

Geometric Modeling in Graphics

Part 1: Polygonal Meshes

Geometric object

- ▶ Set of connected points in space
- ▶ Usually inside Euclidean space (orthonormal basis, coordinates, inner product, norm, distance, angle, ...)
- ▶ Topological dimension – 0D, 1D, 2D, 3D objects
- ▶ Topological dimension defined by open covers

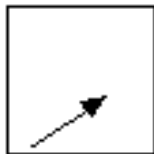
Point



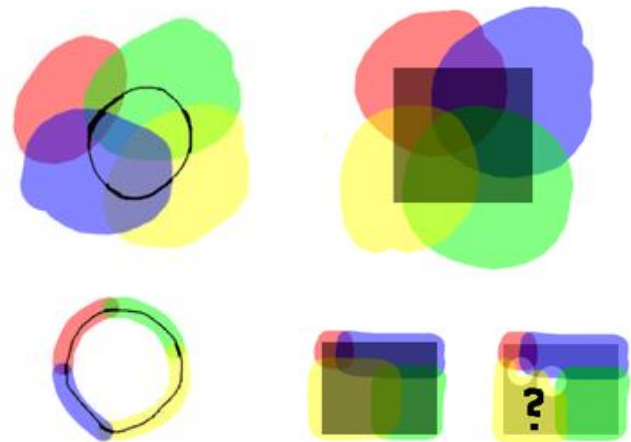
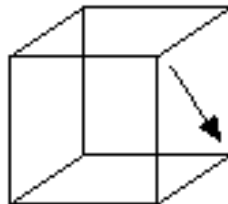
Line



Square



Cube



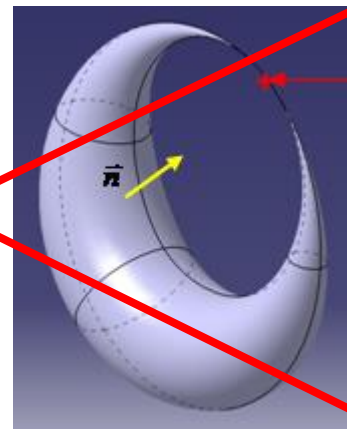
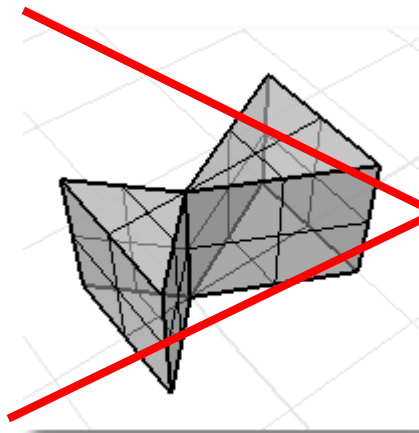
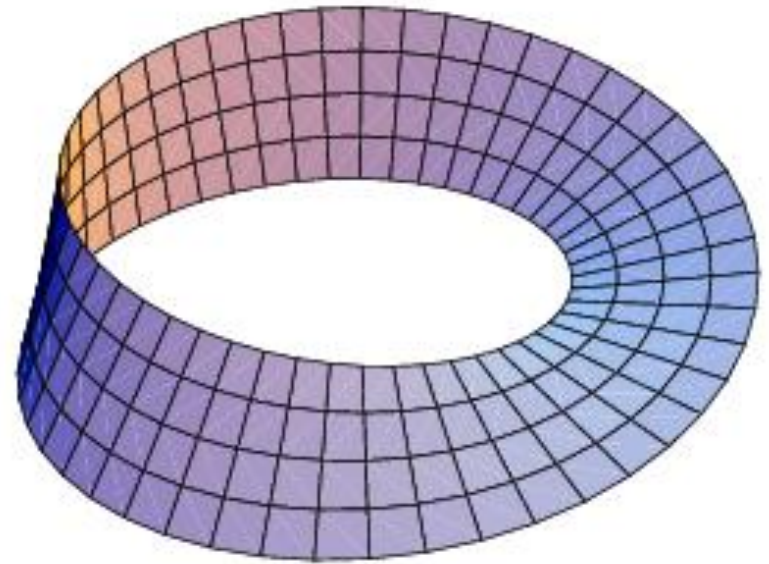
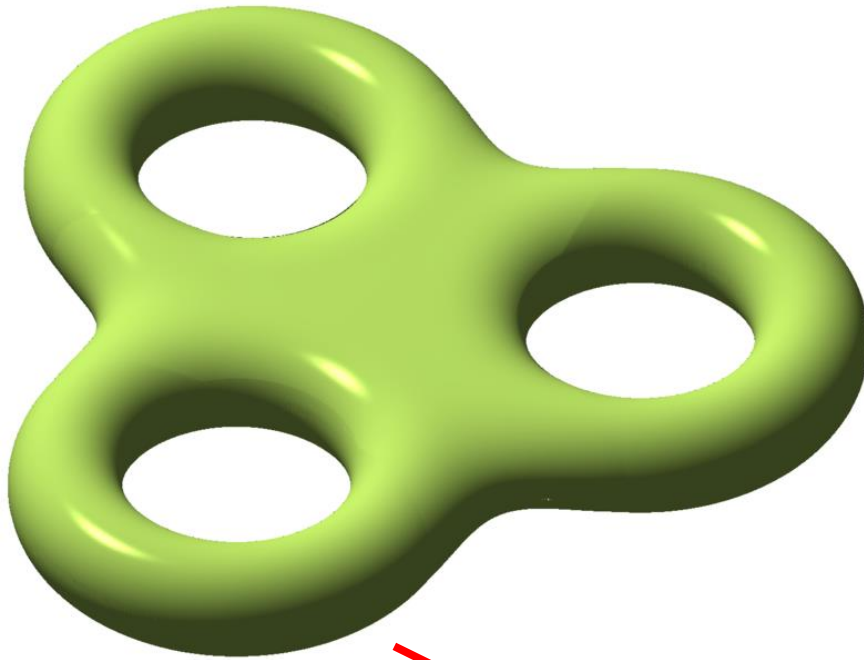
wikipedia.org

Manifold

- ▶ n-manifold – set of points locally homeomorphic to n-dimensional Euclidean space
- ▶ Manifold resembles Euclidean space near each point
- ▶ For each point of n-manifold there exists his neighborhood homeomorphic with open n-dimensional ball $\mathbf{B}^n = \{(x_1, x_2, \dots, x_n) \in \mathbb{R}^n \mid x_1^2 + x_2^2 + \dots + x_n^2 < 1\}$.
- ▶ n-manifold is n dimensional object
- ▶ Homeomorphism – continuous function with continuous inverse function, means topological equivalence

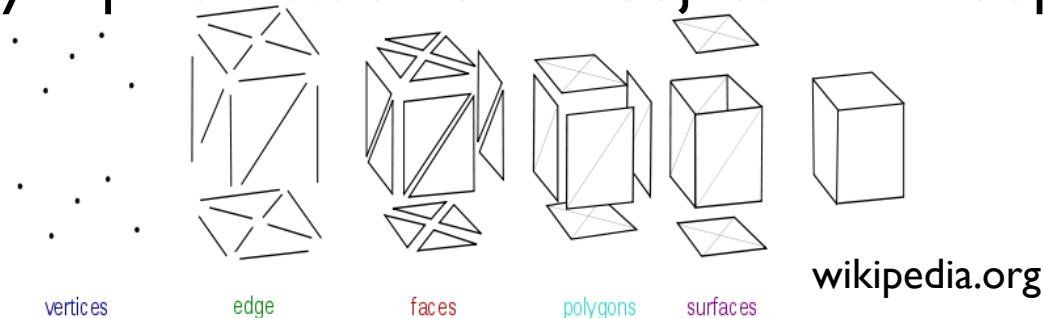


Manifolds & non-manifolds



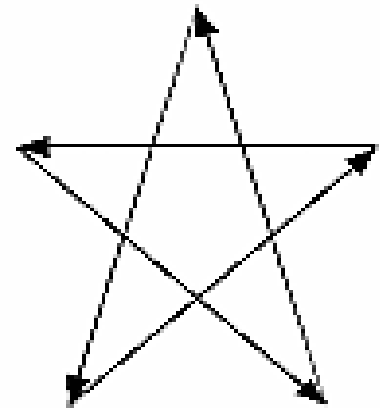
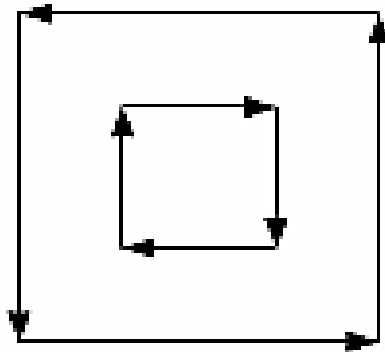
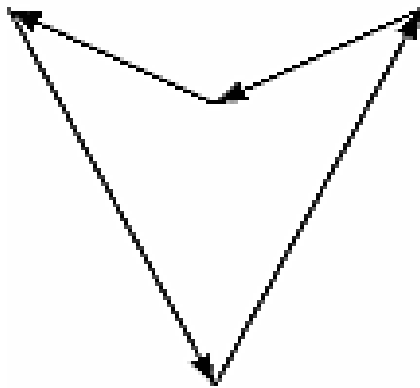
Polygonal mesh

- ▶ Boundary representation of 3D object (polyhedron) or representation of 2D object (surface)
- ▶ Boundary represented as set of polygons (**faces**)
- ▶ Each polygon defined by ordered set of **vertices**
 - ▶ Vertices – coordinates – geometric information
 - ▶ Order of vertices – topological information
- ▶ Possible additional element = **edges** – connecting 2 consecutive vertices in polygon
- ▶ Edges are shared between several neighboring polygons
- ▶ Boundary representation of 2D object – line loop



Extended polygonal mesh

- ▶ Extended faces with holes and self intersecting edges
- ▶ Each face is originally defined as set of contours
- ▶ Type of representation in some modeling packages
- ▶ Can be transformed to simplified mesh with faces without holes – using tessellation algorithms (GLU tessellation, CGAL, Visualization Library, ...)



Polygonal mesh orientation

- ▶ Edge orientation – order of two vertices
- ▶ Polygon orientation – order of vertices (edges) that defines polygon boundary
- ▶ Polygonal mesh orientation – given by orientation of faces, such that polygons on common edge have opposite orientation
- ▶ If orientation exists – orientable – have both sides
- ▶ Computation of orientable area, volume

Oriented Area of simple polygon

$[x_i, y_i]$ is i-th vertex

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

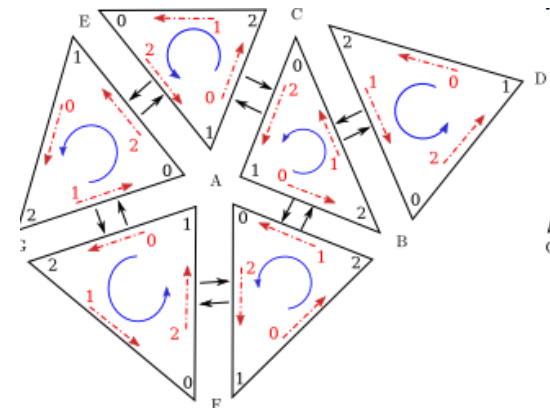
Oriented Volume of convex polyhedron

x_i is any point of i-th face

A_i is area of i-th face

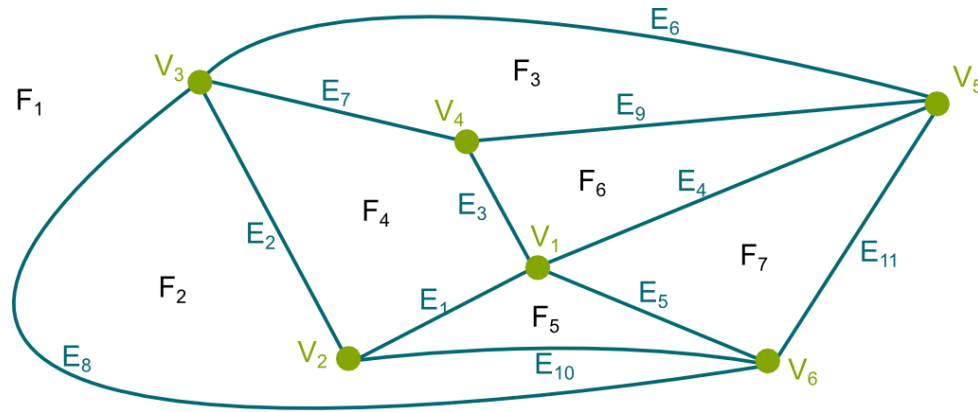
n_i is normal of i-th face

$$\text{volume} = \frac{1}{3} \sum_{\text{face } i} \vec{x}_i \cdot \hat{n}_i A_i$$



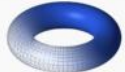


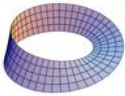
Euler characteristic






- ▶ Boundary representation of 3D object using 2-manifold polygonal mesh
- ▶ Oriented 2-manifold polygonal mesh
- ▶ Works also for planar graphs
- ▶ Genus g – number of holes in 3D object
- ▶ V, E, F – number of vertices, edges, faces in mesh
- ▶ $V - E + F = 2 - 2g$



$$F - E + V = 7 - 11 + 6 = 2$$

Euler characteristic

Name	Image	Euler characteristic
Sphere		2
Torus		0
Double torus		-2
Triple torus		-4
Real projective plane		1
Möbius strip		0
Klein bottle		0
Two spheres (not connected)		$2 + 2 = 4$

Name	Image	Vertices V	Edges E	Faces F	Euler characteristic: $V - E + F$
Tetrahedron		4	6	4	2
Hexahedron or cube		8	12	6	2
Octahedron		6	12	8	2
Dodecahedron		20	30	12	2
Icosahedron		12	30	20	2

wikipedia.org

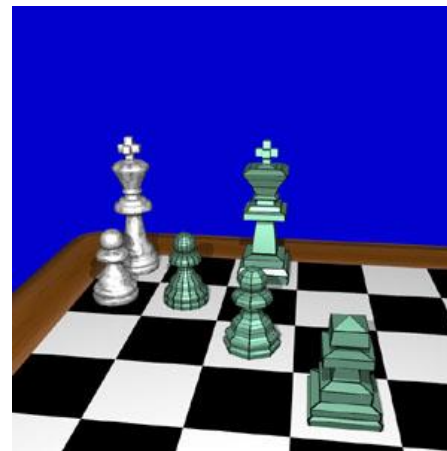
Polygonal mesh structures

- ▶ Structures representing vertices, edges, faces
- ▶ Memory complexity of structures
- ▶ Optimizing algorithms on these structures
- ▶ Algorithms for creation and update
- ▶ Geometric algorithms
 - ▶ Transformations, intersections
- ▶ Topological algorithms
 - ▶ Finding neighborhood elements
- ▶ Visualization algorithms
 - ▶ Usually using graphics cards and 3D APIs

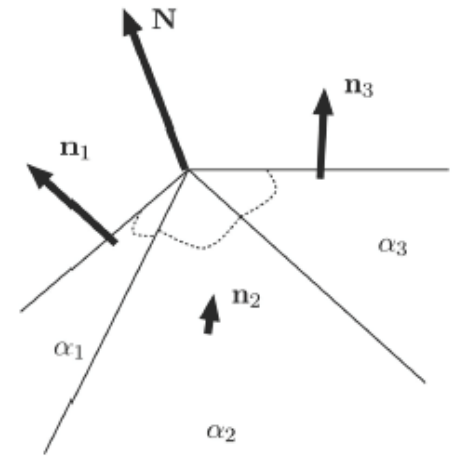
Topological algorithms

- ▶ Find elements (vertices, edges, faces) that are connected with given element
- ▶ Connected through k other elements = searching in k -ring neighborhood
- ▶ Used frequently in many modeling algorithms

	Vertex	Edge	Face
Vertex	VV	VE	VF
Edge	EV	EE	EF
Face	FV	FE	FF



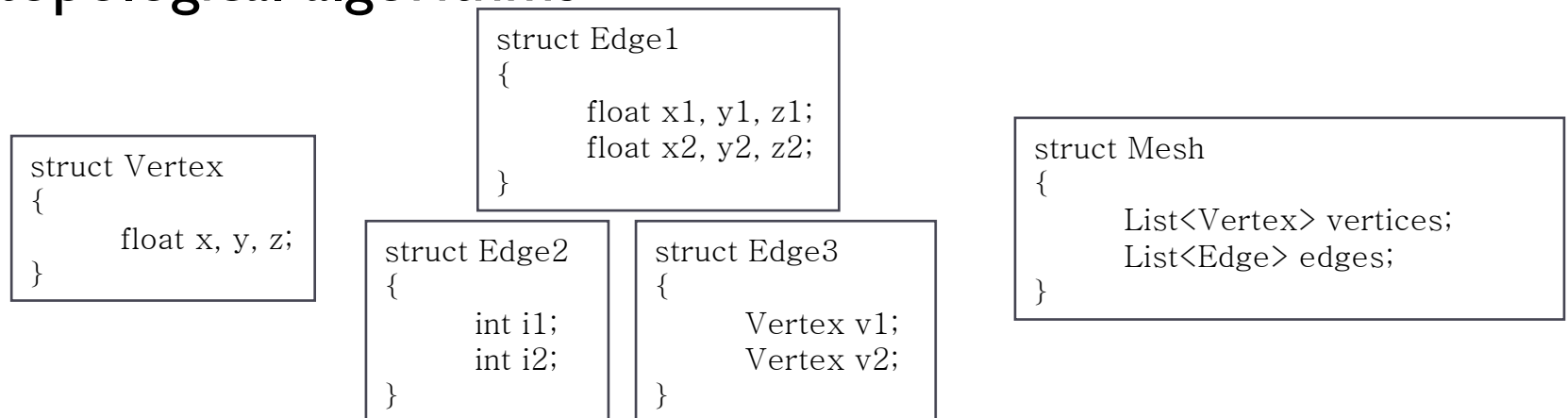
subdivision surfaces



computation of vertex normals

Edge-Vertex meshes

- ▶ Simple representation of polygonal mesh
- ▶ Structure containing two sets
 - ▶ List of vertices
 - ▶ List of edges, where each edge is given by two vertices = two pointers to list of vertices, several types of pointer
- ▶ No implicit representation of faces
- ▶ No information of neighboring elements = slow topological algorithms



Face-Vertex meshes

- ▶ Structure containing two sets
 - ▶ List of vertices
 - ▶ List of faces, where each face is given as ordered list of vertices
- ▶ Order of vertices (edges) in face – orientation of face
- ▶ No implicit representation of edges, but can be added third list of edges
- ▶ No information of neighboring elements = slow topological algorithms

```
struct Vertex
{
    float x, y, z;
}
```

```
struct Edge2
{
    Vertex v1, v2;
}
```

```
struct Face1
{
    List<int> vertices;
}
```

```
struct Face2
{
    List<Edge2> edges;
}
```

```
struct Mesh1
{
    List<Vertex> vertices;
    List<Face1> faces;
}
```

```
struct Mesh2
{
    List<Vertex> vertices;
    List<Edge2> edges;
    List<Face2> faces;
}
```

Face-Vertex meshes

- ▶ Minimal structure for representing vertices, edges and faces
- ▶ Structure best suitable for visualization using graphics card and serialization using files
- ▶ File formats for meshes – Collada, 3DS, OBJ, VRML, ...

```
<mesh>
  <source id="box-lib-positions" name="position">
    <float_array id="box-lib-positions-array" count="24">-1 1 1 1 1 1 -1 -1 1 1 1 1 -1 1 1 1 -1 -1 1 1 1 1 -1 -1</float_array>
    <technique_common>
      <accessor count="8" source="#box-lib-positions-array" stride="3">
        <param name="X" type="float"/>
        <param name="Y" type="float"/>
        <param name="Z" type="float"/>
      </accessor>
    </technique_common>
  </source>
  <vertices id="box-lib-vertices">
    <input semantic="POSITION" source="#box-lib-positions"/>
  </vertices>
  <polylist count="6" material="BlueSG">
    <input offset="0" semantic="VERTEX" source="#box-lib-vertices"/>
    <vcount>4 4 4 4 4 4 </vcount>
    <p>0 2 3 1 0 1 5 4 6 7 3 2 0 4 6 2 3 7 5 1 5 7 6 4 </p>
  </polylist>
</mesh>
```

Face-Vertex meshes

- ▶ Visualization using modern graphics card and 3D APIs (Direct 3D, OpenGL, ...)
- ▶ Using simple list (array) of vertex attributes and list (array) of triangles (polygons)
- ▶ Polygon is given as list of indices to vertex array
- ▶ Use 3D API to send these arrays to graphic card

```
struct VisualizationVertex
{
    float x, y, z;
    // uv coordinates, normals
}
```

```
struct VisualizationTriangle
{
    int i, j, k;
}
```

```
struct VisualizationMesh
{
    int num_vertices;
    Vertex[] vertices;
    int num_triangles;
    Triangle[] triangles;
}
```



Face-Vertex meshes

► Topological algorithms

	Vertex	Edge	Face
Vertex	$O(m)$	$O(m)$	$O(l+k)$
Edge	$O(l)$	$O(m)$	$O(l+k)$
Face	$O(k)$	$O(k)$	$O(m+k)$

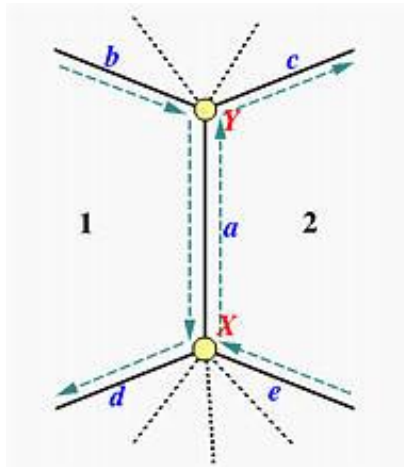
- **n** – number of vertices
- **m** – number of edges
- **l** – number of faces
- **k** – maximal number of vertices (edges) for face

```
FaceVertexMeshFF(Face2 face, Mesh2 mesh)
{
    List<Face2> result;
    for (int i = 0; i < mesh.faces.size(); i++)
        mesh.edges[i].f1 = mesh.edges[i].f2 = NULL;
    for (int i = 0; i < mesh.faces.size(); i++)
        for (int j = 0; j < mesh.faces[i].edges.size(); j++)
        {
            if (mesh.faces[i].edges[j].f1 == NULL)
                mesh.faces[i].edges[j].f1 = mesh.faces[i];
            else if (mesh.faces[i].edges[j].f2 == NULL)
                mesh.faces[i].edges[j].f2 = mesh.faces[i];
        }
    for (int i = 0; i < face.edges.size(); i++)
    {
        if (face == face.edges[i].f1 && face.edges[i].f2 != NULL)
            result.add(face.edges[i].f2);
        if (face == face.edges[i].f2 && face.edges[i].f1 != NULL)
            result.add(face.edges[i].f1);
    }
    return result;
}
```

```
struct Edge2
{
    Vertex v1, v2;
    Face2 f1;
    Face2 f2;
}
```


Winged Edge

- ▶ Structure for representing polygonal orientable 2-manifold mesh
- ▶ Lists of vertices, edges (winged edges), faces
- ▶ Structure for vertex and face contains only one pointer to one incident edge + coordinates of vertex
- ▶ Extended incident data for edge structure, its members are given by edge and polyhedron orientation



- a* – current edge
- X* – begin vertex of current edge
- Y* – end vertex of current edge
- b* – previous edge in orientation from left face
- d* – next edge in orientation from left face
- c* – next edge in orientation from right face
- e* – previous edge in orientation from right face
- 1 – left face
- 2 – right face

Winged Edge

- ▶ Possibility to store only next edges (c, d) from left and right faces, removing previous faces (b, e)
- ▶ For extended meshes store one edge of each contour inside each face
- ▶ Visualization & file serialization – covert between face-vertex mesh and winged-edge mesh

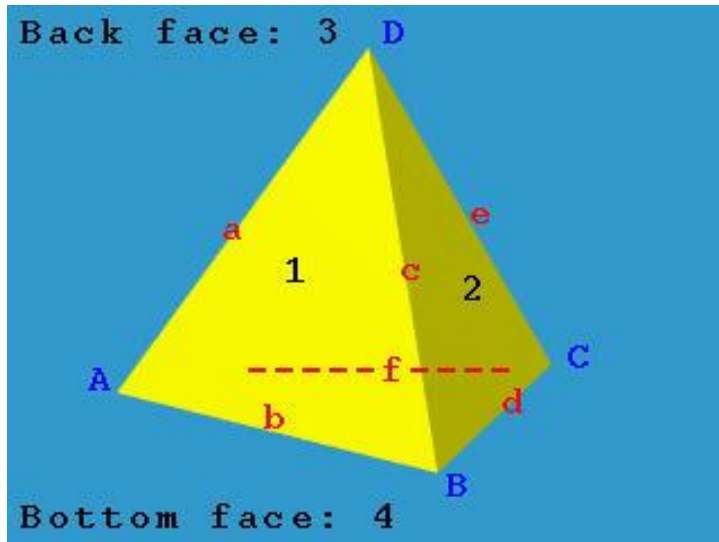
```
struct Vertex
{
    float x, y, z;
    WingedEdge edge;
}
```

```
struct Face
{
    WingedEdge outer_edge;
    //List<WingedEdge> inner_edges;
}
```

```
struct WingedEdge
{
    Vertex X;
    Vertex Y;
    WingedEdge b;
    WingedEdge c;
    WingedEdge d;
    WingedEdge e;
    Face 1;
    Face 2;
}
```

```
struct WingedEdgeMesh
{
    List<Vertex> vertices;
    List<WingedEdge> edges;
    List<Face> faces;
}
```

Winged Edge example



Edge	Vertices		Faces		Left Traverse		Right Traverse	
Name	Start	End	Left	Right	Pred	Succ	Pred	Succ
a	A	D	3	1	e	f	b	c
b	A	B	1	4	c	a	f	d
c	B	D	1	2	a	b	d	e
d	B	C	2	4	e	c	b	f
e	C	D	2	3	c	d	f	a
f	A	C	4	3	d	b	a	e

Vertex Name	Incident Edge
A	a
B	b
C	d
D	e

Face Name	Incident Edge
1	a
2	c
3	a
4	b

<http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html>

Winged Edge

- All topological algorithms in constant time, higher memory

```
WingedEdgeFF(Face face) {
    WingedEdge start_edge = face.outer_edge;
    WingedEdge current_edge;
    if (start_edge.1 == face) {
        result.Add(start_edge.2);
        current_edge = start_edge.d;
    }
    else if (start_edge.2 == face) {
        result.Add(start_edge.2);
        current_edge = start_edge.c;
    }
    else return;
    while (current_edge != start_edge) {
        if (current_edge.1 == face) {
            result.Add(current_edge.2);
            current_edge = current_edge.d;
        }
        else if (current_edge.2 == face) {
            result.Add(current_edge.1);
            current_edge = current_edge.c;
        }
    }
    return result;
}
```

```
WingedEdgeVE(Vertex vertex) {
    WingedEdge start_edge = vertex.edge;
    WingedEdge current_edge;
    WingedEdge prev_edge = start_edge;
    if (vertex == start_edge.X)
        current_edge = start_edge.d;
    else
        current_edge = start_edge.c;
    result.Add(start_edge);
    while (current_edge != start_edge) {
        result.Add(current_edge);
        if (vertex == current_edge.X) {
            if (prev_edge == current_edge.e)
                current_edge = current_edge.d;
            else
                current_edge = current_edge.e;
        }
        else {
            if (prev_edge == current_edge.c)
                current_edge = current_edge.b;
            else
                current_edge = current_edge.c;
        }
        prev_edge = result.Last();
    }
    return result;
}
```

Quad Edge

- ▶ Structure used mainly for representing graphs and its dual graphs – flipping vertices and faces
- ▶ Structure for vertex and face is almost the same, represented by same pointer
- ▶ List of data (vertices and faces) and list of quad edges
- ▶ New structure – half edge – connection from start vertex of edge to end vertex of edge or from one face to second face over edge
- ▶ Half edge holds starting data pointer (element) and pointer to next half-edge around starting vertex or edge
- ▶ Set of 4 half edges – quad edge

Quad Edge

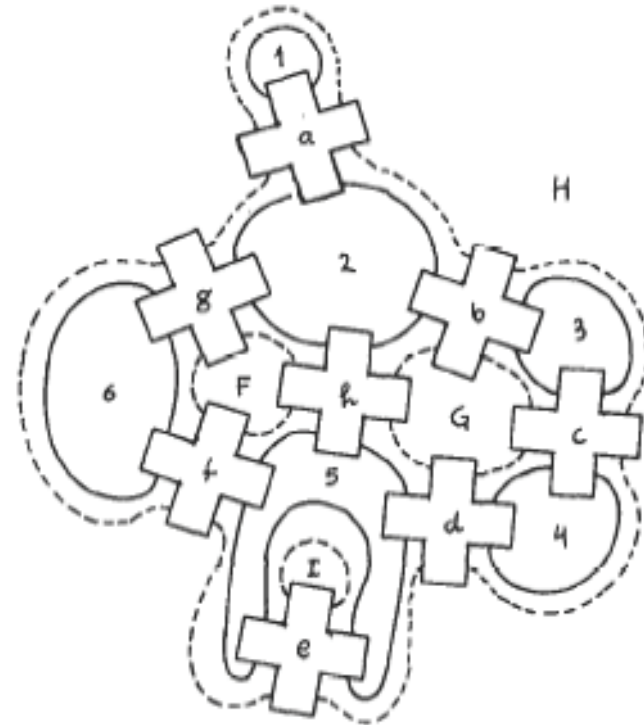
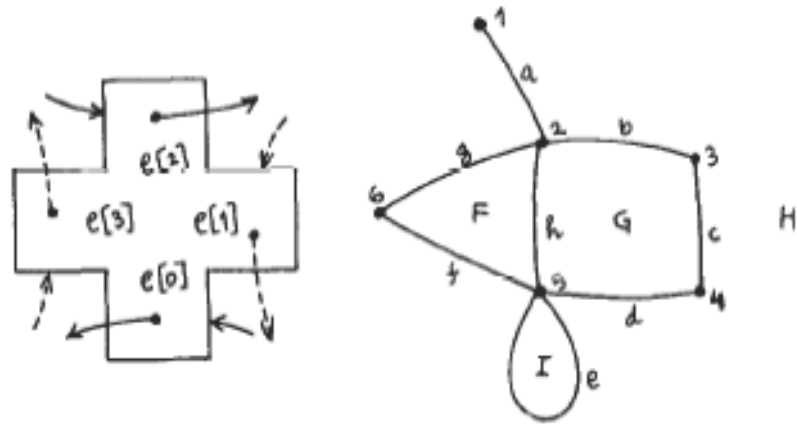
```
struct Vertex: Data
{
    float x, y, z;
    HEdge edge;
}
```

```
struct Face: Data
{
    HEdge edge;
}
```

```
struct QuadEdge
{
    HEdge e[4];
}
```

```
struct HEdge
{
    HEdge next; // Onext
    Data data; // vertex, face info
    QuadEdge parent;
}
```

```
struct QuadEdgeMesh
{
    List<Data> vertices_and_faces;
    List<QuadEdge> edges;
}
```

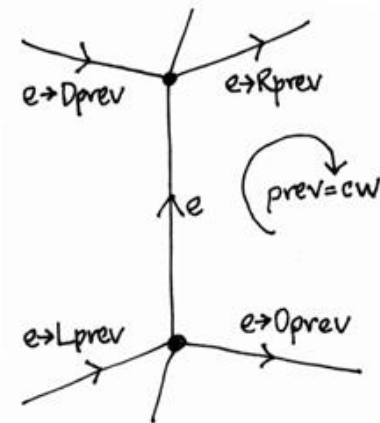
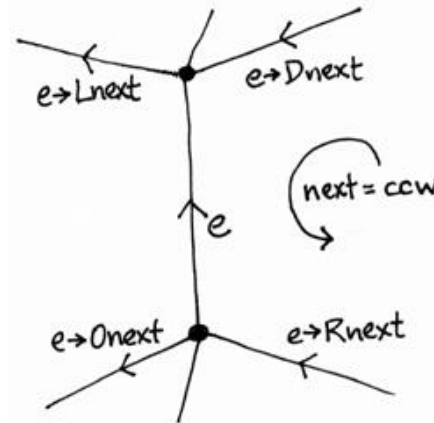
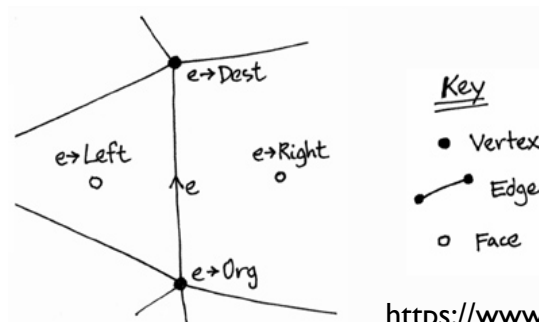


Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi.

Algebra on edges

► Function for given half edge:

- Rot – rotating half edge by 90° counterclockwise
- Sym – symmetrical half edge
- Next – next half edge; can be around origin, destination, left, right object of given half edge (Onext, Dnext, Lnext, Rnext)
- Prev – previous half edge, again around four elements
- Org – origin element, where half edge starts
- Dest – destination element, where half edge ends
- Left – element to the left of half edge
- Right – element to the right of half edge



<https://www.cs.cmu.edu/afs/andrew/scs/cs/15-463/2001/pub/src/a2/quadedge.html>

Algebra on edges

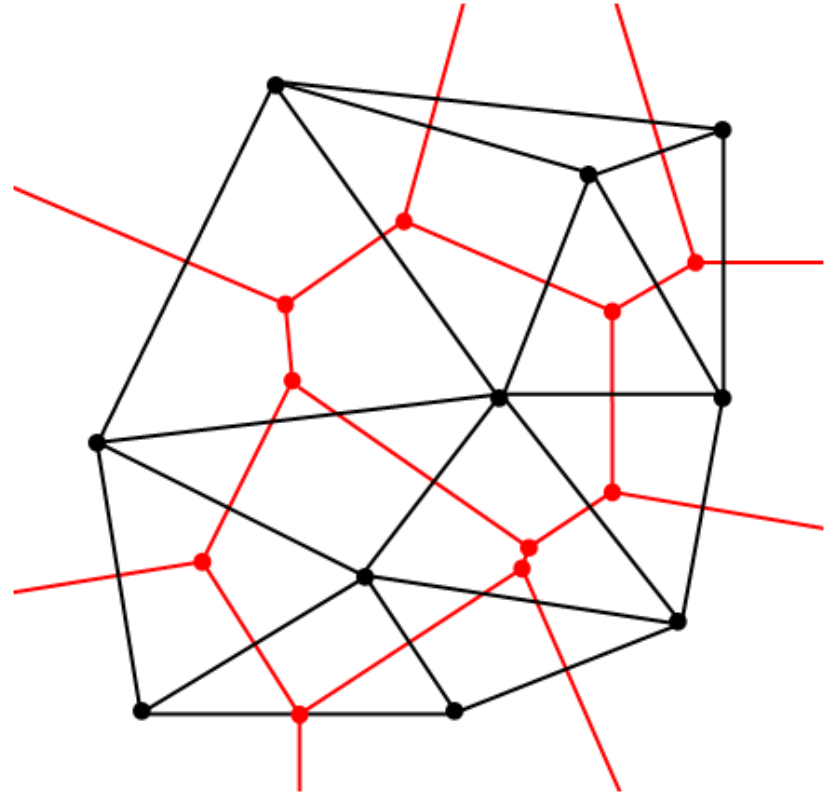
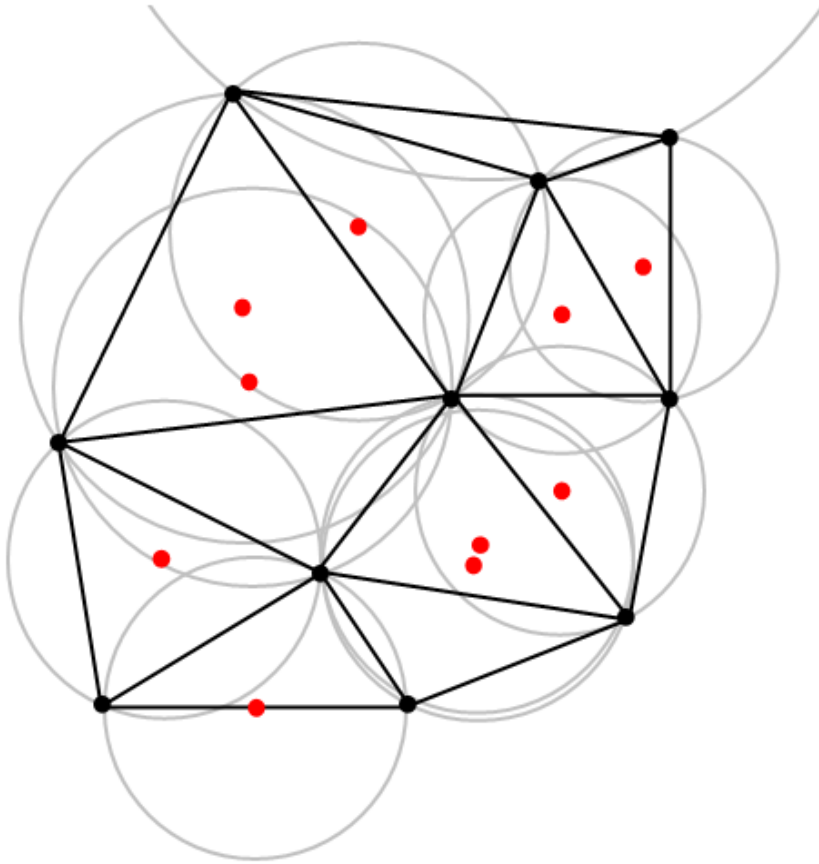
- ▶ $\text{Rot}(e) = e \rightarrow \text{parent} \rightarrow e[(r+1) \bmod 4];$ // r is index of e in $e \rightarrow \text{parent}$ QuadEdge
 - ▶ $\text{Sym}(e) = \text{Rot}(\text{Rot}(e)) = e \rightarrow \text{parent} \rightarrow e[(r+2) \bmod 4];$
 - ▶ $\text{Org}(e) = e \rightarrow \text{data};$
 - ▶ $\text{Dest}(e) = \text{Sym}(e) \rightarrow \text{data};$
 - ▶ $\text{Rot-1}(e) = e \rightarrow \text{parent} \rightarrow e[(r+3) \bmod 4] = \text{Rot}(\text{Rot}(\text{Rot}(e)));$
 - ▶ $\text{Right}(e) = \text{Rot-1}(e) \rightarrow \text{data};$
 - ▶ $\text{Left}(e) = \text{Rot}(e) \rightarrow \text{data};$
 - ▶ $\text{Onext}(e) = e \rightarrow \text{next};$
 - ▶ $\text{Oprev}(e) = \text{Rot}(\text{Onext}(\text{Rot}(e)));$
 - ▶ $\text{Dnext}(e) = \text{Sym}(\text{Onext}(\text{Sym}(e)));$
 - ▶ $\text{Dprev}(e) = \text{Rot-1}(\text{Onext}(\text{Rot-1}(e)));$
 - ▶ $\text{Lnext}(e) = \text{Rot}(\text{Onext}(\text{Rot-1}(e)));$
 - ▶ $\text{Lprev}(e) = \text{Sym}(\text{Onext}(e));$
 - ▶ $\text{Rnext}(e) = \text{Rot-1}(\text{Onext}(\text{Rot}(e)));$
 - ▶ $\text{Rprev}(e) = \text{Onext}(\text{Sym}(e));$
- Only Rot a Onext is needed, all other operators can be computed
 - That is reason for only `next` and `parent` members in QuadEdge structure

Quad Edge

```
QuadEdgeFF(Face face)
{
    HEdge start_edge = face.edge;
    result.Add(Sym(start_edge).data);
    HEdge current_edge = Onext(start_edge); // = start_edge.next
    while (current_edge && current_edge != start_edge)
    {
        result.Add(Sym(current_edge).data);
        current_edge = Onext(current_edge);
    }
    return result;
}
```

```
QuadEdgeVE(Vertex vertex)
{
    HEdge start_edge = vertex.edge;
    result.Add(start_edge);
    HEdge current_edge = Onext(start_edge); // = start_edge.next
    while (current_edge && current_edge != start_edge)
    {
        result.Add(current_edge);
        current_edge = Onext(current_edge);
    }
    return result;
}
```

Delaunay-Voronoi dual graphs



DCEL and Half-Edge

- ▶ Solving problems with orientation in Winged Edge
- ▶ Breaking each edge into two half-edges, „arrows“ or oriented edges
- ▶ DCEL - Double Connected Edge List for 2-manifold polygonal mesh, contains list of vertices, faces and half edges
- ▶ Each Half-edge contains
 - ▶ Pointer to opposite or twin half-edge, together they form whole edge, can be NULL if there is no opposite half-edge
 - ▶ Pointer to vertex where this half-edge starts (or ends)
 - ▶ Pointer to face where half-edge belongs, direction of half-edge is given by orientation inside this face
 - ▶ Pointer to next half-edge in orientation of half-edge's face

DCEL

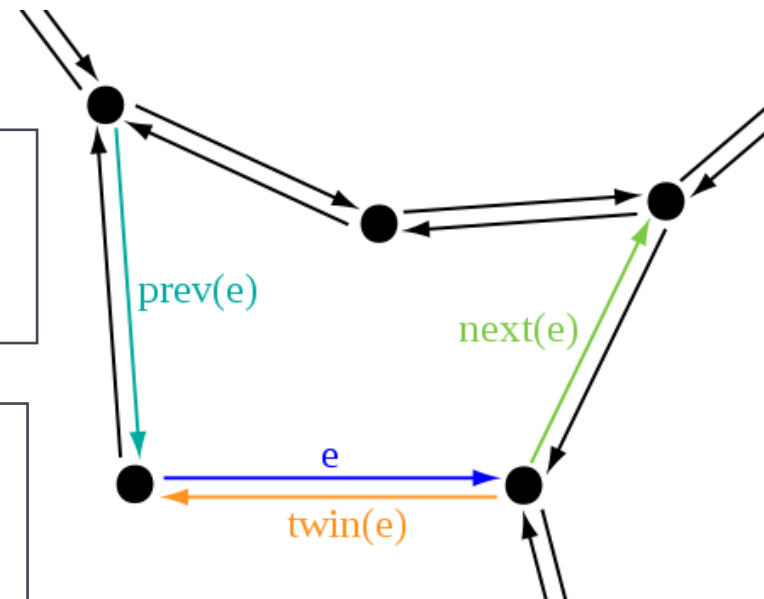
- Can represent also extended polygonal meshes – face then contains one half-edge for each contour

```
struct Vertex
{
    float x, y, z;
    HalfEdge edge;
}
```

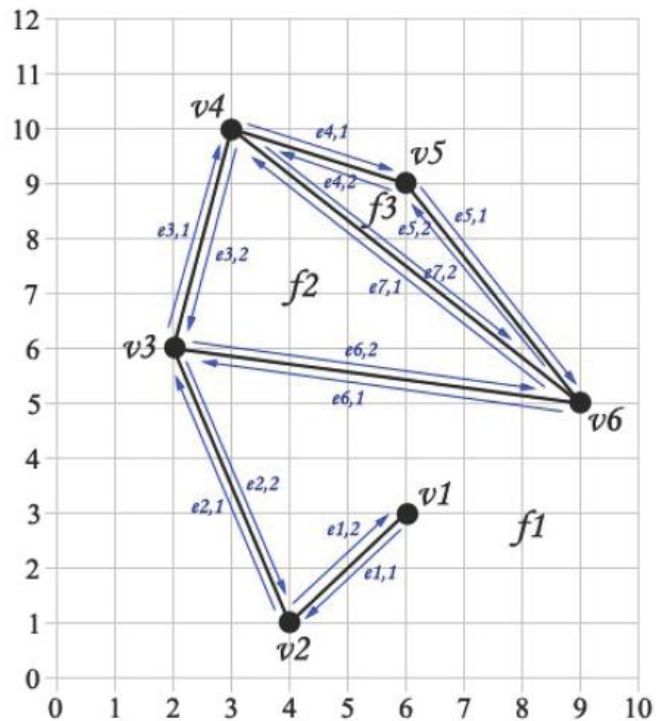
```
struct Face
{
    HalfEdge outer_edge;
    //List< HalfEdge> inner_edges;
}
```

```
struct HalfEdge
{
    Vertex origin;
    HalfEdge opp;
    HalfEdge next;
    //HalfEdge prev;
    Face face;
}
```

```
struct DCEL
{
    List<Vertex> vertices;
    List<HalfEdge> edges;
    List<Face> faces;
}
```



DCEL example



Half-Edge	Origin	Twin	Incident Face	Next	Prev
e1,1	v1	e1,2	f1	e2,1	e1,2
e1,2	v2	e1,1	f1	e1,1	e2,2
e2,1	v2	e2,2	f1	e3,1	e1,1
e2,2	v3	e2,1	f1	e1,2	e6,1
e3,1	v3	e3,2	f1	e4,1	e2,1
e3,2	v4	e3,1	f2	e6,2	e7,1
e4,1	v4	e4,2	f1	e5,1	e3,1
e4,2	v5	e4,1	f3	e7,2	e5,2
e5,1	v5	e5,2	f1	e6,1	e4,1
e5,2	v6	e5,1	f3	e4,2	e7,2
e6,1	v6	e6,2	f1	e2,2	e5,1
e6,2	v3	e6,1	f2	e7,1	e3,2
e7,1	v6	e7,2	f2	e3,2	e6,2
e7,2	v4	e7,1	f3	e5,2	e4,2

Face	Outer Comp.	Inner Comp.
f1	nil	f2, f3 (e3,1) (e4,1)
f2	f1, f3 (e3,2)	nil
f3	f1, f2 (e4,2)	nil

Vertex	Coordinates	IncidentEdge
v1	(6, 3)	e1,1
v2	(4, 1)	e2,1
v3	(2, 6)	e3,1
v4	(3, 10)	e4,1
v5	(6, 9)	e5,1
v6	(9, 5)	e6,1

DCEL topological algorithms

```
HalfEdgeFF(Face face)
{
    HalfEdge start_edge = face.outer_edge;
    if (start_edge.opp)
        result.Add(start_edge.opp.face);
    HalfEdge current_edge = start_edge.next;
    while (current_edge && current_edge != start_edge)
    {
        result.Add(current_edge.opp.face);
        current_edge = current_edge.next;
    }
    return result;
}
```

All in constant time!

```
HalfEdgeVE(Vertex vertex)
{
    HalfEdge start_edge = vertex.edge;
    result.Add(start_edge);
    HalfEdge current_edge = start_edge.opp.next;
    while (current_edge && current_edge != start_edge)
    {
        result.Add(current_edge);
        current_edge = current_edge.opp.next;
    }
    return result;
}
```

Face-Vertex mesh to DCEL mesh

- ▶ Used mainly when importing mesh from file
 - ▶ 1. Copy list of vertices and faces from Face-Vertex mesh to DCEL mesh
 - ▶ 2a. For each face traverse all edges of that face, create half edge for each face vertex and fill origin, next and face pointers
 - ▶ 2b. While traversing faces and its vertices, remember all incident (incoming) half-edges for each vertex
 - ▶ 3. Then for each half-edge, find opposite half-edge by searching incoming half-edges for origin vertex, here we need 2-manifold property to simple achieve this
 - ▶ 4. Add one arbitrary incident half-edge for each face and vertex
- ▶ Computational complexity is linear



The End for today