

---

# Data structures for 3D Meshes

Paolo Cignoni

p.cignoni@isti.cnr.it

<http://vcg.isti.cnr.it/~cignoni>

# Key idea:

---

- ❖ Discretize the surface in a set of simple primitives
  - ❖ Simple!
  - ❖ Polygons
  - ❖ Triangles
  - ❖ Small curved elements (nurbs)
  - ❖ Even points
- ❖ We will focus only on one of them:
  - ❖ simplicial complexes

# Introduction

## ❖ Premise:

- ❖ 3D Graphics involves the rendering of digitally represented objects
- ❖ Rendering means reasoning about how the light interacts with the surface of the objects (mostly).

## ❖ Issue:

- ❖ representing boundary of 3D objects



# Why triangular meshes

- ❖ Perché solo e soltanto mesh triangolari?
  - ❖ In modellazione si vedono spesso mesh composte da poligoni generici...
  - ❖ Risposta teorica
    - ❖ Hanno un bel formalismo (complessi simpliciali)
    - ❖ Meno casi degeneri
      - ❖ Un triangolo è sempre planare
      - ❖ Uniformità se tolgo un vertice ottengo sempre un simpleso...
      - ❖ Estendibilità
      - ❖ Fixed size relations

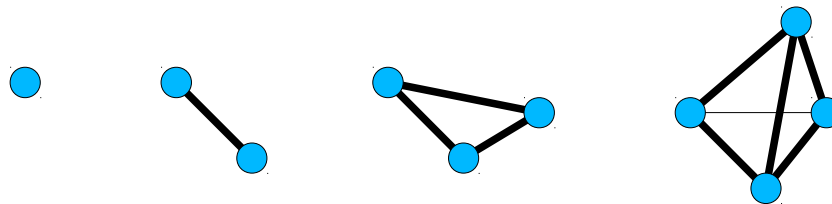
# Why triangular meshes

---

- ❖ Perché solo e soltanto mesh triangolari?
  - ❖ In modellazione si vedono spesso mesh composte da poligoni generici...
  - ❖ Risposta Pratica
    - ❖ Hardware grafico basato solo su triangoli
    - ❖ Strutture dati semplici

# Simplessi

- ❖ Un ***k* simplessso** è definito come la combinazione convessa di  $k+1$  punti non linearmente dipendenti



- ❖  $k$  è l'ordine del simplessso
- ❖ I punti si chiamano vertici

# Sotto-Simplesso

---

- ❖ Un simplesso  $\sigma'$  è detto *faccia* di un simplesso  $\sigma$  se e' definito da un sottoinsieme dei vertici di  $\sigma$
- ❖
- ❖ Se  $\sigma \neq \sigma'$  si dice che é una faccia propria

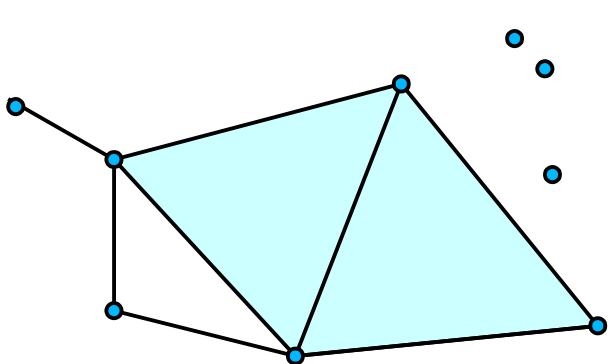
# Complesso Simpliciale

❖ Una collezione di semplici  $\Sigma$  e' un  $k$ -complesso simpliciale se:

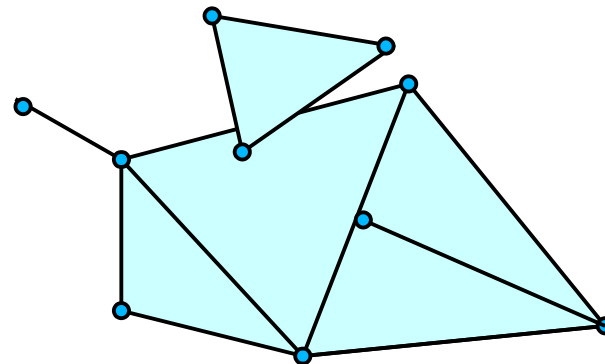
$\forall \sigma_1, \sigma_2 \in \Sigma \quad \sigma_1 \cap \sigma_2 \neq \emptyset \rightarrow \sigma_1 \cap \sigma_2$  is a simplex of  $\Sigma$

$\forall \sigma \in \Sigma$  all the faces of  $\sigma$  belong to  $\Sigma$

$k$  is the maximum order  $\forall \sigma \in \Sigma$



OK



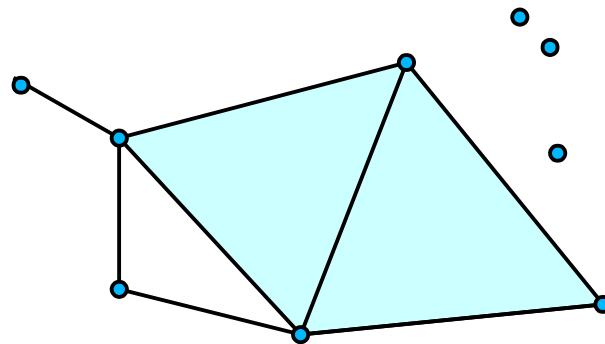
Not



# Complesso Simpliciale

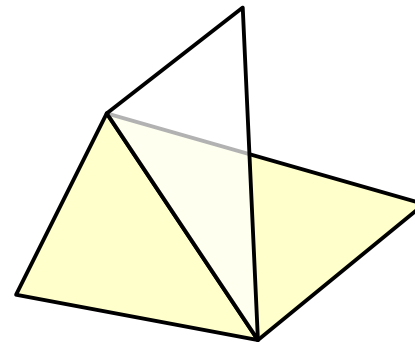
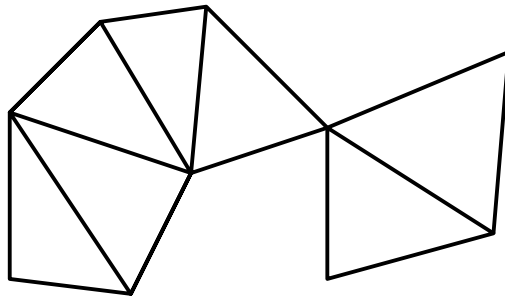
- ❖ Un semplice  $\sigma$  è massimale in un complesso simpliciale  $\Sigma$  se non è faccia propria di nessun altro semplice di  $\Sigma$
- ❖ Un  $k$ -complesso simpliciale  $\Sigma$  è massimale se tutti semplici massimali sono di ordine  $k$ 
  - ❖ In pratica non penzolano pezzi di ordine inferiore

Non maximal 2-simplicial complex



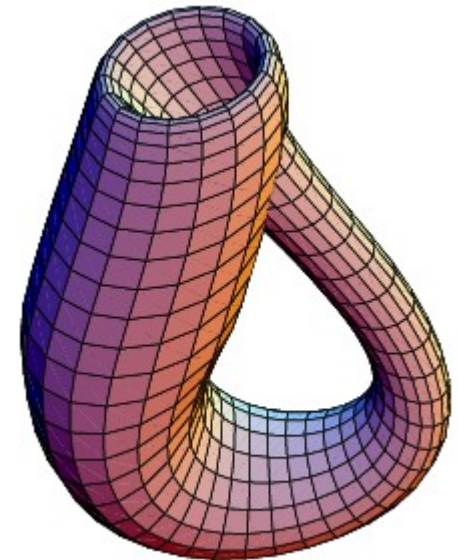
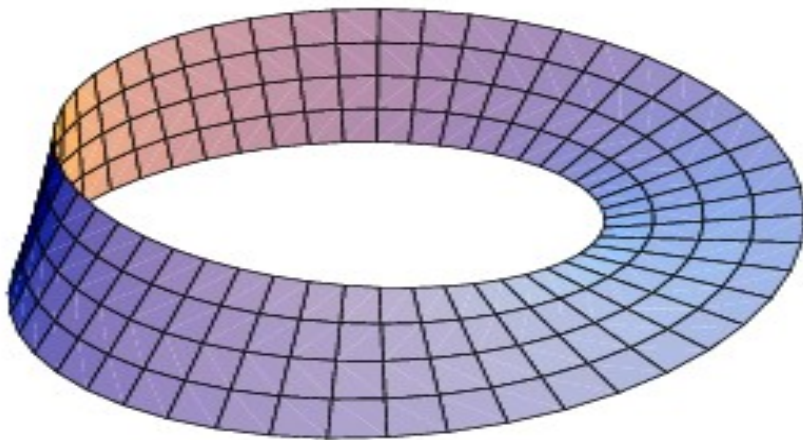
# 2-Manifold

- ❖ Una superficie  $\Sigma$  immersa in  $\mathbf{R}^3$  tale che ogni punto su  $\Sigma$  ha un intorno aperto omeomorfo ad un disco aperto o a un semidisco aperto in  $\mathbf{R}^2$
- ❖ Esempi non manifold



# Orientable

- ❖ If it is possible to set a coherent normal to each point of the surface
  - ❖ Moebius strips, klein bottles, and non manifold surfaces are not orientable



# Mesh

- ❖ Le classiche mesh triangolari cui siamo abituati sono 2-complessi simpliciali massimali la cui realizzazione in  $\mathbf{R}^3$  è una superficie 2-manifold.
- ❖ Note:
  - ❖ A volte (spesso) capitano superfici non 2-manifold
  - ❖ A volte non sono orientabili
  - ❖ Che siano massimali invece lo assumiamo
    - ❖ e' facile trasformale in massimali distruttivamente...

# Topology vs Geometry

---

- ❖ Di un complesso simpliciale e' buona norma distinguere
  - ❖ Realizzazione geometrica
    - ❖ Dove stanno effettivamente nello spazio i vertici del nostro complesso simpliciale
  - ❖ Caratterizzazione topologica
    - ❖ Come sono connessi combinatoriamente i vari elementi

# Topology vs geometry 2

---

Nota: Di uno stesso oggetto e' possibile dare rappresentazioni con eguale realizzazione geometrica ma differente caratterizzazione topologica (molto differente!) Demo kleine

Nota: Di un oggetto si puo' dire molte cose considerandone solo la componente topologica

- Orientabilita

- componenti connesse

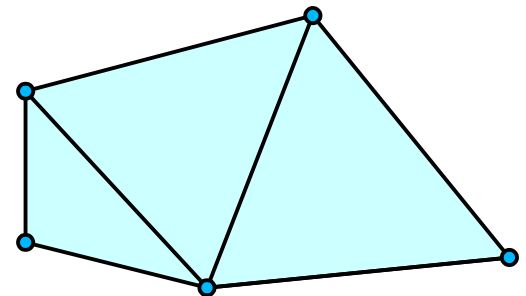
- bordi

# Cell Complexes

- ❖ Esistono anche generalizzazioni di questi concetti basate sul concetto di generici sottoinsiemi di uno spazio legati tra loro in maniera analoga ai simplicial complexes
  - ❖ Formalizzazione teorica di mesh non basate su triangoli
  - ❖ Il concetto di realizzazione geometrica e' **molto** piu' delicato (sono patch generiche in effetti)
  - ❖ Noi qui non ne parleremo...

# Incidenza Adiacenza

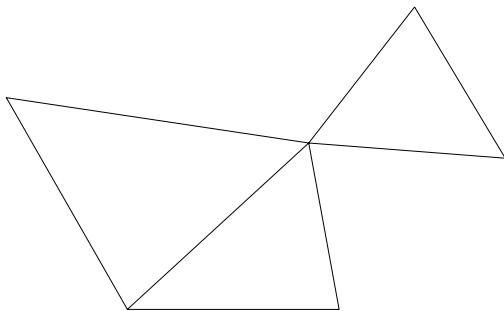
- ❖ Due semplici  $\sigma$  e  $\sigma'$  sono incidenti se  $\sigma$  è una faccia propria di  $\sigma'$  o vale il viceversa.
- ❖ Due  $k$ -simplessi sono  $m$ -adiacenti ( $k > m$ ) se esiste un  $m$ -simpleso che è una faccia propria di entrambi.
  - ❖ Due triangoli che condividono un edge sono 1-adiacenti
  - ❖ Due triangoli che condividono un vertice sono 0-adiacenti





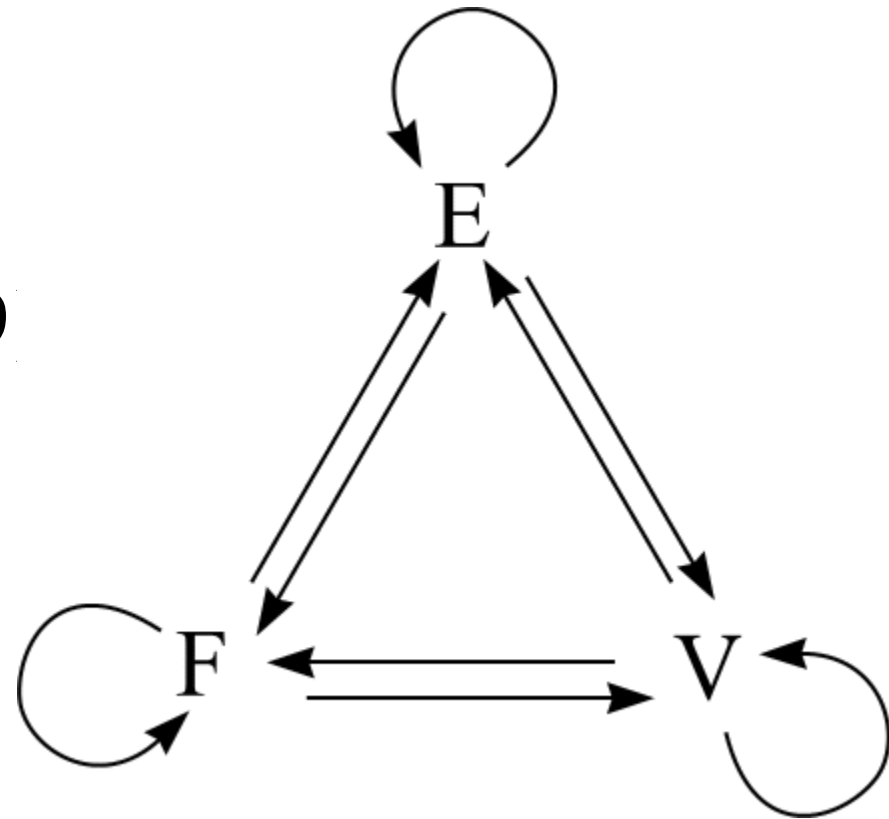
# Relazioni di Adiacenza

- ❖ Per semplicità nel caso di mesh si una relazione di adiacenza con un una coppia (ordinata!) di lettere che indicano le entità coinvolte
  - ❖ FF adiacenza tra triangoli
  - ❖ FV i vertici che compongono un triangolo
  - ❖ VF i triangoli incidenti su un dato vertice



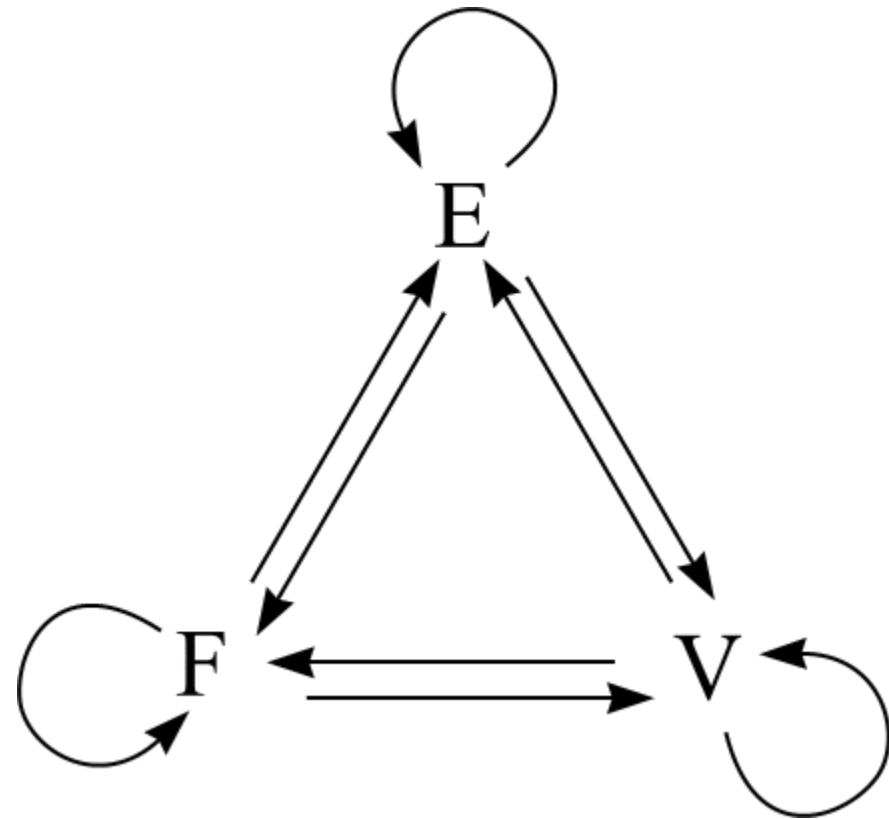
# Relazioni di adiacenza

- ❖ Di tutte le possibili relazioni di adiacenza di solito vale la pena se ne considera solo un sottoinsieme (su 9 e ricavare le altre proceduralmente)



# Relazioni di adiacenza

- ❖  $FF \sim 1$ -adiacenza
- ❖  $EE \sim 0$  adiacenza
- ❖  $FE \sim$  sottofacce proprie di  $F$  con  $\dim 1$
- ❖  $FV \sim$  sottofacce proprie di  $F$  con  $\dim 0$
- ❖  $EV \sim$  sottofacce proprie di  $E$  con  $\dim 0$
- ❖  $VF \sim F$  in  $\Sigma$  :  $V$  sub faccia di  $F$
- ❖  $VE \sim E$  in  $\Sigma$  :  $V$  sub faccia di  $E$
- ❖  $EF \sim F$  in  $\Sigma$  :  $E$  sub faccia di  $F$
- ❖  $VV \sim V'$  in  $\Sigma$  : Esiste  $E (V, V')$



# Partial adjacency

- ❖ Per risparmiare a volte si mantiene una informazione di adiacenza parziale
  - ❖  $VF^*$  memorizzo solo un riferimento dal vertice ad una delle facce e poi 'navigo' sulla mesh usando la FF per trovare le altre facce incidenti su  $V$

# Relazioni di adiacenza

- ❖ In un 2-complesso simpliciale immerso in  $R^3$ , che sia 2 manifold
  - ❖ FV FE FF EF EV sono di cardinalità bounded (costante nel caso non abbia bordi)
    - ❖  $|FV| = 3$   $|EV| = 2$   $|FE| = 3$
    - ❖  $|FF| \leq 3$
    - ❖  $|EF| \leq 2$
  - ❖ VV VE VF EE sono di card. variabile ma in stimabile in media
    - ❖  $|VV| \sim |VE| \sim |VF| \sim 6$
    - ❖  $|EE| \sim 10$
    - ❖  $F \sim 2V$

# Euler characteristic

- ❖ Invariante topologico

$$\chi = V - E + F$$

$$\chi = |\Sigma_0| - |\Sigma_1| + |\Sigma_2| - \dots$$

- ❖ Per tutto quello omeomorfo ad una sfera

$$\chi = 2$$

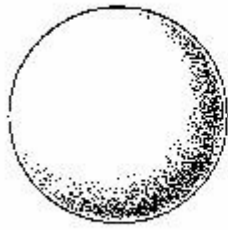
- ❖ In generale per una sup qualsiasi

$$\chi = 2 - 2g$$

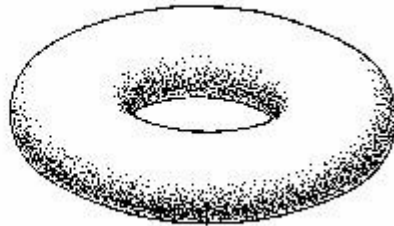
- ❖ Con  $g$  genus della superficie

# Genus

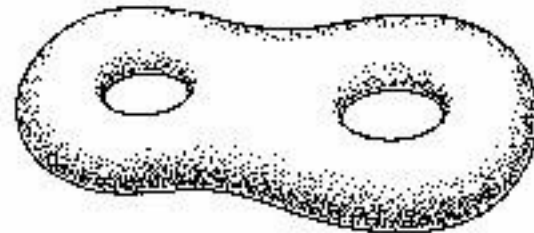
- ❖ Il genus di una superficie chiusa, orientabile 2-manifold: e' il massimo numero di tagli lungo curve chiuse semplici che si possono fare senza rendere l'insieme sconnesso



0



1



2

- ❖ Per i profani numero di maniglie sulla superficie

# Euler characteristic

---

- ❖ Se la superficie non è chiusa

$$\chi = 2 - 2g - B$$

- ❖ Dove  $B$  è il numero di bordi
  - ❖ (non di elementi sul bordo)

- ❖ Per multiple connected surfaces

$$\chi = 2C - 2 \sum g_i$$

- ❖ Con  $C$  numero delle componenti connesse



Name	Image	V (vertices)	E (edges)	F (faces)	Euler characteristic: $V - E + F$
<a href="#">Tetrahedron</a>		4	6	4	<b>2</b>
<a href="#">Hexahedron</a> or <a href="#">cube</a>		8	12	6	<b>2</b>
<a href="#">Octahedron</a>		6	12	8	<b>2</b>
<a href="#">Dodecahedron</a>		20	30	12	<b>2</b>
<a href="#">Icosahedron</a>		12	30	20	<b>2</b>

# Example data structure

- ❖ Simplest
- ❖ List of triangles:
  - ❖ For each triangle store its coords.
  - ❖
  - ❖ 1.  $(3, -2, 5), (3, 6, 2), (-6, 2, 4)$
  - ❖ 2.  $(2, 2, 4), (0, -1, -2), (9, 4, 0)$
  - ❖ 3.  $(1, 2, -2), (3, 6, 2), (-4, -5, 1)$
  - ❖ 4.  $(-8, 2, 7), (-2, 3, 9), (1, 2, -2)$
- ❖ How to find any adjacency?
- ❖ Does it store FV?

# Example data structure

- ❖ Slightly better
- ❖ List of unique vertices with indexed faces
  - ❖ Storing the FV relation

- ❖ Vertices:

- ❖ 1. (-1.0, -1.0, -1.0)
- ❖ 2. (-1.0, -1.0, 1.0)
- ❖ 3. (-1.0, 1.0, -1.0)
- ❖ 4. (-1, 1, 1.0)
- ❖ 5. (1.0, -1.0, -1.0)
- ❖ 6. (1.0, -1.0, 1.0)
- ❖ 7. (1.0, 1.0, -1.0)
- ❖ 8. (1.0, 1.0, 1.0)

- ❖ Faces:

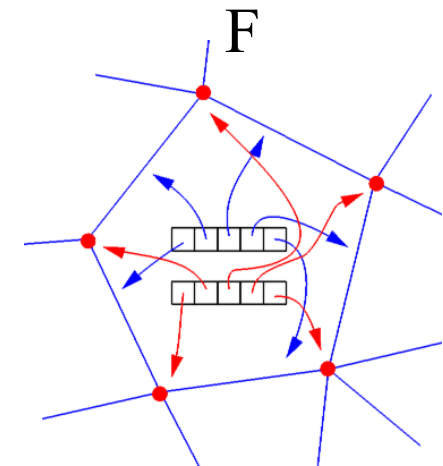
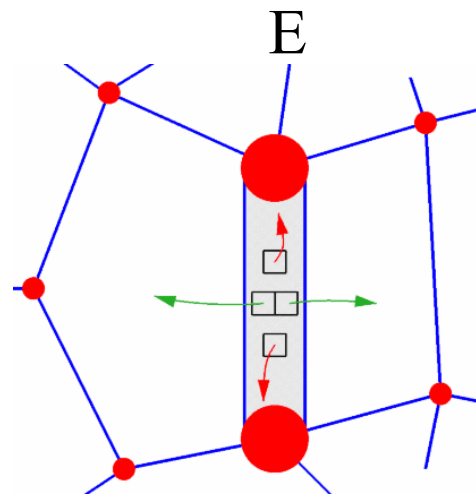
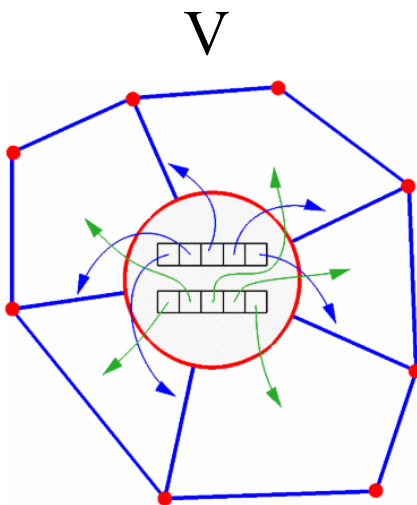
- ❖ 1. 1 2 4
- ❖ 2. 5 7 6
- ❖ 3. 1 5 2
- ❖ 4. 3 4 7
- ❖ 5. 1 7 5

# Example data structure

## ❖ Issue of Adjacency

### ❖ Vertex, Edge, and Face Structures

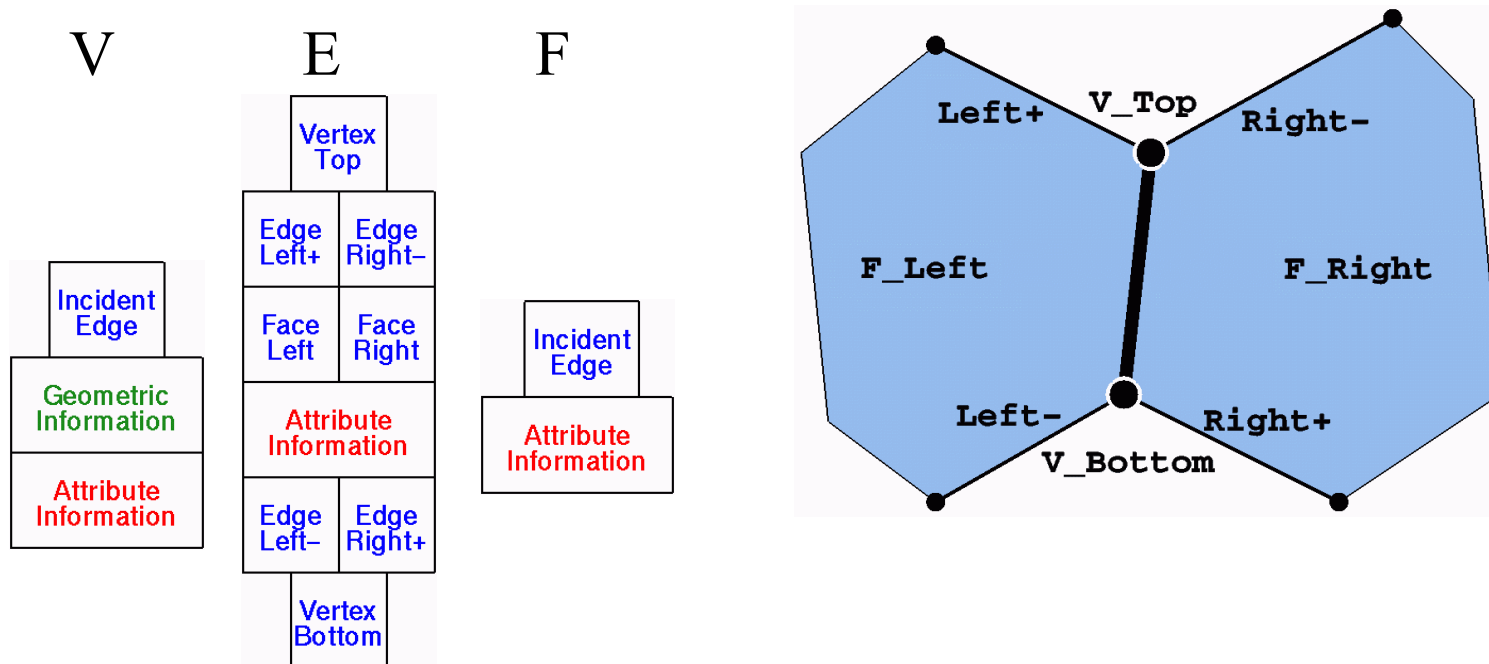
- ❖ Each element has list of pointers to all incident elements
- ❖ Queries depend only on local complexity of mesh!
- ❖ Slow! Big! Too much work to maintain!
- ❖ Data structures do not have fixed size



# Example data structure

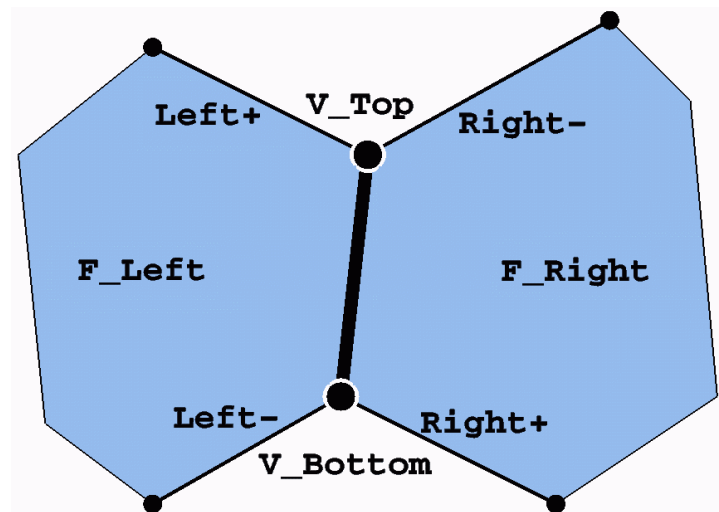
## ❖ Winged edge

- ❖ Classical real smart structure
- ❖ Nice for generic polygonal meshes
- ❖ Used in many sw packages



# Winged Edge

- ❖ Winged edge
  - ❖ Compact
  - ❖ All the query requires some kind of "traversal"
  - ❖ Not fitted for rendering...



# What is a mesh processing algorithm?

---

- ❖ Finding the border
  - ❖ Using various Adjacency relations
- ❖ Filling holes
- ❖ Cutting a mesh
- ❖ Finding Corners or computing curvature
- ❖ Remove noise
- ❖ Deform the mesh
- ❖ Remesh
  - ❖ Etc etc

# Designing data structures

---

## ❖ Data

- ❖ What i am going to keep behind to represent all the information
  - ❖ Redundancy vs efficiency
  - ❖ Explicit vs implicit

## ❖ Iterator/circulator

- ❖ How can i access to my mesh
  - ❖ Navigating your mesh
  - ❖ What is a *position* over a mesh



# The goal

---

- ❖ A framework to implement algorithms on Simplicial Complexes of order  $d=0..3$  in  $R^n$ :
  - ❖ Efficient code
  - ❖ Easy to understand
  - ❖ Flexible
  - ❖ Reusable

# Representing Simplicial Complexes

---

- ❖ A good problem.
- ❖ Meshes requires different information for different algorithms and purposes
  - ❖ Topology
  - ❖ Different geometric informations
  - ❖ Additional datas
- ❖ Templated solutions.
  - ❖ Generic algorithms on generic meshes

# Vertex

---

- ❖ What is a vertex?
  - ❖ position in n-space (almost always)
  - ❖ normal
  - ❖ color
  - ❖ quality
  - ❖ quadric
  - ❖ connectivity information (topology)
  - ❖ ...
- ❖ One may want any combination of attributes

# Vertex (cntd)

---

## ❖ How to do it?

- ❖ every user derives an empty VertexBase class to implement the vertex type
  - ❖ annoying cut and paste of code
  - ❖ need to agree upon interface (name of access functions)
  - ❖ potentially memory consuming (memory padding)
- ❖ multiple inheritance: not well supported in old compilers. It could be done now, but still dangerous sometimes...

# vertex (cntd)

- ❖ Classical MetaProgramming approach
  - ❖ Build a compile time a linear derivation chain from a set of attributes
  - ❖ All the desired attributes are passed as template class to the vertex:

```
typedef VertexSimpl< Vertex0, EdgeProto,  
vcg::vert::VFAdj, vcg::vert::Normal3f, vcg::vert::Color4b> MyVertex;
```

- ❖ The template parameters can be passed in any order
- ❖ More elegant
- ❖ Much more complex implementation
- ❖ Better typed: ex. the normal and the position have different type even if their structure is the very same

# Complexes

---

- ❖ PointSet
- ❖ EdgeMesh
- ❖ TriMesh
- ❖ TetraMesh
- ❖ As for simplices, they could be:  

```
template <int order,....> Complex{...};
```
- ❖ but they are not (at the moment)

```

class MyEdge;
class MyFace;
class MyVertex;
struct MyUsedTypes : public UsedTypes<
    Use<MyVertex>      ::AsVertexType,
    Use<MyEdge>        ::AsEdgeType,
    Use<MyFace>        ::AsFaceType>{};

class MyVertex : public Vertex<MyUsedTypes, vertex::Coord3f,
    vertex::Normal3f, vertex::BitFlags >{};

class MyFace : public Face< MyUsedTypes, face::FFAdj, face::VertexRef,
    face::BitFlags > {};

class MyEdge : public Edge<MyUsedTypes>{};

class MyMesh : public tri::TriMesh< vector<MyVertex>, vector<MyFace> ,
    vector<MyEdge> > {};

```

# Complex (ex: TriMesh)

- ❖ A complex is just a collection of simplices

```
public:  
  
    /// Set of vertices  
    VertContainer vert;  
    /// Real number of vertices  
    int vn;  
    /// Set of faces  
    FaceContainer face;  
    /// Real number of faces  
    int fn;  
    ...  
};
```

- ❖ Max and min order simplices are the only ones explicitly kept



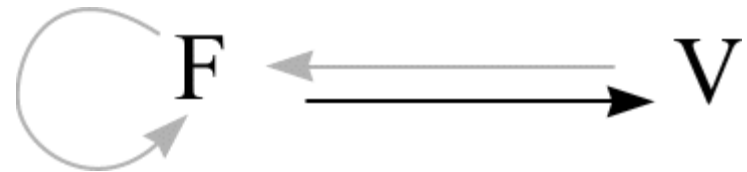
# containers

---

- ❖ Usually vectors
  - ❖ Most of the code works also with other generic containers
- ❖ Lazy deletion strategy
  - ❖ Object that have to be deleted are just marked and purged away later...
    - ❖ SetD() IsD() function over simplices
- ❖ No edges.
  - ❖ Only maximal complexes are usually kept
  - ❖ It could be discussed...

# Complex

- ❖ Topological relations stored optionally inside the simplexes
- ❖ Each Simplex knows its own geometric realization  
(a face contains pointers instead of indexes)

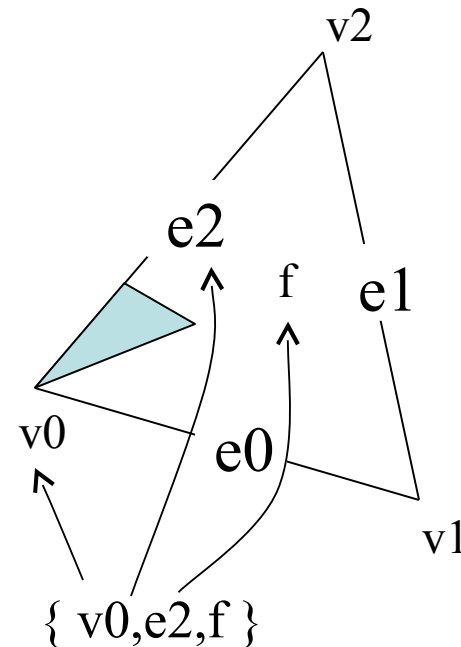


# Surfing a the mesh (I)

- ❖ All based on the concept of `pos` (position)
- ❖ a `pos` is a  $d$ -tuple:

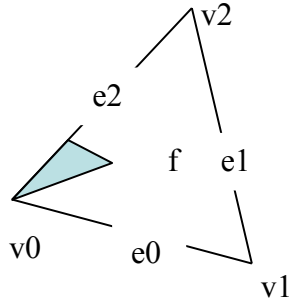
$$p = \{ s_0, \dots, s_d \}$$

such that each  $s_i$  is a reference to a  $d$ -simplex  
For triangle meshes is a triple of references to  
***vertex, edge, face***

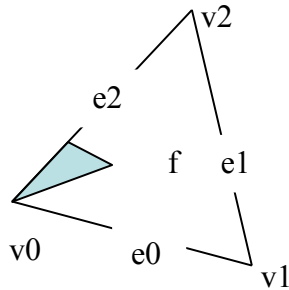
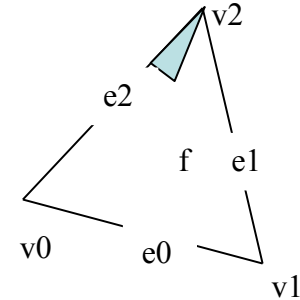


# Pos

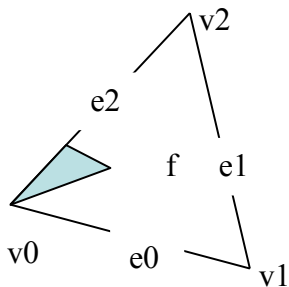
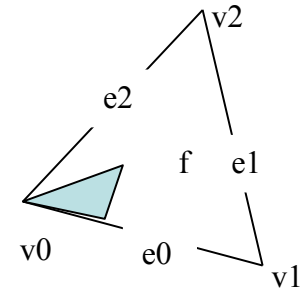
- ❖ any component of a pos can be changed only into another value to obtain another pos



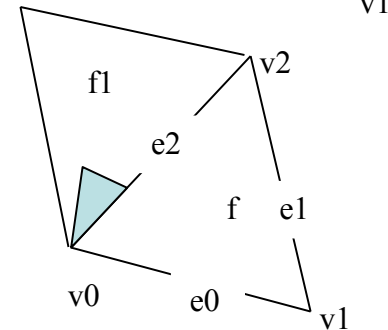
$$\underline{p.FlipV()} \\ \{v0, e2, f\} \rightarrow \{v2, e2, f\}$$



$$\underline{p.FlipE()} \\ \{v0, e2, f\} \rightarrow \{v0, e0, f\}$$



$$\underline{p.FlipE()} \\ \{v0, e2, f\} \rightarrow \{v2, e2, f1\}$$



# Pos

## ❖ Example: running over the faces around a vertex

<vcg/simplex/face/pos.h>

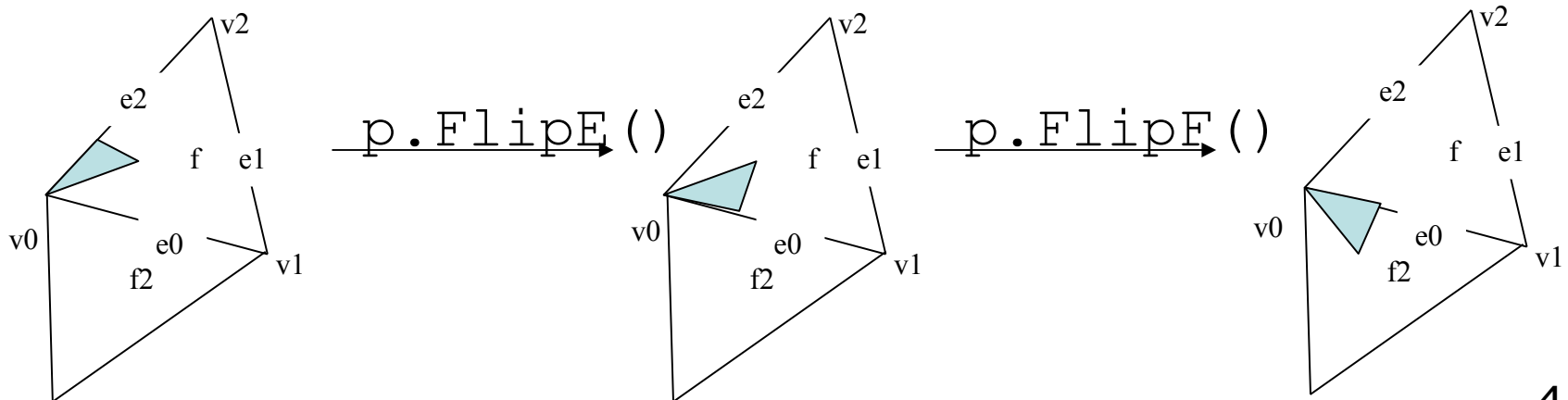
```
template <typename FaceType>  
class Pos {
```

...

```
void NextE()  
{  
    assert( f->V(z)==v || f->V((z+1)%3)==v );  
    FlipE();  
    FlipF();  
    assert( f->V(z)==v || f->V((z+1)%3)==v );  
}
```

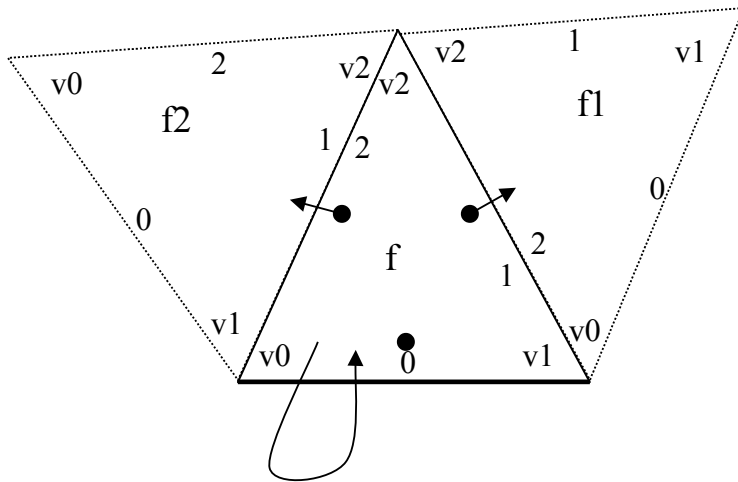
...

```
};
```



# FF Implementation

- ❖ Three pointers to face and three integers in each face:



`f.FFp(1) == &f1`

`f.FFi(1) == 2`

`f.FFp(2) == &f2`

`f.FFi(2) == 1`

`f.FFp(0) == &f`

`f.FFi(0) == -1`

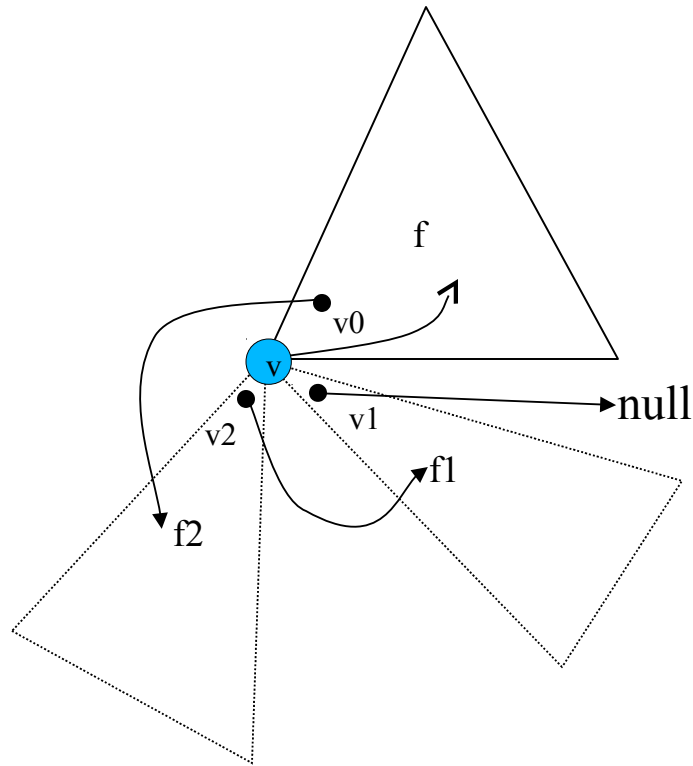
`f.FFp(i) -> f.FFp(f.FFi(i)) == &f`

# FF implementation

- ❖ Works also for non manifold situations.
  - ❖ The pointers in the FF forms a circular list ordered, monodirectional list.
- ❖ It does not hold anymore that
$$f.FFp(i) \rightarrow f.FFp(f.FFi(i)) == \&f$$
- ❖ The pos flip property do not hold any more...

# VF Implementation

- ❖ The list is distributed over the involved faces: No dynamic allocation



```
v.VFp() == &f
```

```
v.VFi() == 0
```

```
f.VFp(0) == &f2
```

```
f.VFi(0) == 2
```

```
f2.VFp(2) == &f2
```

```
f2.VFi(2) == 1
```

```
f1.VFp(1) == null
```

```
f1.VFi(1) == -1
```



# References

---

- ❖ <http://vcg.sf.net>
  - ❖ A wiki with tutorials
  - ❖ Feel free to spot out missing parts.
- ❖ [vcglib/apps/samples](http://vcglib/apps/samples)
  - ❖ A set of more or less simple examples.