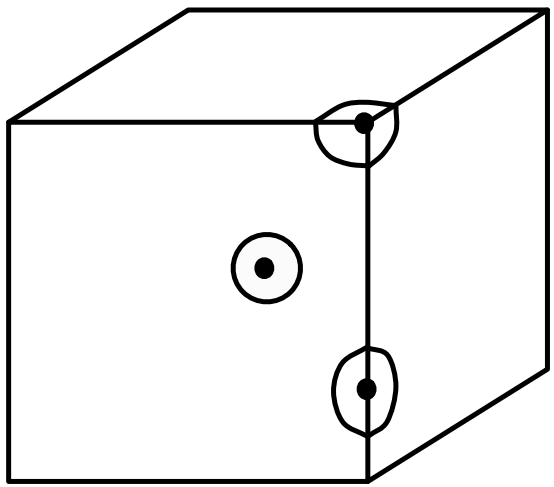
# Widged-Edge Data Structure

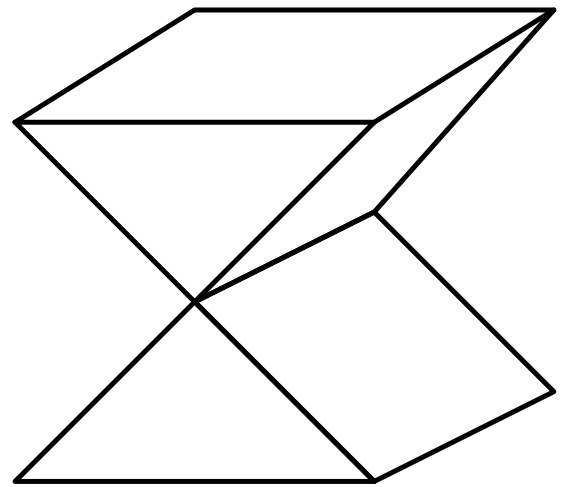
CS/CptS 442/542

October 18, 2011

## 2-Manifold Surfaces



2-Manifold



Not 2-Manifold

The neighborhood of surface points within radius $R$ of any (non-boundary) point $x$ on the surface form a disk topologicaly. Each (non-boundary) edge is adjacent to exactly two faces.

## Motivation: $O(1)$ adjacency queries for mesh elements

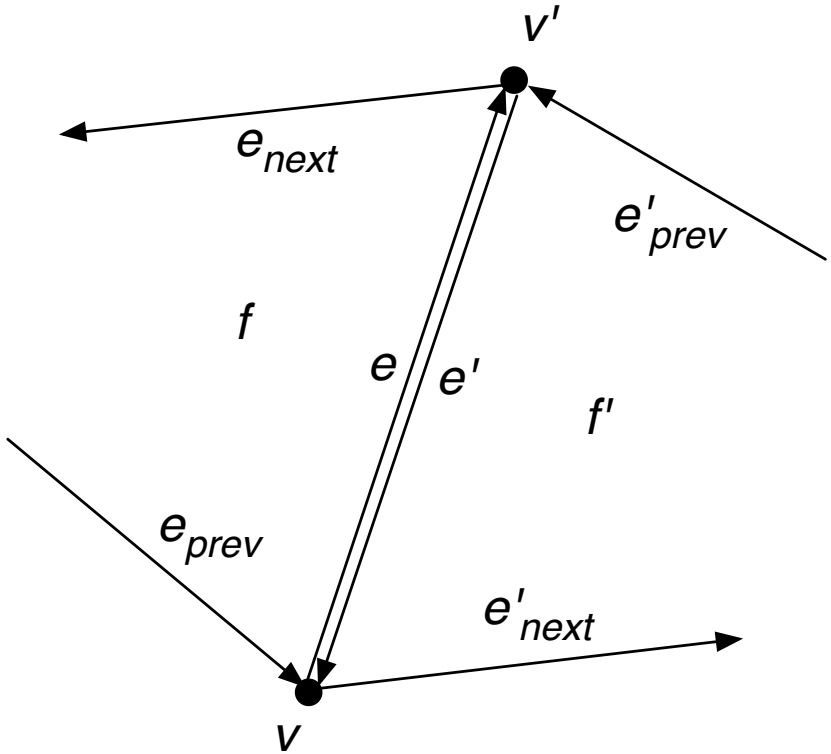
Given polygonal mesh with with vertices $V$, faces $F$, and edges $E$, we want to be able to perform the following queries in constant time:

- Given face $f \in F$, iterate over $f$'s vertices $v \in V$.
- Given vertex $v \in V$, find all adjacent faces $f \in F$.
- Given edge $e \in E$, find both adjacent faces $f \in F$.
- etc. . .

This will allow us to quickly process connected mesh primitives and/or perform local modifications.

## "Winged Edge" Data Structure Elements

The edge is the central element.

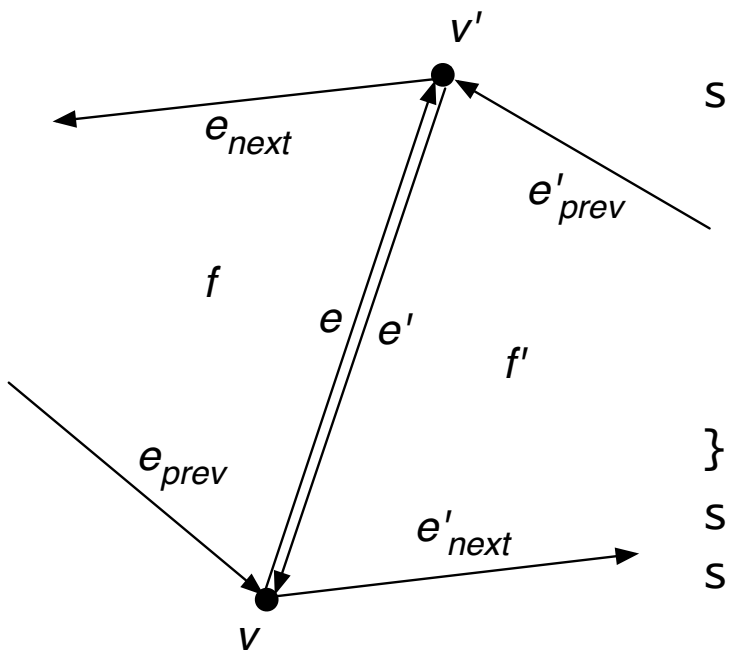



| | |
|---|---|
| $e$ | : edge |
| $v$ | : vertex |
| $f$ | : face on left |
| $e_{prev}$ | : previous edge |
| $e_{next}$ | : next edge |
| $e'$ | : symmetric edge |
| $f'$ | : face on left of $e'$ |
| $e'_{prev}$ | : previous edge to $e'$ |
| $e'_{next}$ | : next edge for $e'$ |

Each edge is represented twice (edge + symmetric edge).

## "Winged Edge" Data Structure

Co-recursive data structures.




```
struct Edge {
  Vertex *v;
  Face *f;
  Edge *prev, *next;
  Edge *sym;
  ...
};
struct Vertex { Edge *e;...};
struct Face {Edge *e; ...};
```

The `Edge` structure encodes most of the topology. The `Vertex` and `Face` structure reference one associated edge.

## Iterating over the edges of face `f`

```
Edge *start = f->e;
Edge *e = start;
do {
  visit(e);
  e = e->next;  // CCW order
} while (e != start);
```

- Iterate over vertices: `visit(e->v)`
- Iterate over neighboring faces: `visit(e->sym->f)`
- For CW order: `e = e->prev`

## Example: Compute normal for face `f`

Newell's algorithm: iterate over `f`'s vertices

```
f->norm = Vec3(0,0,0);
Edge *start = f->e;
Edge *e = start;
do {
   const Vec3& v = e->v->pos; // vertex position
   const Vec3& vnext = e->next->v->pos;
   f->norm.x += (v.y - vnext.y)*(v.z + vnext.z);
   f->norm.y += (v.z - vnext.z)*(v.x + vnext.x);
   f->norm.z += (v.x - vnext.x)*(v.y + vnext.y);
   e = e->next;
} while (e != start);
f->norm.normalize();
```

## Iterating over the edges emanating from vertex `v`

```
Edge *start = v->e;
Edge *e = start;
do {
  visit(e);
  e = e->prev->sym;  // CCW order
} while (e != start);
```

- Iterate over adjacent faces: `visit(e->f)`
- For CW order: `e = e->sym->next`
- What if mesh has boundary (not a closed mesh)? Symmetric edge could be null.

## Example: Compute vertex `v`'s normal

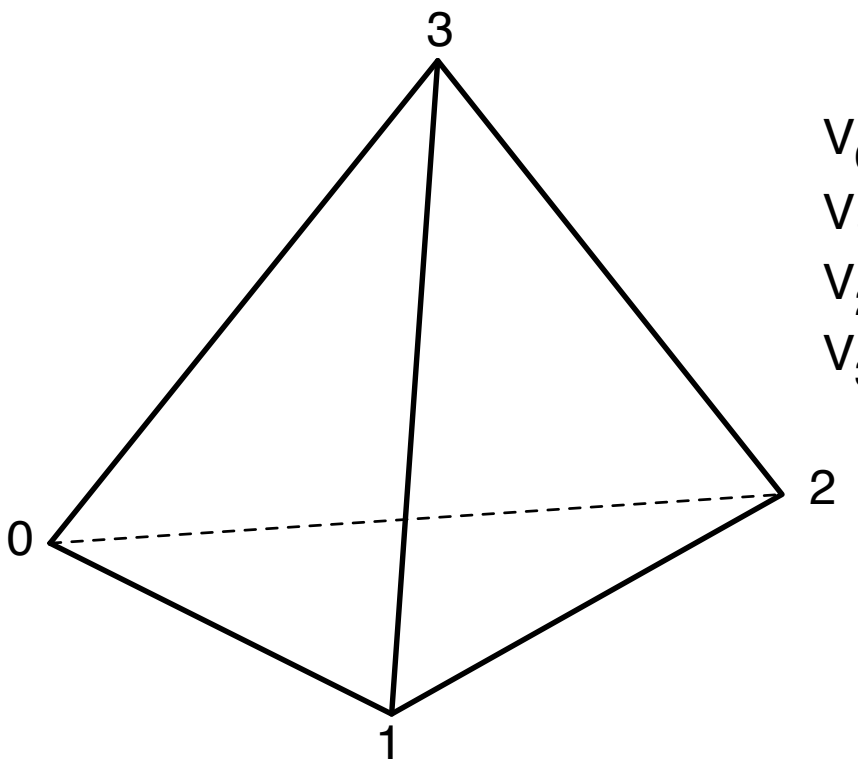
Average `v`'s adjacent face normals

(assumes face normals have been computed)

```
v->norm = Vec3(0,0,0);
Edge *e, *start;
e = start = v->e;
do {
   v->norm += e->f->norm;
   e = e->prev->sym;
} while (e != start);
v->norm.normalize();
```

# Tetrahedon

$|V| = 4,\ |F| = 4,\ |E| = 6.$



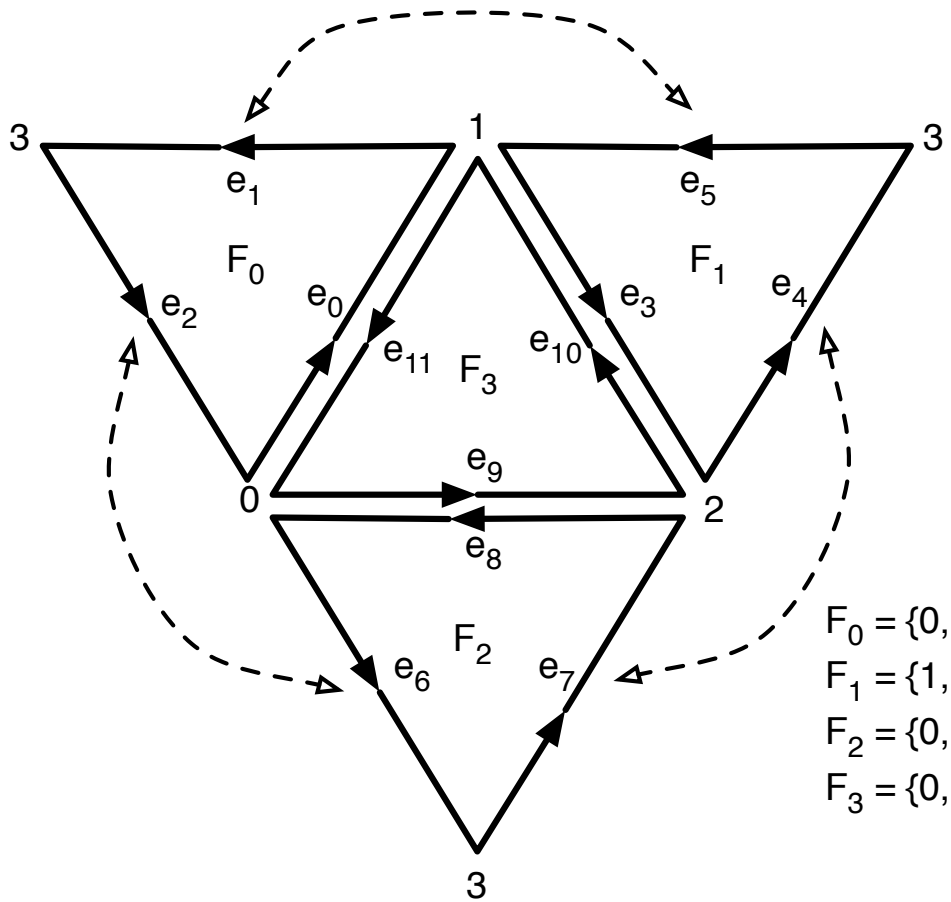

$V_0 = (\sqrt{3}/3, 0, 0)$ $F_0 = \{0, 1, 3\}$,

$V_1 = (-\sqrt{3}/6, 1/2, 0)$ $F_1 = \{1, 2, 3\}$,

$V_2 = (-\sqrt{3}/6, -1/2, 0)$ $F_2 = \{0, 3, 2\}$,

$V_3 = (0, 0, \sqrt{6}/3)$ $F_3 = \{0, 2, 1\}$

# Exploded Tetrahedon with "Winged-Edges"




CCW orientation
from visible side

| | "winged edges" |
|---|---|
| $F_0 = \{0, 1, 3\}$ | $0 \to 1,\ 1 \to 3,\ 3 \to 0$ |
| $F_1 = \{1, 2, 3\}$ | $1 \to 2,\ 2 \to 3,\ 3 \to 1$ |
| $F_2 = \{0, 3, 2\}$ | $0 \to 3,\ 3 \to 2,\ 2 \to 0$ |
| $F_3 = \{0, 2, 1\}$ | $0 \to 2,\ 2 \to 1,\ 1 \to 0$ |

## Mesh Data Structure

```
class Mesh {
  vector<Edge*> E;    // edges + symmetric edges
  vector<Vertex*> V;
  vector<Face*> F;
  ...
};
```

Adjusted Euler's formula for "winged-edge" polyhedra (closed mesh with no boundary):

$$|V| + |F| - |E|/2 = 2 - 2G$$

Genus $G = 0$ for sphere, $G = 1$ for torus, ...

## Build "Winged-Edge" Tetrahedron

1. Allocate space for mesh elements.

```
E.resize(12);
V.resize(4);
F.resize(4);

for (int i = 0; i < 12; i++)
  E[i] = new Edge;
for (int i = 0; i < 4; i++)
  V[i] = new Vertex;
for (int i = 0; i < 4; i++)
  F[i] = new Face;
```

## Build Tetrahedron continued...

2. Connect all edge ptrs

```
                v      f     prev  next   sym
E[0]->set(V[0], F[0], E[2], E[1], E[11]);
E[1]->set(V[1], F[0], E[0], E[2], E[5]);
E[2]->set(V[3], F[0], E[1], E[0], E[6]);
E[3]->set(V[1], F[1], E[5], E[4], E[10]);
...
E[11]->set(V[1], F[3], E[10], E[9], E[0]);
```

3. Connect each face's edge ptr

```
F[0]->e = E[0];
F[1]->e = E[2];
F[2]->e = E[7];
F[3]->e = E[9];
```

## Build Tetrahedron continued...

4. Connect each vertex's edge ptr

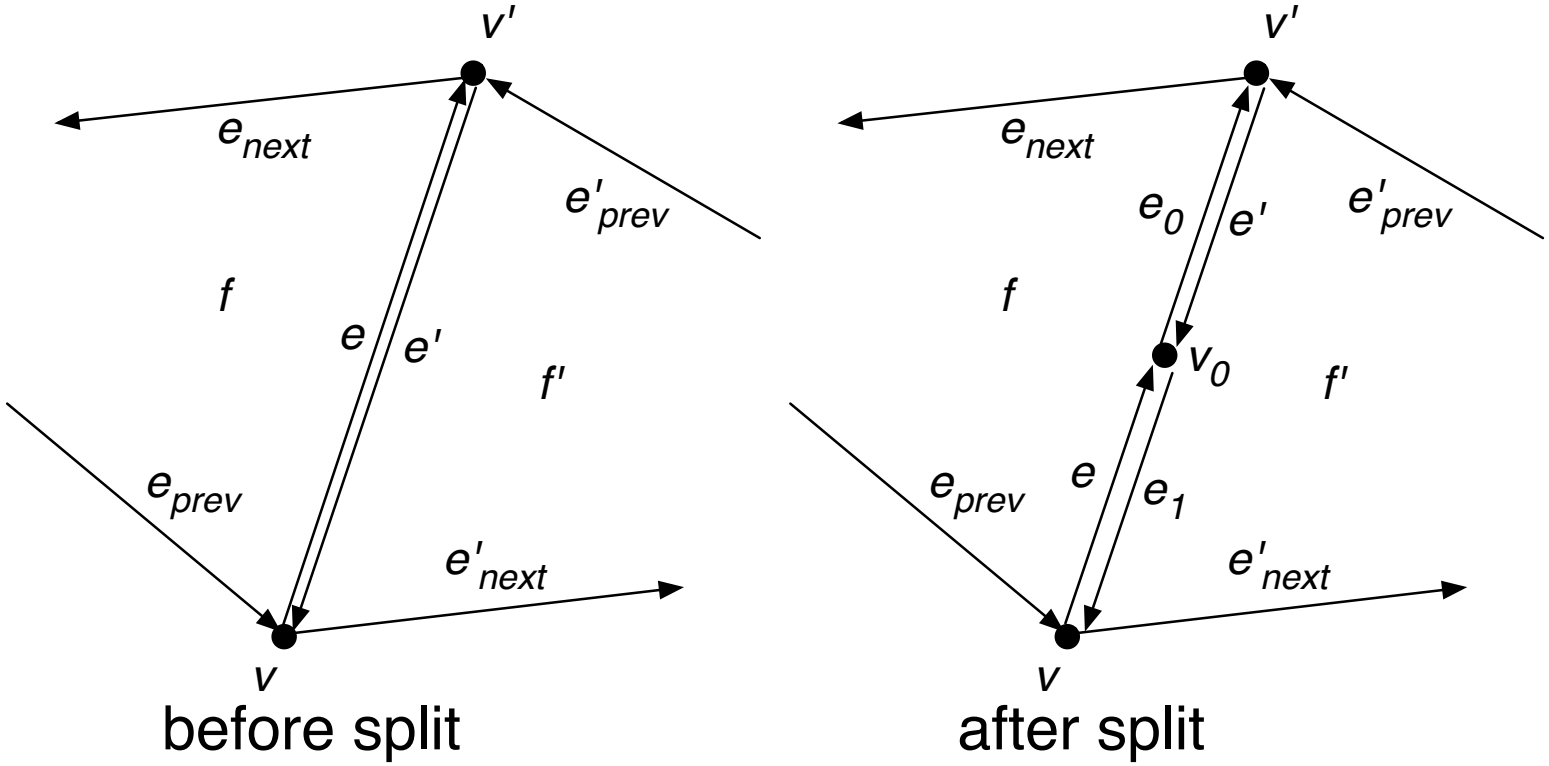
```
V[0]->e = E[0];
V[1]->e = E[1];
V[2]->e = E[3];
V[3]->e = E[2];
```

5. Set other attribute information

```
V[0]->pos = Vec3(SQRT3/3, 0, 0);
V[1]->pos = Vec3(-SQRT3/6, 0.5, 0);
V[2]->pos = Vec3(-SQRT3/6, -0.5, 0);
V[3]->pos = Vec3(0, 0, SQRT6/3);
```

   May also include colors, normals, texture coordinates, etc...

## Edge Split




Two new edges $e_0$, $e_1$ (we recycle edges $e$ and $e'$) and one new vertex $v_0$.

`splitEdge(Edge *e, const Vec3& pos)`

`pos` = position of new vertex

1. Allocate two new edges and a vertex.

```
Edge *e0 = new Edge;
Edge *e1 = new Edge;
Vertex *v0 = new Vertex;
E.push_back(e0);
E.push_back(e1);
V.push_back(v0);
```

2. Set new attribute information, *e.g.* vertex position

```
v0->pos = pos;
```

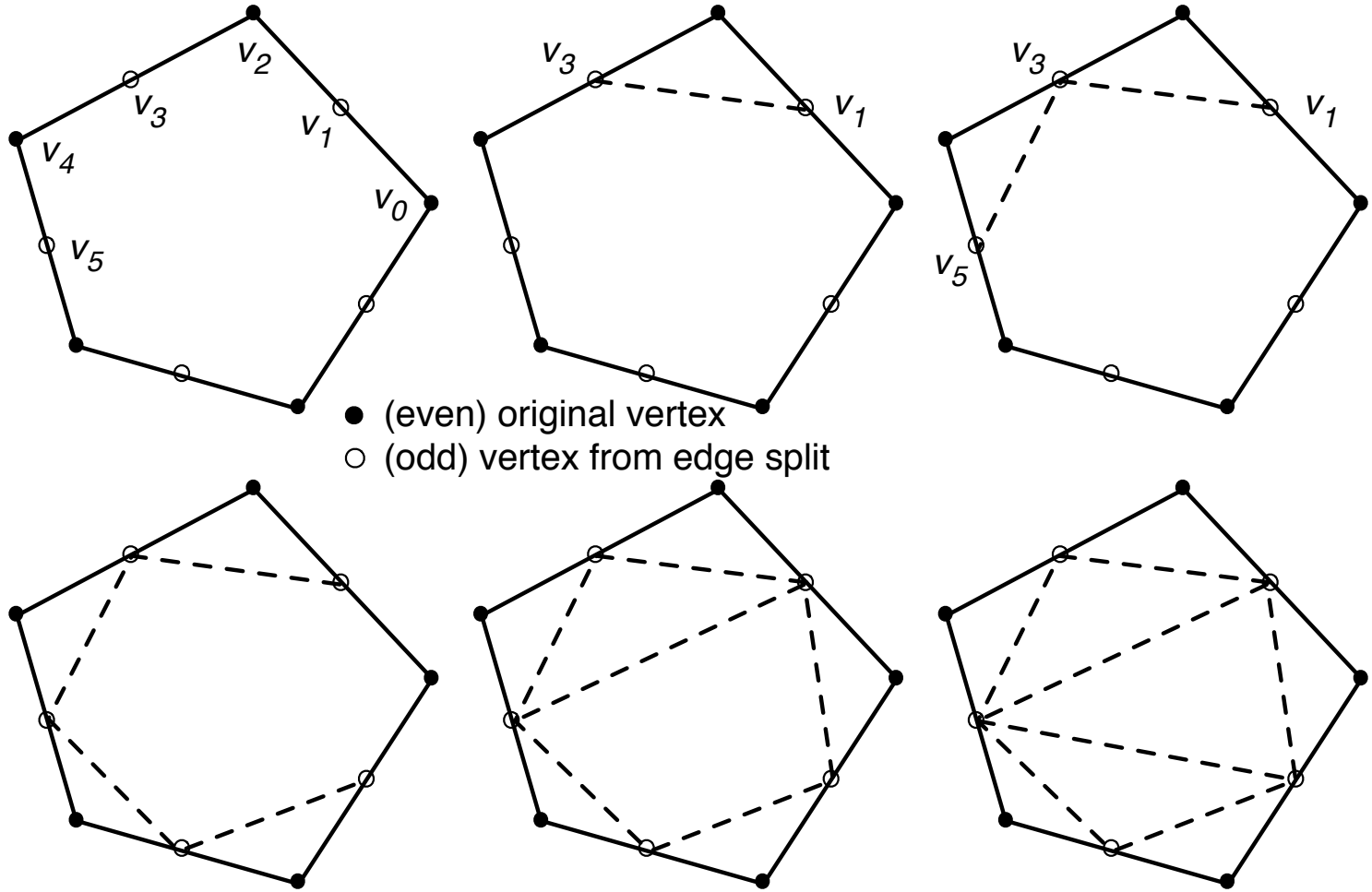
`splitEdge(Edge *e, const Vec3& pos)` **continued...**

3. Set and modify ptrs, be very careful!

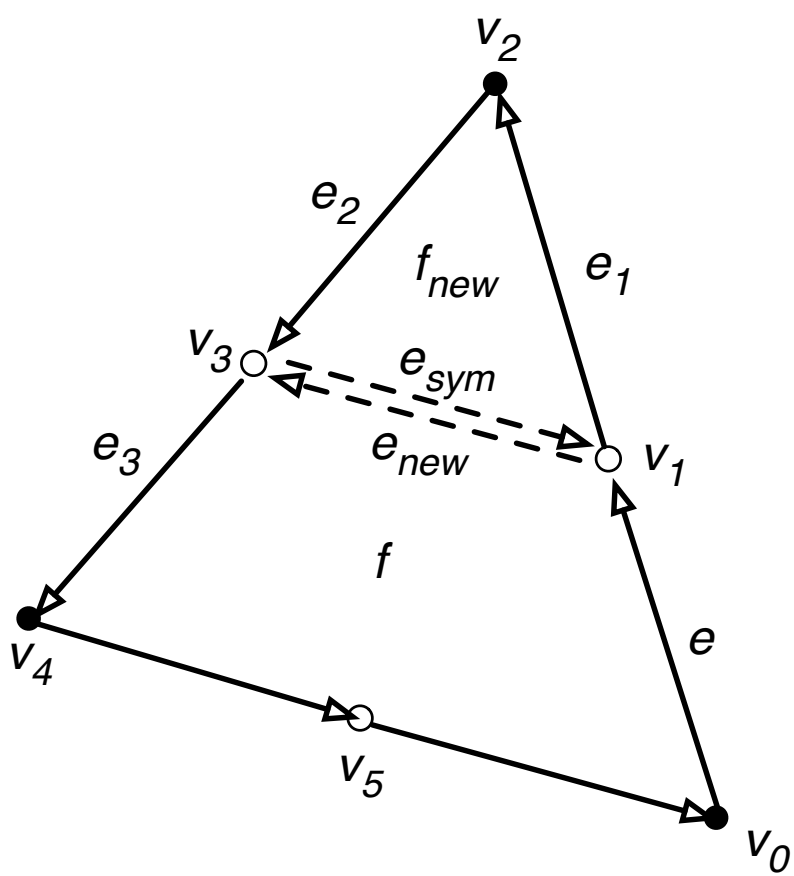
```
v0->e = e0;
e0->set(v0, e->f, e, e->next, e->sym);
e1->set(v0, e->sym->f, e->sym, e->sym->next, e);

e->next->prev = e0;
e->sym->next->prev = e1;
e->next = e0;
e->sym->next = e1;
e->sym->sym = e0;
e->sym = e1;
```

# Triangulating face after edge split



# Triangulating face `f`

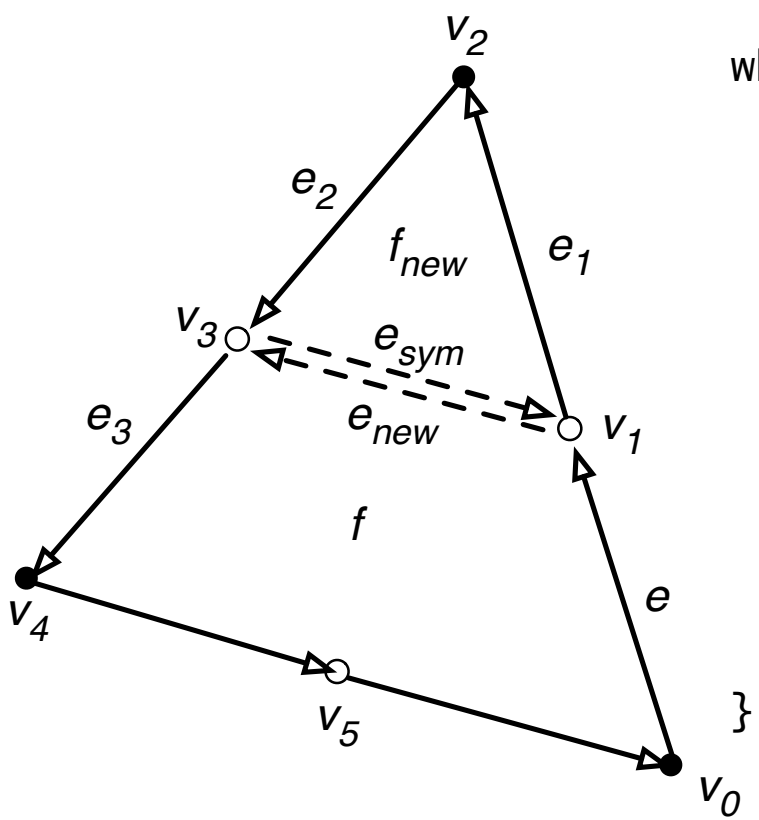



```
while (true) {
    Edge *e = f->e;
    Edge *e1 = e->next;
    Edge *e2 = e1->next;
    Edge *e3 = e2->next;

    if (e == e3) break;

    Vertex *v1 = e1->v;
    Vertex *v3 = e3->v;
    Edge *enew = new Edge;
    Edge *esym = new Edge;
    Face *fnew = new Face;
    ...
}
```

# Triangulating face `f` continued...




```
while (true) {
    ...
    enew->set(v1, f, e, e3, esym);
    esym->set(v3, fnew, e2, e1, enew);
    fnew->e = e1;
    e1->prev = e2->next = esym;
    e3->prev = e->next = enew;
    e1->f = e2->f = fnew;
    f->e = enew;
    F.push_back(fnew);
    E.push_back(enew);
    E.push_back(esym);
}
```