3-D Graphics

- Polygon Mesh Models
- Bicubic Surface Patches
- 3D Geometric Transformations
Polygonal Models

- Object surfaces approximated by a mesh of planar polygons

Scene -->
Objects -->
Subobjects -->
Polygons -->
Vertices (points)

3-D Modeling with Polygons

- Polygon Mesh
  - Store the polygon faces:
  - Array of vertex lists
  - One list for each polygon

Data structures

- Polygons represent/approximate object surfaces
- In either case we must store 3-D world coordinates of each vertex
  - Use an array of 3-D points:
    • struct point3d {float x; float y; float z};
      // a single 3-D point
    • struct point3d w_pts[ ];
      // w_pts is the 3-D points array

Storing Polygons in a Polygon Mesh Model

- Object: Can be represented as an array of polygons
- Each polygon consists of:
  - (a) the number of vertices in the polygon
  - (b) a list of indices into the 3-D points array
    • (An index gives the position of a vertex in the 3-D points array)

Example--A Pyramid

- Following pyramid has 5 vertices, 8 edges and 5 polygon faces

```
struct polygon {int n; int *inds};
// n: The number of vertices
// inds: A list of indices into
// the points array,
// Specifies which vertices form
// the polygon

struct polygon object[ ];
// The object being modeled
// An array of polygons
```
**Vertex Coordinates**

<table>
<thead>
<tr>
<th>vertex</th>
<th>xw</th>
<th>yw</th>
<th>zw</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>150</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>150</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>75</td>
<td>75</td>
<td>150</td>
</tr>
</tbody>
</table>

**The Pyramid’s Points Array**

```c
struct point3d w_pts[5];
// Pyramid vertices in world coords.
int b=150, h=75;    // Dimensions of pyramid

// Set up world coordinate points array
w_pts[0].x=w_pts[0].y=w_pts[0].z=0;
w_pts[1].x=b; w_pts[1].y=w_pts[1].z=0;
w_pts[2].x=b; w_pts[2].y=w_pts[2].z=0;
w_pts[3].x=w_pts[3].z=0; w_pts[3].y=b;
w_pts[4].x=w_pts[4].y=b; w_pts[4].z=h;
```

**Polygons Array (Mesh)**

<table>
<thead>
<tr>
<th>polygon</th>
<th># vertices</th>
<th>vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0,1,4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1,2,4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2,3,4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0,4,3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0,3,2,1</td>
</tr>
</tbody>
</table>

**Polygon array could be generated by:**

```c
struct polygon *object;
// Allocate Space:
for (i=0;i<=4;i++)
{
    object[i].n=3; object[i].inds = (int *) calloc(3,sizeof(int));
}
object[4].n=4; object[4].inds = (int *) calloc(4,sizeof(int));

// Define the polygons in the object
// define the side triangles
object[0].inds[0]=0; object[0].inds[1]=1;  object[0].inds[2]=4;
// define the square base
object[4].inds[3]=1;
```

**More Complex 3-D Objects**

- Approximate surfaces with polygons
- Often points, edges, and/or polygons arrays can be generated procedurally
Example 1: A Cone
- Approximate with n triangular sides
- $n+1$ vertices (apex + n in the base)
- And a Base polygon with n sides (example, n=12)

Cone Points Array
- Base points:
  - $x = R \cos (i \cdot \theta)$
  - $y = R \sin (i \cdot \theta)$
  - $z = 0$
- Apex point:
  - $x = y = 0$
  - $z = h$ (height of cone)

Cone Polygons Array
- poly[0] = {12, {12,11,10,9,8,7,6,5,4,3,2,1}}
- poly[1] = {3, {1,2,0}}
- poly[2] = {3, {2,3,0}}
- poly[3] = {3, {3,4,0}}
- poly[4] = {3, {4,5,0}}
- ...
- poly[12] = {3, {12,1,0}}
- The triangles can be generated in a loop

Example 2: A Sphere
- Divide with n lines of latitude and m lines of longitude
- Gives triangles and quadrilaterals
- Latitude/Longitude intersection points used as approximating-polygon vertices
- Number of vertices = $m \cdot n + 2$
- Number of polygons = $(n+1) \cdot m$
- Example n=3, m=8

Example: n=3, m=8
8 * 3 + 2 = 26 vertices
Can get x, y, z from spherical coordinates
Loop j: 0 -> n-1 (latitudes), i: 0 -> m-1 (longitudes)
- $x = R \sin(i \cdot \theta) \cdot \cos(j \cdot \theta)$
- $y = R \sin(i \cdot \theta) \cdot \sin(j \cdot \theta)$
- $z = R \cos(j \cdot \theta)$

(3+1) * 8 = 32 polygons
Number them in a consistent way
poly[0] = {4, {1,2,10,9}}
poly[1] = {4, {2,3,11,10}}
etc.
poly[8] = {3, {0,9,10}}
poly[9] = {3, {0,10,11}}
etc.
These can be generated in a loop
3-D Surfaces

- Explicit Representation
  \[ z = f(x,y) \]

- Plotting
  - Fix values of \( y \) and vary \( x \)
  - Gives a family of curves
    - \( z_0 = f(x,0) \)
    - \( z_1 = f(x,1) \)
    - \( z_2 = f(x,2) \)
    - \( z_3 = f(x,3) \)
    - etc.

Plotting 3D Surfaces, continued

- Then fix values of \( x \) and vary \( y \)
- Gives another family of curves
  - \( z_0' = f(0,y) \)
  - \( z_1' = f(1,y) \)
  - \( z_2' = f(2,y) \)
  - \( z_3' = f(3,y) \)
  - etc.

Plotting 3D Surfaces, continued

- Result is a wireframe that represents the surface
- Could be broken up into polygons

Parametric Representation of 3D Surfaces

- Need two parameters, say \( t \) and \( s \)
- \( x = x(t,s), \ y = y(t,s), \ z = z(t,s) \)
- both \( t \) and \( s \) vary over a range (0 to 1)
- To plot:
  - Fix values of \( s \) and for each vary \( t \) over range
    - gives one family of isoparametric curves
  - Fix values of \( t \) and for each vary \( s \) over range
    - gives another family of isoparametric curves

Bicubic Bezier Surface Patches

- Define 4-vectors \( S \) and \( T \):
  - \( S = [s^3 \ s^2 \ s \ 1] \), \( 0 \leq s \leq 1 \)
  - \( T = [t^3 \ t^2 \ t \ 1] \), \( 0 \leq t \leq 1 \)
- Define points on surface patch \( Q(s,t) \) as:
  \[
  Q(s,t) = S \cdot M_B \cdot T
  \]
- Control points \( P_i \) are themselves parameterized by \( t \)
- \( M_B \) is the Bezier Geometry Matrix we’ve seen before

\[
\begin{align*}
Q_0(t) &= T \cdot M_B \cdot P_0(t) \\
Q_1(t) &= T \cdot M_B \cdot P_1(t) \\
Q_2(t) &= T \cdot M_B \cdot P_2(t) \\
Q_3(t) &= T \cdot M_B \cdot P_3(t)
\end{align*}
\]

Transposing:

\[
\begin{align*}
P_0(t) &= [P_{00} \ P_{01} \ P_{02} \ P_{03}] \cdot M_B^T \cdot T^T \\
P_1(t) &= [P_{10} \ P_{11} \ P_{12} \ P_{13}] \cdot M_B^T \cdot T^T \\
P_2(t) &= [P_{20} \ P_{21} \ P_{22} \ P_{23}] \cdot M_B^T \cdot T^T \\
P_3(t) &= [P_{30} \ P_{31} \ P_{32} \ P_{33}] \cdot M_B^T \cdot T^T
\end{align*}
\]

Result:

\[
\begin{align*}
Q(s,t) &= S \cdot M_B \cdot T
\end{align*}
\]
A Bicubic Bezier Surface Patch

Expanding and Rearranging Terms -- x(s,t) Equation

\[
x(s,t) = (1-s)^2 [x_0 (1-t)^2 + 2 x_1 (1-t) t + x_2 (1-t)^2 t^2] + 2 (1-s) [x_0 (1-t)^2 t + x_1 (1-t)^2 t^2 + x_2 (1-t)^2 t^3] + 2 s [x_0 (1-t)^2 t^2 + x_1 (1-t)^2 t^3 + x_2 (1-t)^2 t^4] + s^2 [x_0 + x_1 (1-t) + x_2 (1-t)^2]
\]

Plotting One Set of Isoparametric Curves
For (s=0; s<=1; s+=?)
Compute & store x(s,0), y(s,0), z(s,0)
Project to screen and store --> xs(s,0), ys(s,0)
MoveTo(xs(s,0), ys(s,0))
For (t=0; t<=1; t+=?)
Compute & store x(s,t), y(s,t), z(s,t)
Project to screen and store --> xs(s,t), ys(s,t)
LineTo(xs(s,t), ys(s,t))

Plotting the Other Set of Isoparametric Curves
For (t=0; t<=1; t+=?)
MoveTo(xs(0,t), ys(0,t))
For (s=0; s<=1; s+=?)
LineTo(xs(s,t), ys(s,t))

3-D Geometric Transformations
- Move objects in a 3-D scene
- Extension of 2-D Affine Transformations
- Three important ones:
  - Translation
  - Scaling
  - Rotations

Representing 3-D Points
- Homogeneous coordinates
  \[ P (x,y,z) \rightarrow P' (x',y',z') \]
  \[
  \begin{pmatrix}
  x' \\
  y' \\
  z'
  \end{pmatrix}
  =
  \begin{pmatrix}
  x \\
  y \\
  z
  \end{pmatrix}
  \begin{pmatrix}
  x' & y' & z' \\
  1 & 1 & 1
  \end{pmatrix}
\]
Translations

- Given 3-D translation vector \( T = (tx, ty, tz) \)
- Component equations
  \[
  x' = x + tx \\
  y' = y + ty \\
  z' = z + tz
  \]
- Represent translation as matrix equation
  \[
  P' = T \cdot P
  \]
- \( T \) is a 4 X 4 Homogeneous Matrix

Homogeneous Translation Matrix

\[
\begin{bmatrix}
1 & 0 & 0 & tx \\
0 & 1 & 0 & ty \\
0 & 0 & 1 & tz \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- Notice obvious extension from 2-D to 3-D

Scaling with respect to origin

- Given three scaling factors \( sx, sy, sz \)
  \( P' = S \cdot P \)
- \( S \) is the following 4 X 4 scaling matrix:
  \[
  S = \begin{bmatrix}
sx & 0 & 0 & 0 \\
0 & sy & 0 & 0 \\
0 & 0 & sz & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
- Again obvious extension from 2D

Rotations

- Need to specify angle of rotation
- And axis about which the rotation is to be performed
- Infinite number of possible rotation axes
  - Rotation about any axis: linear combinations of rotations about x-axis, y-axis, z-axis

Rotations about z-axis

- Consider rotation of point \( P = (x, y, z) \) by angle theta about the z-axis giving rotated point \( P' = (x', y', z') \)
  - Same x, y equations as in the 2-D case
  - z will not change

Z-Axis Rotation Component Equations

\[
\begin{align*}
x' &= x \cdot \cos(\theta) - y \cdot \sin(\theta) \\
y' &= x \cdot \sin(\theta) + y \cdot \cos(\theta) \\
z' &= z
\end{align*}
\]

- Represented as homogeneous matrix equation:
  \[
  P' = R_z \cdot P
  \]
**Z-Axis Rotation Matrix**

\[
R_z = \begin{bmatrix}
\cos(\theta) & -\sin(\theta) & 0 & 0 \\
\sin(\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**Rx Matrix for rotations about x-axis**

- **Symmetry argument**
- \(x \rightarrow y\), \(y \rightarrow z\), \(z \rightarrow x\)

\[
R_x = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\theta) & -\sin(\theta) & 0 \\
0 & \sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**Original rotation about z-axis equations:**

- \(x' = x \cos(\theta) - y \sin(\theta)\)
- \(y' = x \sin(\theta) + y \cos(\theta)\)
- \(z' = z\)

**x\rightarrow y, y \rightarrow z, z\rightarrow x** transformed equations:

- \(y' = y \cos(\theta) - z \sin(\theta)\)
- \(z' = y \sin(\theta) + z \cos(\theta)\)
- \(x' = x\)

**Represented as matrix equation:**

\[
P' = R_x \times P
\]

**Ry Rotation Matrix**

- **Symmetry:**
- \(x \rightarrow z\), \(y \rightarrow x\), \(z \rightarrow y\)

\[
R_y = \begin{bmatrix}
\cos(\theta) & 0 & \sin(\theta) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\theta) & 0 & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**P' = Ry \times P**

\[
P' = \begin{bmatrix}
\cos(\theta) & 0 & \sin(\theta) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\theta) & 0 & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Rotation Sense

- Positive sense
  - Defined as counter clockwise as we look down the rotation axis toward the origin

Composite 3-D Geometric Transformations

- Series of consecutive transformations
  - Represented by homogeneous transformation matrices $T_1$, $T_2$, ..., $T_n$
- Equivalent to a single transformation
  - Represented by composite transformation matrix $T$
  - $T$ is given by the matrix product:
    $$ T = T_n \ldots T_2 T_1 $$
- Just like in 2-D, except matrices are 4 X 4

Example

- Rotate the line segment:
  $$(0,2,0) \rightarrow (0,4,3)$$
  by 90 degrees about an axis parallel to the z-axis and passing through its left endpoint.
  1. Translate to origin $(0,-2,0)$
  2. Rotate by 90 degrees about z-axis
  3. Translate back $(0,2,0)$

Library of 3-D Transformation Functions

- 3-D Transformation Package
- Straightforward Package
- Enables setting up and transforming points & polygons
- 4 X 4 Matrices have 12 non-trivial matrix elements
- Package Might contain the following functions:

3-D Transformation Functions

```c
void settranslate3d(a[12], tx, ty, tz);
void setscale3d(a[12], sx, sy, sz);
void setrotateX3d(a[12], theta);
void setrotateY3d(a[12], theta);
void setrotateZ3d(a[12], theta);
void combine3d(c[12], a[12], b[12]); // C = A * B
void xformcoord3d(c[12], vi, vo); // vo = C * vi
void xformpoly3d(inpoly[], outpoly[], float c[12]);
```

- $a$, $b$, and $c$ are arrays
  - Contain 12 non-trivial matrix elements of a 4 X4 homogeneous transformation matrix
- $v_i$ and $v_o$ are 3-D point structures;
  inpoly and outpoly are polygons
Rotation about an Arbitrary Axis

- Rotate point P by angle \( \theta \) about a line
- Given: endpoints \( P1=(x1,y1,z1) \) & \( P2=(x2,y2,z2) \)
- Convert problem into rotation about x-axis
  1. Translate so that \( P1 \) is at origin: \( T1 = T(-x1,-y1,-z1) \)
  2. Compute spherical coordinates of the other endpoint:
     \( \rho = \sqrt{(x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2} \)
     \( \phi = \arccos\left(\frac{z2-z1}{\rho}\right) \)
     \( \theta = \arctan\left(\frac{y2-y1}{x2-x1}\right) \)
  3. Rotate about z-axis by \(-\theta\) so line lies in x-z plane:
     \( T2 = Rz(-\theta) \)
  4. Rotate about y-axis by \((90-\phi)\) to make line coincide with x-axis
     \( T3 = Ry(90-\phi) \)
  5. Rotate about x-axis by given angle \( \theta \)
     \( T4 = Rx(\theta) \)
  6. Rotate back to undo step 4:
     \( T5 = Ry(90) \)
  7. Rotate back to undo step 3:
     \( T6 = Rz(\phi) \)
  8. Translate back to undo step 1:
     \( T7 = T(x1,y1,z1) \)
- Composite transformation then will be:
  \( T = T7*T6*T5*T4*T3*T2*T1 \)

3-D Coordinate System Transformations

- There's a symmetrical relationship between 3-D geometric transformations
  - (moving the object)
  and 3-D coordinate system transformations
  - (moving the coordinate system)
- For translations, relationship is:
  \( T_{coord}(x,y,z) = T_{geom}(-x,-y,-z) \)
- For each principal-axis, rotation relationship is:
  \( R_{coord}(\theta) = R_{geom}(\theta) \)
- Useful in deriving 3-D viewing transformation