# Triangle Meshes: Representations

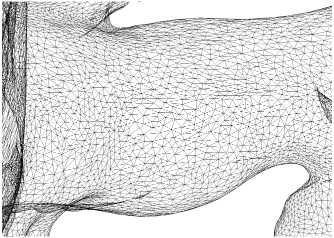
Andrzej Szymczak

October 6, 2010

# Triangle Soup

- For any triangle, list coordinates of vertices of that triangle
- Can be viewed as an array of floats with $T$ rows and 9 columns; each row of the table contains coordinates of 3 points or 9 integers
- Requires 9 floats per triangle
- 36 bytes per triangle if 4-byte floats are used for the coordinates
- Simple, but inefficient in terms of space (see below); also, makes it tricky to do any kind of computation on the mesh
  - No information on adacent triangles, triangles sharing a vertex etc
  - Possible to find them, but it requires searching the triangle list (slow!)

# Triangle+Vertex Tables



- Typically meshes close to watertight are used; triangles share edges and vertices
- Using triangle soup representation for such triangles is a waste of space (vertex coordinates repeated several times)
- For a large watertight meshes, a vertex is shared by about 6 triangles on average

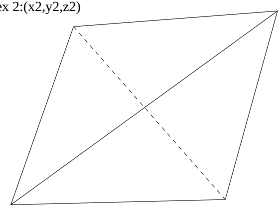
# Triangle+Vertex Tables

- Idea: Number the vertices, specify mesh using vertex table and triangle table
- Vertex table
  - List coordinates of vertices in order
  - Requires $3V$ floats, where $V$ is the number of vertices
  - Array with $V$ rows and 3 columns; entries are floats
- Triangle table
  - List IDs of vertices of every triangle
  - $3T$ integers, where $T$ is the number of triangles
- Total size: $12V + 12T$ if 4-byte integers are used for vertex IDs and 4-byte floats for vertex coordinates
- Typically, number of triangles is roughly twice the number of vertices (will prove it later based on Euler's formula)
- Size $=$ 18 bytes per triangle
- Still, not too convenient for geometry processing

# Triangle+Vertex+Adjacency table

- Add information about adjacent triangles: *adjacency table*
- Vertices have integer IDs (vertex table)
- Triangles have IDs (triangle table row number)
- For any triangle, vertices come in a certain order (the order of being listed in a row of the triangle table)
- Assume that any triangle has exactly one triangle adjacent across any of its edges
- Take a triangle with ID $t$ and its edge connecting $i$-th and $j$-th vertex; put the ID of the triangle adjacent to $t$ across that edge in row $t$ and column $3 - i - j$ of the adjacency table

# Example




vertex table:

| | | |
|---|---|---|
| x0 | y0 | z0 |
| x1 | y1 | z1 |
| x2 | y2 | z2 |
| x3 | y3 | z3 |

triangle table:

| | | |
|---|---|---|
| 2 | 0 | 1 |
| 3 | 2 | 1 |
| 3 | 0 | 2 |
| 3 | 1 | 0 |

adjacency table:

| | | |
|---|---|---|
| 3 | 1 | 2 |
| 0 | 3 | 2 |
| 0 | 1 | 3 |
| 0 | 2 | 1 |

Triangle soup

| | | |
|---|---|---|
| x2 y2 z2 | x0 y0 z0 | x1 y1 z1 |
| x3 y3 z3 | x2 y2 z2 | x1 y1 z1 |
| x3 y3 z3 | x0 y0 z0 | x2 y2 z2 |
| x3 y3 z3 | x1 y1 z1 | x0 y0 z0 |

# Triangle Table from Triangle Soup

- Very easy to do, but somewhat harder to do well
- Here is a bad example (for the tetrahedron mesh):

$$\text{Vertex table}: \begin{bmatrix} x_2 & y_2 & z_2 \\ x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_3 & y_3 & z_3 \\ x_2 & y_2 & z_2 \\ x_1 & y_1 & z_1 \\ x_3 & y_3 & z_3 \\ x_0 & y_0 & z_0 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_1 & y_1 & z_1 \\ x_0 & y_0 & z_0 \end{bmatrix}, \qquad \text{triangle table}: \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix}$$

# Triangle Table from Triangle Soup: Better approach

- For the $i$-th row

$$(x_i, y_i, z_i)(x_i', y_i', z_i')(x_i'', y_i'', z_i'')$$

of the triangle soup array generate three rows:

$$\begin{bmatrix} x_i & y_i & z_i & i & 0 \\ x_i' & y_i' & z_i' & i & 1 \\ x_i'' & y_i'' & z_i'' & i & 2 \end{bmatrix}$$

- Result: array with $3T$ rows, 5 numbers in each row; first three are vertex coordinates, the other two tell where the vertex came from
- Sort the array with respect to lexicographical order
- Result?

# Example

$$\left[\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{array}\right] \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 & 1 \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 3 & 0 \\ 1 & 0 & 0 & 3 & 1 \\ 0 & 1 & 0 & 3 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 3 & 0 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 & 1 \\ 0 & 1 & 0 & 3 & 2 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 3 & 1 \end{bmatrix}$$

$$\text{result}: \begin{bmatrix} 0 & 3 & 2 \\ 0 & 3 & 1 \\ 0 & 2 & 1 \\ 1 & 3 & 2 \end{bmatrix}$$

# Triangle Table from Triangle Soup: Better approach

- Vertices with the same coordinates appear in adjacent rows!
- This is because the coordinates are most significant entries (for lexicographical order) in each row
- Now, build the triangle and vertex table from the sorted array
- Vertex table: forget the last two entries and remove repeating vertices
- Triangle table:
  - `ID:=0`
  - `for i:=0 to 3T do:`
    - Let $x$, $y$, $z$, $i$, $j$ are the entries of the current row; write `ID` into the $i$-th row and $j$-th column of the triangle table
    - If the first three entries of row `i` and row `i+1` are the same, increment `ID`.

# Adjacency table from triangle table

- Similar idea (sort!)
- If $i$-th row of the triangle table is $a, b, c$, generate three rows:

$$\begin{array}{cccc} \min(a,b) & \max(a,b) & i & 2 \\ \min(a,c) & \max(a,c) & i & 1 \\ \min(b,c) & \max(b,c) & i & 0 \end{array}$$

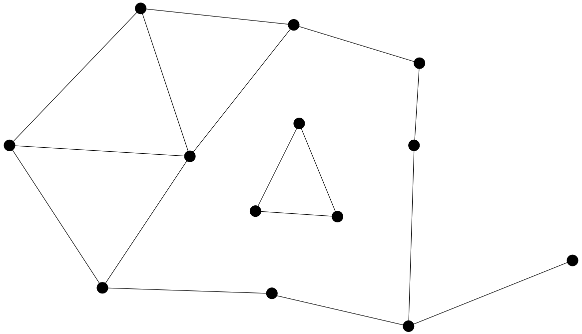
- Sort the table lexicographically
- If each edge has exactly 2 incident triangles, the rows sorted table will come in pairs with the same first two entries
- Take such pair of rows; suppose last two coordinates of these rows are $i$, $j$ and $i'$, $j'$
- Put $i$ into $(i', j')$ and $i'$ at $(i, j)$ (coordinates mean row-column number)

# Example

$$
\begin{bmatrix} 0 & 3 & 2 \\ 0 & 3 & 1 \\ 0 & 2 & 1 \\ 1 & 3 & 2 \end{bmatrix}
\rightarrow
\begin{bmatrix} 0 & 3 & 0 & 2 \\ 0 & 2 & 0 & 1 \\ 2 & 3 & 0 & 0 \\ 0 & 3 & 1 & 2 \\ 0 & 1 & 1 & 1 \\ 1 & 3 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 1 & 2 & 1 \\ 1 & 2 & 2 & 0 \\ 1 & 3 & 3 & 2 \\ 1 & 2 & 3 & 1 \\ 2 & 3 & 3 & 0 \end{bmatrix}
\rightarrow
\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 2 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 2 & 2 & 2 \\ 0 & 3 & 0 & 2 \\ 0 & 3 & 1 & 2 \\ 1 & 2 & 2 & 0 \\ 1 & 2 & 3 & 1 \\ 1 & 3 & 1 & 0 \\ 1 & 3 & 3 & 2 \\ 2 & 3 & 0 & 0 \\ 2 & 3 & 3 & 0 \end{bmatrix}
\rightarrow
\begin{bmatrix} 3 & 2 & 1 \\ 3 & 2 & 0 \\ 3 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix}
$$

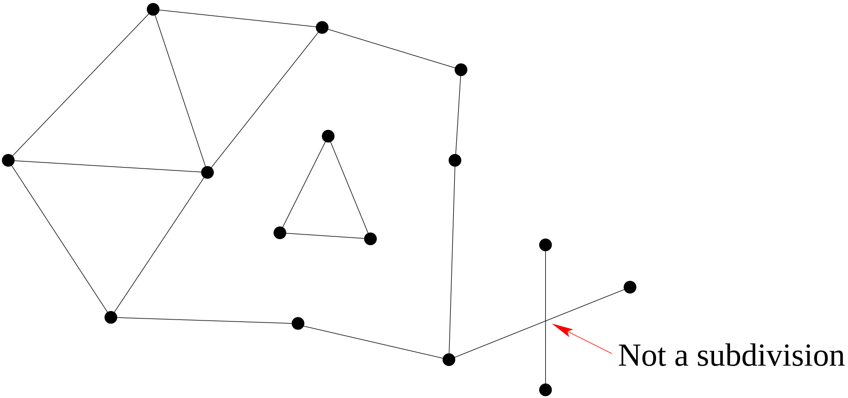
# Half-Edge Data Structure

- A general datastructure for representing planar subdivisions
- Planar subdivision is defined by a finite number of line segments in the plane. Any two segments are either disjoint or they intersect at a common endpoint

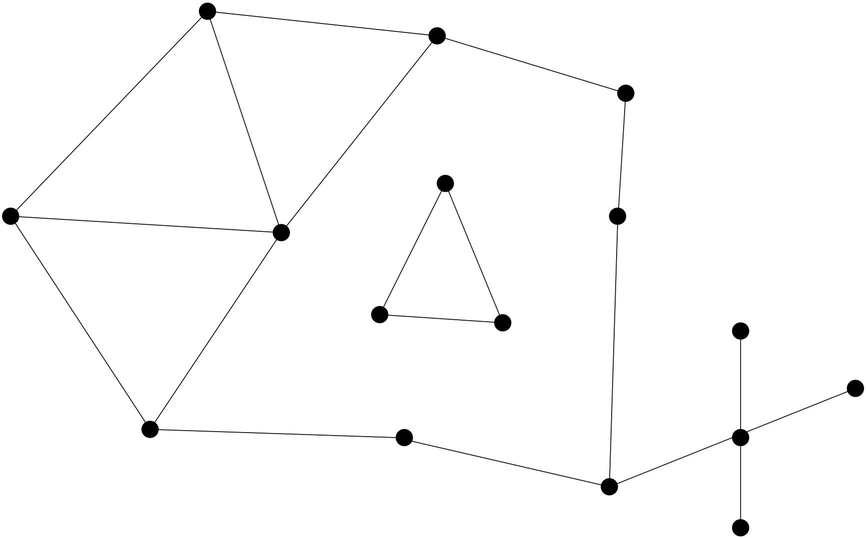


# Not a planar subdivision

- No two intervals can intersect at a point that is not a common endpoint

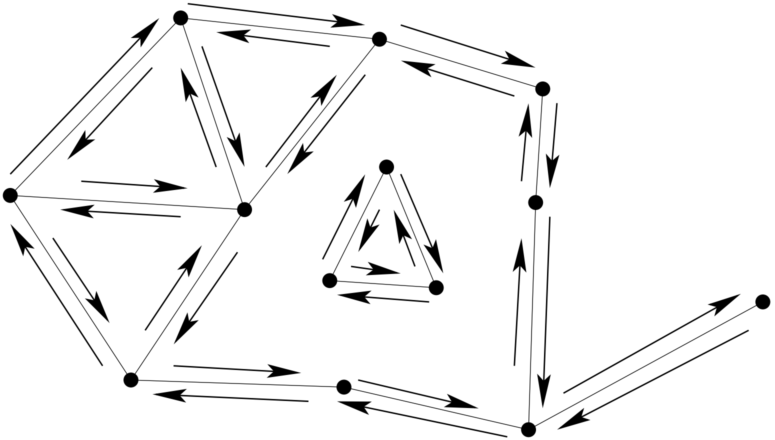


# This one is a planar subdivision



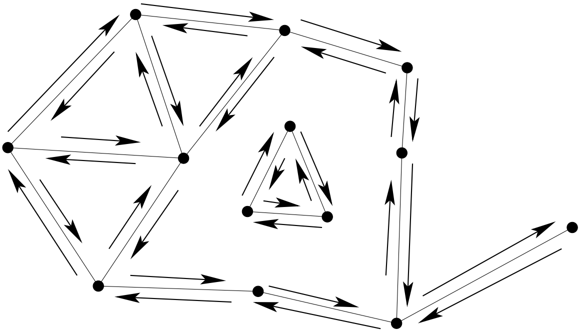
# Half-edges

- Draw arrows on both sides of the edges
- Edge on the right of the arrow



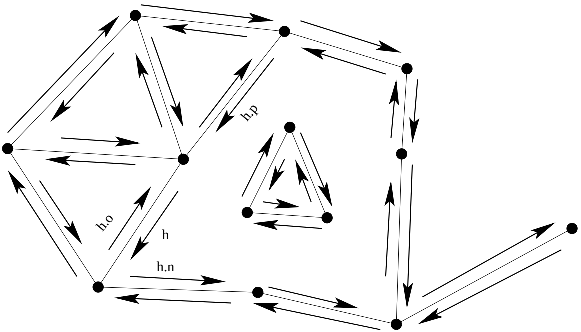
# Half-edges

- Faces: connected pieces obtained by cutting the plane along the edges
- Each face has bounding loops
  - a bounded face: one outer bounding loop and some number (possibly, zero) of inner loops
  - the unbounded face: no outer loop, some number of inner loops



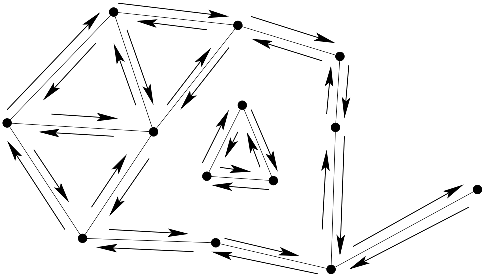
# Half-edge datastructure

- Basic building block: a half-edge (arrow)
- With each half-edge $h$ keep:
  - pointer to the next half edge (in the same face), $h.n$
  - pointer to the previous half-edge (in the same face), $h.p$
  - pointer to the opposite half-edge, $h.o$
  - pointer to the starting vertex, $h.s$
  - pointer to its face, $h.f$.



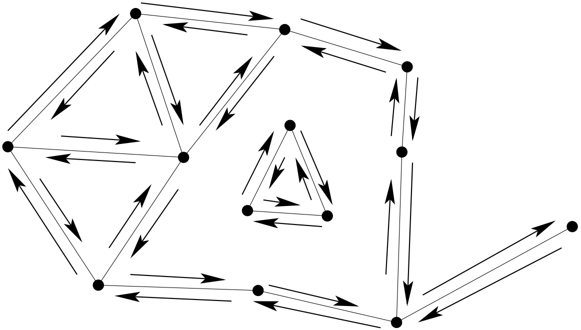
# Half-edge datastructure

- Vertex record
  - vertex data (coordinates etc)
  - pointer to a half-edge out of the vertex (an arbitrary one)
- Face record
  - face data
  - pointer to a half edge on the outer bounding loop (**nil** if the face is unbounded)
  - pointer to one half edge per inner bounding loop



# Half-edge datastructure: properties

- Efficient local queries, such as:
  - output all vertices adjacent to a given one
  - output all faces adjacent to a given face across an edge
  - output all half edges bounding a face.



# Half-edge datastructure: properties

- Easy to adapt to subdivisions with curved edges
- Easy to adapt to subdivisions on surfaces, not on the plane
- Subdivision on a surface could be defined by its polygonal mesh representation
- Useful for representing polygonal meshes