**Binghamton University**

**EngiNet™**

**State University of New York**

Thomas J. Watson

School of Engineering
and Applied Science

**EngiNet™**

**WARNING**



**CS 460/560**

**Computer Graphics**

**Professor Richard Eckert**

**Lecture # 19**

**April 11, 2001**

## Lecture 19

- Overview of 3-D Graphics
- 3-D Modeling with Polygons
- 3-D Geometric Transformations
- See CS-460/560 Notes:

http://www.cs.binghamton.edu/~reckert/460/3d0.htm
http://www.cs.binghamton.edu/~reckert/460/3dpoly.htm
http://www.cs.binghamton.edu/~reckert/460/3dxform.htm

## Overview of 3-D Computer Graphics

- Display image of real or imagined 3-D scene on a 2-D screen

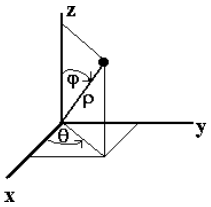
## Problem # 1: Modeling

- Representing objects in 3-D space
- First need to represent points
- Use a 3-D coordinate system
  - Cartesian: (x, y, z)
  - Spherical: (rho, theta, phi)
  - Cylindrical: (r, theta, z)

## Conversions

- Spherical to Cartesian

$x = \rho * \sin(\phi) * \cos(\theta)$
$y = \rho * \sin(\phi) * \sin(\theta)$
$z = \rho * \cos(\phi)$




## Types of 3-D Models

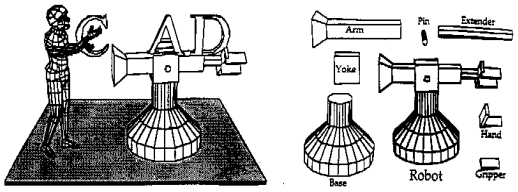
- 1. Boundary Representation (B-Rep)
  - Surface descriptions
  - Two common ones:
    - A. Polygonal
    - B. Bicubic parametric surface patches
- 2. Solid Representation
  - Solid modeling

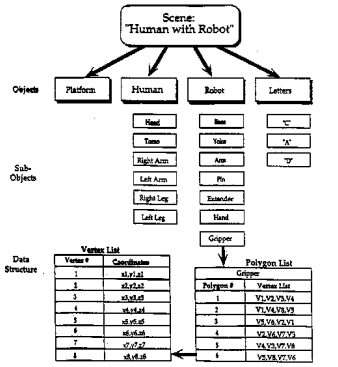
## Polygonal Models

- Object surfaces approximated by a mesh of planar polygons

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

## Polygon Mesh Model Example Scene

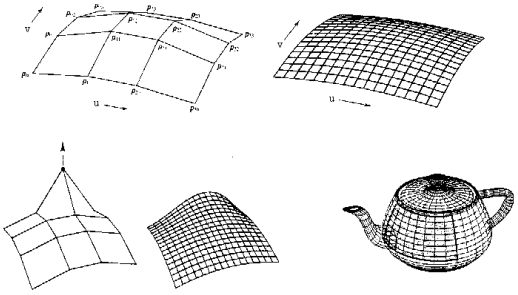



## Bicubic Parametric Surface Patches

- Objects represented by nets of elements called surface patches
  - Polynomials in two parametric variables
  - Usually cubic
    - Bezier surface patches
    - B-Spline surface patches

## Bicubic Parametric Surface Patches

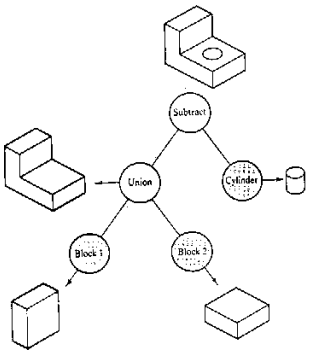


## Solid Representation-- Solid Modeling

- Objects represented exactly by combinations of elementary solid objects
  - e.g., spheres, cylinders, boxes, etc
  - Called geometric primitives

## Constructive Solid Geometry (CSG)

- Complex objects built up by combining geometric primitives using Boolean set operations
  - union, intersection, difference
- and linear transformations
- Object stored as a tree
  - Leaves contain primitives
  - Nodes store set operators or transformations

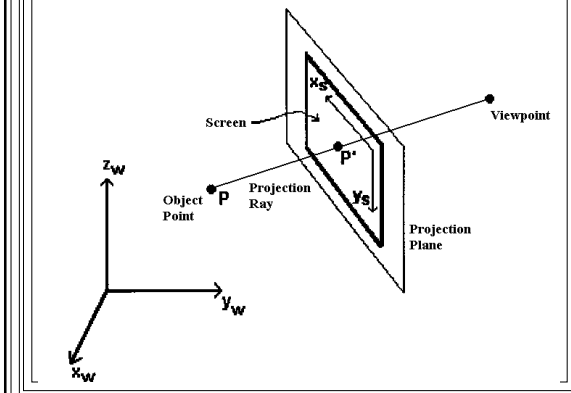



## Problem # 2: Rendering

- Displaying a 2-D view of a 3-D model

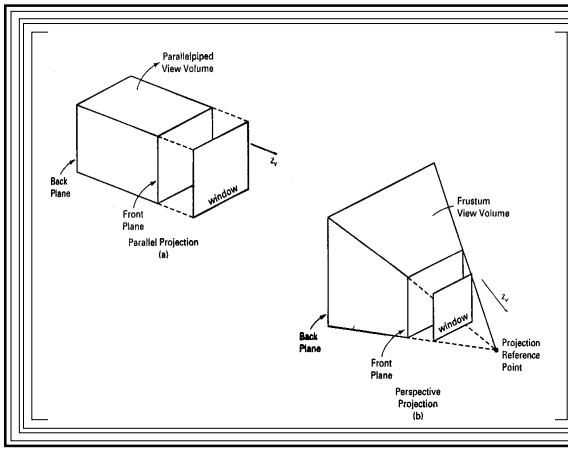
## A. Projection

- Going from 3-D to 2-D
  - Every world coordinate point in scene (xw,yw,zw) maps to a point on device viewing screen (xs,ys)
  - Must construct projection rays
    - From point in scene thru projection plane terminating on “Center of Projection"
  - Projection Point:
    - Intersection of projection ray with projection plane




## Two Basic Types of Projection

- 1. Parallel projection
  - Center of Projection at infinity
  - So projection rays are parallel
  - Equal-size objects at different distances from screen project to same size images
  - Parallel lines in scene project to parallel lines on screen
  - Useful in CAD



- 2. Perspective projection
  - Center of Projection at finite distance from screen
  - Far objects project to smaller images than close objects
    - Farther objects appear to be smaller
    - More realistic images
    - Parallel lines in scene don't necessarily project to parallel lines on screen

## B. Hidden surface removal

- Surfaces facing away from viewer are invisible
  - Should not be displayed
    - Backface culling
- Surfaces blocked by objects closer to viewer are invisible
  - Should not be displayed
    - General hidden surface removal algorithms

## C. Shading

- Projections of surfaces should be colored (shaded)
- Color depends on intensity of light reflected from surface into viewer's eye
- Need an illumination/reflection model
  - Must take into account:
    - Material properties of surfaces
    - How light interacts with them

## D. Other effects

- Shadows
- Transparency
- Multiple reflections
- Atmospheric absorption
- Surface textures
- Lots of others
- Physics and Optics!!

## The Viewing Pipeline

- Chain of transformations/operations needed to go from a 3-D model to a 2-D image on the viewing screen

1. Local coordinate space (3-D):
   Individual object descriptions given
   |
   | Modeling Transformations
   | (Geometric transformations)
   v
2. World coordinate space (3-D):
   Scene is composed
   Objects, lights positioned
   |
   | 3-D Viewing Transformation
   |
   v

3. Viewing coordinate space (3-D):
Eye/camera coordinate system
|
| 3-D clipping
| Backface culling
|
v
4. 3-D viewing volume:
Eye/camera coordinate system
|
| Projection Transformation
|
v

5. 2-D projection plane description:
2-D World coordinate system window
|
| 2-D Viewing xformation (window to viewport)
| 2-D clipping
| Hidden surface removal
| Shading
| Other effects
|
v
6. 2-D Device coordinate space:
2-D Viewport coordinate system window

## 3-D Modeling with Polygons

- Two types of polygon models
  1. Wireframe
     - Store the polygon edges
     - List of edge endpoints
     - Not useful for shaded images
  2. 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 Wireframe Model

- Store polygon edges as an array
- Each element a pair of indices into the 3-D points array:

```
int edges[][2];  // Each second-index value gives the
                 // position of an edge's endpoint vertex
                 // in the 3-D points array
```

## Storing Polygons in a Polygon Mesh Model

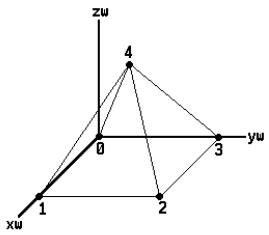
- Object: Can be represented as an array of polygons
- Each element consists of:
  - (a) the number of vertices in the polygon
  - (b) a list of indices into the 3-D points array
    - (An index gives the position of a vertex in the 3-D points array)

```
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
```

## Example--A Pyramid

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




## Vertex Coordinates

| vertex | xw | yw | zw |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 150 | 0 | 0 |
| 2 | 150 | 150 | 150 |
| 3 | 0 | 150 | 0 |
| 4 | 75 | 75 | 150 |

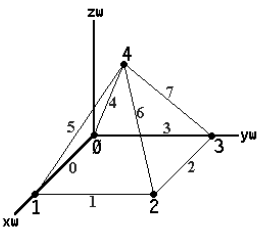
## The Pyramid's Points Array

```
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=w_pts[2].y=b; 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/2; w_pts[4].z=h;
```

## Edge Array (Wireframe)

| Edge | Endpoints (points array indices) |
|---|---|
| 0 | 0, 1 |
| 1 | 1, 2 |
| 2 | 2, 3 |
| 3 | 3, 0 |
| 4 | 0, 4 |
| 5 | 1, 4 |
| 6 | 2, 4 |
| 7 | 3, 4 |




## Edge Array

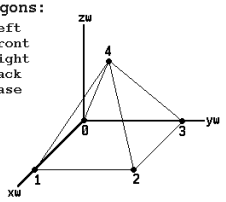
- Edge array could be generated by:

```
int edges[8][2] =
   {{0,1},{1,2},{2,3},{3,0},{0,4},{1,4},{2,4},{3,4}};
```

## Polygons Array (Mesh)

| polygon | # vertices | vertices |
|---|---|---|
| 0 | 3 | 0,1,4 |
| 1 | 3 | 1,2,4 |
| 2 | 3 | 2,3,4 |
| 3 | 3 | 0,4,3 |
| 4 | 4 | 0,3,2,1 |

```
Polygons:
0: Left
1: Front
2: Right
3: Back
4: Base
```

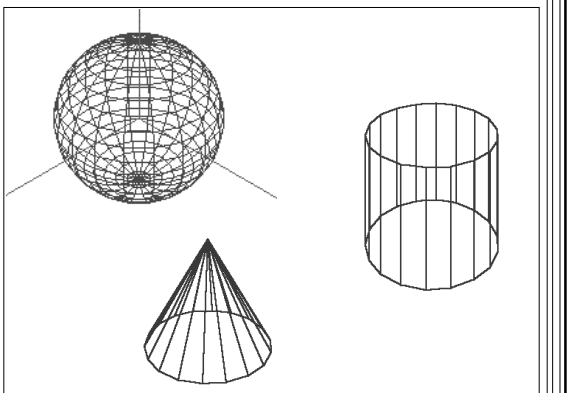



- Polygon array could be generated by:

```
// Allocate Space:
for (i=0;i<=3;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;
object[1].inds[0]=1; object[1].inds[1]=2; object[1].inds[2]=4;
object[2].inds[0]=2; object[2].inds[1]=3; object[2].inds[2]=4;
object[3].inds[0]=0; object[3].inds[1]=4; object[3].inds[2]=3;
 // define the square base
object[4].inds[0]=0; object[4].inds[1]=3;object[4].inds[2]=2;
object[4].inds[3]=1;
```

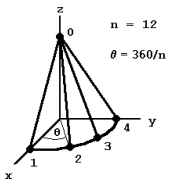
## More Complex 3-D Objects

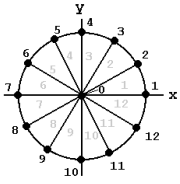
- Approximate surfaces with polygons
- Often Points, edges, and/or polygons arrays can be generated procedurally



## Example 1: A Cone

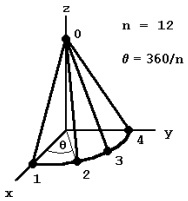
- Approximate with n triangular sides
- n+1 vertices (apex + n in the base)
- And a Base polygon with n sides (example, n=12)



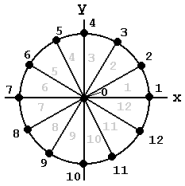

## Cone Points Array

- Base points:
  - x = R * cos ( i * θ )
  - y = R * sin (i * θ )
  - 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}}
- 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*n+2
- Number of polygons = (n+1)*m
- Example n=3, m=8

8 * 3 + 2 = 26 vertices

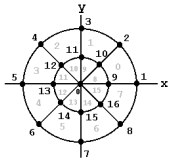
Can get x, y, z from spherical coordinates

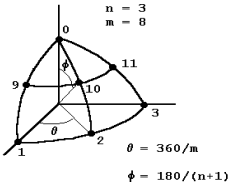
Loop j:0->n-1 latitudes, i:0->m-1 longitudes

x = R * sin(i*θ) * cos(j*ϕ)

y = R * sin(i*θ) * sin(j*ϕ)

z = R * cos(j*ϕ)




(3+1)*8 = 32 polygons

Number in consistent way

poly[0] = {4, {1,2,10,9}}

poly[1] = {4. {2,3,11,10}}

etc.

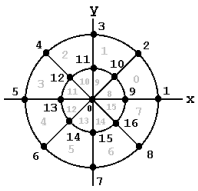
poly[8] = {3, {0,9,10}}

poly[9] = {3, {0,10,11}}

etc.

These can be generated in a loop

Upper Hemisphere




## 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) --> P' (x',y',z')

$$\begin{vmatrix} x \\ y \\ z \\ 1 \end{vmatrix} \rightarrow \begin{vmatrix} x' \\ y' \\ z' \\ 1 \end{vmatrix}$$

## 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 * P
- T is a 4 X 4 Homogeneous Matrix

## Homogeneous Translation Matrix

$$T = \begin{| 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{|}$$

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

## Scaling with respect to origin

- Given three scaling factors sx, sy, sz

  P' = S * P
- S is the following 4 X 4 scaling matrix:

$$S = \begin{| sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{|}$$

- 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

x' = x*cos(theta) - y*sin(theta)

y' = x*sin(theta) + y*cos(theta)

z' = z

- Represented as homogeneous matrix equation:

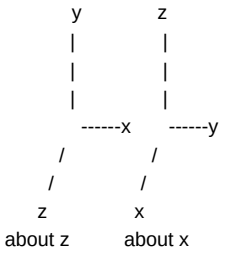
  P' = Rz * P

## Z-Axis Rotation Matrix

$$Rz = \begin{vmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rx Matrix for rotations about x-axis

- Symmetry argument




Make replacements:
x --> y
y --> z
z --> x

- Resulting equations:

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

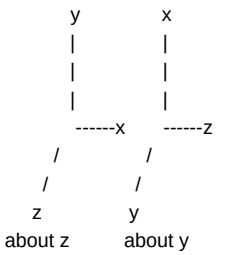
- Represented as matrix equation:

  P' = Rx * P

$$Rx = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Ry Rotation Matrix

- Symmetry:




Replacements:
x --> z
y --> x
z --> y

x --> z
y --> x
z --> y

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

P' = Ry * P

$$Ry = \begin{vmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

## Rotation Sense

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