Overview

• 3D model representations
• Mesh formats
• Bicubic surfaces
• Bezier surfaces
• Normals to surfaces
• Direct surface rendering
3D Modeling

• 3D Representations
  – Wireframe models
  – Surface Models
  – Solid Models
  – Meshes and Polygon soups
  – Voxel/Volume models
  – Decomposition-based
    • Octrees, voxels

• Modeling in 3D
  – Constructive Solid Geometry (CSG),
    Breps and feature-based
Representing 3D Objects

- **Exact**
  - Wireframe
  - Parametric Surface
  - Solid Model
    - CSG
    - BRep
    - Implicit Solid Modeling

- **Approximate**
  - Facet / Mesh
    - Just surfaces
  - Voxel
    - Volume info
Representing 3D Objects

• Exact
  – Precise model of object topology
  – Mathematically represent all geometry

• Approximate
  – A discretization of the 3D object
  – Use simple primitives to model topology and geometry
Positives when Representing 3D Objects

• Exact
  – Precision
    • Simulation, modeling, etc
  – Lots of modeling environments
  – Physical properties
  – High-level control
  – Many applications (tool path generation, motion, etc.)
  – Compact

• Approximate
  – Easy to implement
  – Easy to acquire
    • 3D scanner, CT
  – Easy to render
    • Direct mapping to the graphics pipeline
  – Lots of algorithms
Negatives when Representing 3D Objects

• Exact
  – Complex data structures
  – Expensive algorithms
  – Wide variety of formats, each with subtle nuances
  – Hard to acquire data
  – Translation required for rendering

• Approximate
  – Lossy
  – Data structure sizes can get HUGE, if you want good fidelity
  – Easy to break (i.e. cracks can appear)
  – Not good for certain applications
    • Lots of interpolation and guess work
Exact Representations

- Wireframe
- Parametric Surface
- Solid Model
  - operations
  - CSG, BRep, implicit geometry
Wireframes

• Basic idea:
  – Represent the model as the set of all of its edges

• Example:
  A simple cube
  – 12 lines
  – 8 vertices

• How about the faces?
Issues with Wireframes

• Visually ambiguous
• No surfaces!
  – What’s inside? What’s outside?
  – Hidden line removal?
• What does validity entail?
  – Don’t we just have a bunch of wires?
  – Do they need to add up to something?
• How to model wireframe shapes?
  – Wire by wire? Not very easy!
Surface Models

• Basic idea:
  – Represent a model as a set of faces/patches

• Limitations:
  – Topological integrity; how do faces “line up”; which way is ‘inside’ / ‘outside’?

• Used in many CAD applications
  – Why? They are fine for drafting and rendering, not as good for creating true physical models
3D Mesh File Formats

Some common formats

- STL
- SMF
- OpenInventor
- VRML
- X3D
Minimal

- Vertex + Face

- No colors, normals, or texture

- Primarily used to demonstrate geometry algorithms
Full-Featured

- Colors / Transparency
- Vertex-Face Normals (optional, can be computed)
- Scene Graph
- Lights
- Textures
- Views and Navigation
Simple Mesh Format (SMF)

- Michael Garland
  http://graphics.cs.uiuc.edu/~garland/

- Triangle data

- Vertex indices begin at 1

```
#$SMF 1.0
#$vertices 5
#$faces 6
v 2.0 0.0 2.0
v 2.0 0.0 -2.0
v -2.0 0.0 -2.0
v -2.0 0.0 2.0
v 0.0 5.0 0.0
f 1 3 2
f 1 4 3
f 3 5 2
f 2 5 1
f 1 5 4
f 4 5 3
```
Stereolithography (STL)

- Triangle data + Face Normal
- The de-facto standard for rapid prototyping

```plaintext
solid
...
facet normal 0.00 0.00 1.00
  outer loop
    vertex 2.00 2.00 0.00
    vertex -1.00 1.00 0.00
    vertex 0.00 -1.00 0.00
  endloop
endfacet
...
endsolid
```
Open Inventor

- Developed by SGI
- Predecessor to VRML
  - Scene Graph
Virtual Reality Modeling Language (VRML)

- SGML Based
- Scene-Graph
- Full Featured

```xml
#VRML V2.0 utf8
# A Cylinder
Shape {
    appearance Appearance {
        material Material {
        }
    }
    geometry Cylinder {
        height 2.0
        radius 1.5
    }
}
```
X3D

- Open standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML
- Supports
  - 2D/3D graphics, programmable shaders
  - 2D/3D compositing, CAD data, Animation
  - Spatialized audio and video, User interaction
  - Navigation, Scripting, Networking, Simulation
- See www.web3d.org for more info
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

- **Face structure**
  - Pointers to $m$ edges

- **Edge structure**
  - 2 pointers to end-point vertices
  - 2 pointers to adjacent faces
  - Pointer to next edge
  - Pointer to previous edge
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
  \[ F(u,v) = (x,y,z) = (x(u,v), y(u,v), z(u,v)) \]
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

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

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

- Substituting $G_i(t)$ into $Q(s) = G \cdot M \cdot S$, we get $Q(s, t)$
- The $g_{11}$, 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

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

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

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

\[
z(s,t) = T^T \cdot M^T \cdot G_z \cdot M \cdot S
\]
Bicubic Bézier Patch

- Bézier Surfaces
  (similar definition)

\[
x(s, t) = T^T \cdot M_B^T \cdot G_{Bx} \cdot M_B \cdot S \\
y(s, t) = T^T \cdot M_B^T \cdot G_{By} \cdot M_B \cdot S \\
z(s, t) = T^T \cdot M_B^T \cdot G_{Bz} \cdot M_B \cdot S
\]
Bicubic Bezier Patch

Using data array \( P = [\bar{p}_{ij}] \)

\[
\bar{p}(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)\bar{p}_{ij} = u^T \mathbf{M_B} \mathbf{P} \mathbf{M_B}^T v
\]
Bicubic Bézier Patches

- Expanding the summation

\[
\tilde{p}(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} b_i(u)b_j(v)\tilde{p}_{ij} = b_0(u)b_0(v)\tilde{p}_{00} + b_0(u)b_1(v)\tilde{p}_{01} + b_0(u)b_2(v)\tilde{p}_{02} + b_0(u)b_3(v)\tilde{p}_{03} + b_1(u)b_0(v)\tilde{p}_{10} + \text{ etc.}
\]

\[0 \leq u, v \leq 1\]
Cubic 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)
Bezier Patch Matrix Form

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

\[
= \begin{bmatrix}
1 & u & u^2 & u^3
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
-3 & 3 & 0 & 0 \\
3 & -6 & 3 & 0 \\
-1 & 3 & -3 & 1
\end{bmatrix}
\begin{bmatrix}
P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\
P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\
P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\
P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3}
\end{bmatrix}
\begin{bmatrix}
1 & -3 & 3 & -1 \\
0 & 3 & -6 & 3 \\
0 & 0 & 3 & -3 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 \\
v \\
v^2 \\
v^3
\end{bmatrix}
\]
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
Plotting Isolines
Points on Patch
Faceting Animation
Faceting
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

// This assumes that indices to the vertices are
// in a 2D array, verts(i,j)

num_tri = 0
for i = 0 to (num_u - 2)
    for j = 0 to (num_v - 2)
        triangles[num_tri++] = (verts[i,j], verts[i+1,j], verts[i+1,j+1])
        triangles[num_tri++] = (verts[i,j], verts[i+1,j+1], verts[i,j+1])
Composite 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 modeled by 32 Bézier Patches with $G^1$ continuity
Beziers Surface: Example

- Increased facet resolution
- Rendered
B-spline Surfaces

\[ x(s,t) = T^T \cdot M_{Bs}^T \cdot G_{Bs_x} \cdot M_{Bs} \cdot S \]

\[ y(s,t) = T^T \cdot M_{Bs}^T \cdot G_{Bs_y} \cdot M_{Bs} \cdot S \]

\[ z(s,t) = T^T \cdot M_{Bs}^T \cdot G_{Bs_z} \cdot M_{Bs} \cdot S \]

- Representation for B-spline patches
- \( C^2 \) continuity across boundaries is automatic with B-splines
Normals to Surfaces

• Normals used for
  – Shading
  – Interference detection in robotics
  – Calculating offsets for numerically controlled machining
Computing the Normals to Surfaces

• For a bicubic surface, first, compute the \( s \) tangent vector

\[
\frac{\delta}{\delta s} Q(s,t) = \frac{\delta}{\delta s} \left( T^T \cdot M^T \cdot G \cdot M \cdot S \right)
\]

\[
= T^T \cdot M^T \cdot G \cdot M \cdot \frac{\delta}{\delta s}(S)
\]

\[
= T^T \cdot M^T \cdot G \cdot M \cdot \begin{bmatrix} 3s^2 & 2s & 1 & 0 \end{bmatrix}
\]
Computing the Normals to Surfaces

• Next, compute the $t$ tangent vector:

$$\frac{\delta}{\delta t} Q(s,t)$$

$$= \frac{\delta}{\delta t} \left( T^T \cdot M^T \cdot G \cdot M \cdot S \right)$$

$$= \frac{\delta}{\delta t} (T^T) \cdot M^T \cdot G \cdot M \cdot S$$

$$= \left[ \begin{array}{cccc} 3t^2 & 2t & 1 & 0 \end{array} \right]^T \cdot M^T \cdot G \cdot M \cdot S$$
Computing the Normals to Surfaces

- Since \( s \) and \( t \) are tangent to the surface, their cross product is the normal vector to the surface!

\[
\frac{\delta}{\delta s} Q(s,t) \times \frac{\delta}{\delta t} Q(s,t) = \begin{bmatrix} y_s z_t - y_t z_s , & z_s x_t - z_t x_s , & x_s y_t - x_t y_s \end{bmatrix}
\]

- \( x_s \) - x component of \( s \) tangent
- \( y_s \) - y component of \( s \) tangent
- \( z_s \) - z component of \( s \) tangent
Surface of Revolution

• Rotate planar curve (*directrix*) around an *axis of revolution* (z axis)
  – Cross-section is a circle
• Biparametric surface
  – *u* of curve
  – *θ* of angle of rotation
• Examples: cylinder, cone, sphere, torus
Surface of Revolution

- Directrix:
  \[- D(u) = (f(u), 0, g(u)) \]

- Surface:
  \[- S(u, \theta) = (f(u)\cos(\theta), f(u)\sin(\theta), g(u)) \]
  \[- 0 \leq u \leq 1, \ 0 \leq \theta \leq 2\pi \]

- Tangents:
  \[- \frac{\partial S}{\partial u} = (f'(u)\cos(\theta), f'(u)\sin(\theta), g'(u)) \]
  \[- \frac{\partial S}{\partial \theta} = (-f(u)\sin(\theta), f(u)\cos(\theta), 0) \]
  \[- N(u, \theta) = \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial \theta} \]
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
    • What if they intersect?
    • Note: patch edges need not be monotonic in x or y

• Idea: Scan convert patch \textit{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
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
Silhouette Determination

\[ \mathbf{N} \cdot \mathbf{L} = 0 \]

Brenner & Hughes, Brown U.

Xu, et al., U. of Minnesota

Kowalski, et al.
Programming Assignment 3

- Process command line arguments
- Read in control points from file
- Double loop through u & v parameters
- For each (u,v) pair compute 3D point on Bezier patch
- Once you’ve computed the 3D points, define the triangles that connect them
- If shading, compute exact normals at each mesh vertex
- Output all data as Open Inventor