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

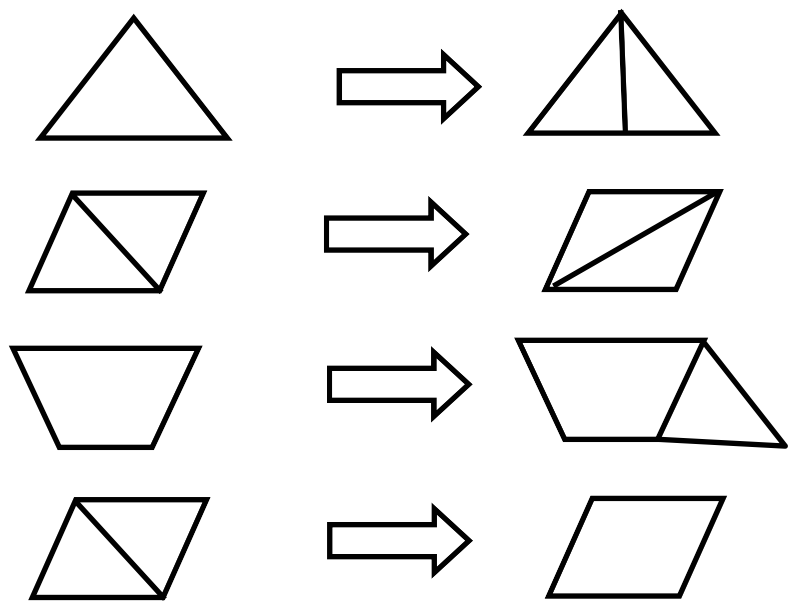
  Face with n vertices:

  f   i1 i2 i3 … in

where i1.. in, are vertex indices; the indices are obtained by numbering all vertices sequentially as they appear in a file

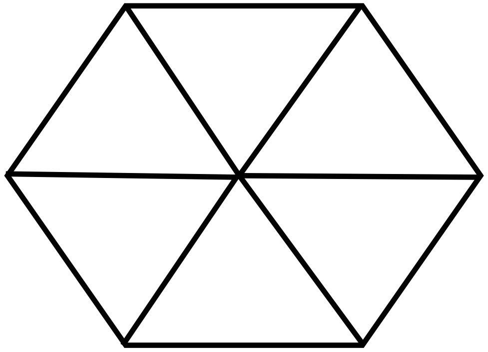
Mesh operations

- 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

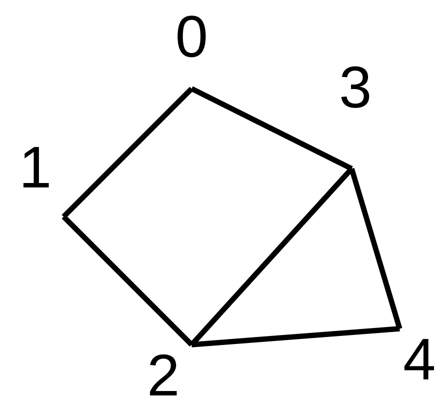
- Iterate over all vertices, faces, edge
  - visit each only once
  - iterate over all elements (faces, vertices, edges) adjacent to an element

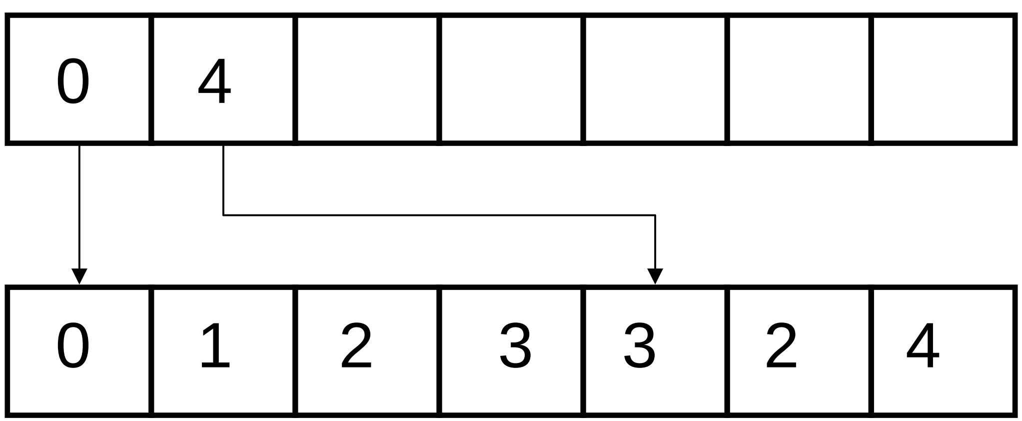


A simple mesh representation

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

Traversal operations

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

iterate over / collect adjacent	V	E	F
V	quadratic	quadratic	linear
E	quadratic	quadratic	linear
F	quadratic	quadratic	linear

Traversal operations

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 const. time

Face-based mesh representation

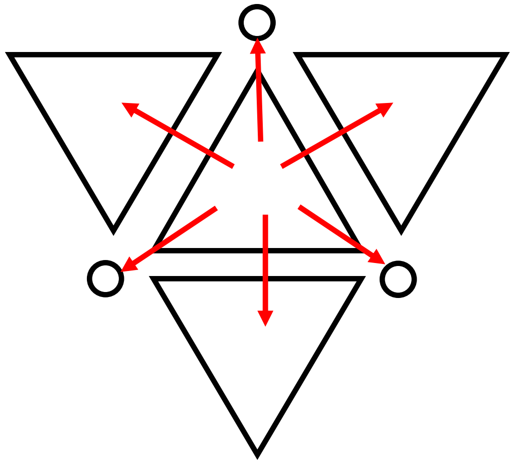
Useful primarily for triangle or quad. meshes

Triangle meshes:

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

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:

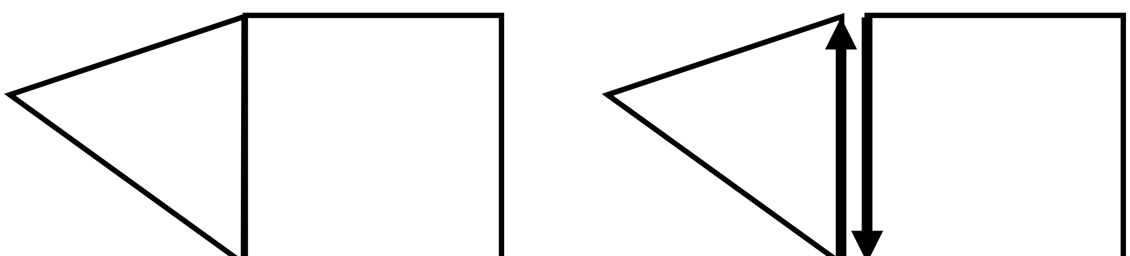
```
fstart = v->face;
f = fstart;
do {
    ... // perform operations with *f
    // assume that vertex i is across edge i
    if (f->vertex[0]== v)
      f = f->face[1]; // crossing edge #1  vert. 0 - vert. 2
    else if (f->v[1] == v)
      f = f->face[2]; // crossing edge #2  vert. 1 - vert. 0
   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

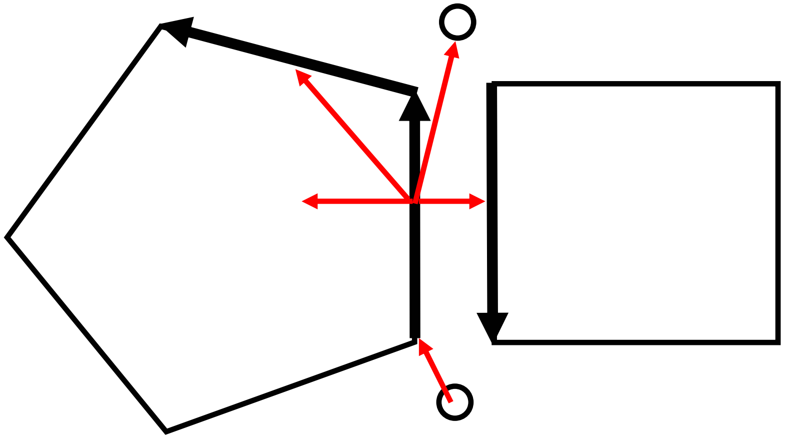
Half-edge data structure

- 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



Half-edge data structure

```
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
     }
```



Traversal operations

Vertices adjacent to a vertex v, mesh without boundary

```
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_i indices of vertices in the face; `other(i1,i2)` for i1,i2 = 0..2, i1 ≠ i2 is the third vertex of the triangle i3 ≠ i1,i2

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 `map` details of use)

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

```
for each face
  create face structure f1, initialize neighbors to 0
  for each triangle vertex i=0..2
    edgemap(v_i, v_{(i+1)%3}) := (f1, other(i, (i+1)%3) )
  endfor
endfor
for each entry (i,j) of the map edgemap
  edgemap(i,j)
  (f2,e2) := edgemap(j,i);
  if  f2 != 0 then
   f1->f[e1] := f2
   f2->f[e2] := f1
  endif
endfor
```

Building a half-edge data structure

- Similar to building face-based triangular mesh
- Input: a list of vertices, a list of faces, each face is a list of vertex indices enumerated CCW
- 1. Create arrays of vertices, faces and halfedges, one half-edge for every seq. pair of vertices of every face; initialize all pointers to zero.
- 2. For each face **`f`**, with n vertices
- assign **`f.halfedge`** to its first half-edge;
  for each vertex **`v`** of a face, assign **`v->halfedge`** to the halfedge starting at it if nothing is assigned to it yet;
  for each half-fedge **`he`** of a face, assign
  `he.face =f, he->next =`next half-edge in the face,
  `he->vertex =` next vertex in the face;
  record half-edge pointer he in the edge map:
- **`edgemap(v[i],v[i+1]) = he`**
- 3. Go over all entries of the edge map, assign for half-edges **`edgemap(i,j) edgemap(j,i)`** links to each other, if both exist

Dealing with boundaries

- To minimize implementation effort it is useful to create two halfedges for boundary edges, one of which has zero face pointer;
- A boundary vertex **`v`** should always have **`v.halfedge`**
- pointing to a boundary halfedge.
- Then it is easy e.g. to find two boundary neighbors of a vertex.