CS 430
Computer Graphics

3D Modeling: Surfaces

3D Clipping

Week 7, Lecture 14

David Breen, William Regli and Maxim Peysakhov
Department of Computer Science
Drexel University

Simple Mesh Format (SMF)

- Michael Garland
  http://graphics.cs.uiuc.edu/~garland/
- Triangle data
- Vertex indices begin at 1

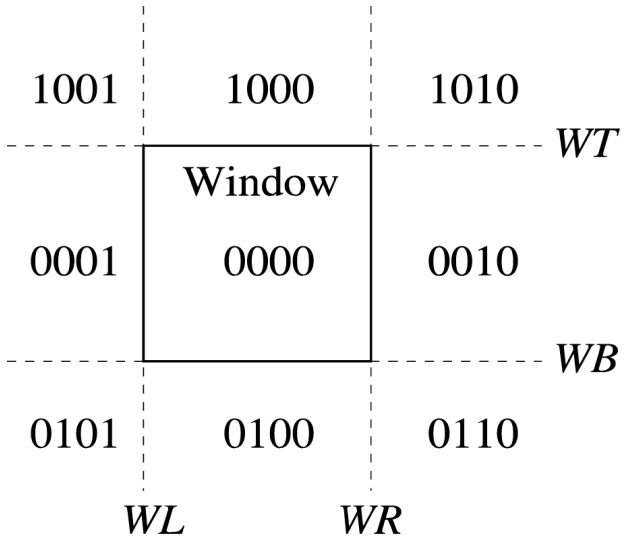
```
#$SMF 1.0
#$vertices 5
#$faces 6
v 2.0 0.0 2.0
v 2.0 0.0 -2.0
v -2.0 0.0 -2.0
v -2.0 0.0 2.0
v 0.0 5.0 0.0
f 1 3 2
f 1 4 3
f 3 5 2
f 2 5 1
f 1 5 4
f 4 5 3
```

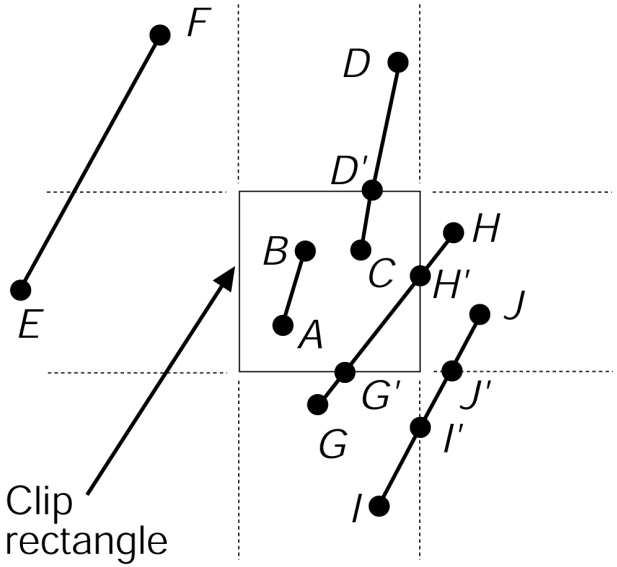
3D Clipping

- Cohen-Sutherland and Cyrus-Beck can be trivially extended to 3D
- We will cover:
  - Cohen-Sutherland for 3D, (parallel projection)
  - Cohen-Sutherland for 3D, (perspective projection)

Recall: Cohen-Sutherland

- Line is completely visible iff both code values of endpoints are *0,* i.e. $C_0 \vee C_1 = 0$
- If line segments are completely outside the window, then $C_0 \wedge C_1 \neq 0$





Pics/Math courtesy of Dave Mount @ UMD-CP

Cohen-Sutherland for 3D, Parallel Projection

- Use 6 bits
- Trivially accept if all end-codes are *0*
- Trivially reject if bit-by-bit *AND* of end-codes is not *0*
- Up to *6* intersections may have to be computed

bit 1	point ABOVE the view volume	y > 1
bit 2	point BELOW the view volume	y < -1
bit 3	point RIGHT OF the view volume	x > 1
bit 4	point LEFT OF the view volume	x < -1
bit 5	point BEHIND the view volume	z < -1
bit 6	point IN FRONT the view volume	z > 0

Cohen-Sutherland for 3D computing intersection points.

- Use parametric representation of the line to compute intersections
- So for *y=1* replace *y* with *1* and solve for *t*
- If $1 \geq t \geq 0$ use it to find *x* and *z*
- Test if *x* and *z* are in valid range
- Repeat for planes *y=-1, x=1, x=-1, z=-1, z=0*

$$x = x_0 + t(x_1 - x_0)$$
$$y = y_0 + t(y_1 - y_0)$$
$$z = z_0 + t(z_1 - z_0)$$

$$t = \frac{(1 - y_0)}{(y_1 - y_0)}$$

Cohen-Sutherland for 3D, Perspective Projection

- Use 6 bits identical to parallel view volume clipping
- Conditions on the codes are different
- Trivially accept/reject lines using same roles
- Intersection points computed differently

bit 1	point ABOVE the view volume	$y > -z$
bit 2	point BELOW the view volume	$y < z$
bit 3	point RIGHT OF the view volume	$x > -z$
bit 4	point LEFT OF the view volume	$x < z$
bit 5	point BEHIND the view volume	$z < -1$
bit 6	point IN FRONT the view volume	$z > z_{min}$

Cohen-Sutherland for 3D computing intersection points.

- Intersections with planes *z=-1, z=z*$_{min}$ is the same.
- Calculating intersections with a sloping plane …
- For plane y=z these two equations are equal
- Repeat for planes *y=-z, x=z, x=-z*

$$x = x_0 + t(x_1 - x_0)$$
$$y = y_0 + t(y_1 - y_0)$$
$$z = z_0 + t(z_1 - z_0)$$

$$t = \frac{(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$$

“3D Clipping” for HWs 4 & 5

- Only do trivial reject test
- For HW4 just do X and Y tests
- ‘AND’ all vertex bit codes for a polygon
- If result != 0, then reject polygon
  - i.e. remove from projection pipeline

More Efficient Alternative?

- Use 3D Cohen-Sutherland to do trivial reject
- Project remaining polygons onto view plane
- Clip polygons in 2D
- Remember that user-defined window is redefined for canonical view volumes!

end clipping

Overview

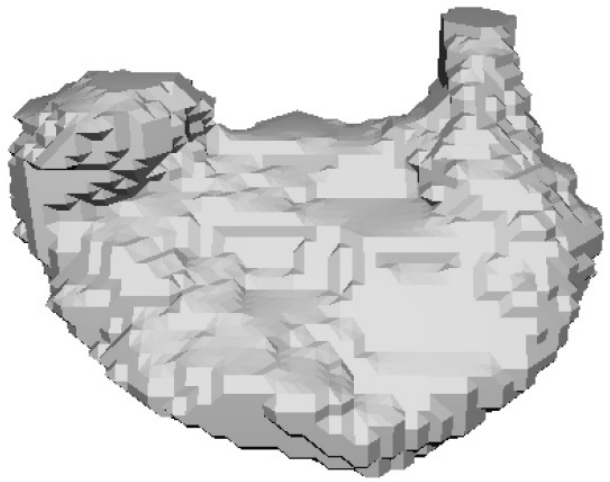
- 3D model representations
- Mesh formats
- Bicubic surfaces
- Bezier surfaces
- Normals to surfaces
- Direct surface rendering

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

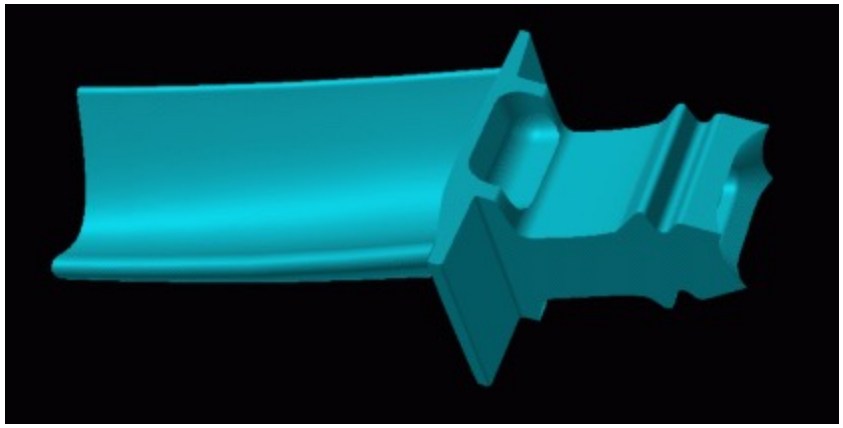


Representing 3D Objects

- 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

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

Positives when Representing 3D Objects

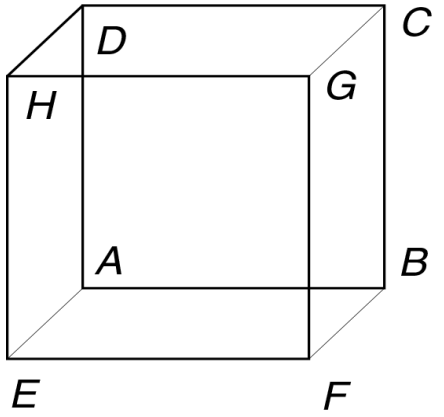
- Exact
  - Precision
    - Simulation, modeling, etc
  - Lots of modeling environments
  - Physical properties
  - High-level control
  - Many applications (tool path generation, motion, etc.)
  - Compact
- Approximate
  - Easy to implement
  - Easy to acquire
    - 3D scanner, CT
  - Easy to render
    - Direct mapping to the graphics pipeline
  - Lots of algorithms

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?

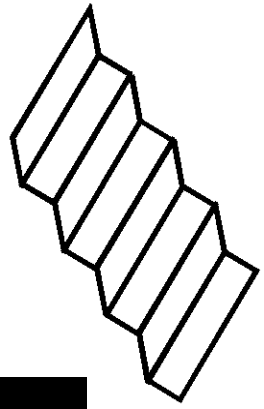


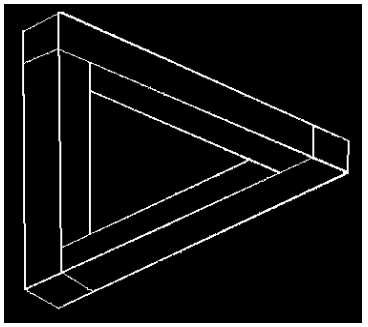
Vertices		Lines
A	(0, 0, 0)	*AB*
B	(1, 0, 0)	*BC*
C	(1, 1, 0)	*CD*
D	(0, 1, 0)	*DA*
E	(0, 0, 1)	*EF*
F	(1, 0, 1)	*FG*
G	(1, 1, 1)	*GH*
H	(0, 1, 1)	*HE*
		AE
		BF
		CG
		DH

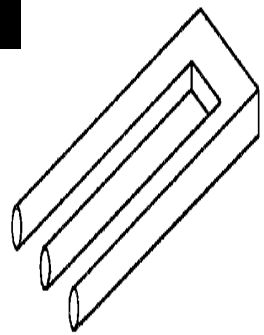
(a)

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!







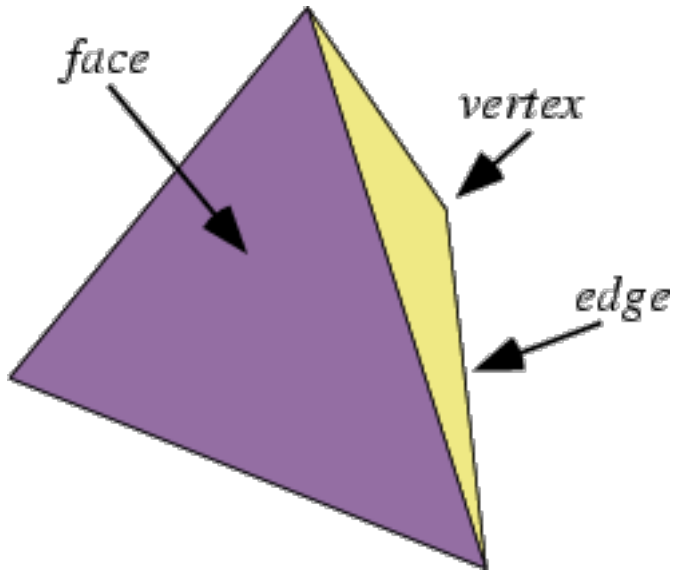
Surface Models

- 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

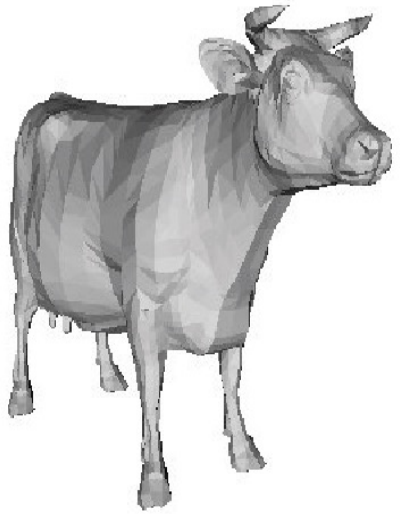
Some common formats

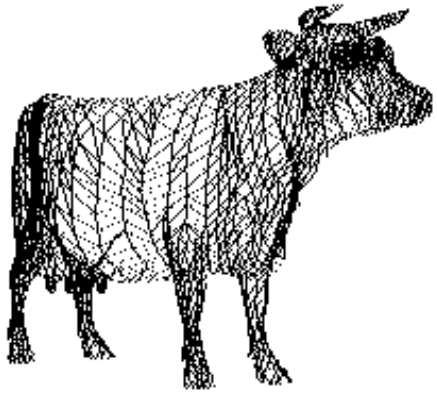
- STL
- SMF
- OpenInventor
- VRML
- X3D



Minimal

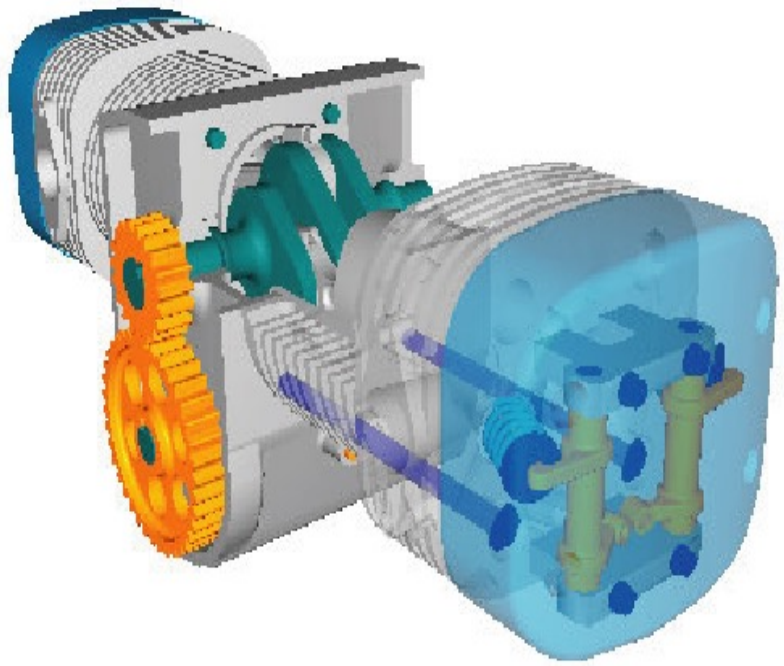
- Vertex + Face
- No colors, normals, or texture
- Primarily used to demonstrate geometry algorithms





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

```
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

```
Separator {
  Coordinate3 {
    point [ -1  1  1,  -1 -1  1,   1 -1  1,   1  1  1,
            -1  1 -1,  -1 -1 -1,   1 -1 -1,   1  1 -1 ]
  }
  Material { diffuseColor [ 1 0 0,  0 1 0,  0 0 1, 1 1 0 ] }# indices 0,1,2,3
  Normal {
    vector [  0.0  0.0  1.0,   1.0  0.0  0.0,    # front and right faces
              0.0  0.0 -1.0,  -1.0  0.0  0.0,    # back and left faces
              0.0  1.0  0.0,   0.0 -1.0  0.0 ]   # top and bottom faces
  }
  NormalBinding { value PER_FACE_INDEXED }
  MaterialBinding { value OVERALL }
  IndexedFaceSet {
    coordIndex [ 0, 1, 2, 3, -1,  3, 2, 6, 7, -1,  # front and right faces
                 7, 6, 5, 4, -1,  4, 5, 1, 0, -1,  # back and left faces
                 0, 3, 7, 4, -1,  1, 5, 6, 2, -1 ] # top and bottom faces
    normalIndex [ 0, 1, 2, 3, 4, 5 ]    # Apply normals to faces, in order
  }
  Translation { translation 3 0 0 }
  MaterialBinding { value PER_VERTEX_INDEXED }
  IndexedFaceSet {
    coordIndex [ 0, 1, 2, 3, -1,  3, 2, 6, 7, -1,  # front and right faces
                 7, 6, 5, 4, -1,  4, 5, 1, 0, -1,  # back and left faces
                 0, 3, 7, 4, -1,  1, 5, 6, 2, -1 ] # top and bottom faces
    materialIndex [ 0, 0, 1, 1, -1,     # red/green front
                    2, 2, 3, 3, -1,     # blue/yellow right
                    0, 0, 1, 1, -1,     # red/green back
                    2, 2, 3, 3, -1,     # blue/yellow left
                    0, 0, 0, 0, -1,     # red top
                    2, 2, 2, 2, -1 ]    # blue bottom
  }
}
```

Virtual Reality Modeling Language (VRML)

- SGML Based
- Scene-Graph
- Full Featured

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

X3D

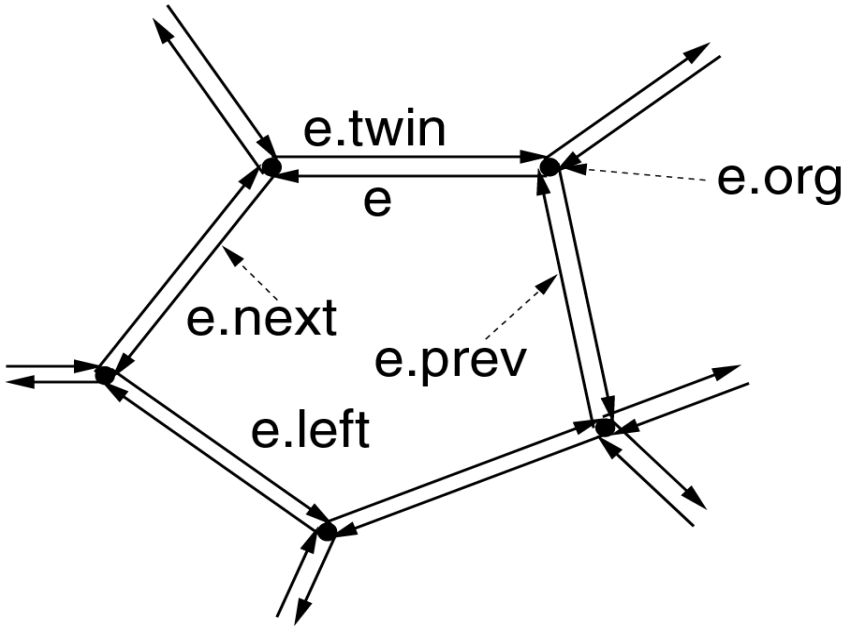
- Open standards file format and run-time architecture to represent and communicate 3D scenes and objects using XML
- Supports
  - 2D/3D graphics, programmable shaders
  - 2D/3D compositing, CAD data, Animation
  - Spatialized audio and video, User interaction
  - Navigation, Scripting, Networking, Simulation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

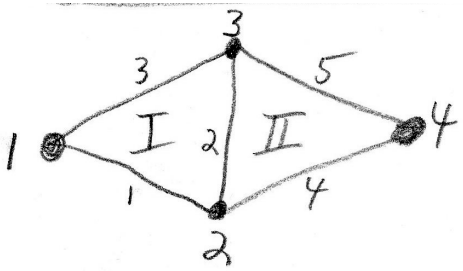
- Winged-Edge Data Structure (Weiler)
- Vertex
  - n edges
- Edge
  - 2 vertices
  - 2 faces
- Face
  - m edges



Pics/Math courtesy of Dave Mount @ UMD-CP

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



VERTICES

V1	X,Y,Z	E3	E1		
V2	X,Y,Z	E1	E2	E4	
V3	X,Y,Z	E2	E3	E5	
V4	X,Y,Z	E4	E5		

EDGES

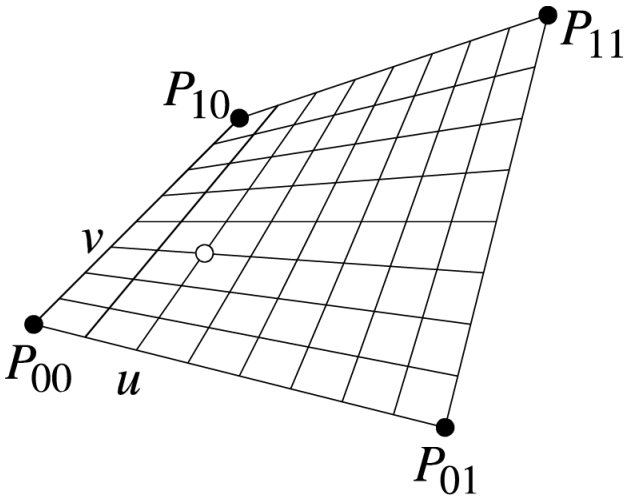
E1	V1	V2	F1		E2	E3
E2	V2	V3	F1	F2	E3	E1
E3	V3	V1	F1		E1	E2
E4	V2	V4	F2		E5	E2
E5	V4	V3	F2		E2	E4

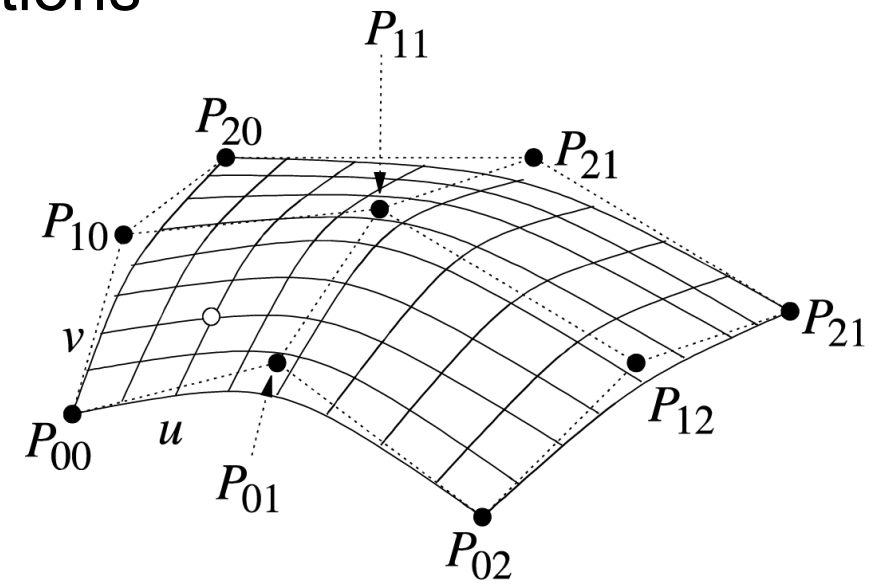
FACES

F1	E1	E2	E3
F2	E2	E4	E5

Biparametric Surfaces

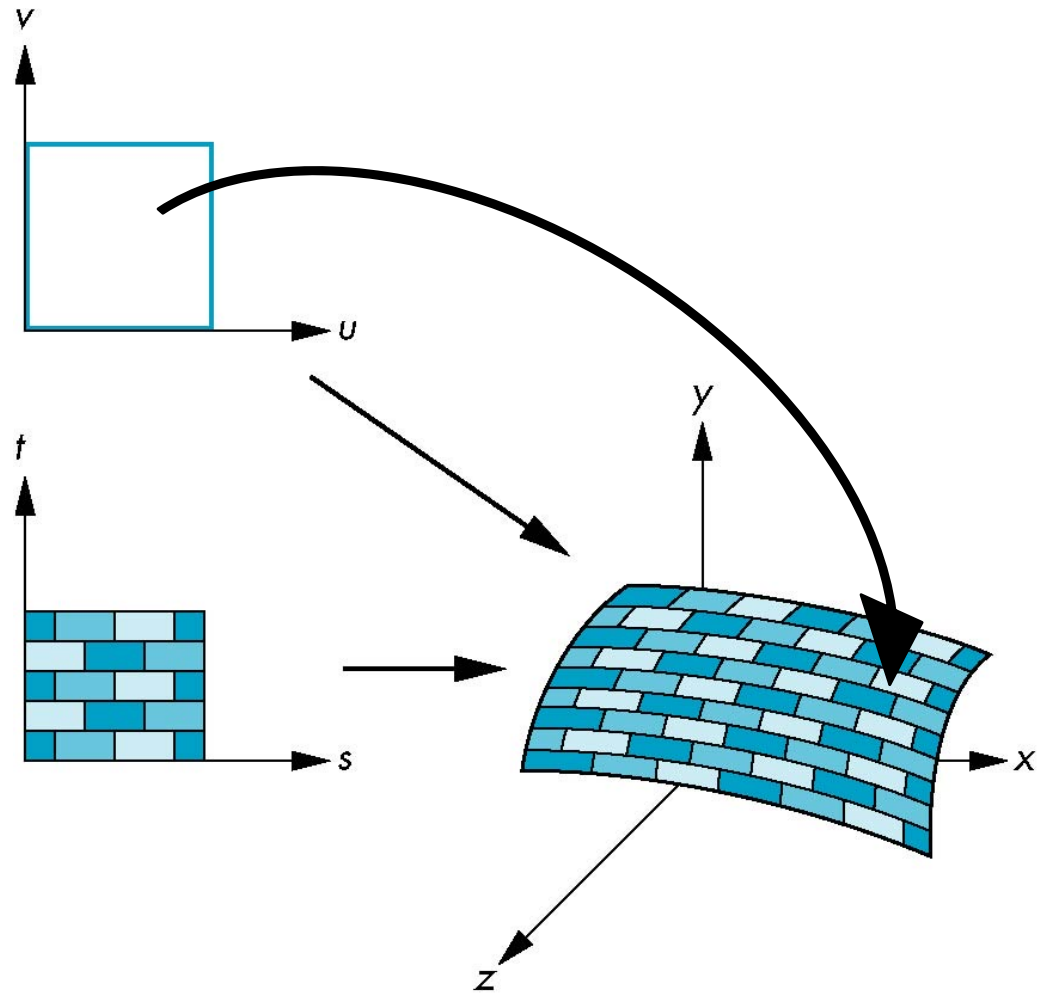
- Biparametric surfaces
  - A generalization of parametric curves
  - 2 parameters: *s, t* (or *u, v*)
  - Two parametric functions





Biparametric Patch

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



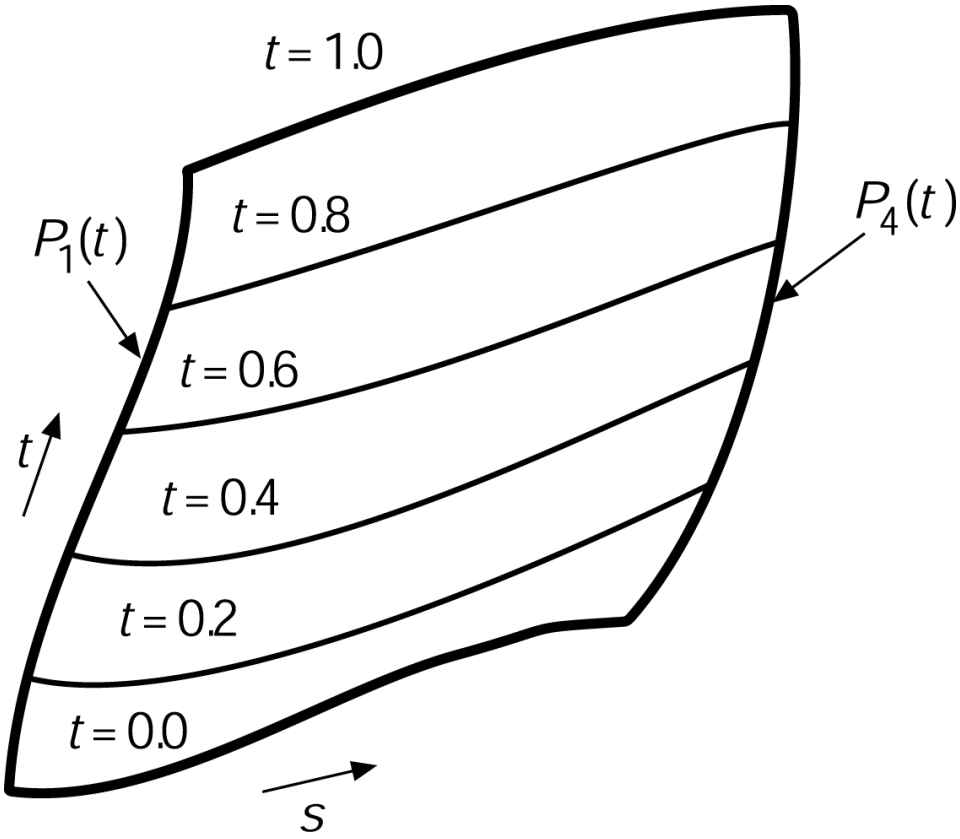
Bicubic Surfaces

- 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:

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

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 $G_i(t) = \mathbf{G_i} \cdot \mathrm{M} \cdot \mathrm{T}$, where

$$\mathbf{G_i} = \begin{bmatrix} g_{i1} & g_{i2} & g_{i3} & g_{i4} \end{bmatrix}$$

- Transposing $G_i(t)$, we get

$$G_i(t) = T^T \cdot M^T \cdot \mathbf{G_i}^{\mathrm{T}}$$

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

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 $Q(s,t) = T^T \cdot M^T \cdot \mathbf{G} \cdot \mathrm{M} \cdot \mathrm{S} \quad 0 \le \mathrm{s,t} \le 1$ gives

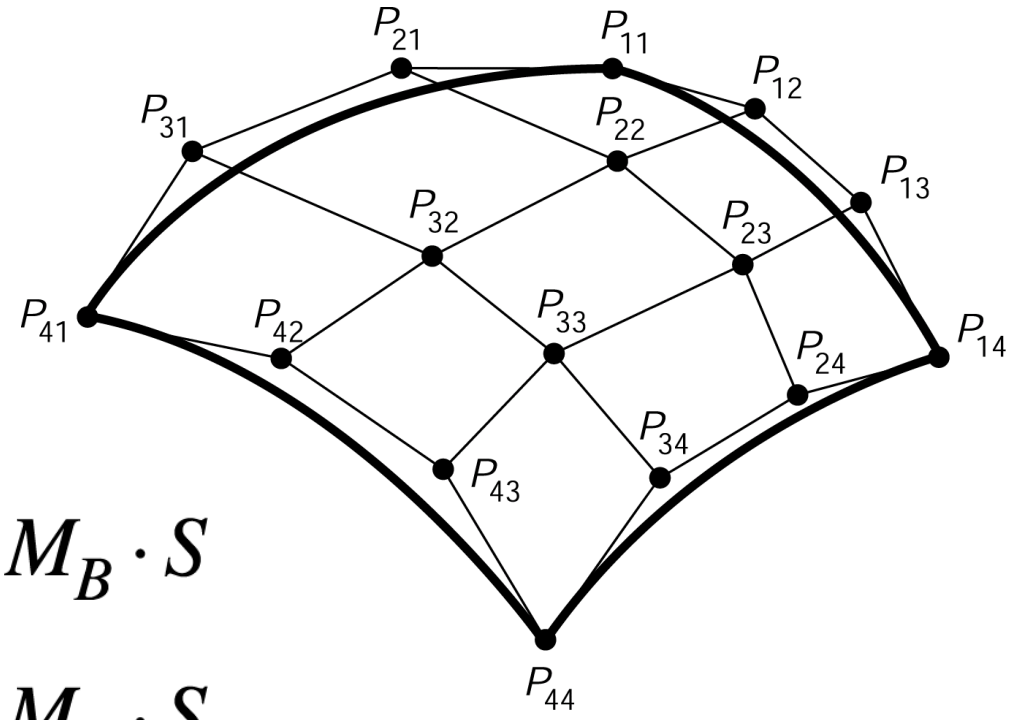
$$x(s,t) = T^T \cdot M^T \cdot G_x \cdot M \cdot S$$

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

$$z(s,t) = T^T \cdot M^T \cdot G_z \cdot M \cdot S$$

Bézier Surfaces

- Bézier Surfaces (similar definition)



$$x(s,t) = T^T \cdot M_B^T \cdot G_{B_x} \cdot M_B \cdot S$$

$$y(s,t) = T^T \cdot M_B^T \cdot G_{B_y} \cdot M_B \cdot S$$

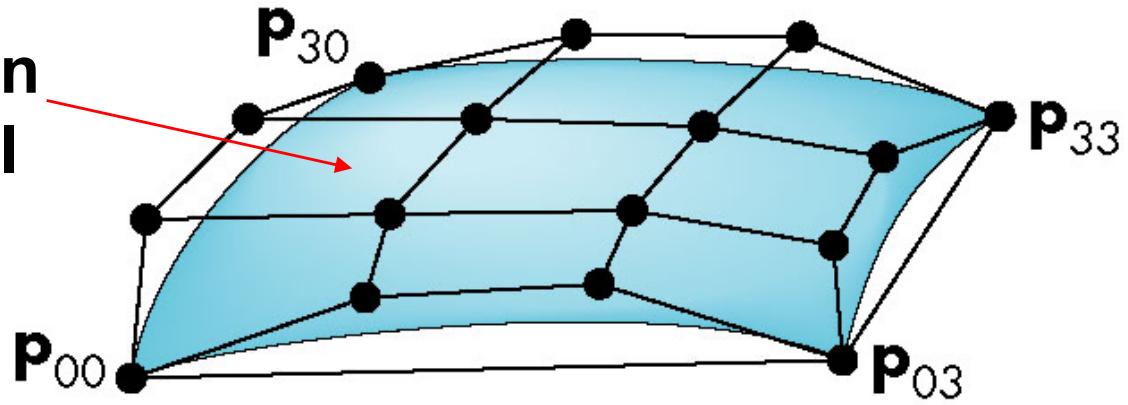
$$z(s,t) = T^T \cdot M_B^T \cdot G_{B_z} \cdot M_B \cdot S$$

Bicubic Bezier Patches

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

$$\vec{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} b_i(u) b_j(v) \vec{p}_{ij} = u^T \mathbf{M}_B \mathbf{P} \mathbf{M}_B^T v$$

Patch lies in convex hull



Bicubic Bézier Patches

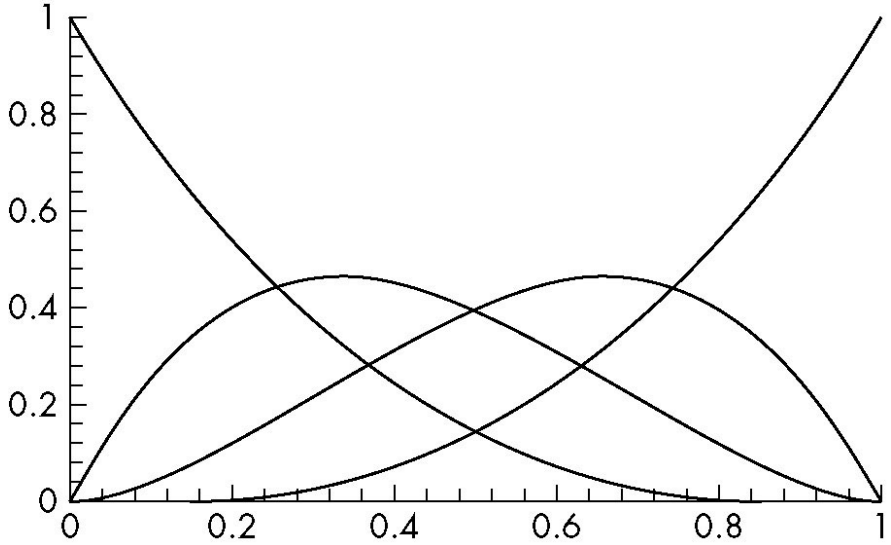
- Expanding the summation

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

$$b_0(u)b_0(v)\vec{p}_{00} +$$
$$b_0(u)b_1(v)\vec{p}_{01} +$$
$$b_0(u)b_2(v)\vec{p}_{02} +$$
$$b_0(u)b_3(v)\vec{p}_{03} +$$
$$b_1(u)b_0(v)\vec{p}_{10} +$$
$$etc.$$

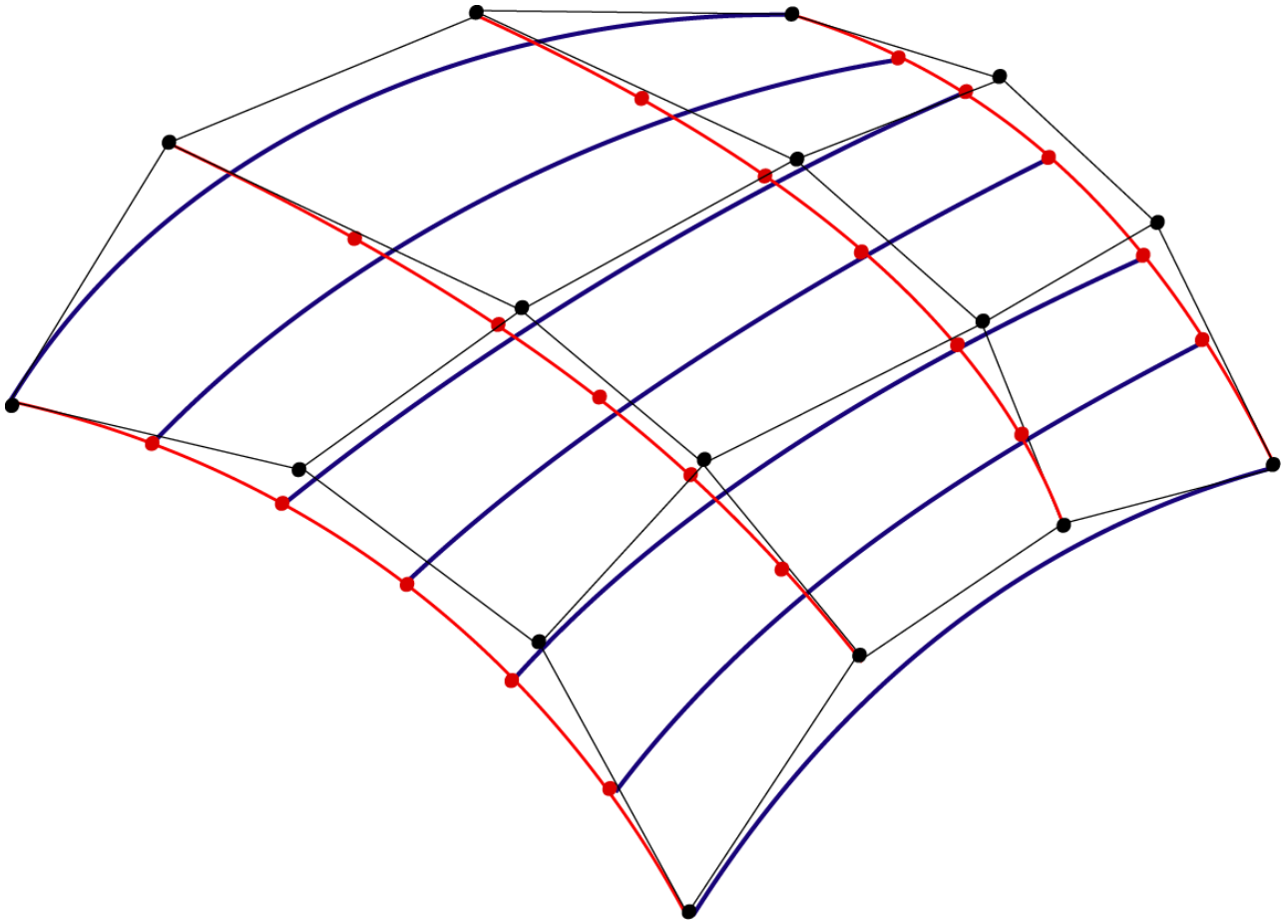
Cubic Bezier Blending Functions

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



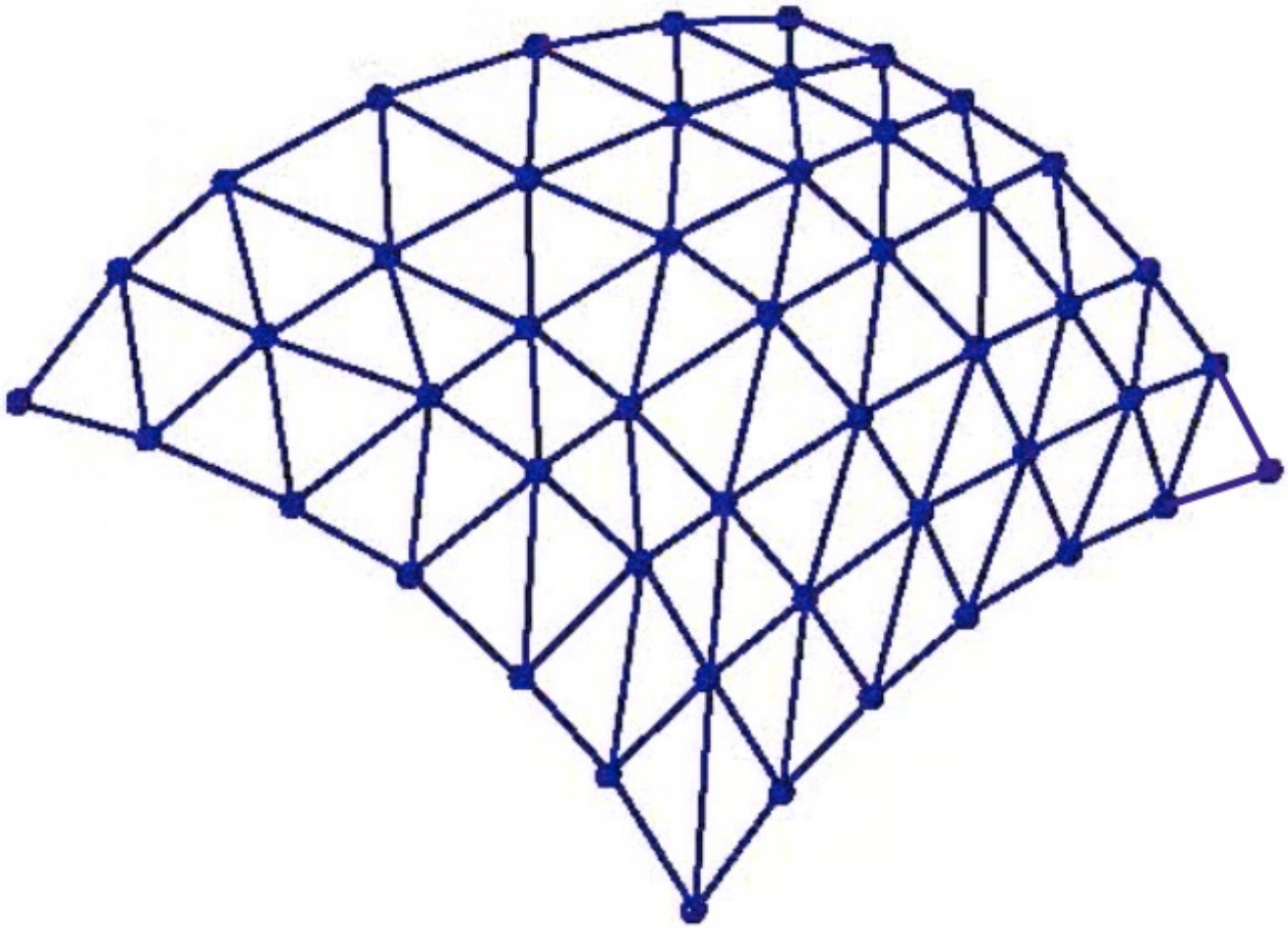
Note that all zeros are at 0 and 1 which forces the functions to be smooth over (0,1)

Plotting Isolines



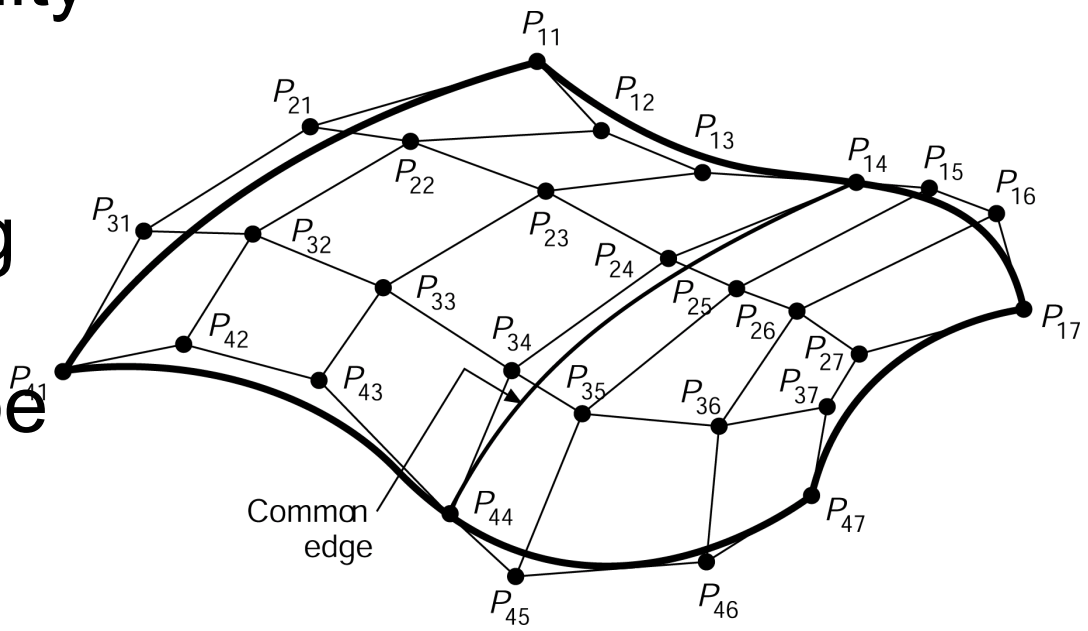
Faceting

Faceting



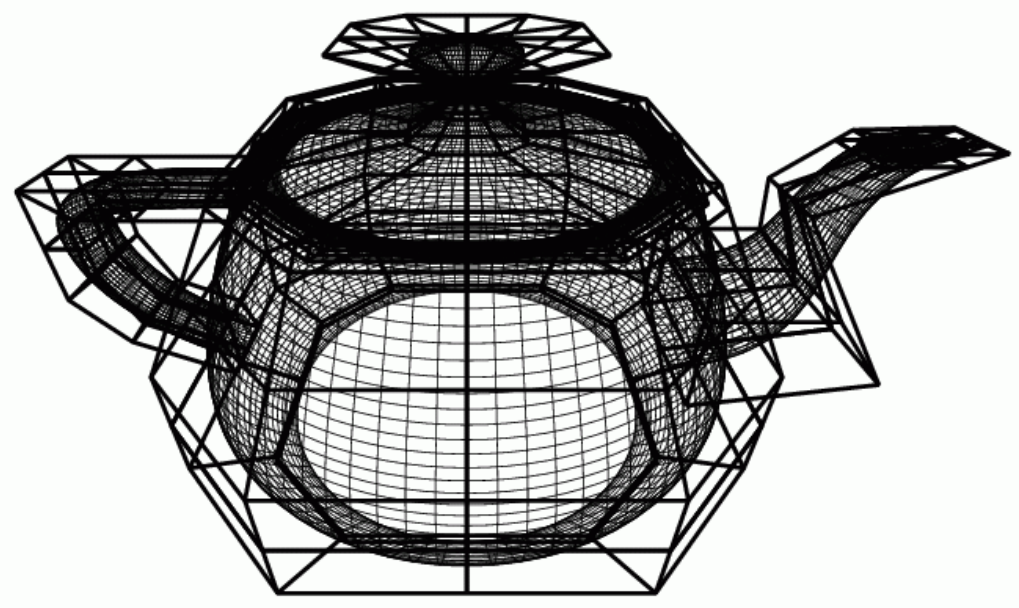
Composite Bézier Surfaces

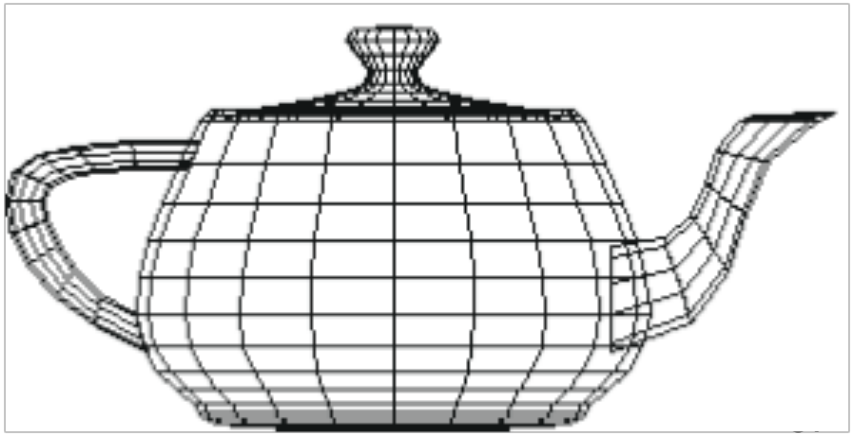
- C^0 and G^0 continuity can be achieved between two patches by setting the 4 boundary control points to be equal
- G^1 continuity achieved when cross-wise CPs are co-linear



Bézier Surfaces: Example

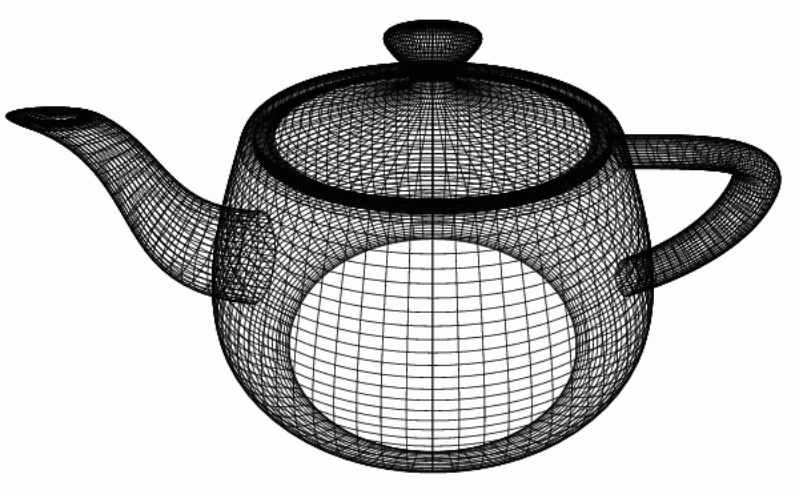
- Utah Teapot modeled by 32 Bézier Patches with G^1 continuity





Bezier Surface: Example

- Increased facet resolution
- Rendered



B-spline Surfaces

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

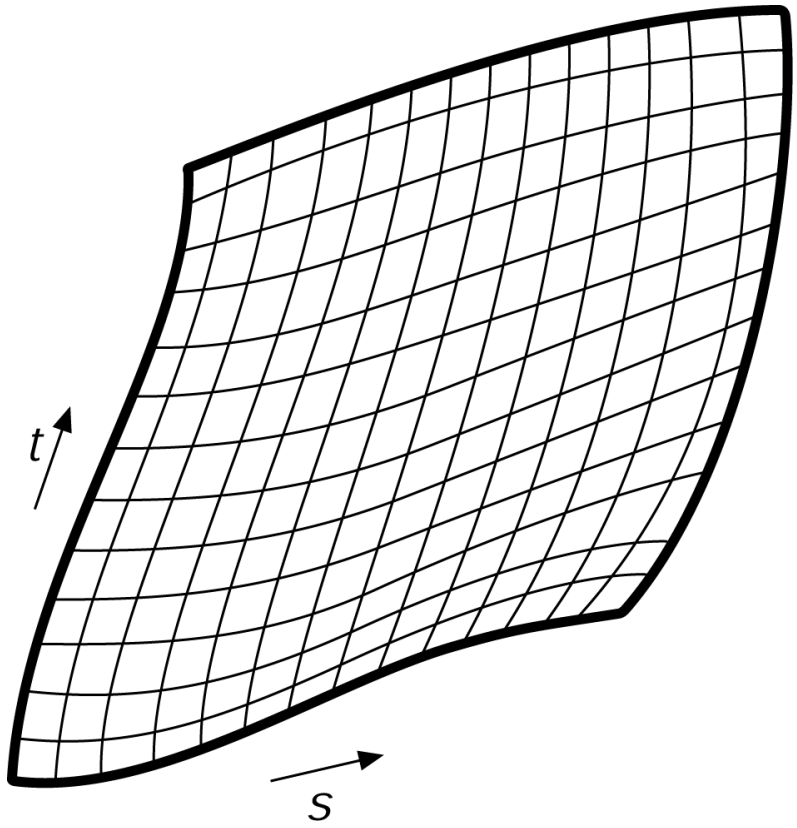
$$y(s,t) = T^T \cdot M_{Bs}^T \cdot G_{Bs_y} \cdot M_{Bs} \cdot S$$

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

- 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

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

$$= \frac{\delta}{\delta s} \left(\; T^T \cdot M^T \cdot \mathbf{G} \cdot \mathrm{M} \cdot \mathrm{S} \; \right)$$

$$= T^T \cdot M^T \cdot \mathbf{G} \cdot \mathrm{M} \cdot \frac{\delta}{\delta \mathrm{s}} (\mathrm{S})$$

$$= \mathrm{T}^{\mathrm{T}} \cdot \mathrm{M}^{\mathrm{T}} \cdot \mathbf{G} \cdot \mathrm{M} \cdot \begin{bmatrix} 3s^2 & 2s & 1 & 0 \end{bmatrix}^{\mathrm{T}}$$

Computing the Normals to Surfaces

- Next, compute the ***t*** tangent vector:

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

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

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

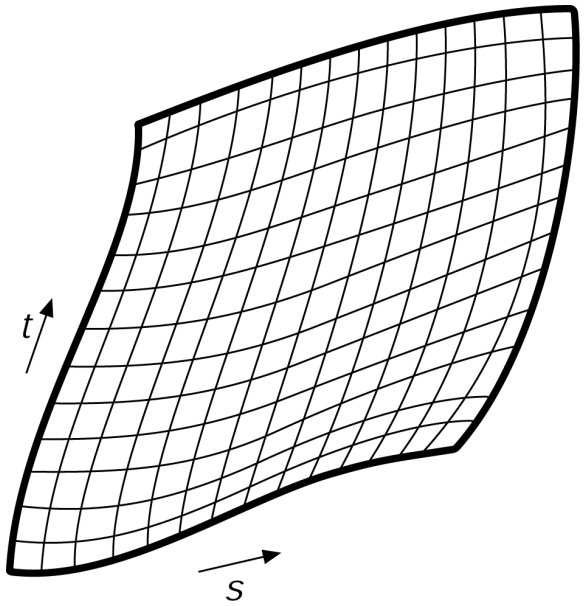
$$= \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix}^{\mathrm{T}} \cdot \mathrm{M}^{\mathrm{T}} \cdot \mathbf{G} \cdot \mathrm{M} \cdot \mathrm{S}$$

Computing the Normals to Surfaces

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

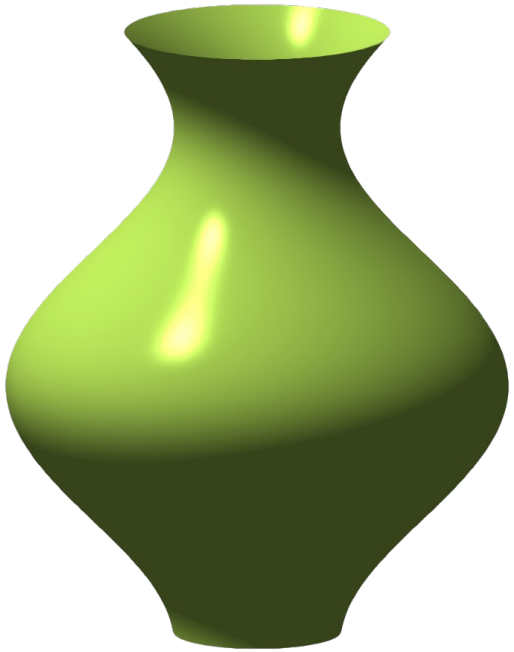
$$\frac{\delta}{\delta s}Q(s,t) \times \frac{\delta}{\delta t}Q(s,t) = \begin{bmatrix} y_s z_t - y_t z_s & z_s x_t - z_t x_s & x_s y_t - x_t y_s \end{bmatrix}$$

- x_s - x component of ***s*** tangent
- y_s - y component of ***s*** tangent
- z_s - z component of ***s*** tangent

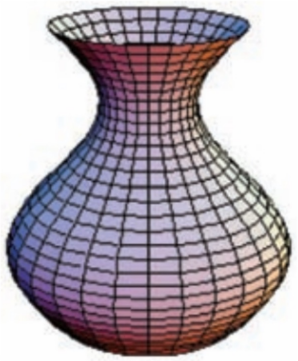


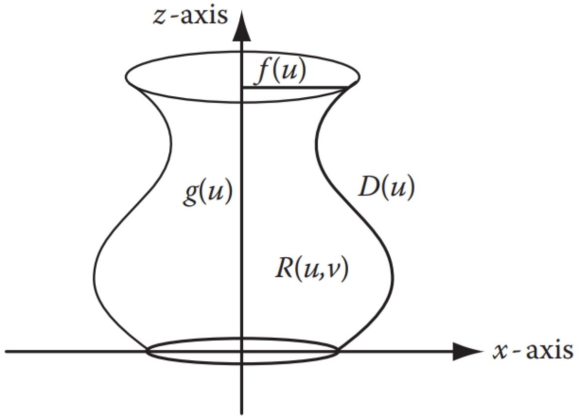
Surface of Revolution

- 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,θ) = (f(u)cos(θ), f(u)sin(θ), g(u))
  - 0 <= u <= 1, 0 <= θ <= 2π
- Tangents:
  - $\partial S/\partial u$ = (f'(u)cos(θ), f'(u)sin(θ), g'(u))
  - $\partial S/\partial \theta$ = (-f(u)sin(θ), f(u)cos(θ), 0)
  - N(u,θ) = $\partial S/\partial u \times \partial S/\partial \theta$

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

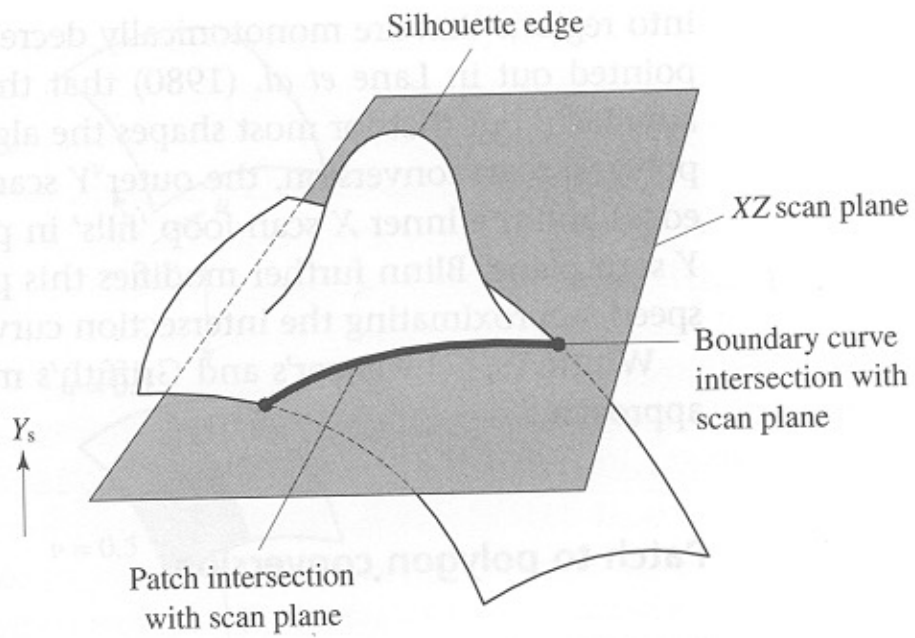
Issues for Direct Rendering

- 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 *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

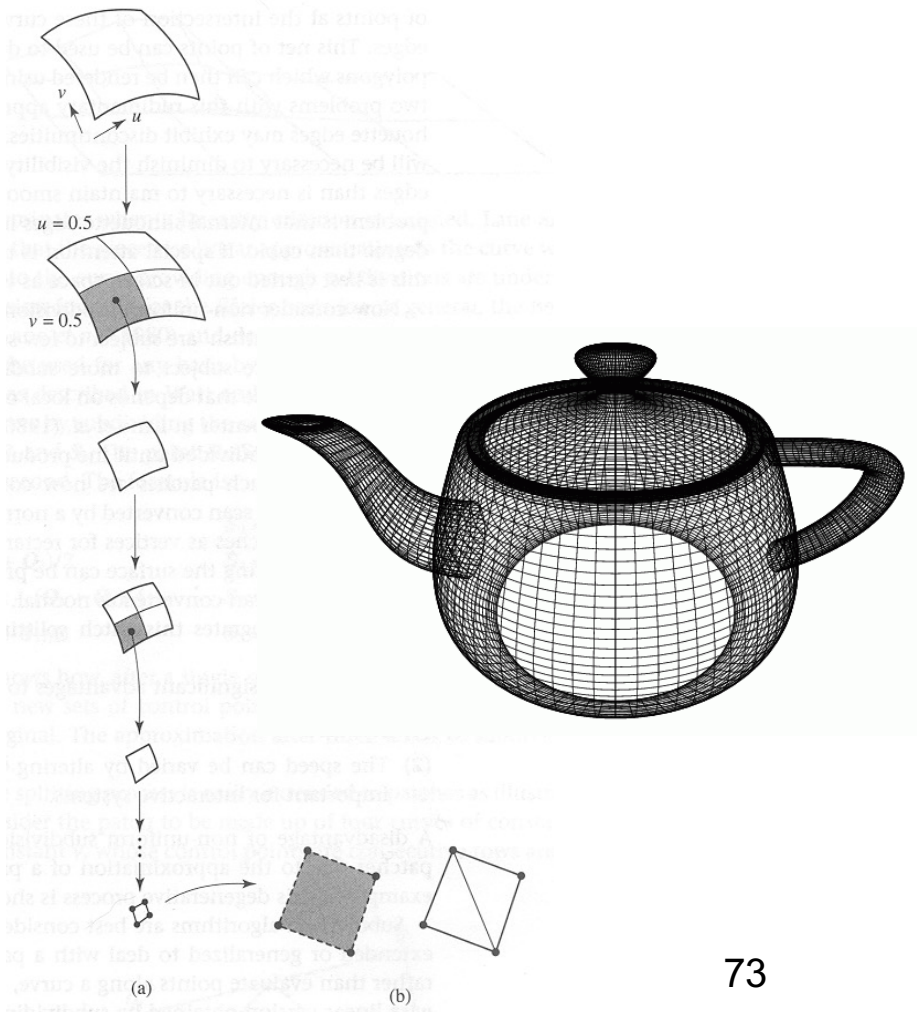
Two methods:

- **Object Space Conversion**
  - Techniques
    - 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

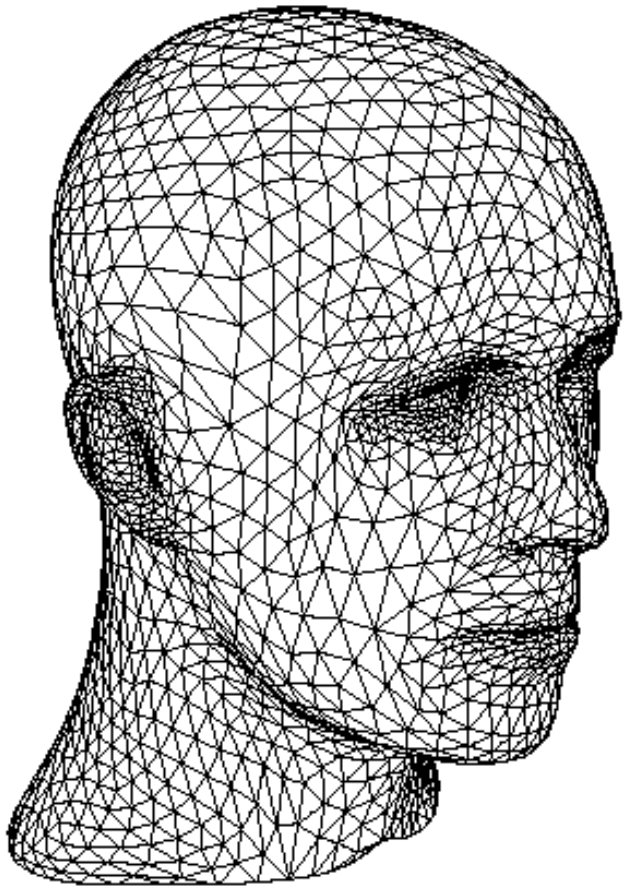


Image Space Conversion

- 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

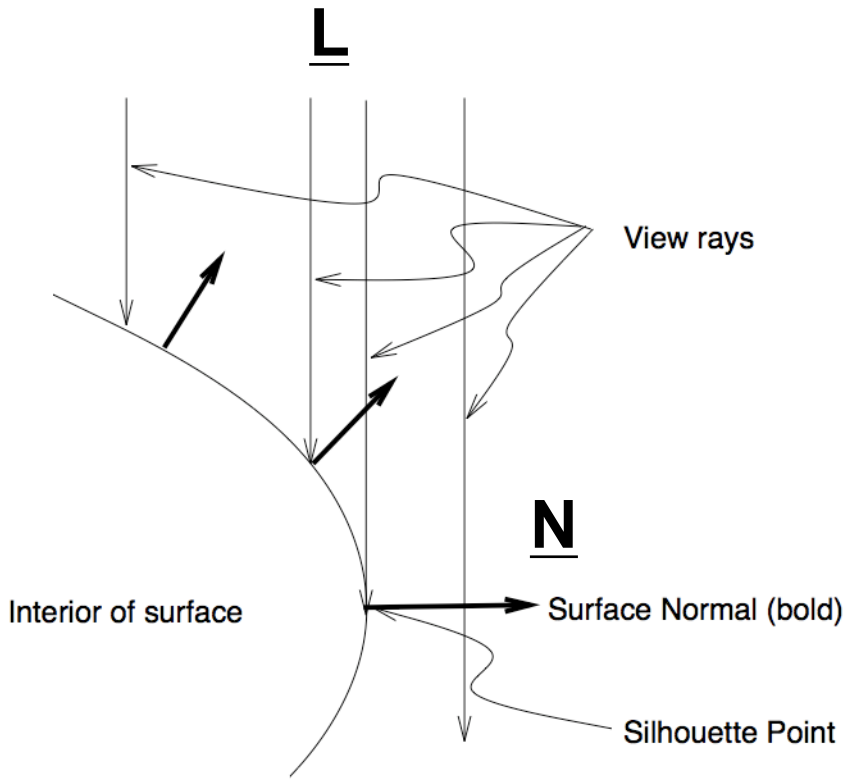
How do I know if I've found a silhouette edge?

- If the viewing ray is tangent to the surface at the point it hits the surface!

$$N \bullet L = 0$$

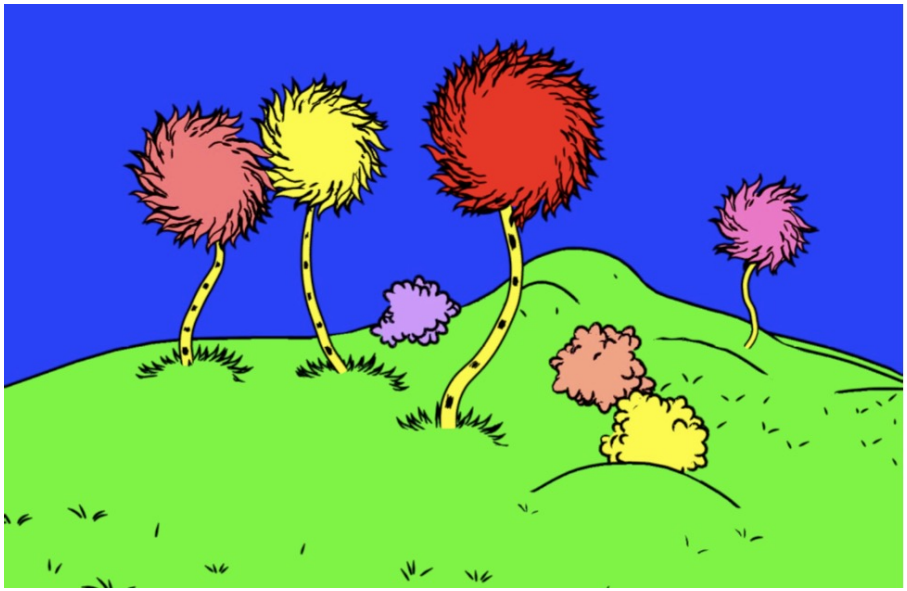
  - Where N is the normal at the point where L, the line of sight, hits the surface

Silhouette Determination



Xu, et al., U. of Minnesota

N•L = 0



Brenner & Hughes, Brown U.

Kowalski, et al.