half \( l(u) \) and right half \( r(u) \)

Since \( l(u) \) and \( r(u) \) are Bezier curves, we should be able to find two sets of control points \( \{l_0, l_1, l_2, l_3\} \) and \( \{r_0, r_1, r_2, r_3\} \) that determine them
Convex Hulls

\{l_0, l_1, l_2, l_3\} and \{r_0, r_1, r_2, r_3\} each have a convex hull that is closer to \(p(u)\) than the convex hull of \{\(p_0, p_1, p_2, p_3\)\}. This is known as the variation diminishing property.

The polyline from \(l_0\) to \(l_3\) (\(= r_0\)) to \(r_3\) is an approximation to \(p(u)\). Repeating recursively we get better approximations.

Efficient Form

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

Requires only shifts and adds!

Drawing Parametric Curves

Two basic ways:

- **Iterative evaluation** of \(x(t), y(t), z(t)\) for incrementally spaced values of \(t\)
  - can’t easily control segment lengths and error

- **Recursive Subdivision** via de Casteljau, that stops when control points get sufficiently close to the curve
  - i.e. when the curve is nearly a straight line

Drawing Parametric Curves via Recursive Subdivision

- **Idea:** stop subdivision when segment is flat enough to be drawn w/ straight line

- **Curve Flatness Test:**
  - based on the convex hull
  - if \(d_2\) and \(d_3\) are both less than some \(e\), then the curve is declared flat

FYI: Computing the Distance from a Point to a Line

- **Line** is defined with two points
- **Basic idea:**
  - Project point \(P\) onto the line
  - Find the location of the projection

\[
d(P, L) = \frac{(y_0 - y_1)x + (x_0 - x_1)y + (x_0y_1 - x_1y_0)}{\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}}
\]
### Bezier and Spline Surfaces

- **Issues with 3D “mesh” formats**
  - Easy to acquire
  - Easy to render
  - Harder to model with
  - Error prone
    - split faces, holes, gaps, etc

### BRep Data Structures

- **Winged-Edge Data Structure (Weiler)**
- **Vertex**
  - \( n \) edges
- **Edge**
  - 2 vertices
  - 2 faces
- **Face**
  - \( m \) edges

### BRep Data Structure

- **Vertex structure**
  - X,Y,Z point
  - Pointers to \( n \) coincident edges
- **Edge structure**
  - 2 pointers to end-point vertices
  - 2 pointers to adjacent faces
  - Pointer to next edge
  - Pointer to previous edge
- **Face structure**
  - Pointers to \( m \) edges

### Biparametric Surfaces

- **Biparametric surfaces**
  - A generalization of parametric curves
  - 2 parameters: \( s \), \( t \) (or \( u \), \( v \))
  - Two parametric functions

### Biparametric Patch

- \((u,v)\) pair maps to a 3D point on patch
Bicubic Surfaces

• Recall the 2D curve: \( Q(s) = G \cdot M \cdot S \)
  - \( G \): Geometry Matrix
  - \( M \): Basis Matrix
  - \( S \): Polynomial Terms \([s^3 \ s^2 \ s \ 1]\)

• For 3D, we allow the points in \( G \) to vary in 3D along \( t \) as well:

\[
Q(s, t) = \begin{bmatrix} G_1(t) & G_2(t) & G_3(t) & G_4(t) \end{bmatrix} \cdot M \cdot S
\]

96

Observations About Bicubic Surfaces

• For a fixed \( t_i \), \( Q(s, t_i) \) is a curve
• Gradually incrementing \( t_i \) to \( t_{i+1} \) we get a new curve
• The combination of these curves is a surface
• \( G_i(t) \) are 3D curves

97

Bicubic Surfaces

• Each \( G_i(t) \) is \( G_i(t) = G_i \cdot M \cdot T \), where

\[
G_i = \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}
\]

• Transposing \( G_i(t) \), we get

\[
G_i(t) = T^T \cdot M^T \cdot G_i^T = T^T \cdot M^T \cdot \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}^T
\]

98

Bicubic Surfaces

• Substituting \( G_i(t) \) into \( Q(s) = G \cdot M \cdot S \) we get \( Q(s, t) \)
• The \( g_{ij} \), etc. are the control points for the bicubic surface patch:

\[
Q(s, t) = T^T \cdot M^T \cdot \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \\ g_{i5} & g_{i6} & g_{i7} & g_{i8} \\ g_{i9} & g_{i10} & g_{i11} & g_{i12} \\ g_{i13} & g_{i14} & g_{i15} & g_{i16} \end{bmatrix} \cdot M \cdot S
\]

99

Bicubic Surfaces

• Writing out gives \( Q(s, t) = T^T \cdot M^T \cdot G \cdot M \cdot S \) \( 0 \leq s, t \leq 1 \)

\[
x(s, t) = T^T \cdot M^T \cdot G_x \cdot M \cdot S \\
y(s, t) = T^T \cdot M^T \cdot G_y \cdot M \cdot S \\
z(s, t) = T^T \cdot M^T \cdot G_z \cdot M \cdot S
\]

100

Interpolating Patch

\[
p(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} c_{ij} u^i v^j
\]

Need 16 conditions to determine the 16 coefficients \( c_{ij} \)
Choose at \( u, v = 0, 1/3, 2/3, 1 \)
Matrix Form

Define \( v = [1 \ v \ v^2 \ v^3]^T \)
\[
C = [c_i] \quad P = [p_i]
\]
\[
p(u,v) = u^T Cv
\]

If we observe that for constant \( u (v) \), we obtain interpolating curve in \( v (u) \), we can show
\[
C=MPM^T
\]
\[
p(u,v) = u^T MPM^T v
\]

Blending Patches

Each \( b_i(u) \) is a blending patch
Shows that we can build and analyze surfaces from our knowledge of curves

Bézier Patches

- Bézier Surfaces (similar definition)

\[
x(s,t) = T^T \cdot M_B^T \cdot G_B \cdot M_B \cdot S
\]
\[
y(s,t) = T^T \cdot M_B^T \cdot G_B \cdot M_B \cdot S
\]
\[
z(s,t) = T^T \cdot M_B^T \cdot G_B \cdot M_B \cdot S
\]

Bezier 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)

Bézier Patches

- Expanding the summation

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

Each \( b_i(u)b_j(v) \) is a blending patch
Beziers Patch Matrix Form

\[ P(u, v) = u^T M_B P M_B^T v \]

Features of Bicubic Bezier Patch

- Interpolates 4 corner control points
- 4 edges are Bezier curves
- Lies within convex hull of control points
- Normal at 4 corners from nearby CPs

Bézier Surfaces

- \( C^0 \) and \( G^0 \) continuity
  can be achieved
  between two patches
  by setting the 4
  boundary control
  points to be equal
- \( G^1 \) continuity
  achieved when
  cross-wise CPs are
  co-linear

Bézier Surfaces: Example Utah Teapot

- Utah Teapot modeled
  with 306 3D control
  points that define 32
  Bézier patches with \( G^1 \)
  continuity

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
Faceting Overview

- Double loop that increments through the u and v parameters
  - Values between 0 and 1
- For each (u,v) pair calculate 3D point on patch. Keep track of linear index.
- This produces a 2-D array of 3D points on the patch and their indices to the linear array
- Define triangles that tessellate the patch

Defining the Triangles

```cpp
// This assumes that the vertices are in a 2D array, verts[i,j]
for i = 0 to (num_u - 2)
  for j = 0 to (num_v - 2)
    triangle0 = (verts[i,j], verts[i+1,j], verts[i+1,j+1])
    triangle1 = (verts[i,j], verts[i+1,j+1], verts[i,j+1])
```

Normals

- For rendering we need the normals if we want to shade
  - Can compute from parametric equations
    \[
    \mathbf{n} = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}
    \]
  - Can approximate with triangle normals

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

Bezier Surface: Example

- Increased facet resolution
- Rendered
Drawing Parametric Surfaces

- Usually done “patch by patch”
- Two choices
  - Draw/render directly from the parametric description
  - Approximate the surface with a polygon mesh, then draw/render the mesh

Direct Rendering

- Use a scan-line algorithm
  - Evaluate pixel by pixel
  - Problem: How to go from (x,y) “screen space” to point on the 3D patch
    - Easy for a planar polygon where we know max/min y, equations for edges, screen depth
    - Not as easy for parametric surfaces

Issues for Direct Rendering

- Max/Min y coords may not lie on boundaries
- Silhouette edges result from patch bulges
  - Need to track both silhouettes and boundaries
  - Note: patch edges need not be monotonic in x or y
- Idea: Scan convert patch plane-by-plane, using scan planes instead of scan lines

Direct Scan Conversion of Patches

- Basic idea
  - Find intersection of patch with XZ plane
  - Producing a planar curve
  - Draw the curve
    - De Boor, D’ Casteljeau
    - Note: if doing rendering, one can compute pixel-by-pixel color values this way
      - Patch: \( x=X(u,v), y=Y(u,v), z=Z(u,v) \)

Patch to Polygon Conversion

Two methods:
- **Object Space Conversion**
  - Techniques
    - Iterative evaluation
    - Uniform subdivision
    - Non-uniform subdivision
  - Resolution: depends on object space
- **Image Space Conversion**
  - Resolution: depends on pixels and screen

Object Space Conversion:
Uniform Subdivision

Basic Procedure
- Cut parameter space into equal parts
- Find new points on the surface
- Recurse/Repeat “until done”
- Split squares into triangles
- Render triangles
**Object Space Conversion: Non-Uniform Subdivision**

- **Basic idea**
  - More facets in areas of high curvature
  - Use change in normals to surface to assess curvature
    - More derivatives
  - Break patch into sub-patches based on curvature changes

**Image Space Conversion**

- **Idea:** control subdivision based on screen criteria
  - Minimum pixel area
    - Stop when patch is basically one pixel
  - Screen flatness
    - Stop when patch converges to a polygon
  - Screen flatness of silhouette edges
    - Stop when edge is straight or size of pixel

**How do I know if I’ve found a silhouette edge?**

- If the viewing ray is tangent to the surface at the point it hits the surface!
  
  \[ \mathbf{N}(X) \cdot \mathbf{L} = 0 \]
  
  - Where \( \mathbf{N} \) is the normal at the point where \( \mathbf{L} \), the line of sight, hits the surface

**Suggestions for HW7**

- Write a function that takes control points and a \((u,v)\) pair and returns a 3D point on patch
- Use formula or de Casteljau Algorithm to compute point
- Compute an array of points that lie on the patch with a double loop that increments through \(u\) and \(v\), from 0 to 1
- Iterate over integers!
- This would be an \((n+1) \times (m+1)\) array, where \(n\) is the number of quads in the \(u\) direction and \(m\) is the number in the \(v\) direction

**Silhouette Determination**

- \( \mathbf{N} \cdot \mathbf{L} = 0 \)

**Suggestions for HW7**

- Use a double loop to iterate through \(i\) & \(j\) = 0 to \(n-1\) & 0 to \(m-1\)
- For each \((i, j)\) pair you define two triangles. The first has vertices \((i,j), [i+1, j], [i, j+1]\). The 2nd triangle is defined with vertices \([i+1, j], [i+1, j+1], [i, j+1]\).
- Now you have a mesh defined like an SMF model. Modify your HW5 code to render it.
Suggestions for HW7

• Implement the interface that allows the user to change n and m (the resolution of the mesh)
• When these values are changed by the user, you'll need to regenerate the mesh
• Flip normals that face away from the eye point, so both sides of the mesh are shaded
• Or take abs(L • N)