Computer Graphics for Digital Games
Video Game Technologies

14475: MSc in Computer Science and Engineering
13828: MSc in Game Design and Development

Chap. 3 — Geometric Object Data Structures

Geometric Object Data Structures

Outline

- Motivation.

- Geometric structures versus topological structures.

- Topological data structures: introduction.

- Incidence and adjacency relationships.

- Triangle soup (or spaghetti) data structure.

- Full incidence and adjacency data structure.

- Partial incidence and adjacency data structure.

- Winged-edge data structure.

- Half-edge data structure.

- Symmetric data structure.

- Topological inference and reasoning on incidence and adjacency.

- Euler operators (still incomplete!: not considered for teaching and learning)

Geometric models

Motivation:
- Representing and manipulating geometric objects in space.
- "Representing" means mathematical and computational representations of geometric objects.
- "Manipulating" means operators to model and construct complex objects from simpler ones.
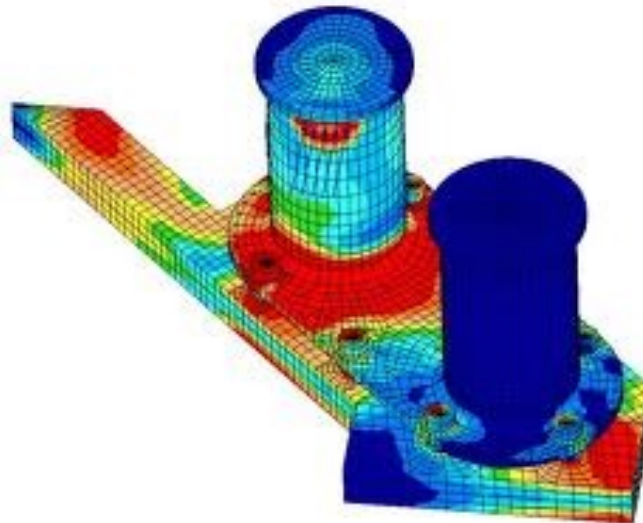
Geometric coverage:

- Points, straight and curved lines, planes and surfaces, solids, and n-D objects
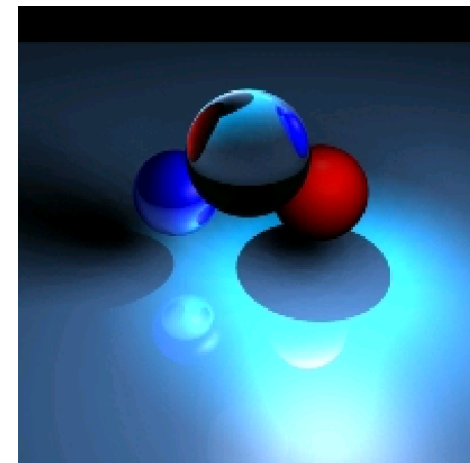
Solids' motivation:

- CT (computerized tomography) produces voxelized solids

- CAD systems require solid data for engineering analysis

- Ray tracing requires solid interiors for refraction purposes

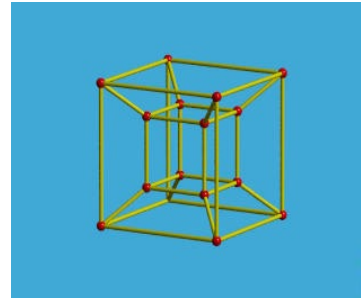cross-sectional image of abdomen
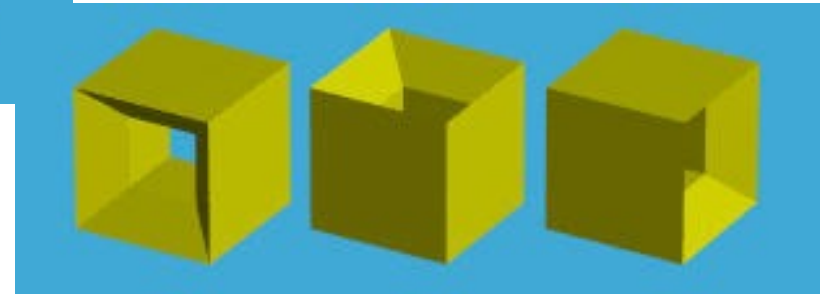
finite elements in eng. analysis

ray tracing solids with refraction
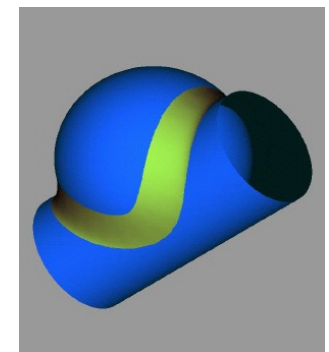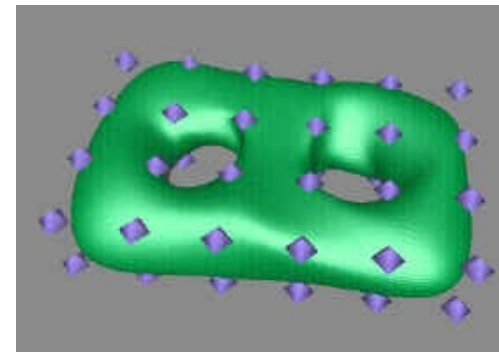
Brief history of geometric models

Wireframe models:

- 1950, vertices (points) and edges (lines)
- Solids are represented with ambiguities

Free-form surfaces:

- 1960's (Bézier surfaces and B-splines)
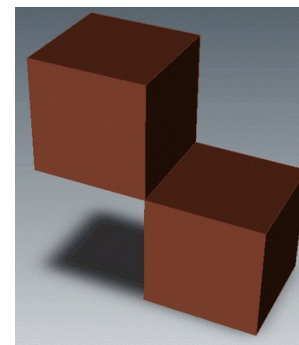- Mathematical (parametric) description of the surface

Solid models:

- 1970's, dimensional homogeneity, manifoldness condition (B-reps and CSG)

Non-manifold models:

- 1980's, relaxation of manifoldness condition

Geometric kernel (in computer graphics)

A geometry is a pair (G,∗), where:
−G is a set of transformations;
−∗ is an operation of concatenation.

Euclidean geometry:

- translation

- rotation

Affine geometry:

- scaling

- shearing

Projective geometry:

- orthogonal projection
- perspective projection

Topological Data Structures

Topology / connectivity

- Generic sets of entities: *vertices*, *edges* and *faces*

- Overlayed sets of entities: only *meet* and *disjoin*

- **Meet**: topological relation that defines <u>connectivity</u> between entities. Entities of different dimension are "connected" in different ways: relations (vertex-, edge-, face-based)

- **Disjoin**: topological relation that defines the entities of lower dimension are in the boundary of higher dimension entities.

Triangle soup data structure

- − Independent faces
- − Redundant vertices (because they are not shared by triangles).
- − No topology/connectivity information.
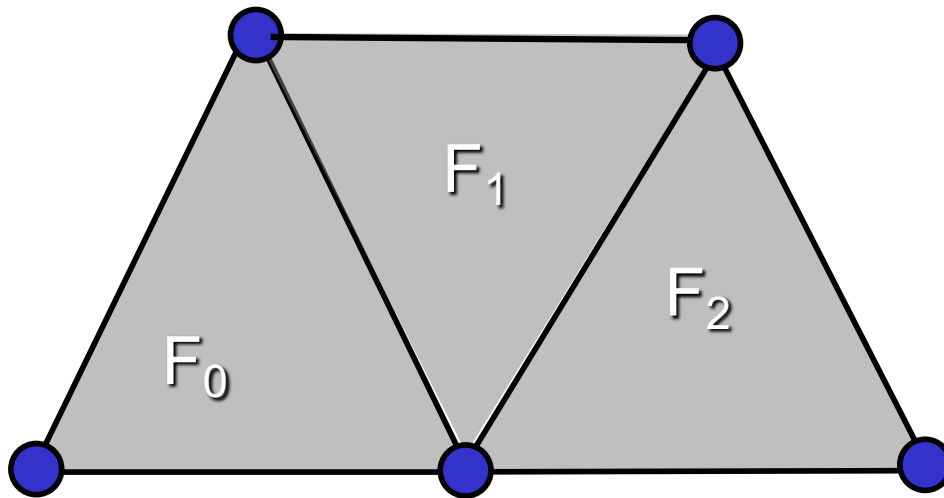
Face Table
F_0: (x_0,y_0,z_0), (x_1,y_1,z_1), (x_2,y_2,z_2)
F_1: (x_3,y_3,z_3), (x_4,y_4,z_4), (x_5,y_5,z_5)
F_2: (x_6,y_6,z_6), (x_7,y_7,z_7), (x_8,y_8,z_8)

... also called Spaghetti data structure

- Spaghetti data structure: represents sets of points, lines and polygons
- Can be used for both generic sets of entities and overlayed sets (plane subdivisions)
- The geometry of any spatial entity is described independently of other entities
- No topology/connectivity information is recorded

- Points, lines and polygons are stored separately.
- For each polygon, we store a (ordered) list of coordinates of points on its boundary.
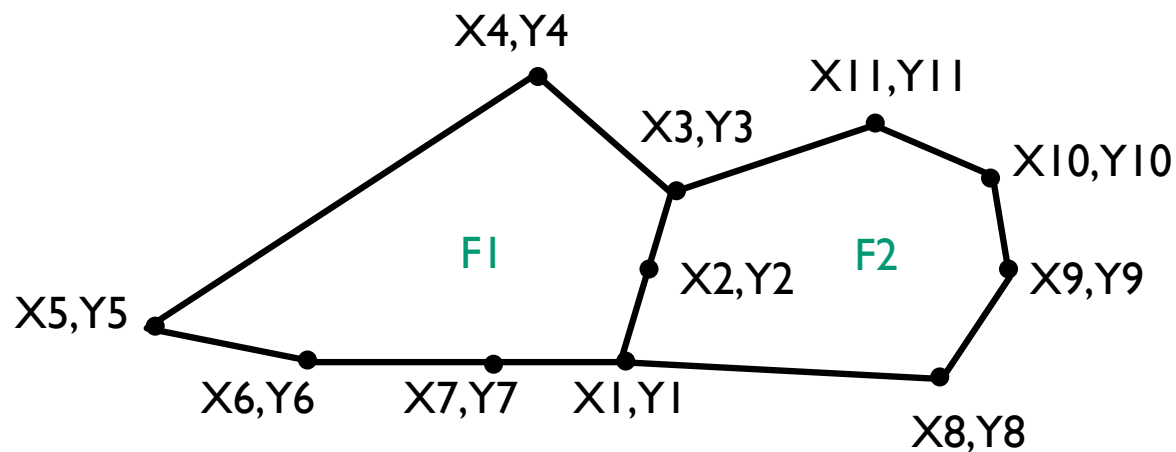
F1	F2
X1,Y1	X8,Y8
X2,Y2	X9,Y9
X3,Y3	X10,Y10
X4,Y4	X11,Y11
X5,Y5	X3,Y3
X6,Y6	X2,Y2
X7,Y7	X1,Y1
X1,Y1	X8,Y8

Spaghetti data structure: pros & cons

Advantages:

- simplicity

- easy insertions of new entities (all entities are independent)

**Disadvantages:**

- inefficient for topological queries

  No easy way of solving queries such as: "do Polygon 1 and 2 share a common bounding line?"
  Need to analyse all coordinates of points of Polygon 1 and compare with those of Polygon 2 and see if two consecutive pairs are the same: inefficient!!

- redundancies (and consequently, possible inconsistencies!)

  Coordinates of points along common boundary are recorded twice!
  Redundancy: if we update coordinates of a point, we need to update them everywhere!

Indexed face data structure

- Faces list vertex references – "shared vertices"
- Commonly used (e.g. OFF file format itself)
- Augmented versions simple for mesh processing
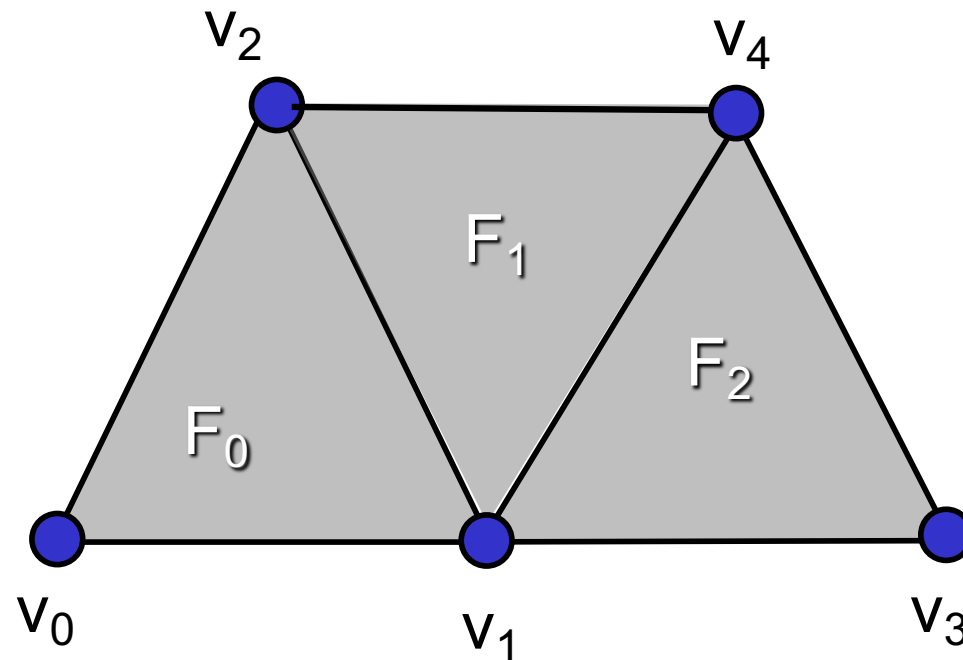
Vertex Table
v_0: (x_0, y_0, z_0)
v_1: (x_1, y_1, z_1)
v_2: (x_2, y_2, z_2)
v_3: (x_3, y_3, z_3)
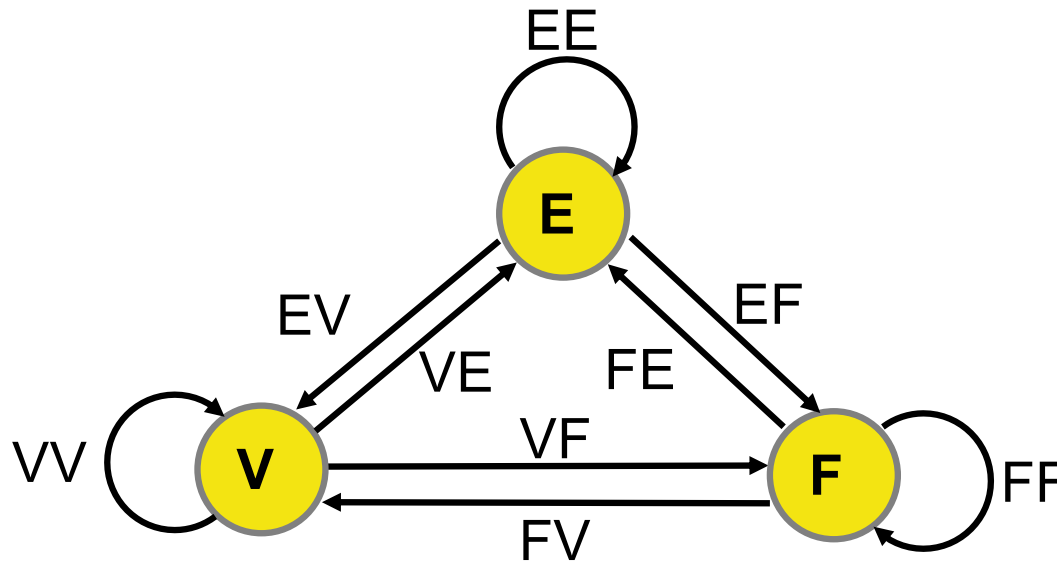v_4: (x_4, y_4, z_4)

Face Table
F_0: 0, 1, 2
F_1: 1, 4, 2
F_2: 1, 3, 4

Note CCW ordering

Incidence and Adjacency
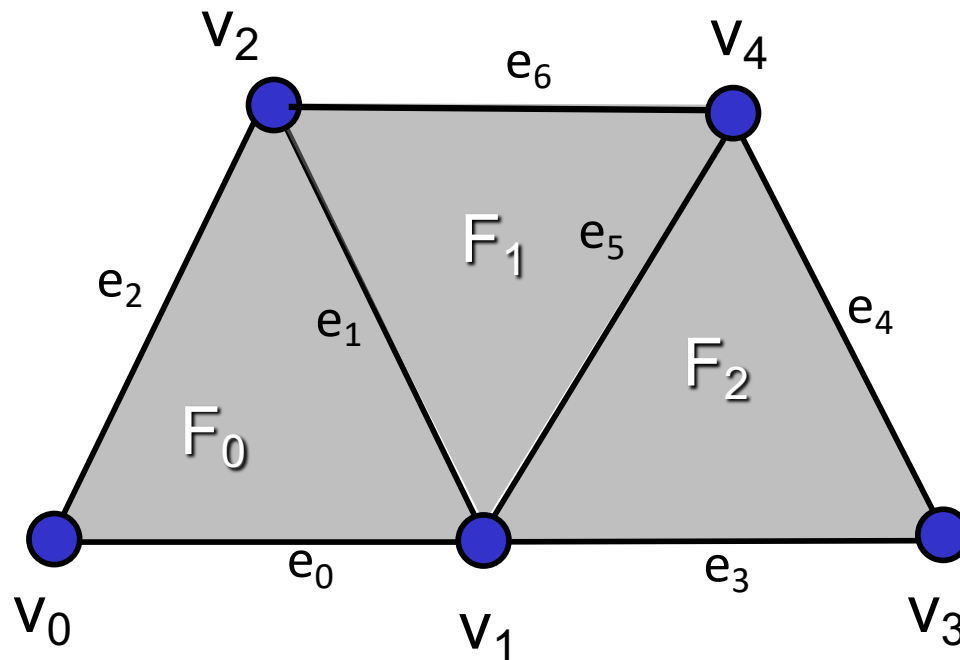
Incidence: V   : E   : F (decreasing dimension to left)
Adjacency: F   : E   : V (increasing dimension to left)

Full incidence and adjacency data structure

- Store all vertex, face, and edge incidences/adjacencies: VV, VE, VF, EV, EE, EF, FV, FE, FF
- Implement something simpler (like indexed face set plus vertex to face adjacency)
- Storage is the issue

Edge Adjacency Table
e_0: v_0, v_1; $F_0, \varnothing$; $\varnothing, e_2, e_1, \varnothing$
e_1: v_1, v_2; F_0, F_1; e_5, e_0, e_2, e_6
…

Face Adjacency Table
F_0: v_0, v_1, v_2; $F_1, \varnothing, \varnothing$; e_0, e_2, e_0
F_1: v_1, v_4, v_2; $\varnothing, F_0, F_2$; e_6, e_1, e_5
F_2: v_1, v_3, v_4; $\varnothing, F_1, \varnothing$; e_4, e_5, e_3

Vertex Adjacency Table
v_0: v_1, v_2; F_0; e_0, e_2
v_1: v_3, v_4, v_2, v_0; F_2, F_1, F_0; e_3, e_5, e_1, e_0

…

Topological data structures: motivation

- Storing connectivity information explicitly allows for more efficient spatial _queries_.
- Topology/connectivity: important criterion to establish the _correctness_ (integrity, consistency) of geometric objects, with applications in CAD, geographical databases, etc.

Example:

If we store relation FE explicitly (i.e., for each polygon we store a list of IDs of edges bounding it), the query: _"do Polygon 1 and 2 share a common bounding line segment?"_ only requires checking whether the two lists contain any common IDs
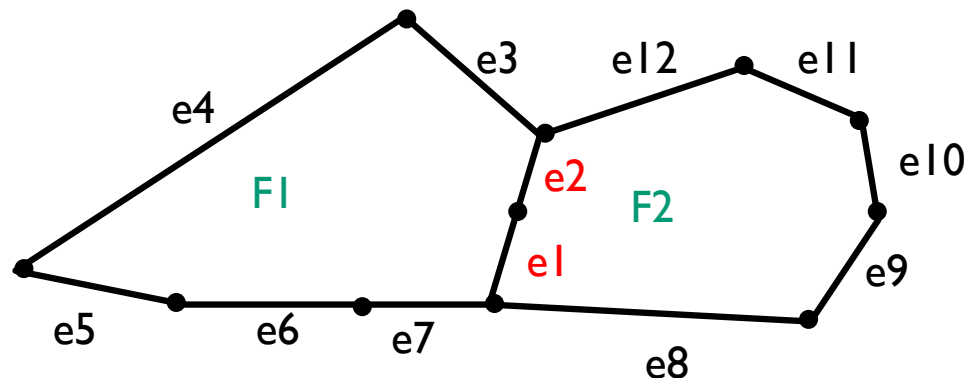
| F1	F2
e1	e8
e2	e9
e3	e10
e4	e11
e5	e12
e6	e2
e7	e1

Symmetric data structure

– Woo (1985)

<u>Symmetric</u> structure stores:

- three sets of entities: V, E, F

- relation EV and its inverse VE

- relation FE and its inverse EF
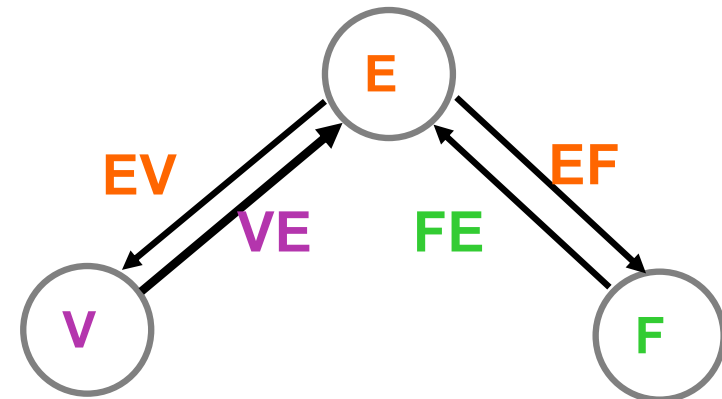
Example

	EV	EF		VE		FE
e1	v1,v2	f1,f2	v1	e1,e7,e8	f0	e3, e4,e5,e6,e7,e8,e9,e10,e11,e12
e2	v2,v3	f1,f2	v2	e2,e1	f1	e3,e4,e5,e6,e7,e1,e2
e3	v3,v4	f1,f0	v3	e3,e2,e12	f2	e1,e8,e9,e10,e11,e12,e2
e4	v4,v5	f1,f0	v4	e4,e3		
e5	v5,v6	f1,f0	v5	e5,e4		
e6	v6,v7	f1,f0	v6	e6,e5		
e7	v7,v1	f1,f0	v7	e7,e6		
e8	v1,v8	f2,f0	v8	e8,e9		
e9	v8,v9	f2,f0	v9	e9,e10		
e10	v9,v10	f2,f0	v10	e10,e11		
e11	v10,v11	f2,f0	v11	e11,e12		
e12	v11,v3	f2,f0				
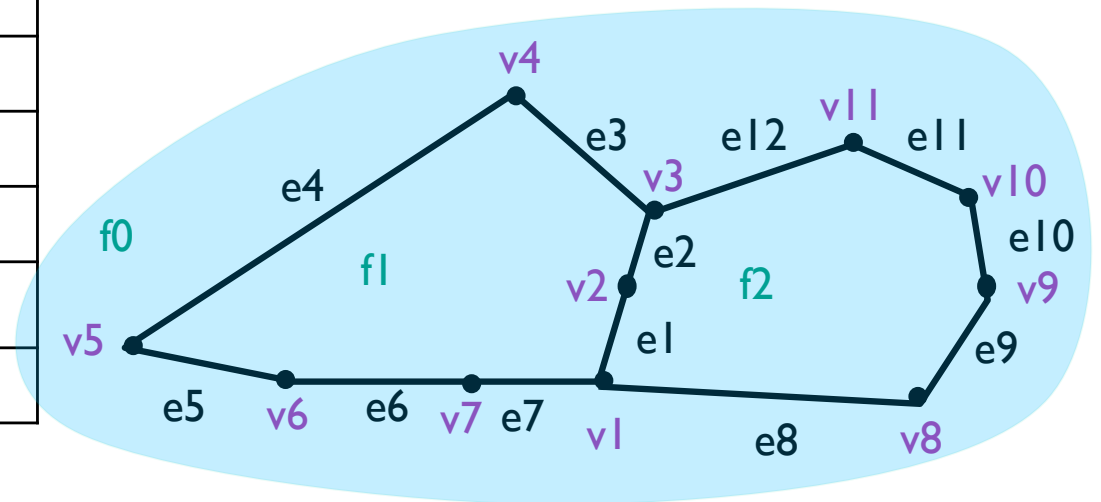
Symmetric structure: space complexity

For every **edge**:

- 2 constant relations are stored (involving 2 entities): 4e

For every **face**:

- 1 variable relation (FE). Every edge is common to two faces, so each edge is stored twice: 2e

For every **vertex**:

- 1 variable relation (VE). Every edge has two endpoints, so each edge is stored twice: 2e

TOTAL space required to represent relations: 8e

For each vertex we also store the two geometric coordinates: 2n

Symmetric structure: EE

Calculating relation EE: *Obtained by combining EV and VE (or EF and FE)*

- For example, if we want to calculate EE(e2)=(e1,e3,e12), we retrieve the endpoints v2 and v3 of e2 using EV. To retrieve e1 we consider the successor of e2 in the list associated with v2 through VE (for e3 and e12 the successors of e2 in the list associated with v3). To do this in constant time, for each edge we need to store the position of the edge in the lists associated to its endpoints through VE.
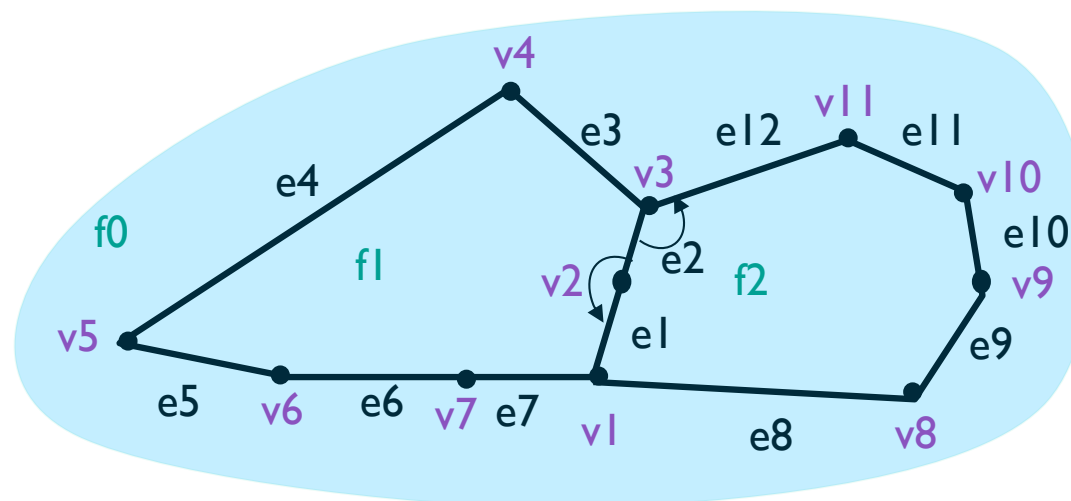
Symmetric structure: FF , FV, VV, VF

As in DCEL:

- FF: FE+EF
- FV: FE+EV
- VV: VE+EV
- VF: VE+EF

Example: FF

FF(f1)=(f0,f2) obtained combining:

- FE(f1)=(e3,e4,e5,e6,e7,e1,e2)
- EF(e3)=(f1,f0)
- EF(e4)=(f1,f0)
- EF(e5)=(f1,f0)
- EF(e6)=(f1,f0)
- EF(e7)=(f1,f0)
- EF(e1)=(f1,f2)
- EF(e2)=(f1,f2)
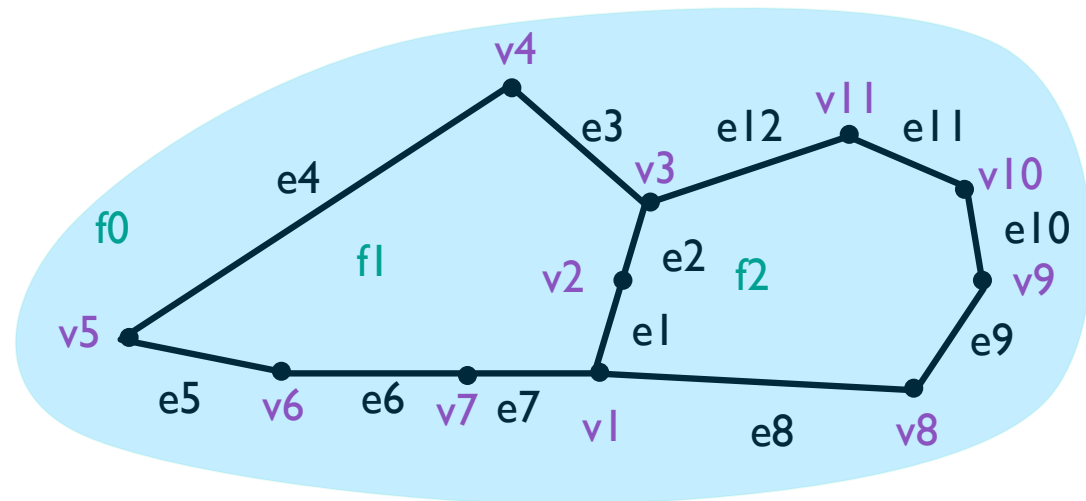
Winged-edge data structure (1/4)

– Baumgart (1975)

<u>Entities and topological relations:</u>

- three sets of entities: V, E, F

- relation EV and its inverse partial VE

- relation EF and its inverse partial EF

- partial EE
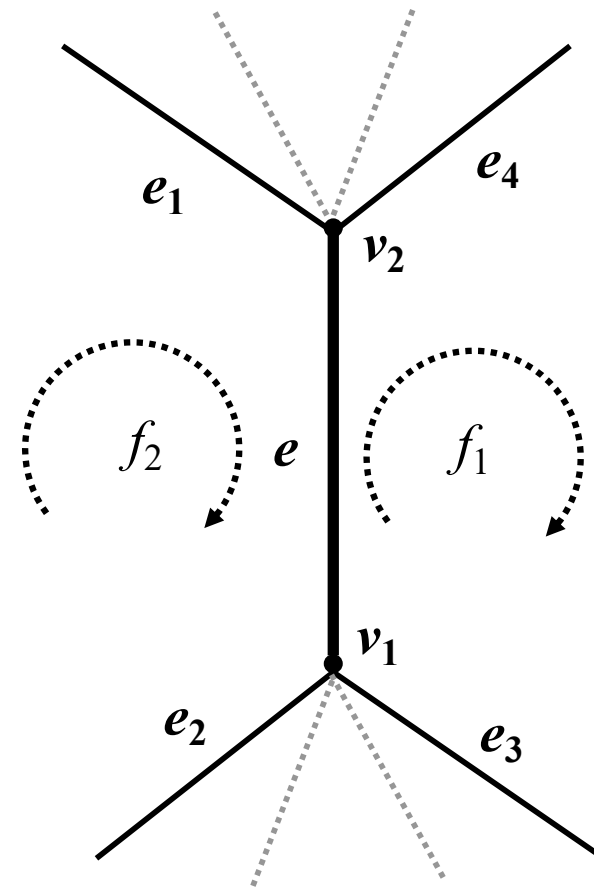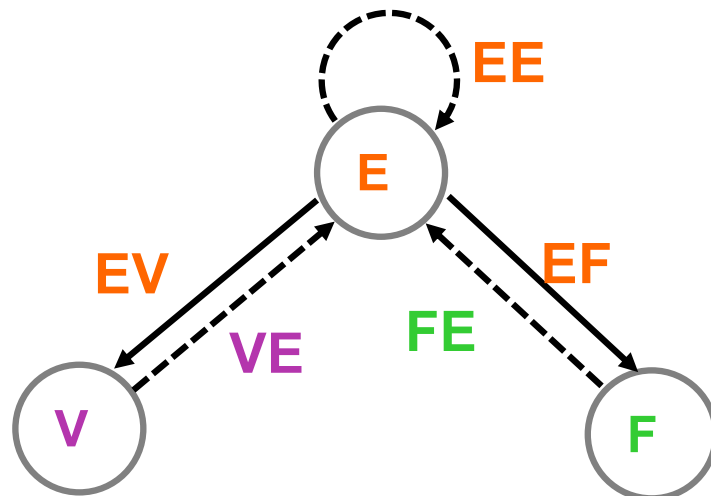
Winged-edge data structure (2/4)

Edge {
  Edge e1, e2, e3, e4;
  Face f2, f1;
  Vertex v1, v2;
}

Face {
  Edge edge;   // any adjacent (boundary) edge
}

Vertex {
  Edge edge;   // any incident edge
}
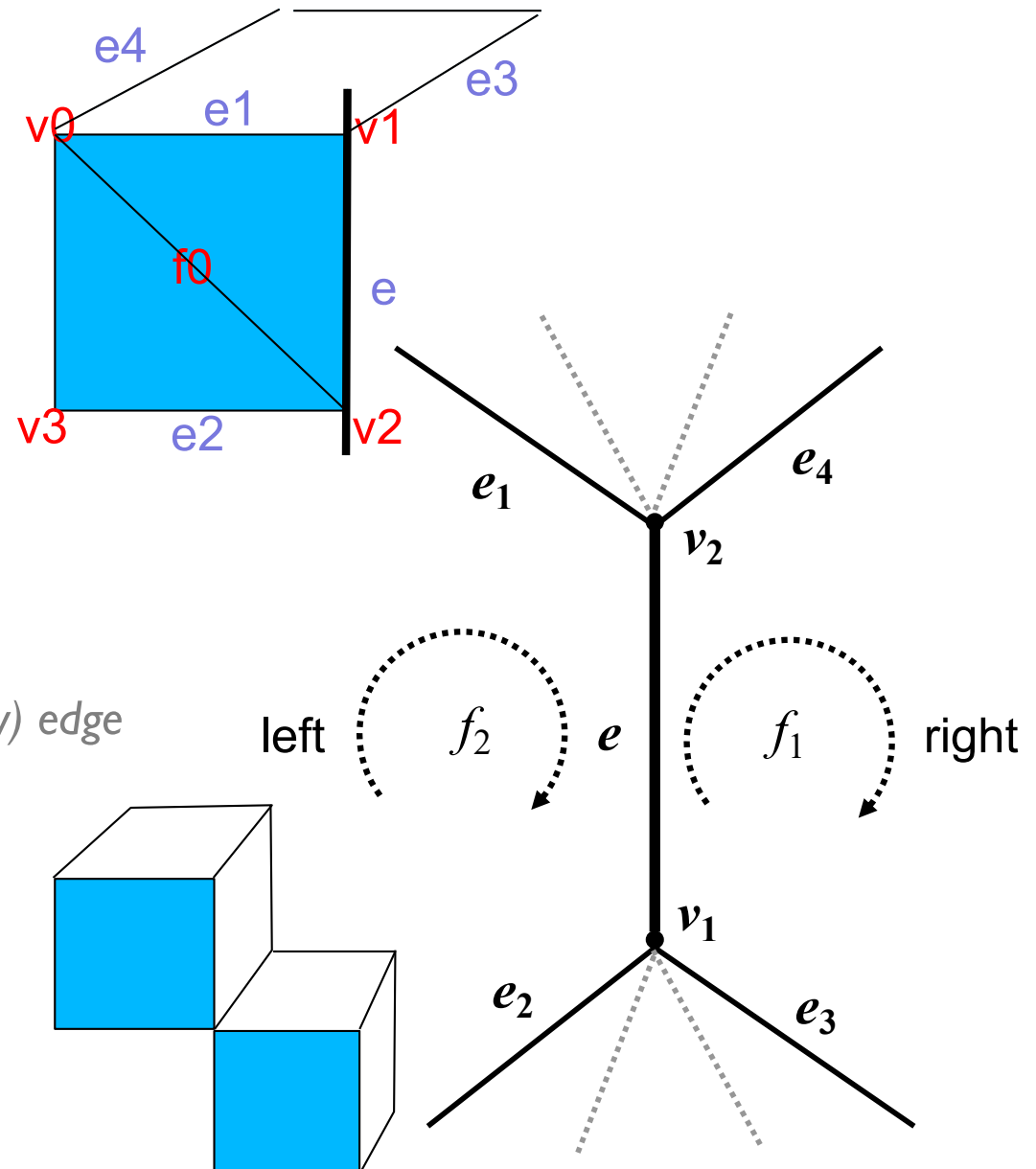
Winged-edge data structure (3/4)

Mesh and its arrays:

- – The winged-edge data structure has three arrays, edge array, vertex array, and face array.

- – Each edge contains the eight pieces information.

- – Each vertex has one entry in the vertex array. Each entry has a (pointer to an) incident edge of that vertex.

- – Each face has one entry in the face array. Each entry has a (pointer to a) boundary edge.

e_1 e_4 v_2

left f_2 e f_1 right

v_1

e_2 e_3
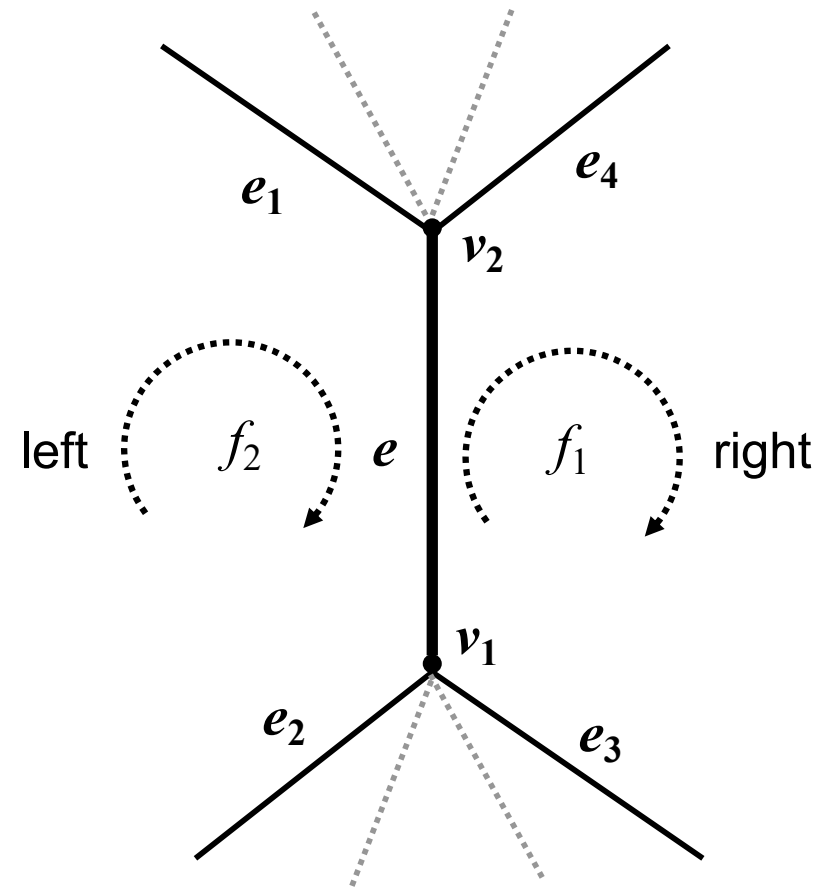
Winged-edge data structure (4/4)

Queries to get other topological relations:

- – Given a face, find all vertices bordering that face: FV = FE + EV

- – Given a vertex, find all edge-incident (or neighbor) vertices: VV

- – Given a face, find all neighbor faces: FF
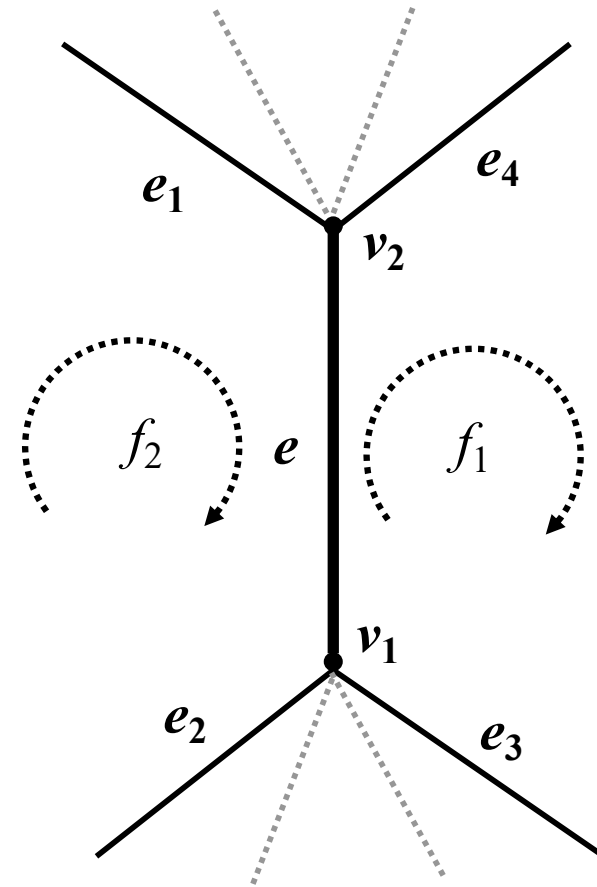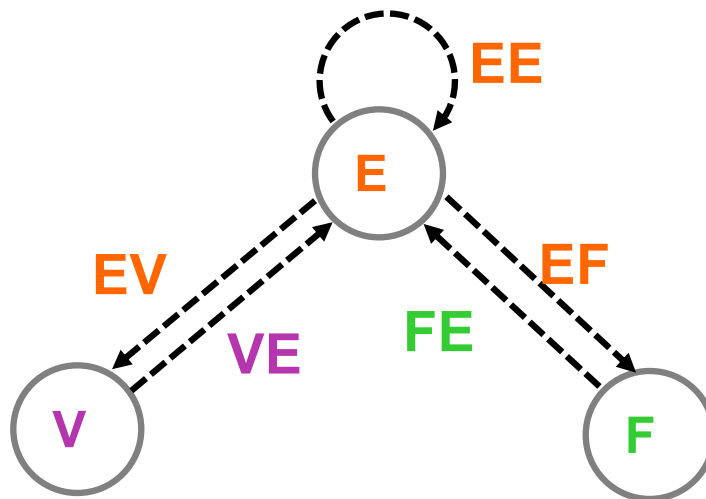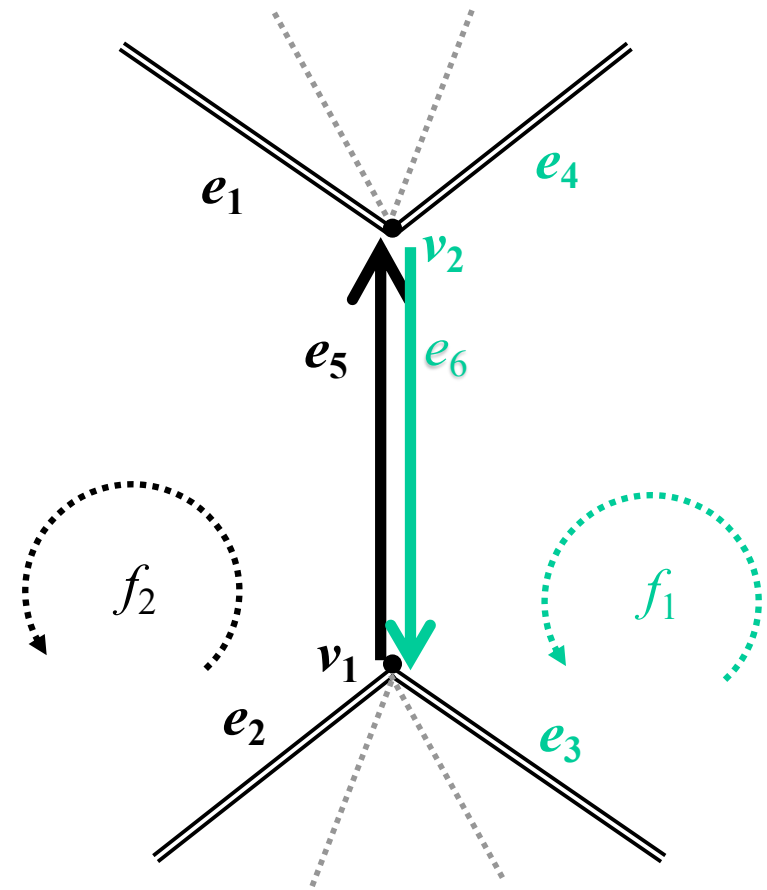
Half-edge data structure (1/2)

- Eastman(19??)
- Each <u>edge</u> is a pair of symmetric oriented edges, called half-edges.

<u>Entities and topological relations:</u>

- three sets of entities: V, <u>E</u>, F

- partial relations: EV, VE, EF, FE, EE

- Similar to Winged Edge Data Structure, but edges have unique orientation

- It requires slightly more storage
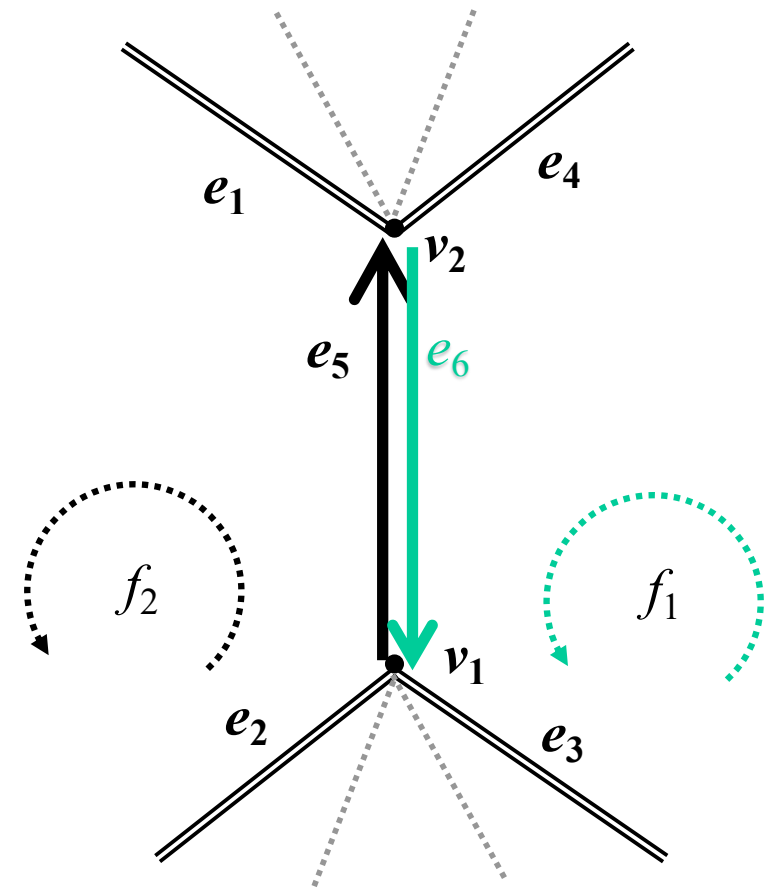
Half-edge data structure (2/2)

Edge {
  Edge previous_e, next_e, twin_e;
  Face f;
  Vertex v;
}

Face {
  Edge edge;   // any adjacent (boundary) edge
}

Vertex {
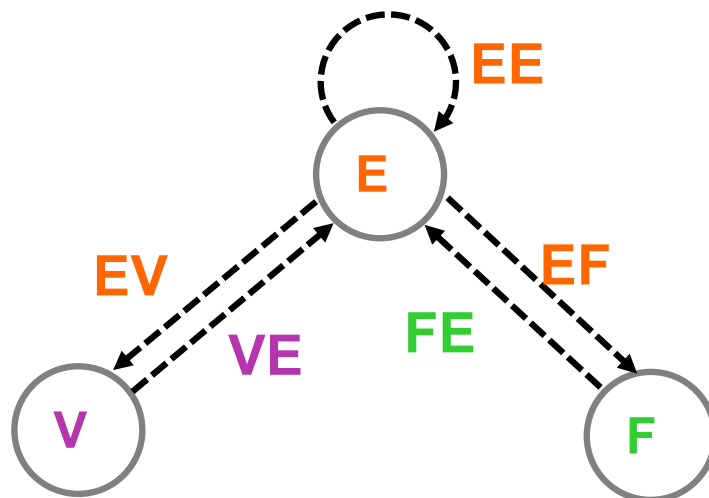 Edge edge;   // any incident edge
}

Quad-edge data structure (1/2)

- Guibas and Stolfi (1985)
- Each <u>edge is directed or oriented</u>.

<u>Entities and topological relations:</u>

- three sets of entities: V, <u>E</u>, F

- partial relations: EV, VE, EF, FE, EE

- **Quad-edge**: two primal twin edges + two dual twin edges
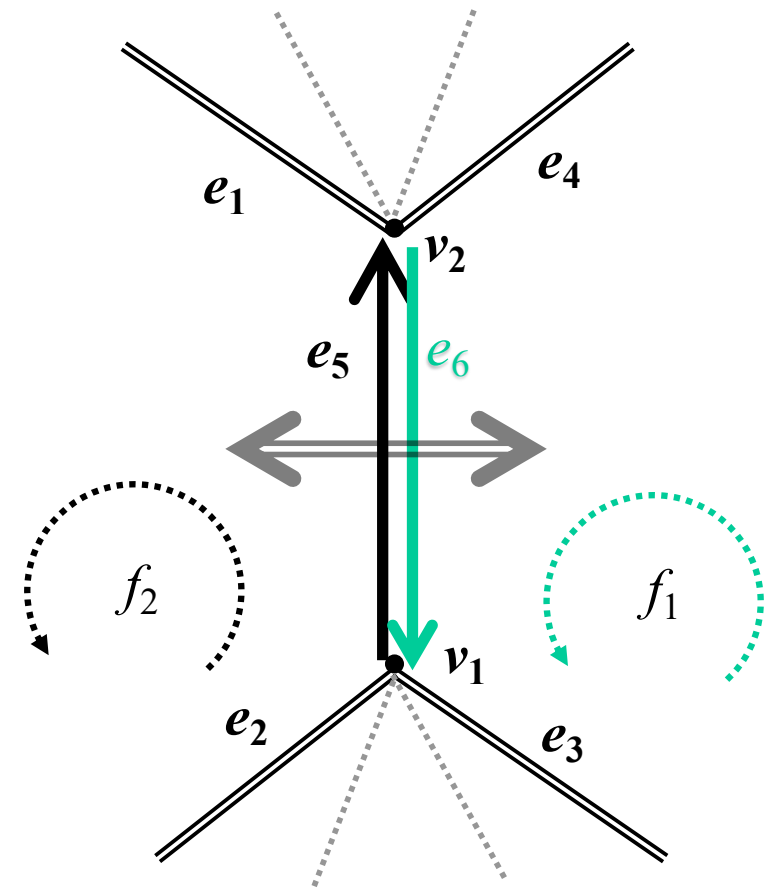
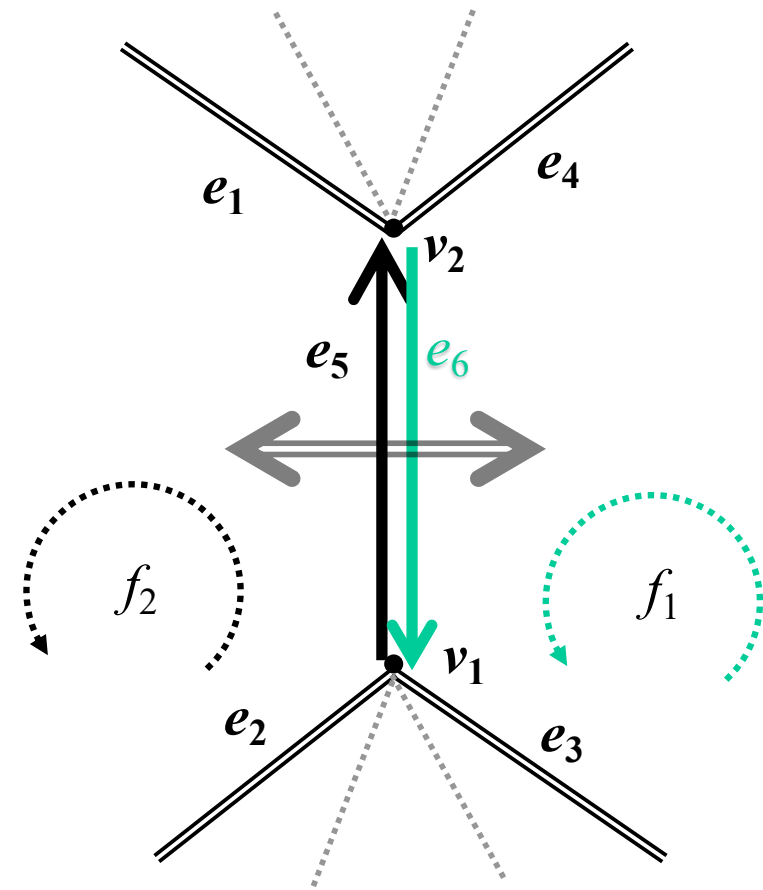# Quad-edge data structure (2/2)

– Guibas and Stolfi (1975)

Edge {
  Vertex v;   *// origin vertex*
  Edge e;   *// next edge*
  Face f2, f1;   *// its incident faces*
}

Face {
  Edge edge;   *// any adjacent (boundary) edge*
}

Vertex {
 Edge edge;   *// any (outgoing) incident edge*
}

QuadEdge{
  Edge E[4];
}

Cell-tuple data structure (1/4)

- Brisson (1989)
- A cell is an abstraction of vertex, edge, facet, etc.

Entities and topological relations:

- three sets of entities: V, E, F

- basic tools: tuples of cells and switch operators

- **_cell-tuple:_** $(d+1)$-tuple of cells $(c_0, ..., c_i, ..., c_d)$, where c_i is an i-cell, and c_i belongs to the boundary of c_{i+1} $(i = 0, 1,..., d-1)$
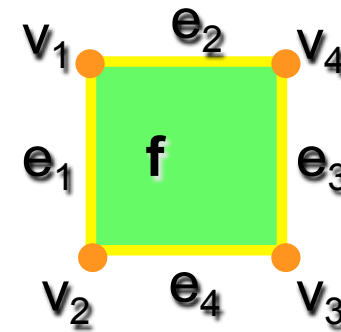
- total relations: EV, VE, EF, FE

- dimension-independent
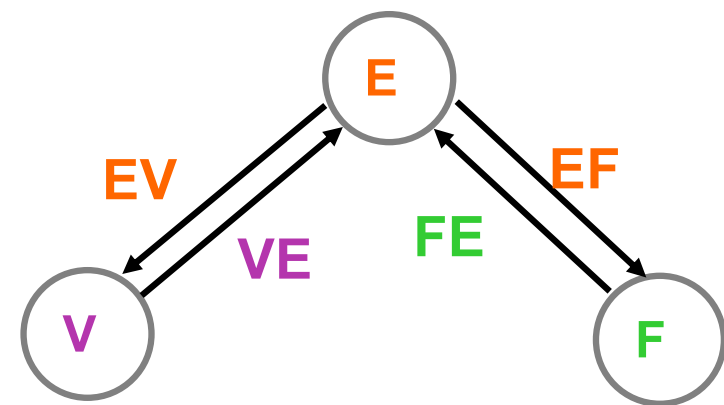
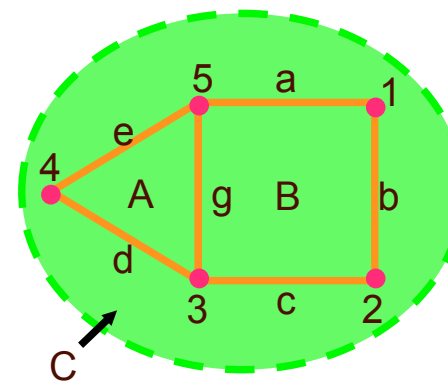# Cell-tuple data structure (2/4)

– Brisson (1989)

Ctuple{
  Cell tuple[]; // origin vertex
}

Cell{
  int d;   // dimension
}

Full set of cell-tuples:

(v1,e1,f)
(v1,e2,f)
(v2,e1,f)
(v2,e4,f)
(v3,e3,f)
(v3,e4,f)
(v4,e2,f)
(v4,e3,f)

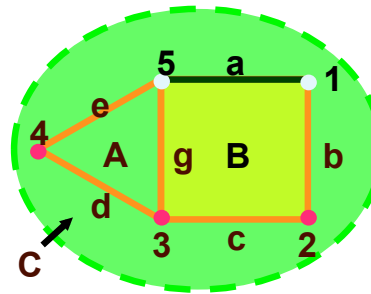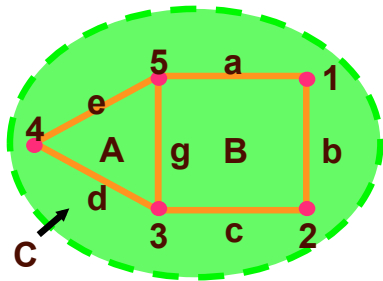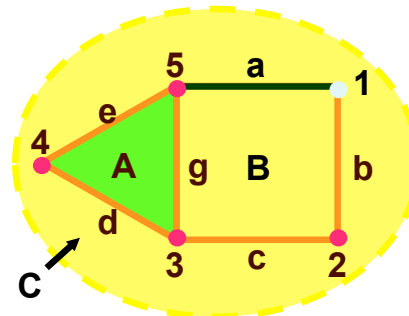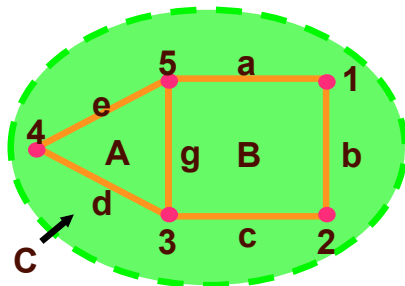*Set of cell-tuples for g:*

(v3,g,A)
(v3,g,B)
(v5,g,A)
(v5,g,B)

Cell-tu

switch operator:

- $switch_i(\sigma_0, ..., \sigma_i, ..., \sigma_d) = (\sigma_0, ..., \gamma_i, ..., \sigma_d)$

  - where γ_i is an i-cell different from σ_i

  - and $(\sigma_0, ..., \gamma_i, ..., \sigma_d)$ is another cell-tuple



$switch_0(1,a,B) = (5,a,B)$

$switch_2(1,a,B) = (1,a,C)$

Cell-tuple data structure (4/4): graph

Cell-tuple graph:

- − Cell tuples are represented as nodes of a labeled graph, where each arc of the graph represents a switch operator.

- − The label (=0,1or2 in the 2D case) of an arc; index of the switch operator described by the arc.

- − Verbose representation, but all topological relations can be retrieved in optimal time

0   1

3   2
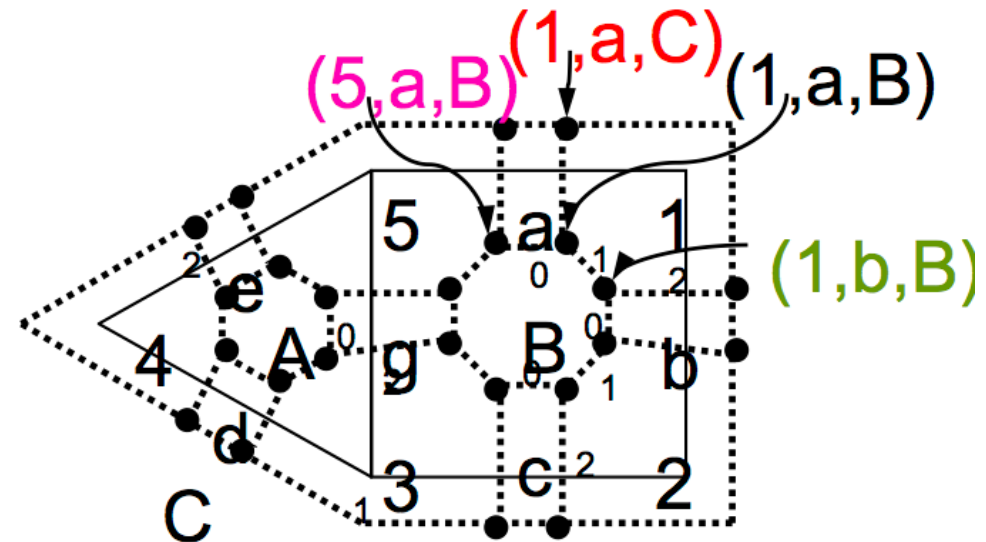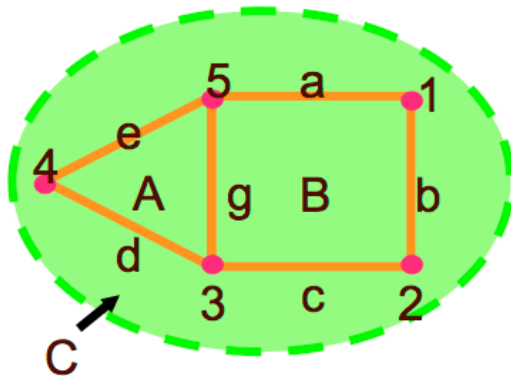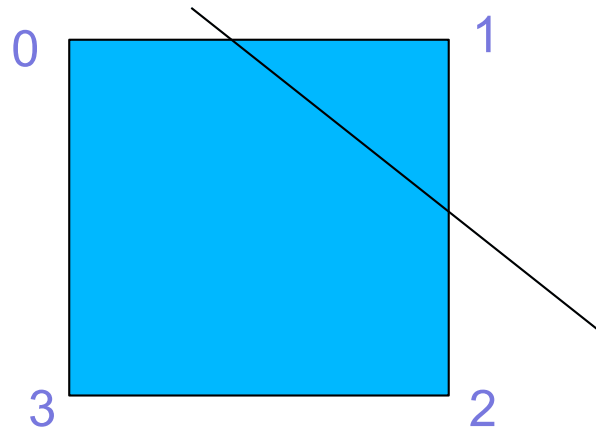
v-e+f=C-H

0-1+1

mef= make edge and face

Euler operators

Motivation for studying Euler operators:
- Allow the incremental construction of complex objects from basic building blocks such as vertices, edges and faces.
- Applications: geometric CAD kernels, computational animation systems, etc.

Summary:

....:

- Motivation.

- Geometric structures versus topological structures.

- Topological data structures: introduction.

- Incidence and adjacency relationships.

- Spaghetti data structure.

- DCEL data structure.

- Symmetric data structure.

- Topological inference and reasoning on incidence and adjacency.

- Euler operators (still incomplete!: not considered for teaching and learning)