Introduction to 3-D Graphics
- Modeling and Rendering
- Types of Projections
- The Viewing Pipeline
- Wireframe Models
- Polygon Mesh Models

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
  \[ x = \rho \sin(\theta) \cos(\phi) \]
  \[ y = \rho \sin(\theta) \sin(\phi) \]
  \[ z = \rho \cos(\theta) \]
- RH Coord System
- Could be LH
- Viewing system

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 -->
- Subobjects -->
- 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 $(x_w, y_w, z_w)$ maps to a point on device viewing screen $(x_s, y_s)$

Camera model

- Construct projection rays
  - From points in scene through projection plane terminating on “Center of Projection”
    - Camera point or viewpoint
- 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 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)

```c
struct polygon {int n; int *inds};
// n: The number of vertices
// inds: 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

Pyramid below has 5 vertices, 8 edges and 5 polygon faces

### 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>150</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=b; w_pts[2].z=0;
w_pts[3].x=0; w_pts[3].y=b; w_pts[3].z=0;
w_pts[4].x=b/2; w_pts[4].y=b/2; w_pts[4].z=h;
```

Edge Array (Wireframe)

```c
int edge[7] = {0, 1, 1, 2, 2, 3, 3, 0, 0, 4, 1, 4, 2, 4, 3, 4};
```

```c
// 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=b; w_pts[2].z=0;
w_pts[3].x=0; w_pts[3].y=b; w_pts[3].z=0;
w_pts[4].x=b/2; w_pts[4].y=b/2; w_pts[4].z=h;
```
**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)**

<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[5];
// 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;
// define the square base
object[4].inds[3]=1;
```