Shape model

• You have some experience with shape modeling
  – Rails as curves
  – Tree = cone + cylinder

• There are many ways to represent the shape of an object

• choosing a representation depends on application and requirement
Boundary vs. Solid Representations

• B-rep: boundary representation
  – Sometimes we only care about the surface
  – Rendering opaque objects

• Solid modeling
  – Some representations are best thought of defining the space filled, rather than the surface around the space
  – Medical data with information attached to the space
  – Transparent objects with internal structure
  – Taking cuts out of an object; “What will I see if I break this object?”
Shape Representation

- Parametric models
- Implicit models
- Procedural models
Parametric Model

• generates all the points on a surface (volume) by “plugging in a parameter”
  – Eg \((\sin \phi \cos \theta, \sin \phi \sin \theta, \cos \phi)\)
    \[0 \leq \theta < 2\pi, \quad 0 \leq \phi \leq \pi\]
  – Easy to render, how?
  – Easy to texture map
Implicit Models

- Implicit models use an equation that is 0 if the point is on the surface
  - Essentially a function to test the status of a point
  - Eg \( x^2 + y^2 + z^2 - 1 = 0 \)
  - Easy to test inside/outside/on
  - Hard to?
    - Render
    - Texture map
Parametric Model

• generates all the points on a surface (volume) by “plugging in a parameter”
  – Eg \((\sin \phi \cos \theta, \sin \phi \sin \theta, \cos \phi)\)
    \[ 0 \leq \theta < 2\pi, \quad 0 \leq \phi \leq \pi \]
  – Easy to render, how?
  – Easy to texture map
  – Hard to
    • Test inside/outside/on
Procedural Modeling

• a procedure is used to describe how the shape is formed
Parameterization

• Parameterization is the process of associating a set of parameters with every point on an object
  – For instance, a line is easily parameterized by a single value
  – Triangles in 2D can be parameterized by their barycentric coordinates
  – Triangles in 3D can be parameterized by 3 vertices and the barycentric coordinates (need both to locate a point in 3D space)

• Several properties of a parameterization are important:
  – The smoothness of the mapping from parameter space to 3D points
  – The ease with which the parameter mapping can be inverted

• We care about parameterizations for several reasons
  – Texture mapping is the most obvious one you have seen so far; require \((s,t)\) parameters for every point in a triangle
Polygon Meshes

• A *mesh* is a set of polygons connected to form an object
• A mesh has several components, or geometric entities:
  – Faces
  – Edges
    • the boundary between faces
  – Vertices
    • the boundaries between edges,
    • or where three or more faces meet
  – Normals, Texture coordinates, colors, shading coefficients, etc
• What is the counterpart of a polygon mesh in curve modeling?
Polygonal Data Structures

- Polygon mesh data structures are **application dependent**
- Different applications require different operations to be fast
  - Find the neighbor of a given face
  - Find the faces that surround a vertex
  - Intersect two polygon meshes
- You typically choose:
  - Which features to store explicitly (vertices, faces, normals, etc)
  - Which relationships you want to be explicit (vertices belonging to faces, neighbors, faces at a vertex, etc)
Polygon Soup

- Many polygon models are just lists of polygons

```c
struct Vertex {
    float coords[3];
}
struct Triangle {
    struct Vertex verts[3];
}
struct Triangle mesh[n];

glBegin(GL_TRIANGLES)
    for ( i = 0 ; i < n ; i++ )
    {
        glVertex3fv(mesh[i].verts[0]);
        glVertex3fv(mesh[i].verts[1]);
        glVertex3fv(mesh[i].verts[2]);
    }
glEnd();
```

**Important Point:** OpenGL, and almost everything else, assumes a constant vertex ordering: clockwise or counter-clockwise. Default, and slightly more standard, is counter-clockwise
Cube Soup

```c
struct Triangle Cube[12] =
  {{{1,1,1},{1,0,0},{1,1,0}},
   {{1,1,1},{1,0,1},{1,0,0}},
   {{0,1,1},{1,1,1},{0,1,0}},
   {{1,1,1},{1,1,0},{0,1,0}},
   ...
  };
```
Polygon Soup Evaluation

• What are the advantages?
• What are the disadvantages?
Polygon Soup Evaluation

• What are the advantages?
  – It’s very simple to read, write, transmit, etc.
  – A common output format from CAD modelers
  – The format required for OpenGL

• BIG disadvantage: No higher order information
  – No information about neighbors
  – No open/closed information
  – No guarantees on degeneracies
There are reasons not to store the vertices explicitly at each polygon

- Wastes memory - each vertex repeated many times
- Very messy to find neighboring polygons
- Difficult to ensure that polygons meet correctly

Solution: Indirection

- Put all the vertices in a list
- Each face stores the indices of its vertices

Advantages? Disadvantages?
Cube with Indirection

```c
struct Vertex CubeVerts[8] =
    {{0,0,0},{1,0,0},{1,1,0},{0,1,0},
     {0,0,1},{1,0,1},{1,1,1},{0,1,1}};

struct Triangle CubeTriangles[12] =
    {{6,1,2},{6,5,1},{6,2,3},{6,3,7},
     {4,7,3},{4,3,0},{4,0,1},{4,1,5},
     {6,4,5},{6,7,4},{1,2,3},{1,3,0}};
```
Indirection Evaluation

• Advantages:
  – Connectivity information is easier to evaluate because vertex equality is obvious
  – Saving in storage:
    • Vertex index might be only 2 bytes, and a vertex is probably 12 bytes
    • Each vertex gets used at least 3 and generally 4-6 times, but is only stored once
  – Normals, texture coordinates, colors etc. can all be stored the same way

• Disadvantages:
  – Connectivity information is not explicit
OpenGL and Vertex Indirection

```c
struct Vertex {
    float coords[3];
}

struct Triangle {
    GLuint verts[3];
}

struct Mesh {
    struct Vertex vertices[m];
    struct Triangle triangles[n];
}

glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, sizeof(struct Vertex), mesh.vertices);

 glBegin(GL_TRIANGLES)
    for ( i = 0 ; i < n ; i++ )
    {
        glArrayElement(mesh.triangles[i].verts[0]);
        glArrayElement(mesh.triangles[i].verts[1]);
        glArrayElement(mesh.triangles[i].verts[2]);
    }
 glEnd();
```
OpenGL and Vertex Indirection (v2)

```c
glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, sizeof(struct Vertex),
               mesh.vertices);
for ( i = 0 ; i < n ; i++ )
    glVertexPointer(3, GL_FLOAT, sizeof(struct Vertex),
                    mesh.vertices);
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT,
                   mesh.triangles[i].verts);
```

- Minimizes amount of data sent to the renderer
- Fewer function calls
- Faster!
- Other tricks to accelerate using array, see Red book, Ch 2 on vertex arrays
Polygon Modeling

- Polygons are the dominant force in modeling for real-time graphics
- Why?
Polygons Dominate because

- Everything can be turned into polygons (almost everything)
  - Normally an error associated with the conversion, but with time and space it may be possible to reduce this error
- We know how to render polygons quickly
- Texture mapping easily
- Memory and disk space is cheap
- Simplicity
What’s Bad About Polygons?

• What are some disadvantages of polygonal representations?
Polygons Aren’t Great

• They are always an approximation to curved surfaces
  – Most real-world surfaces are curved, particularly natural surfaces
  – They throw away information
  – Normal vectors are approximate
  – But can be as good as you want, if you are willing to pay in size
• They can be very unstructured
• They are hard to globally parameterize (complex concept)
  – How do we parameterize them for texture mapping?
• It is difficult to perform many geometric operations
  – Collision, intersection
Normal Vectors in Mesh

• Normal vectors give information about the true surface shape
• Per-Face normals:
  – One normal vector for each face, stored as part of face (Flat shading)

```c
struct Vertex {
    float coords[3];
}
struct Triangle {
    GLuint verts[3];
    float normal[3];
}
struct Mesh {
    struct Vertex vertices[m];
    struct Triangle triangles[n];
}```
Normal Vectors in Mesh

• Normal vectors give information about the true surface shape

• Per-Vertex normals:
  – A normal specified for every vertex (smooth shading)

```c
struct Vertex {
    float coords[3];
    float normal[3];
};
struct Triangle {
    GLuint verts[3];
};
struct Mesh {
    struct Vertex vertices[m];
    struct Triangle triangles[n];
};
```
Storing Other Information

• Colors, Texture coordinates and so on can all be treated like vertices or normals
• Lighting/Shading coefficients may be per-face, per-object, or per-vertex
Other Data in Mesh

• Normal vectors give information about the true surface shape
• Per-Vertex normals:
  – A normal specified for every vertex (smooth shading)
• Per-Vertex Texture Coord

```
struct Vertex {
    float coords[3];
    float normal[3];
    float texCoords[2];
}
struct Triangle {
    GLuint verts[3];
}
struct Mesh {
    Vertex vertices[m];
    Triangle triangles[n];
}
```
Other Data in Mesh

• Normal vectors give information about the true surface shape

• Per-Vertex normals:
  – A normal specified for every vertex (smooth shading)

• Per-Vertex Texture Coord, Shading Coefficients

```c
struct Vertex {
    float coords[3];
    float normal[3];
    float texCoords[2], diffuse[3], shininess;
}
struct Triangle {
    GLuint verts[3];
}
struct Mesh {
    Vertex vertices[m];
    Triangle triangles[n];
}
```
Other Data in Mesh

• Normal vectors give information about the true surface shape
• Per-Vertex normals:
  – A normal specified for every vertex (smooth shading)
• Per-Vertex Texture Coord, Shading Coefficients

```c
struct Vertex {
    float coords[3];
}
struct Triangle {
    GLuint verts[3];
}
struct Mesh {
    Vertex vertices[m];
    float normals[3*m];
    float texCoords[2*m], diffuse[3*m], shininess[m];
    Triangle triangles[n];
}```
Issues with Polygons

• They are inherently an approximation
  – Things like silhouettes can never be perfect without very large numbers of polygons, and corresponding expense
  – Normal vectors are not specified everywhere

• Interaction is a problem
  – Dragging points around is time consuming
  – Maintaining things like smoothness is difficult

• Low level representation
  – Eg: Hard to increase, or decrease, the resolution
  – Hard to extract information like curvature
In Project 3, we use Sweep Objects

- Define a polygon by its edges
- Sweep it along a path
- The path taken by the edges form a surface - the sweep surface
- Special cases
  - Surface of revolution: Rotate edges about an axis
  - Extrusion: Sweep along a straight line
Rendering Sweeps

• Convert to polygons
  – Break path into short segments
  – Create a copy of the sweep polygon at each segment
  – Join the corresponding vertices between the polygons
  – May need things like end-caps on surfaces of revolution and extrusions

• Normals?
  – Normals come from sweep polygon and path orientation

• Texture Coord?
  – Sweep polygon defines one texture parameter, sweep path defines the other
General Sweeps

• The path maybe any curve
General Sweeps

• The path maybe any curve
• The polygon that is swept may be transformed as it is moved along the path
  – Scale, rotate with respect to path orientation, ...

Cube

Twisted Cube
General Sweeps

• The path maybe any curve
• The polygon that is swept may be transformed as it is moved along the path
  – Scale, rotate with respect to path orientation, ...
General Sweeps

• The path maybe any curve
• The polygon that is swept may be transformed as it is moved along the path
  – Scale, rotate with respect to path orientation, ...
• One common way to specify is:
  – Give a poly-line (sequence of line segments) as the path
  – Give a poly-line as the shape to sweep
  – Give a transformation to apply at the vertex of each path segment
• Texture Coord?
• Difficult to avoid self-intersection
Klein Bottle

Torus

Klein Bottle
Mobious Strip

Non-orientable surfaces
Change Topology when Sweeping