CMSC 23700	**Introduction to Computer Graphics**	**Handout 2**
Autumn 2015		**November 12**

Notes on polygon meshes

1 Basic definitions

Definition 1 *A* polygon mesh *(or* polymesh*) is a triple* (V, E, F)*, where*

$$\begin{array}{lll} V & & \textit{a set of vertices (points in space)} \\ E & \subset \ (V \times V) & \textit{a set of edges (line segments)} \\ F & \subset \ E^* & \textit{a set of faces (convex polygons)} \end{array}$$

with the following properties:

1. *for any* $v \in V$*, there exists* $(v_1, v_2) \in E$ *such that* $v = v_1$ *or* $v = v_2$*.*
2. *for and* $e \in E$*, there exists a face* $f \in F$ *such that* e *is in* f*.*
3. *if two faces intersect in space, then the vertex or edge of intersection is in the mesh.*

If all of the faces of a polygon mesh are triangles, then we call it a *triangle mesh* (*trimesh*). Polygons can be *tessellated* to form triangle meshes.

Definition 2 *We classify edges in a mesh based on the number of faces they are part of:*

- *A* boundary edge *is part of exactly one face.*
- *An* interior edge *is part of two or more faces.*
- *A* manifold edge *is part of exactly two faces.*
- *A* junction edge *is part of three or more faces.*

Junction edges are to be avoided; they can cause cracks when rendering the mesh.

Definition 3 *A polymesh is* connected *if the undirected graph* $G = (V_F, E_E)$*, called the* dual graph*, is connected, where*

- V_F *is a set of graph vertices corresponding to the faces of the mesh and*
- E_E *is a set of graph edges connecting adjacent faces.*

Definition 4 *A* polyhedron *is a polymesh that is*

1. *connected and*
2. *each edge is manifold.*

Definition 5 *A* polytope *is a polyhedron that encloses a convex region R of* $\mathbb{R}^3$ *(*i.e.*, any two points in R are connected by a line segment that is wholly contained in R).*

Definition 6 *A connected mesh is* manifold *if every edge in the mesh is either a boundary edge or a manifold edge.*

For most computer graphic applications, we use manifold meshes.

Definition 7 *A manifold mesh is* closed *if every edge is manifold and it is non-intersecting.*

2 Orientation

The *orientation* of a face determines which side is the front and which side is the back. The orientation can either be Counter Clockwise (CCW, which is the OpenGL default) or Clockwise (CW).

Definition 8 *Two faces, f_1 and f_2, that share a common edge e are* consistently *oriented if the head of e in f_1 is the tail of e in f_2 (and vice versa).*

Definition 9 *A manifold mesh is* orientable *if the vertex orderings of its faces can be chosen so that adjacent faces have consistent orderings (*i.e.*, all faces are either CW or CCW).*

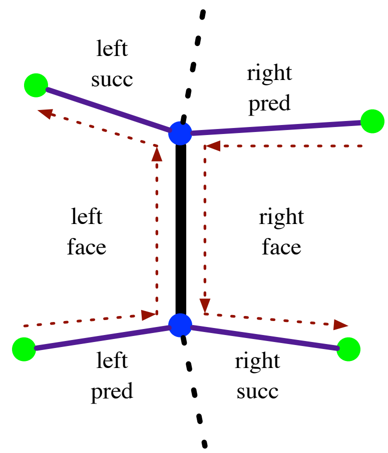
3 Data structures

The data structures used to represent meshes vary by application. For rendering purposes, one might use a wireframe representation (just vertices and edges), or a trimesh representation (vertices and triangles). If we want to do more substantial computation with the mesh, we need to be able to efficiently answer geometric queries, such as

- find the edges or vertices of a face
- find the neighboring vertices of a vertex
- find the faces of an edge
- find the edges of a vertex (c.f., silhouette edges)
- find the next edge in a path around a face

3.1 Winged-edge model

One popular representation is the *winged-edge* data structure. In this representation, the edge is the central part of the representation.



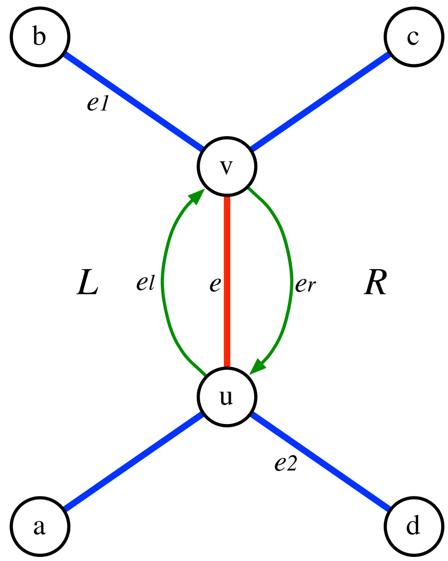
In C, we might use the following pointer-based representation:

```
struct edge {
    struct vertex *v0;     /* endpoints of edge */
    struct vertex *v1;
    struct face   *left;   /* face on left-hand-side of edge */
    struct face   *right;  /* face on right-hand-side of edge */
    struct edge   *lPred;  /* left-most predecessor edge */
    struct edge   *rPred;  /* right-most predecessor edge */
    struct edge   *lSucc;  /* left-most successor edge */
    struct edge   *rSucc;  /* right-most successor edge */
};

struct vert {
    struct edge *e;
    ...
};

struct face {
    struct edge *e;
    ...
};
```

In a graphical application, we will store other information with vertices and faces (colors, normals, texture coordinates, ...), hence the "`...`" in the code. For models where the mesh is static, we can use a table-based representation that is more compact (assuming that we can use the `char` or `short` type as table indices).

3.2 Directed-edge model

The *directed-edge* (or *half-edge*) model splits the representation of each edge into two oriented parts. Consider the following fragment of a mesh:



The half-edge representation uses two half-edges (e_l and e_r) to represent the edge e. For e_l, we record the source vertex (u), the face to the left (L), the next half-edge in the CCW tour of L (e_1), and the other half of the edge (e_r). Likewise, for e_r we record v, R, e_2, and e_l. For a vertex we record an edge that it is a source of, and for a face we record an edge that the face lies to the left of. The C data structures for half-edges look like the following:

```
struct Edge {
    Vertex     *vert;  // source vertex of edge
    Face       *face;  // face to left of edge
    Edge       *next;  // next edge in tour around face
    Edge       *pair;  // half-edge going the other way
};

struct Vertex {
    Edge       *edge;  // An edge with this vertex as its source
    vec3f      v;      // The position of the vertex
    ...
};

struct Face {
    Edge       *edge;  // An edge of the face
    ...
};
```

In this representation, boundary edges will have a null pair pointer.

When restricted to triangle meshes, the directed-edge model can be made very space efficient (at a slight cost in time). Since each half-edge belongs to exactly one triangle, we can group them in triplets and use arithmetic to determine the triangle of an edge, the edges of a triangle, and the next and previous edges of a tour:

$$\begin{aligned}
\mathrm{tri}(e) &= e\,\mathbf{div}\,3 \\
\mathrm{edges}(t) &= (3t, 3t+1, 3t+2) \\
\mathrm{prev}(e) &= \left\{ \begin{array}{ll} e+2 & \text{if } e\,\mathbf{mod}\,3 = 0 \\ e-1 & \text{otherwise} \end{array} \right\} \\
\mathrm{next}(e) &= \left\{ \begin{array}{ll} e-2 & \text{if } e\,\mathbf{mod}\,3 = 2 \\ e+1 & \text{otherwise} \end{array} \right\}
\end{aligned}$$

The edge representation can then become very compact:

```
struct Edge {
    struct Vertex *vert;   // source vertex of edge
    int            pair;   // half-edge going the other way
};
```