Intro to Modeling

Modeling in 3D
Polygon sets – can approximate more complex shapes as *discretized surfaces*
**Modeling in 3D**

**Curve surfaces in 3D**

- **Sphere, ellipsoids, etc**
  \[ (x^2 + y^2 + z^2 = r^2) \]

**Modeling in 3D**

**Solid Models in 3D**

- **Sphere, ellipsoids, etc**
  \[ (x^2 + y^2 + z^2 < r^2) \]
Solid Models in 3D

Boolean solids

Polygons

- Multi-sided planar element composed of edges and vertices.
- Vertices (singular vertex) are represented by points
- Edges connect vertices as line segments
Polygons

- **Simple** polygons - no edges cross
- **Non-simple** - some edges cross

Convex Polygon - a polygon that has no included angles larger than 180 degrees

- **Convex Polygon** - a polygon that has no included angles larger than 180 degrees
- **Concave Polygon** - has at least one included angle greater than 180 degrees
Polygons

• Test for convexity - any line segment drawn between two points inside a polygon must remain inside the polygon if the poly is convex

Convex

Concave

Polygons

• A concave polygon can be broken up into two or more convex polygons

OpenGL assumes all polygons are simple and convex
Rasterizing Polygons

• Assume that all polygons lie on the image plane (2D), then rasterization is to fill in pixels lying within a closed polygon and to do so efficiently.

• To do this, algorithm needs to determine if a pixel is inside or outside a polygon.

• How do we do this?

Scan Line Approaches:

- Inside test: Point P is inside a polygon iff a scanline intersects polygon edges an odd number of times moving from P in either direction.
Rasterizing Polygons

- **Inside test** special cases:
  - Horizontal edges can be ignored
  - Use **Min-max test** when scanline passes thru vertex

Min-max:

- count twice if slope changes sign
- count once otherwise

Rasterizing Polygons

- **Inside test**: To fill polygons, one scanline at a time
determine which pixels are inside and set "on"
Rasterizing Polygons

- Faster is to fill polygons, one scanline at a time
determine which pixels intersect and sort by edge

- **Scanline fill algorithm**
  For each scanline:
  - Find intersections
  - Sort intersections
  - Fill in pixels between
    pairs of intersections

But, intersection calculations are expensive

- Faster still, use **edge coherence** - Many edges that
  intersect scanline $s$ intersect scanline $s+1$

  - Compute scanline intersection:
    - $y = mx + b$ and $y_s = s$, then
    - $s = mx_s + b$ or $x_s = (s-b)/m$

  - Compute difference in next scanline for intersection
    - $y_{s+1} = s+1$,
    - $s+1 = mx_{s+1} + b$ or $x_{s+1} = (s+1 - b)/m$

    and, $x_{s+1} = x_s + 1/m$ <= USE THIS TO UPDATE

Edge tables allow us to define in efficient, easy manner
Modeling in 3D

Defining 3D Polygons

3D Polygons “live” in 3D
Defining 3D Polygons

- 3D Polygons are truncated planes in 3-space

\[ \begin{align*}
E1 & : (x1,y1,z1) \\
E2 & : (x2,y2,z2) \\
E3 & : (x3,y3,z3)
\end{align*} \]

- Express given the vertex and edge coordinates

Start with the generic form of a plane
\[ Ax + By + Cz + D = 0 \]

---

Defining 3D Polygons

- Vertices and edges are ordered, CW or CCW, usually counter clockwise
- Therefore, edges are directed line segments and may be treated as vectors
Defining 3D Polygons

- Every plane has a vector $n$ normal (perpendicular, orthogonal) to it.
- From two vectors that line on the plane, we can use the cross product to find the normal, $n = u \times v$.

![Diagram of plane with vectors and normal vector]

Vector Operations

**Cross Product**

- Apply the right hand rule:
  - Curl fingers from $u$ to $v$;
  - Thumb points to $u \times v$.

Cross product results in a vector.
Vector Operations

**Dot product** can be used to define angles:
For example, assume \( w \) & \( v \) are unit length

\[
\text{projection } w \cdot v
\]

\[
\vec{v} \quad \text{and} \quad \theta = \cos^{-1}( w \cdot v )
\]

Also generalizes for non-normal vectors (see text)

---

Defining 3D Polygons

- In our plane equation: \( Ax + By + Cz + D = 0 \),
  \([A,B,C]^T = \text{the normal of the plane}\)
- From the edges find two vectors
- Use the cross product to find
  the normal, giving us A, B, C

Then, to find D, plug in any vertex into the equation and solve for D
• Once we know our plane equation:
  \[ Ax + By + Cz + D = 0, \]
  we still need to manage the truncation which leads to the polygon itself

Functionally, we will need to do this to know if a point lies in a polygon or not, for example

---

• To do this, we can *project* the 3d polygon into 2d and see if the point is in the 2d using the inside test
Defining 3D Polygons

- Project 3d to 2d based on largest of A, B, C

This example:
Z (or C) is principal component of N, normal so project on to xy-plane

Z-buffer

- For 3D with multiple polygons, must deal with visibility

Project vertices into image plane and with projected vertex include depth in additional depth- or Z-buffer
Z-buffer

Scan convert each projected polygon:

For each polygon in scene
  project vertices
  for each pixel inside poly
    calculate $z$
    if $z < \text{closest}$
      draw into frame buffer
      update z buffer

Curves
and Surfaces
Curves

- Curves are one dimensional entities where the function is nonlinear

Curves

Representations for curves

- Parametric representation
  \[ x = x(u), \ y = y(u) \text{ or simply } p(u) \text{ where } p = \begin{bmatrix} x \\ y \end{bmatrix} \]
  - More robust and general than other forms
  - Gives better control over curves and surfaces

Notation:
\[ p(u) = c_0 + c_1u + c_2u^2 \]
where each \( c \) is a vector, \( c_i = \begin{bmatrix} c_{ix} \\ c_{iy} \end{bmatrix} \)
Representations for curves

• Example, quadric parametric curve:
  \[ p(u) = c_0 + c_1u + c_2u^2 \]

  Like curve:
  \[ x = 3u^2 \]
  \[ y = 2u + 3 \]
  for \( u = [-1,1] \)

  \[
  \begin{bmatrix}
  c_0 \\
  c_1 \\
  c_2
  \end{bmatrix}
  \begin{bmatrix}
  0 \\
  3 \\
  2
  \end{bmatrix}
  \begin{bmatrix}
  0 \\
  2 \\
  0
  \end{bmatrix}
  \begin{bmatrix}
  3 \\
  0
  \end{bmatrix}
  \]

  Note, more coefficients than a quadratic

• Parametric curve, \( p(u) \) can as easily represent a curve in 3D (x,y,z)

  Simply:
  \[ x = f_x(u) \]
  \[ y = f_y(u) \]
  \[ z = f_z(u) \]

  Quadric coefs become:

  \[
  \begin{bmatrix}
  c_{ox} \\
  c_{oy} \\
  c_{oz}
  \end{bmatrix}
  \begin{bmatrix}
  c_{ix} \\
  c_{iy} \\
  c_{iz}
  \end{bmatrix}
  \begin{bmatrix}
  c_{2x} \\
  c_{2y} \\
  c_{2z}
  \end{bmatrix}
  \]
Parametric Cubic (pc) curves

• Extension of quadrics to cubics
• Represented as:
  \[ p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3 \]
  
• Also called Hermite curves after the 17th century mathematician

Parametric Cubic (pc) curves

• Algebraic form:
  \[ p(u) = c_0 + c_1 u + c_2 u^2 + c_3 u^3 \]
  (Note, unless otherwise specified \( u \) goes from 0,1)

• Not very intuitive, 12 values of \( c \)
• Instead want a better specify the curve
• More intuitive: control by start point \( p(0) \) and ending point \( p(1) \) and their derivatives
• From: \[ p(u) = c_0 + c_1u + c_2u^2 + c_3u^3 \]
• Build a geometric form in order to specify a curve by their end points and tangents

![Diagram showing parametric cubic curves with control points and derivatives]

This leads to the Geometric Form:

\[ p(u) = F_1(u)p(0) + F_2(u)p(1) + F_3(u)p'(0) + F_4(u)p'(1) \]

where:
\[
\begin{align*}
F_1(u) &= 2u^3 - 3u^2 + 1 \\
F_2(u) &= -2u^3 + 3u^2 \\
F_3(u) &= u^3 - 2u^2 + u \\
F_4(u) &= u^3 - u^2
\end{align*}
\]
Hermite curves basis

- F curves:

Joining Multiple Segments

use $p = [p_0 \ p_1 \ p_2 \ p_3]^T$

use $p = [p_3 \ p_4 \ p_5 \ p_6]^T$

Get continuity at join points but not continuity of derivatives

$C_0$, $C_1$, $C_2$ continuity
• Family of curves developed in the 1970’s by Bezier, a engineer for Renault, car manufacturer

• Bezier’s curves are only guaranteed to pass through the end points, but other control points controlled the derivative at the end points

• Specifically, the tangent was controlled by the next control point in, the 2nd derivative by the second control point in, and the nth derivative by the nth and so on...
Beziers Curves

- The general form of the Bezier curve is:

\[ p(u) = \sum_{i=0}^{n} p_i f_i(u) \quad 0 < u < 1 \]

- Vertices \( p \) control the curve and blending functions, \( f_i(u) \), that satisfy the "derivative" condition.

- Bernstein polynomials were a family of functions that were chosen by Bezier to satisfy his needs, these are not the only functions that could be used though.

Convex Hull Property

- The properties of the Bernstein polynomials ensure that all Bezier curves lie in the convex hull of their control points.

- Hence, even though we do not reach all the input data, we cannot be too far away.
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)

Beziers Curves

• \[ p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2 \]
Beziers Curves

Beziers curves have intuitive control, are nicely formed, convex hull of all points
• One problem with the curves we have looked at is that changing a single control point affects the whole curve (this is called global propagation.)
• Also, depends on # of control pts
• B-Splines offer an alternative, to only affect the local region if a single control point is modified (i.e. local propagation)

B Splines

• B-splines are also called Basis Splines

• They have a form similar to Bezier, B-splines are defined as:

$$p(u) = \sum_{i=0}^{n} p_i f_{i,k}(u) \quad 0 < u < n + 2 - k$$

where n + 1 is the number of control points and k controls the degree of the blending (or basis) functions
B Splines

• B-splines' blending functions are defined recursively:

\[ f_{i,1}(u) = \begin{cases} 1 & \text{if } t_i < u < t_{i+1} \\ = 0 & \text{otherwise} \end{cases} \]

and

\[ f_{i,m}(u) = \frac{(u - t_i) \cdot f_{i,m-1}(u)}{t_{i+m-1} - t_i} - \frac{(u - t_{i+m}) \cdot f_{i+1,m-1}(u)}{t_{i+m} - t_{i+1}} \]

\(m\) goes from 2 to \(k\)

\(t_i\)'s are knot points relating \(u\) to control points, \(p_i\)

B Splines

• Knot points, \(t_i\)'s, follow along like this:

\[ t_i = \begin{cases} 0 & \text{if } i < k \\ i - k + 1 & \text{if } k \leq i \leq n \\ n - k + 2 & \text{if } i > n \end{cases} \]
• For 6 control pts & $k = 1$, 
  $n= 5$ and $0 < u < 6$

We get the degenerate case:

- $p(u) = p_0$ for $0 < u < 1$
- $p(u) = p_1$ for $1 < u < 2$
- $p(u) = p_2$ for $2 < u < 3$
- $p(u) = p_3$ for $3 < u < 4$
- $p(u) = p_4$ for $4 < u < 5$
- $p(u) = p_5$ for $5 < u < 6$
• For 6 control pts & k = 2,
  \( n = 5 \) and \( 0 < u < 5 \)

We get a linear average of neighbors:

\[
p(u) = (1 - u)p_0 + u p_1 \quad 0 < u < 1
\]
\[
p(u) = (2 - u)p_1 + (u - 1)p_2 \quad 1 < u < 2
\]
\[
p(u) = (3 - u)p_2 + (u - 2)p_3 \quad 2 < u < 3
\]
\[
p(u) = (4 - u)p_3 + (u - 3)p_4 \quad 3 < u < 4
\]
\[
p(u) = (5 - u)p_4 + (u - 4)p_5 \quad 4 < u < 5
\]
• For 6 control pts & $k = 3$, $n=5$ and $0 < u < 4$

We get:

for $0 < u < 1$

$$p_1(u) = (1 - u)^2p_0 + .5u(4 - 3u) p_1 + .5 u^2p_2$$

for $1 < u < 2$

$$p_2(u) = .5(2 - u)^2p_1 + .5(-2u^2 + 6u - 3) p_2 + .5(u - 1)^2p_3$$

for $2 < u < 3$

$$p_3(u) = .5(3 - u)^2p_2 + .5(-2u^2 + 10u - 11) p_3 + .5(u - 2)^2p_4$$

for $3 < u < 4$

$$p_4(u) = .5(4 - u)^2p_3 + .5(-3u^2 + 20u - 32) p_4 + (u - 3)^2p_5$$

B Splines

• Note, influence grows with degree

B Splines

• Note, influence grows with degree
B Splines

• Thus, local influence of control points on curve as:

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

Cubic B-spline
Rendering Curves

- How do we draw the curve, given $p(u)$?

```c
void evaluateCurve(u, pixelX, pixelY)
```

Rendering Curves

- Polyline approximation
Curves vs. Surfaces

• Curves are one dimensional entities where the function is nonlinear
• Surfaces are formed from two-dimensional functions
  - *Linear functions give planes and polygons*

Curves

Surfaces

Curves vs. Surfaces

• Parametric curve, \( p(u) \) can as easily represent a curve in 3D \((x,y,z)\)

\[
\begin{align*}
x &= f_x(u) \\
y &= f_y(u) \\
z &= f_z(u)
\end{align*}
\]
Curves vs. Surfaces

• Parametric surface is very similar:
  \( p(u,v) \)

Simply:
  \[
  x = f(u, v) \\
  y = f(u, v) \\
  z = f(u, v)
  \]

(Just, more difficult to comprehend in a 2D representation, we must go behind the image plane)

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
**Bezier Patches**

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

Patch lies in convex hull

**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 region
Splitting a Cubic Bezier

\( p_0, p_1, p_2, p_3 \) determine a cubic Bezier polynomial and its convex hull.

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

\[ l(u) \quad \text{and} \quad r(u) \]

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.

\[ l(u) \quad \text{and} \quad r(u) \]
Efficient Form

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

Requires only shifts and adds!

Surfaces

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

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

16 final points for 1 of 4 patches created

Normals

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

$$\frac{\partial \mathbf{p}(u,v)}{\partial u} = \begin{bmatrix} \frac{\partial x(u,v)}{\partial u} \\ \frac{\partial y(u,v)}{\partial u} \\ \frac{\partial z(u,v)}{\partial u} \end{bmatrix} \quad \frac{\partial \mathbf{p}(u,v)}{\partial v} = \begin{bmatrix} \frac{\partial x(u,v)}{\partial v} \\ \frac{\partial y(u,v)}{\partial v} \\ \frac{\partial z(u,v)}{\partial v} \end{bmatrix}$$

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