Meshes

• polygonal soup
  – polygons specified one-by-one with no explicit information on
    shared vertices
• polygonal nonmanifold
  – connectivity information is provided (which vertices are shared)
    no restrictions on connections between polygons
• polygonal manifold
  – no edge is shared by more than two polygons; the faces
    adjacent to a vertex form a single ring (incomplete ring for
    boundary vertices)
• triangle manifold
  – in addition, all faces are triangles

Mesh elements

faces, vertices, edges

Each mesh element can have information
associated with it; typical mesh operations
involve visiting (traversing) all vertices, faces, or edges
Mesh descriptions

- **OBJ format**
  each line defines an element (vertex or face); first character defines the type
  
  **Vertex:**
  
  \[ v \ x, \ y, \ z \]

  **Face with n vertices:**
  
  \[ f \ i_1 \ i_2 \ i_3 \ldots \ i_n \]

  where \( i_1 \ldots i_n \) are vertex indices; the indices are obtained by numbering all vertices sequentially as they appear in a file

Mesh operations

- **Types of mesh operations**
  - traversals go over all elements of certain type
  - collect adjacent elements (e.g. all neighbors of a vertex)
  - refinement
    - edge flips
    - face addition/deletion
    - face merge
Traversal operations

- Iterate over all vertices, faces, edge
  - visit each only once
  - iterate over all elements (faces, vertices, edges) adjacent to an element

A simple mesh representation

One-to-one correspondence with OBJ

array of vertices
2 arrays for faces
each face is a list of vertex indices enumerated clockwise

starting indices of face vertex lists
vertex indices of all faces
Traversing operations

Complexity of traversal operations w/o additional data structures as function of the number of vertices, assuming constant vertex/face ratio

<table>
<thead>
<tr>
<th>Iterate over collect adjacent</th>
<th>V</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>quadratic</td>
<td>quadratic</td>
<td>linear</td>
</tr>
<tr>
<td>E</td>
<td>quadratic</td>
<td>quadratic</td>
<td>linear</td>
</tr>
<tr>
<td>F</td>
<td>quadratic</td>
<td>quadratic</td>
<td>linear</td>
</tr>
</tbody>
</table>

Most operations such as collecting all adjacent faces for a vertex are slow, because the connectivity information is not explicit: one needs to search the whole list of faces to find faces with a given vertex; if neighbors are encoded explicitly this can be done in constant time.
Face-based mesh representation

Useful primarily for triangle or quad. meshes

Triangle meshes:

```c
struct Face {
    Face* face[3]; // pointers to neighbors
    Vertex* vertex[3];
}
```

```c
struct Vertex {
    Face* face; // pointer to a triangle adjacent to the vertex
}
```

(not really necessary, can refer to vertices using a handle (Face ptr, vertex index)

Traversing faces sharing a vertex

Assuming a mesh without boundary:

```c
fstart = v->face;
f = fstart;
do {
    ... // perform operations with *f
    // assume that vertex i is across edge i
    if (f->vertex[0]== v)
        f = f->face[1]; // crossing edge #1 vert. 0 - vert. 2
    else if (f->vertex[1] == v)
        f = f->face[2]; // crossing edge #2 vert. 1 - vert. 0
    else
        f = f->face[0]; // crossing edge #0 vert. 2 - vert. 1
} while( f != fstart);
```

Similar for edges and vertices.

All such operations can be done in const. time per vertex/face/edge.
Half-edge data structure

- General manifold polygonal meshes
  - Polygons have variable number of vertices
    variable size;
  - data structures based on faces are inconvenient and inefficient.
- Solution: use edge-based structures (winged edge, half-edge).
  - Half-edge is currently most common
  - Each edge = 2 half edges; can be interpreted either as
    directed edge or face-edge pair

```c
struct HalfEdge {
  Vertex* vertex; // the head vertex the
                  // half edge is pointing to
  Face* face;     // if data stored in faces
  HalfEdge* next; // next halfedge in the face
                  // on the left
  HalfEdge* sym;  // the other half edge for
                  // the same edge
}
struct Vertex {
  HalfEdge* halfedge; // one of the half edges
                      // starting at the vertex
}
```
Traversing operations

Vertices adjacent to a vertex $v$, mesh without boundary

```c
he = v->halfedge;
    do {
        he = he->sym->next;
        ... // perform operations with // he->vertex
    } while (he != v->halfedge)
```

No “if” statements.

Constructing a mesh data structure

Construct face-based structure from a list of triangles and vertices

Assume that vertices are listed counterclockwise for each triangle and $v_1$ indices of vertices in the face: $other(i_1,i_2)$ for $i_1,i_2 = 0..2$, $i_1 \neq i_2$ is the third vertex of the triangle $t_{i_1,i_2}$

Edgemap is a map (associative array) from pairs of vertices (directed edges) to faces; in addition to the face, we also record the number of the edge in the face (See C++ STL details of use)

This is pseudocode (not using C syntax to emphasize this)

```c
for each face
    create face structure $f_1$, initialize neighbors to 0
    for each triangle vertex $i=0..2$
        edgemap[$v_{i_1}, v_{(i+1)%3}$] := ($f_1$, other($i$, (i+1)%3))
    endfor
endfor

for each entry $(i,j)$ of the map edgemap
    edgemap($i,j$)
    $(f_2,e_2) := edgemap(j,i)$;
    if $f_2 != 0$ then
        $f_1->f[e_1] := f_2$
        $f_2->f[e_2] := f_1$
    endif
endfor
```