Curves and Surfaces

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science

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

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 surface to get a curve
- In general, we cannot exactly solve for points that satisfy the equation

Algebraic Surface

\[ \sum_{i,j,k} a_{i,j,k} x^i y^j z^k \]
- 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 \( p_1 \)
  \[ p(u)=(1-u)p_0+up_1 \]
- Ray from \( p_0 \) in the direction \( d \)
  \[ p(u)=p_0+ud \]

Curve Segments

- After normalizing \( u \), each curve is written
  \[ p(u)=[x(u), y(u), z(u)]^T, \quad 1 \geq u \geq 0 \]
- In classical numerical methods, we design a single global curve
- In computer graphics and CAD, it is better to design small connected curve segments
Parametric Polynomial Curves

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

- 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 \( x, y, z \)

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

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

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 \( c_0, c_1, c_2, c_3 \)

Special case of the Lagrange polynomial interpolation

**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

**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

**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

- $p(0) = p_0 = c_0$
- $p(1) = p_3 = c_0 + c_1 + c_2 + c_3$

Approximating derivative conditions
- $p'(0) = 3(p_1 - p_0) = c_1$
- $p'(1) = 3(p_3 - p_2) = c_1 + 2c_2 + 3c_3$

Solve four linear equations for $c = M_B \cdot p$

Bezier Matrix

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

$p(u) = u^3 M_B \cdot p = b(u)^3 p$

blending functions

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

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

\[ 0 \leq u \leq 1 \]

Bernstein Polynomials

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

\[ b_{mk}(u) = \binom{d}{k} u^k (1-u)^{d-k} \]

• 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

\[ \mathbf{p}(u) = \sum_{i=0}^{k} \mathbf{p}_{i+1} \binom{k}{i} (1-u)^{k-i} u^i \]

Convex Hull Property

• The properties of the Bernstein polynomials ensure that all Bezier curves lie within the convex hull of their control points

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 \( C^r \) continuous at a point \( p \) if the \( r \)-th derivatives of the curves are equal at \( p \)
Chaining Bézier Curves

- Fit curve through set of data points

Catmull-Rom Splines

- An interpolating spline through multiple points
- Like Bézier, equivalent to Hermite
  - in fact, all splines of this form are equivalent
- First example of a spline based on just an input point sequence
- Does not have convex hull property
- Only has C1 continuity

Catmull-Rom splines

- Tangents are \((p_{k+1} - p_{k-1})/2\) for interior control points \((p_k)\)
- User specifies tangents at first \((\mathcal{T}_0)\) and last \((\mathcal{T}_n)\) input points
- Or fit parabola to first/last 3 points
  - \(q_0 = p_k\)
  - \(q_1 = p_{k+1}\)
  - \(t_0 = 0.5(p_{k+1} - p_{k-1})\)
  - \(t_1 = 0.5(p_{k+2} - p_k)\)

B-Splines

- Basis splines: use the data at \(p=[p_{i-2} p_{i-1} p_i p_{i+1}]^T\)
  - 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
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

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

B-spline Blending Functions

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

B-spline Blending Functions for 2nd Degree Splines

• Note: can’t define a polynomial with these properties (both 0 and non-zero for ranges)
• Idea: subdivide the parameter space into intervals and build a piecewise polynomial
  - Each interval gets different polynomial function

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

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 or degree \( d \), then:

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

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

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
NURBS

- At the core of several modern CAD systems - I-DEAS, Pro/E, Alpha 1
- Describes analytic and freeform shapes
- Accurate and efficient evaluation algorithms
- Invariant under affine and perspective transformations

Rendering Curves

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

Evaluating Polynomials

\[ p(u) = c_0 + u(c_1 + uc_2 + uc_3) \]
- 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 \( \overline{p_0p_1} \)
- \( p(u) = (1-u)p_0 + up_1 \) for \( 0 \leq u \leq 1 \)

The de Casteljau Algorithm

- The complete solution from the algorithm for three iterations:
  \[ p_{01}(u) = (1-u)p_0 + up_1 \]
  \[ p_{11}(u) = (1-u)p_1 + up_2 \]
  \[ p(u) = (1-u)p_{01}(u) + up_{11}(u) \]
The de Casteljau Algorithm

- The solution after four iterations:

\[ p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12} \]

\[ p_{i0}(t) = (1-t)p_{i-1} + tp_i \]

\[ p_{ir}(t) = (1-t)p_{i(r-1)} + tp_{i+1}(r-1) \]

Then \( p_{i0}(t) \) is the point with parameter value \( t \) on the Bézier curve defined by the \( p_i \)’s.

De Casteljau: Arc Segment Animation

De Casteljau: Cubic Curve Animation

Cubic Bezier Curve

- Multiplying it all out gives

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

\[ 0 \leq u \leq 1 \]

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”

**Bézier Curve Subdivision**

- Observe subdivision:
  - does not affect the shape of the curve
  - partitions one curve into several curved pieces with (collectively) the same shape

**Splitting a Cubic Bezier**

\[ \begin{align*}
\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3 & \text{ determine a cubic Bezier polynomial and its convex hull} \\
\mathbf{p}_0 &= \mathbf{l}_0 \\
\mathbf{r}_3 &= \mathbf{r}_3 \end{align*} \]

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

**Convex Hulls**

\[ \begin{align*}
\{\mathbf{l}_0, \mathbf{l}_1, \mathbf{l}_2, \mathbf{l}_3\} \text{ and } \{\mathbf{r}_0, \mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3\} \text{ each have a convex hull that is closer to } \mathbf{p}(u) \text{ than the convex hull of } \{\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\} \\
\text{This is known as the variation diminishing property.} \\
The polyline from } \mathbf{l}_0 \text{ to } \mathbf{l}_3 \text{ to } \mathbf{r}_1 \text{ to } \mathbf{r}_3 \text{ is an approximation to } \mathbf{p}(u). \text{ Repeating recursively we get better approximations.} \]
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

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

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

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

Biparametric Surfaces

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

 Parametric Surfaces

• Surfaces require 2 parameters
  \[ x = x(u, v) \]
  \[ y = y(u, v) \]
  \[ z = z(u, v) \]
  \[ p(u, v) = [x(u, v), y(u, v), z(u, v)]^T \]
• Want same properties as curves:
  - Smoothness
  - Differentiability
  - Ease of evaluation

Parametric Planes

point-vector form

\[ p(u, v) = p_0 + uq + vr \]
\[ n = q \times r \]
three-point form

\[ q = p_1 - p_0 \]
\[ r = p_2 - p_0 \]

Parametric Sphere

\[ x(u, v) = r \cos \theta \sin \phi \]
\[ y(u, v) = r \sin \theta \sin \phi \]
\[ z(u, v) = r \cos \phi \]
\[ 360 \geq \theta \geq 0 \]
\[ 180 \geq \phi \geq 0 \]
\[ \theta \text{ constant: circles of constant longitude} \]
\[ \phi \text{ constant: circles of constant latitude} \]

differentiate to show \[ n = \vec{p} \]
Normals

We can differentiate with respect to $u$ and $v$ to obtain the normal at any point $p$:

$$\frac{\partial p(u,v)}{\partial u} = \begin{bmatrix} \frac{\partial p_x(u,v)}{\partial u} \\ \frac{\partial p_y(u,v)}{\partial u} \\ \frac{\partial p_z(u,v)}{\partial u} \end{bmatrix}$$

$$\frac{\partial p(u,v)}{\partial v} = \begin{bmatrix} \frac{\partial p_x(u,v)}{\partial v} \\ \frac{\partial p_y(u,v)}{\partial v} \\ \frac{\partial p_z(u,v)}{\partial v} \end{bmatrix}$$

Tangents in $u$ and $v$ directions:

$$n = \frac{\partial p(u,v)}{\partial u} \times \frac{\partial p(u,v)}{\partial v}$$

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:

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

Observations About Bicubic Surfaces

- For a fixed $t_1$, $Q(s,t_1)$ is a curve
- Gradually incrementing $t_1$ to $t_2$, we get a new curve
- The combination of these curves is a surface
- $G_i(t)$ are 3D curves

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

$$G_i(t) = T^T \cdot M^T \cdot G_i^T$$

$$= T^T \cdot M^T \cdot \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}^T$$

- 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:

$$Q(s,t) = T^T \cdot M^T \cdot \begin{bmatrix} g_{11} & g_{21} & g_{31} & g_{41} \\ g_{12} & g_{22} & g_{32} & g_{42} \\ g_{13} & g_{23} & g_{33} & g_{43} \\ g_{14} & g_{24} & g_{34} & g_{44} \end{bmatrix} \cdot M \cdot S$$
Bicubic Surfaces

\[ Q(s,t) = T^T \cdot M^T \cdot G \cdot M \cdot S \quad 0 \leq s,t \leq 1 \]

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

Blending Functions

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

Each \( b_i(u)b_j(v) \) is a blending function

Shows that we can build and analyze surfaces from our knowledge of curves

A point on the patch is a weighted sum of the control points

Beziers Blending Functions

\[ b(u) = \begin{cases} 
(1-u)^3 & 0 \\ 3u(1-u)^2 & 0 \\ u^3 & 0 
\end{cases} \]

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

Beziers Patches

Using same data array \( \mathbf{p}=[\hat{p}_i] \) as with interpolating form

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

Patch lies in convex hull
### Bezier Patch Matrix Form

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

\[
= \begin{bmatrix} 1 & u & u^2 \end{bmatrix} \begin{bmatrix}
0 & 0 & 0 & \psi_0 & \psi_1 & \psi_2 & \psi_3 \\
-3 & 3 & 0 & 0 & 0 & 0 & 0 \\
3 & -6 & 3 & 0 & 0 & 0 & 0 \\
-1 & 3 & -3 & 1 & 0 & 0 & 0 \\
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 & 0 \\
-3 & 3 & 0 & 0 \\
3 & -6 & 3 & 0 \\
-1 & 3 & -3 & 1 \\
\end{bmatrix} \begin{bmatrix} \psi_0 & \psi_1 & \psi_2 & \psi_3 \end{bmatrix}
\]

### 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

### Faceting

- Faceting
Defining the Triangles

// This assumes that the vertices are in a 2D array, verts(i,j)
// num_u & num_v are the number of points in u and v directions

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])

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

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 by averaging 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
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!

\[ N(X) \cdot L = 0 \]

- Where \( N \) is the normal at the point where \( L \), the line of sight, hits the surface
Modified Gouraud Shading
(Verteex Shader)

```
flat out vec3 vColor;
vec3 ambient = ambientProduct;
// If normal pointing away from the eye, flip it
if (dot(E,N) < 0.0) N = vec3(-1,-1,-1) * N;
float diffuseTerm = max( dot(L, N), 0.0 );
vec3 diffuse = diffuseTerm*diffuseProduct;
float specularTerm = pow( max(dot(N, H), 0.0), shininess );
vec3 specular = specularTerm * specularProduct;
if (dot(L, N) < 0.0 ) specular = vec3(0.0, 0.0, 0.0);
gl_Position = projectionMatrix * vec4(pos, 1.0);

vColor = min(ambient + diffuse + specular, 1.0);
```

Suggestions for HW7

• Write a function that takes control points and a (u,v) pair and returns a 3D point on patch
• Use formula to compute point
• Compute an array of 3D 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 x m array, where n is the number of points in the u direction and m is the number in the v direction

Suggestions for HW7

• Display with gl.POINTS to test
• Use a double loop to iterate through i & j = 0 to n-2 & 0 to m-2
• 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 HW6 code to render it.

Suggestions for HW7

• Use flat qualifier to flat-shade mesh
• 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