Convex Hulls (3D)

O'Rourke, Chapter 4

Outline

- Correction
- Polyhedra
  - Polytopes
  - Euler Characteristic
- (Oriented) Mesh Representation

Correction

For implementing the trapezoidalization, we described using a sorting function which changes dynamically:

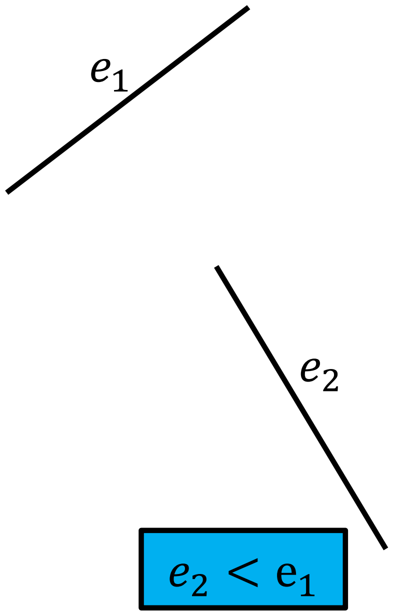
```
float sweepHeight;
typedef function< bool ( const EKey & , const EKey & ) > EComparator;
EComparator eComparator = [&]( const EKey &k1 , const EKey &k2 )
{
        // Compare the keys using the current value of sweepHeight
};
```

Correction

This is not necessary.

We could check if the y-spans of the two edges overlap.

- If they do not, call the lower edge "first".

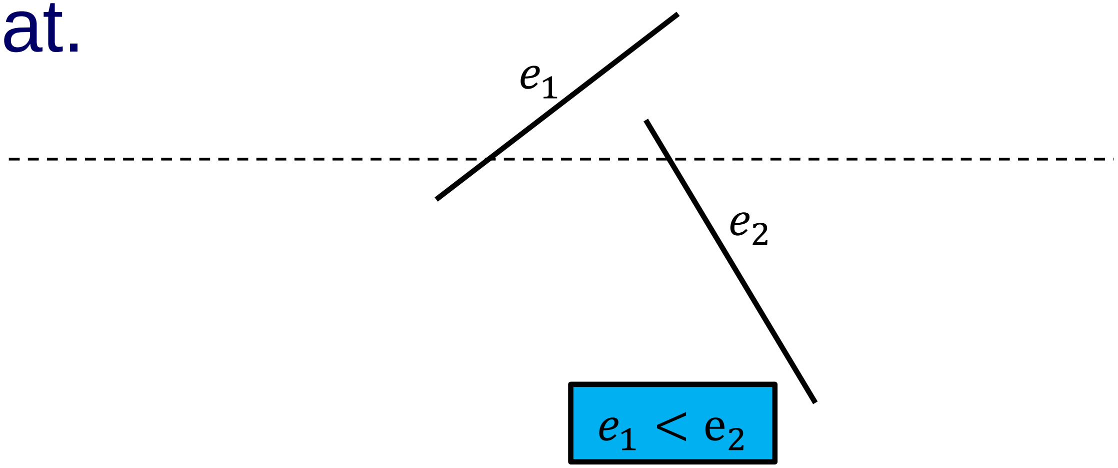


Correction

This is not necessary.

We could check if the y-spans of the two edges overlap.

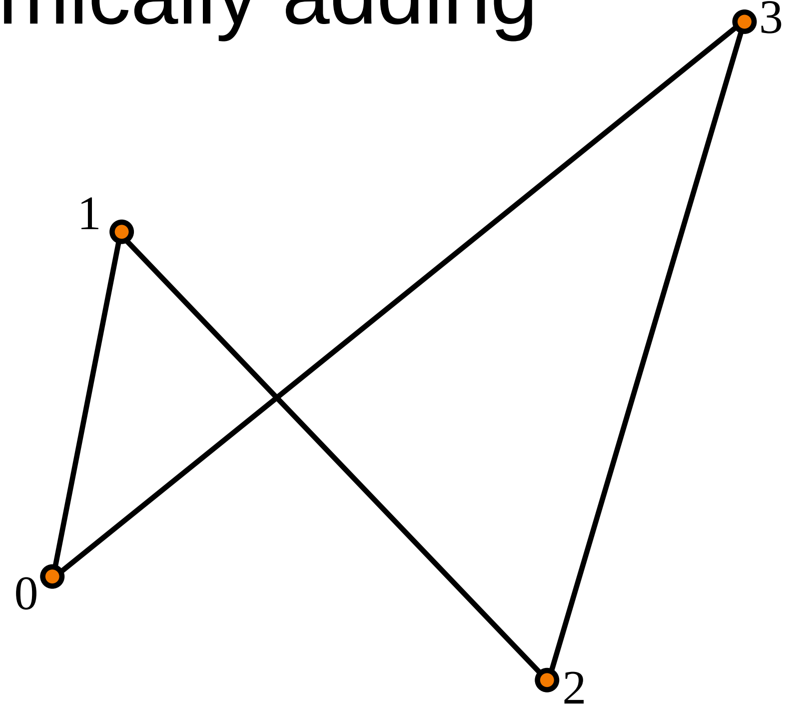
- If they do not, call the lower edge "first".
- Otherwise, draw a horizontal line through some point on the overlap of the y-spans and sort based on that.



Correction

However…

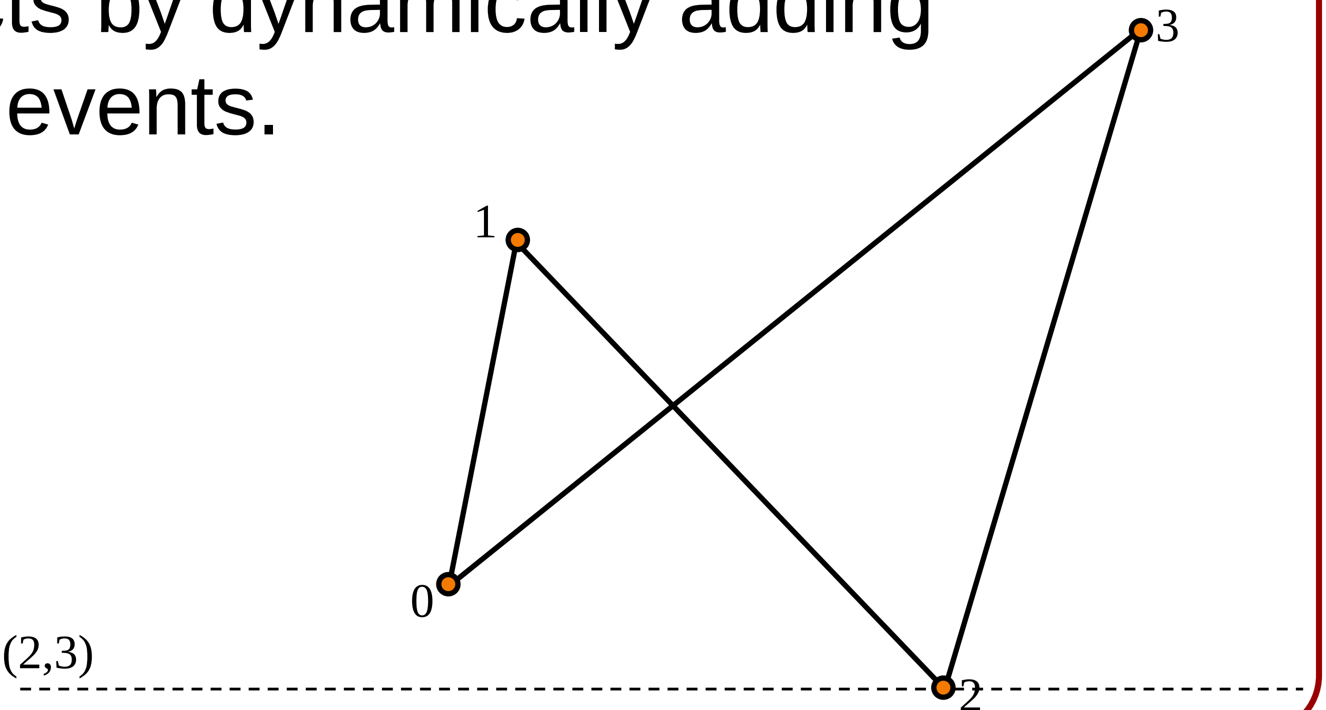
We can also use sweep-line algorithm to check if a closed (piecewise-linear) curve self-intersects by dynamically adding intersection events.



Correction

However…

We can also use sweep-line algorithm to check if a closed (piecewise-linear) curve self-intersects by dynamically adding intersection events.

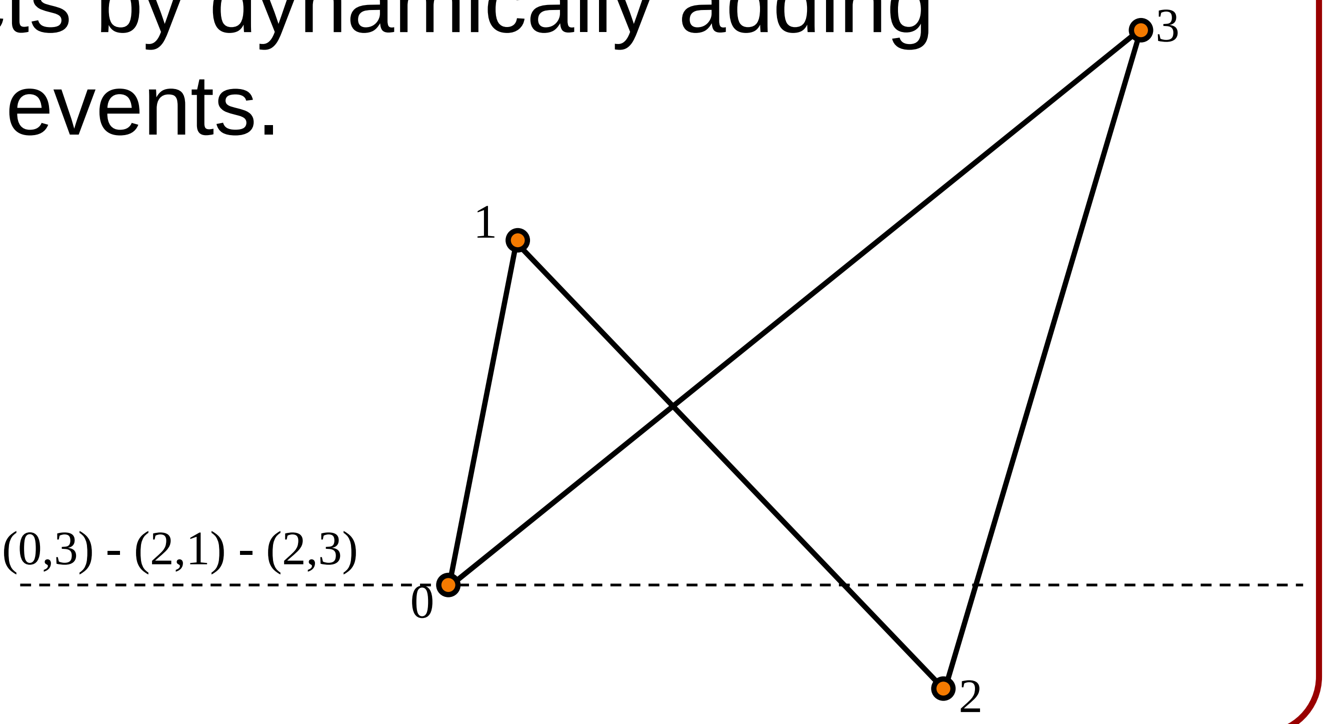


(2,1) - (2,3)

Correction

However…

We can also use sweep-line algorithm to check if a closed (piecewise-linear) curve self-intersects by dynamically adding intersection events.

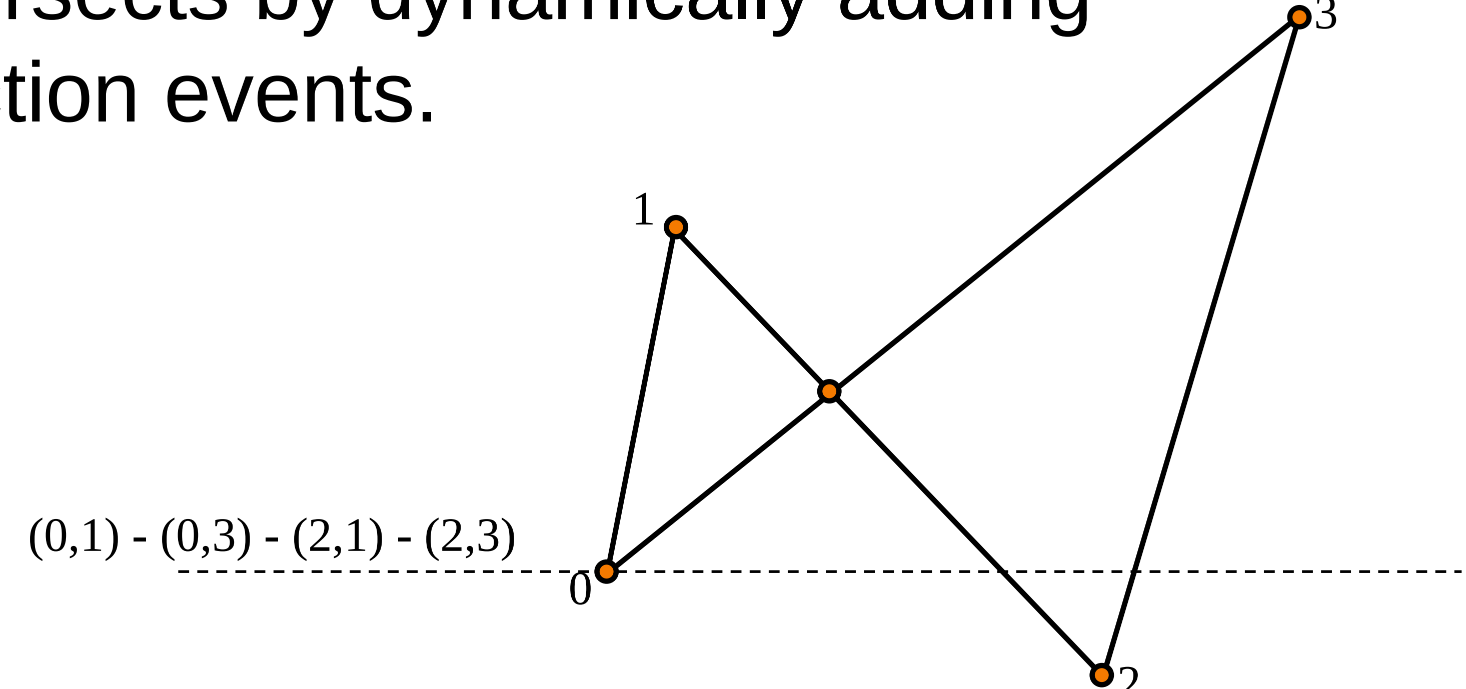


(0,1) - (0,3) - (2,1) - (2,3)

Correction

However…

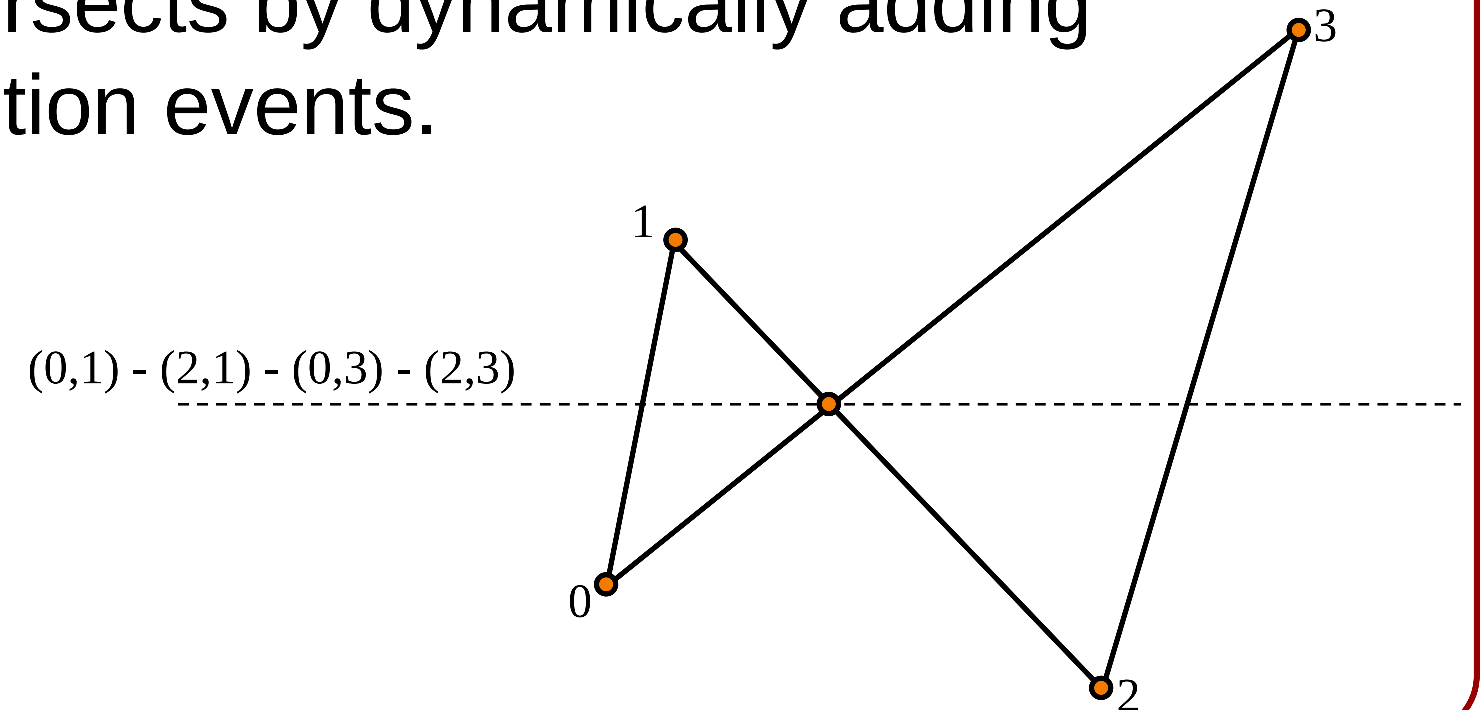
We can also use sweep-line algorithm to check if a closed (piecewise-linear) curve self-intersects by dynamically adding intersection events.



Correction

However…

We can also use sweep-line algorithm to check if a closed (piecewise-linear) curve self-intersects by dynamically adding intersection events.

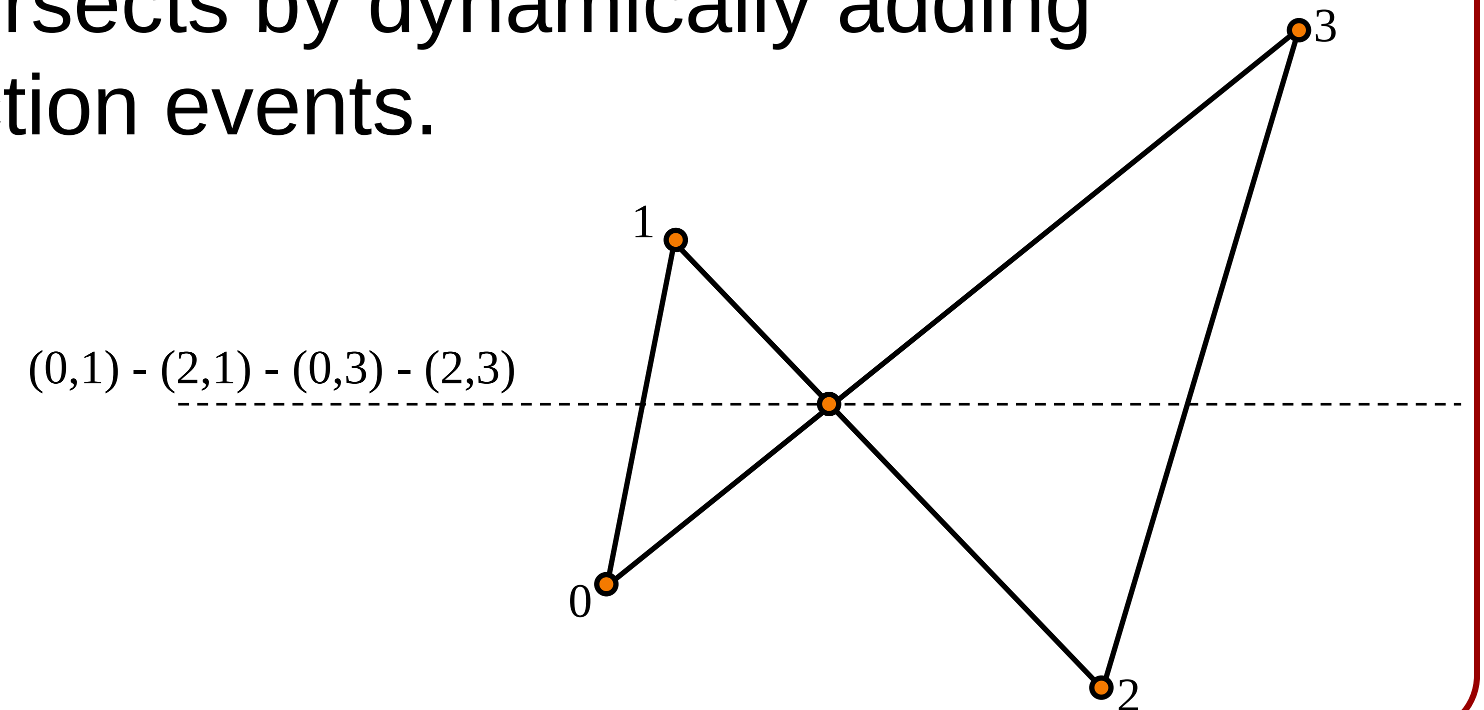


Correction

However…

We
ch
self-intersects by dynamically adding intersection events.

> Note: Only need to check for intersections
> 1. Between adjacent edges in the active-edge list
> 2. Around newly added/removed edges



Polyhedra

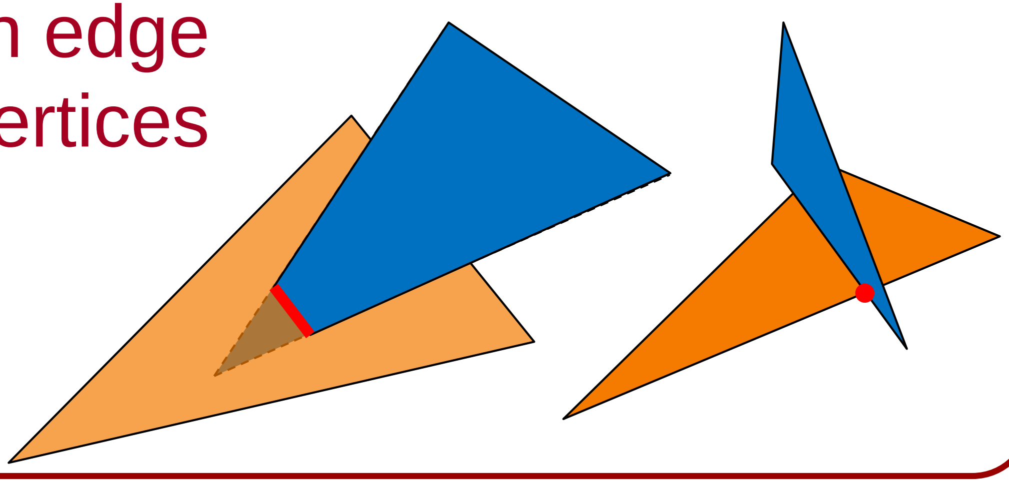
Definition:

A *polyhedron* is a solid region in 3D space whose boundary is made up of planar polygonal faces comprising a connected 2D manifold.

Polyhedra

The boundary of a polyhedron can be expressed as a combination of vertices, edges, and faces:

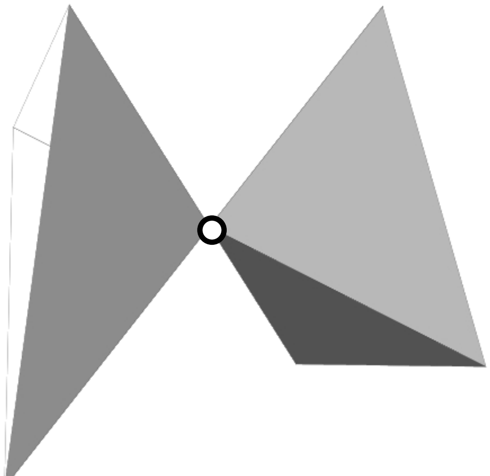
- Intersections are proper:
  - Elements don't overlap, or
  - They share a single vertex, or
  - They share an edge and the two vertices



Polyhedra

The boundary of a polyhedron can be expressed as a combination of vertices, edges, and faces:

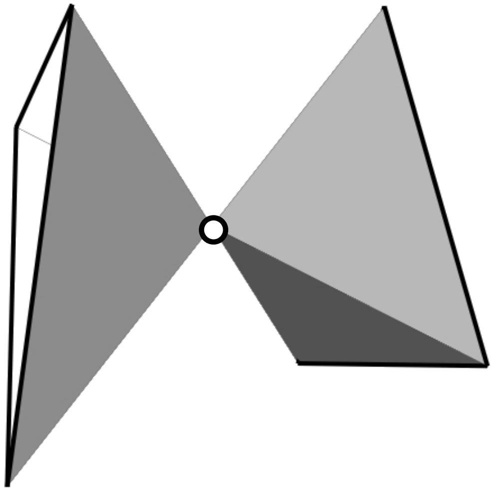
- Intersections are proper
- Locally manifold:
  - Edges around a vertex can be sorted to match their incidence on adjacent faces.



Polyhedra

The boundary of a polyhedron can be expressed as a combination of vertices, edges, and faces:

- Intersections are proper
- Locally manifold:
  - Edges around a vertex can be sorted to match their incidence on adjacent faces.

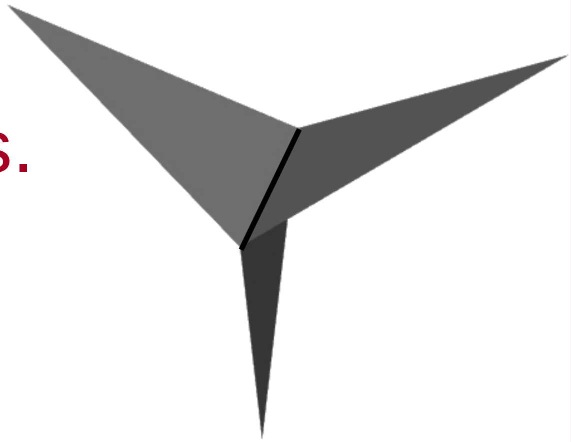


Alternatively, the subgraph of the dual obtained by restricting to the adjacent faces (the link) is connected.

Polyhedra

The boundary of a polyhedron can be expressed as a combination of vertices, edges, and faces:

- Intersections are proper
- Locally manifold:
  - Edges around a vertex can be sorted to match their incidence on adjacent faces.
  - Exactly two faces meet at each edge.



Polyhedra

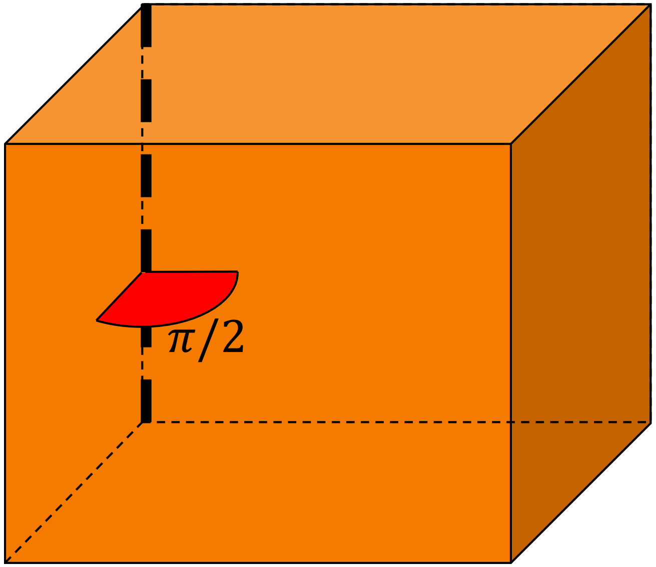
The boundary of a polyhedron can be expressed as a combination of vertices, edges, and faces:

- Intersections are proper
- Locally manifold
- Globally connected

Definition

Definition:

Given an edge on a polyhedron, the *dihedral angle* of the edge is the internal angle between the two adjacent faces.

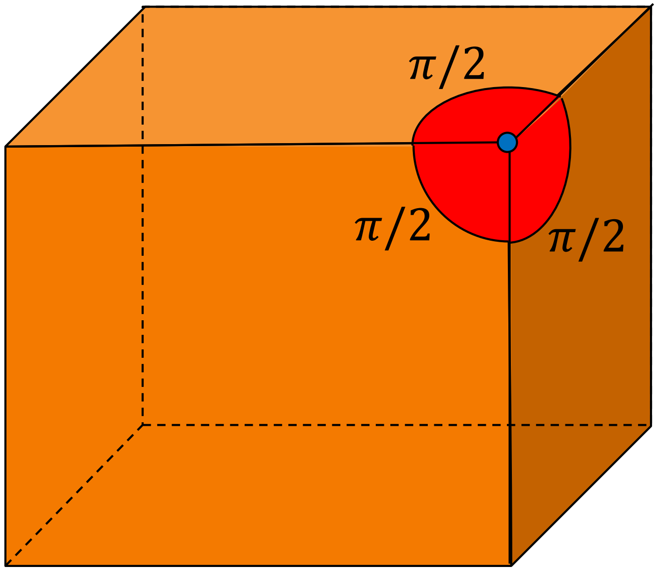


Aside:
The dihedral angle is a discrete measure of mean curvature.

Definition

<u>Definition</u>:

Given a vertex on a polyhedron, the *deficit angle* at the vertex is 2π minus the sum of angles around the vertex.



$\Rightarrow \pi/2$

<u>Aside</u>:
The deficit angle is a discrete measure of Gauss curvature.

Polytopes

A convex polyhedron is a *polytope*:

- Non-negative mean curvature:
  All dihedral angles are less than or equal to π.
  (Necessary and sufficient.)
- Non-negative Gaussian curvature:
  Sum of angles around a vertex is at most 2π.
  (Necessary but not sufficient).

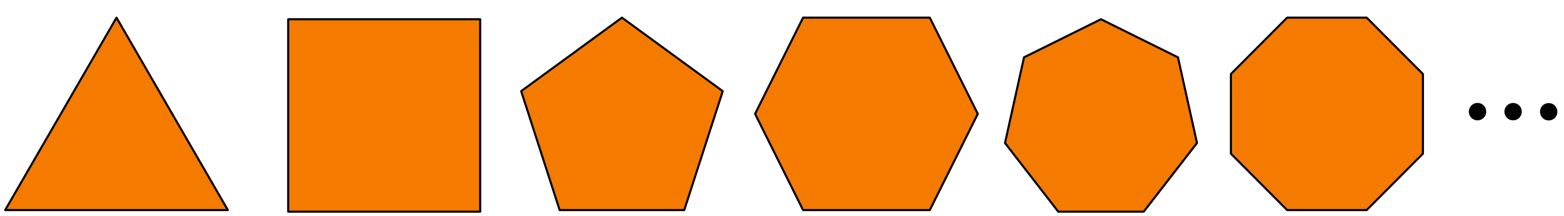
Schwarz Lantern

Image courtesy of https://www.cut-the-knot.org/

Platonic Solids

Definition:

A *regular polygon* is a polygon with equal sides and equal angles.



Platonic Solids

Definition:

A *regular polygon* is a polygon with equal sides and equal angles.

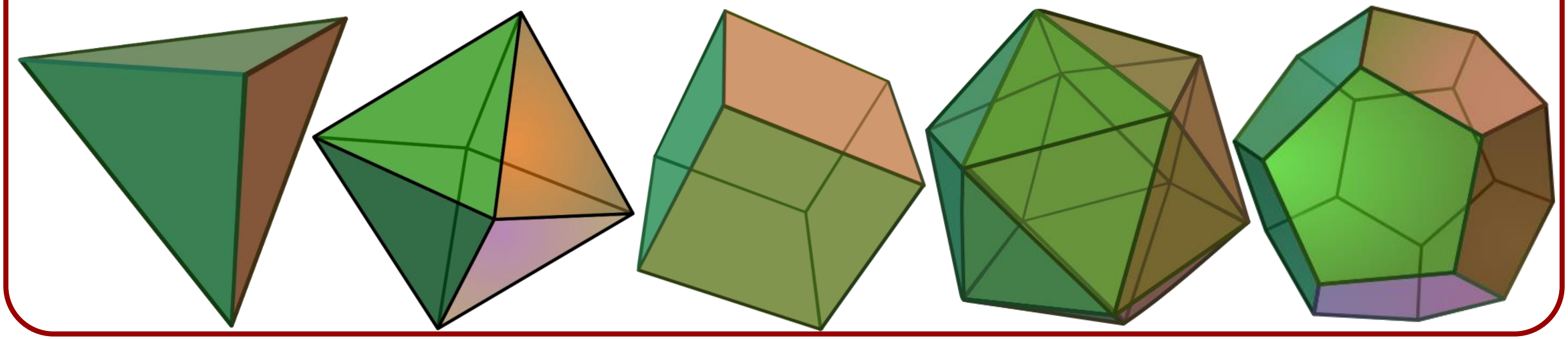
A *regular polyhedron* is a convex polyhedron, with all faces congruent regular polygons and vertices having the same valence.

Platonic Solids

Claim:

The five platonic solids are the only regular polyhedra.

[Images courtesy of Wikipedia]



Platonic Solids

Proof:

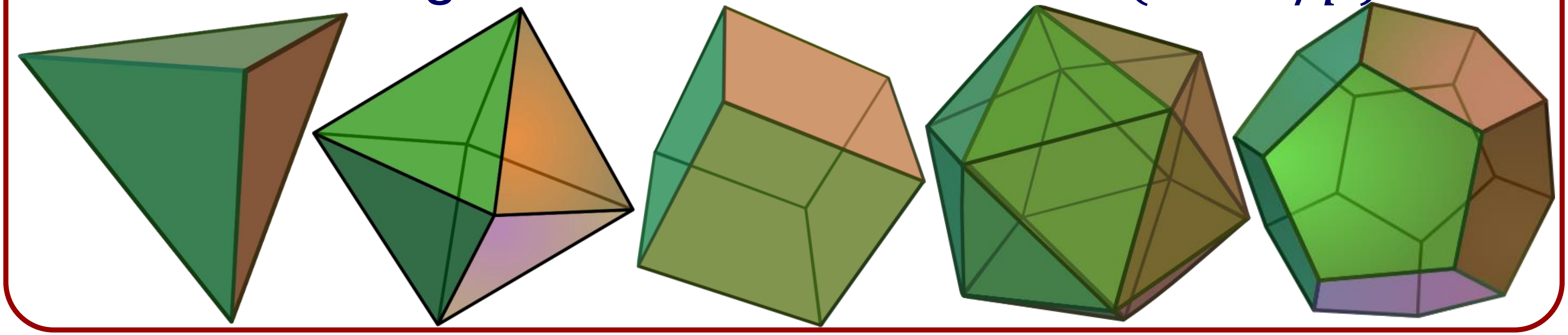
Assume each face is p-sided:

$\Rightarrow$ The sum of angles in a face is $\pi(p-2)$

$\Rightarrow$ The angle at each vertex is $\pi(1-2/p)$

Assume each vertex has valence v:

$\Rightarrow$ The angle-sum at a vertex is $v\pi\,(1-2/p)$

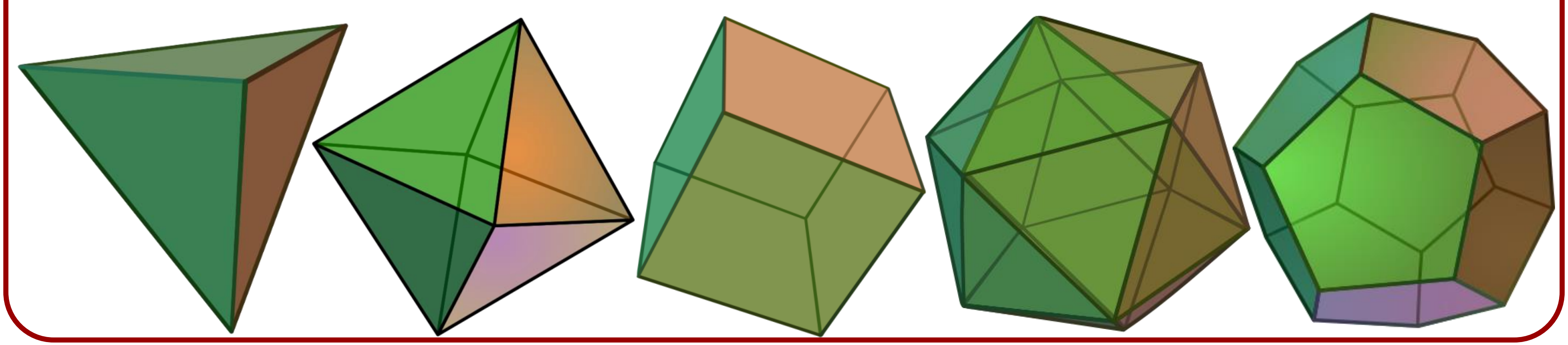


Platonic Solids

Proof:

Since the polyhedron is convex:

$$v\pi(1 - 2/p) < 2\pi \iff v(1 - 2/p) < 2$$
$$\iff v(p - 2) < 2p$$
$$\iff vp - 2v - 2p < 0$$
$$\iff (p - 2)(v - 2) - 4 < 0$$



Platonic Solids

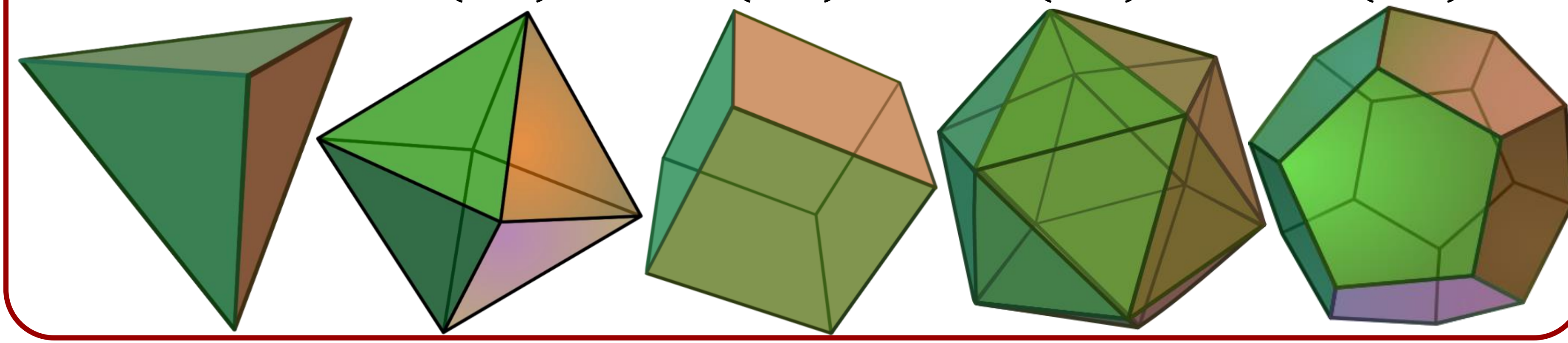
Proof:

Since the polyhedron is convex:

$$(p-2)(v-2)-4<0$$

Since $p, v \geq 3$, valid options are (p, v):

$(3,3)$ $(3,4)$ $(4,3)$ $(3,5)$ $(5,3)$



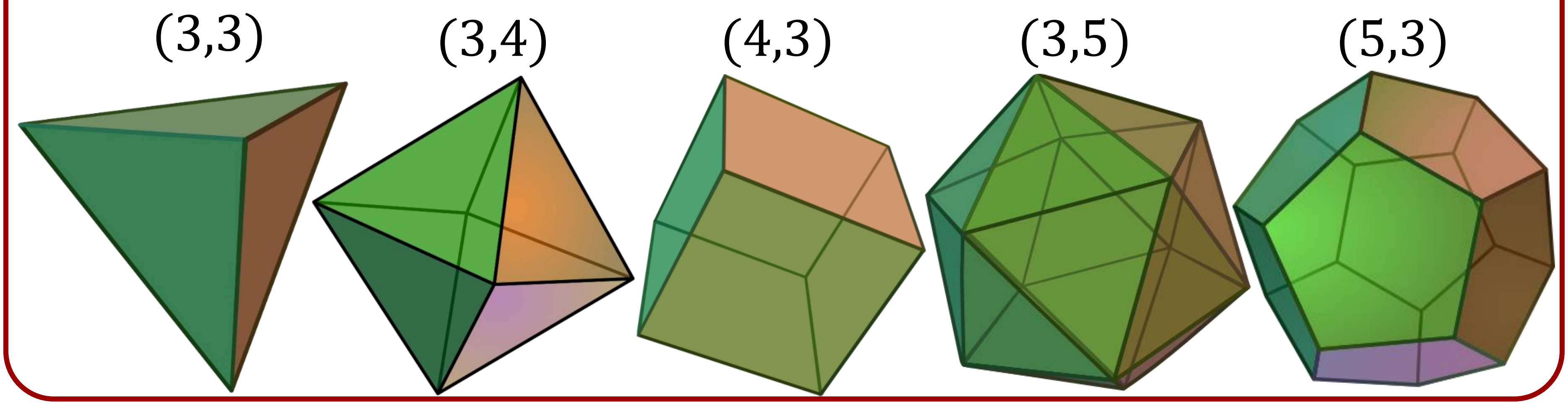
Platonic Solids

The platonic solids come in dual pairs, where one solid is obtained from the other by replacing faces with vertices:

Cube ↔ Octahedron

Icosahedron ↔ Dodecahedron

Tetrahedron ↔ Tetrahedron



Topological Polyhedra

The boundary of a polyhedron can be expressed as a combination of vertices, edges, and faces:

- Intersections are proper } Geometric
- Locally manifold
- Globally connected } Topological

Topological Polyhedra

If we ignore the vertex positions, we get a combinatorial structure composed of faces (cells), edges, and vertices.*

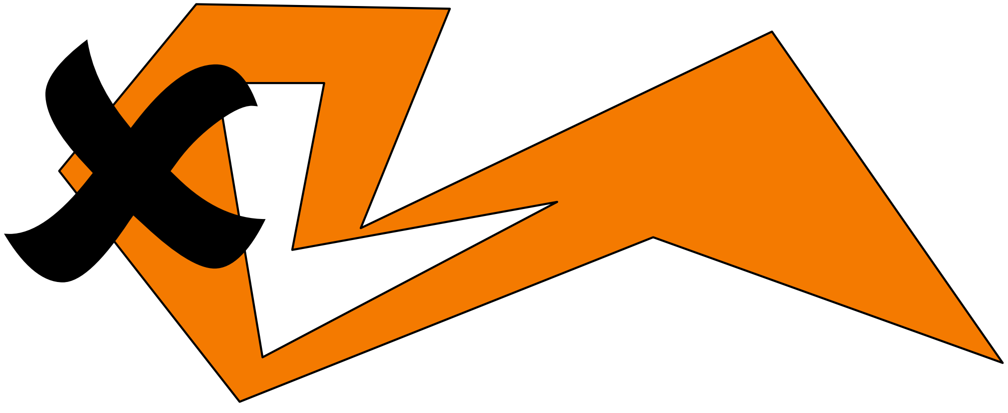
[Nivoliers and Levy, 2013]

*These are CW complexes. (And, if faces are triangles, these are simplicial complexes).

Topological Polyhedra

<u>Properties (CW Complex)</u>:

- Faces intersect at edges and vertices.
- Edges are topologically line segments and intersect at vertices.
- Interiors of faces have disk-topology and the boundary is a polygon made up of edges.



Topological Polyhedra

Properties (Manifold):

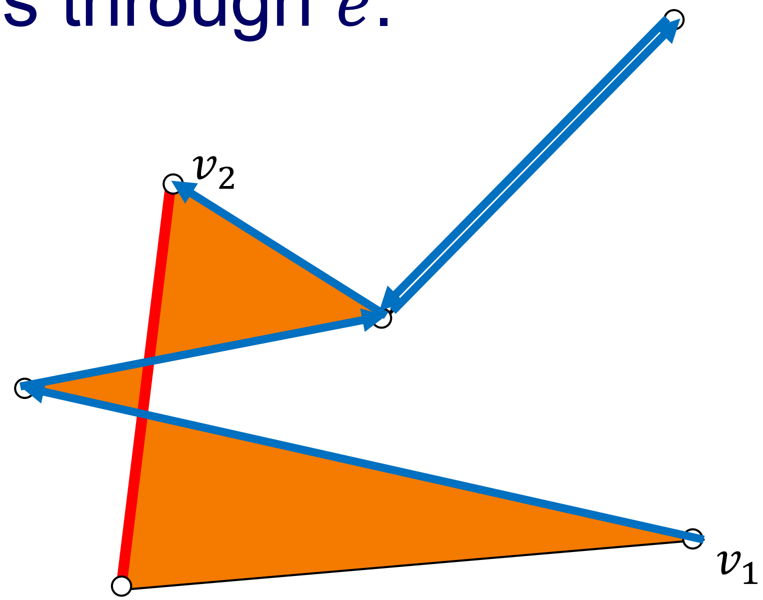
- Each vertex is on the boundary of some edge.
- Each edge is on the boundary of some face.
- Edges around a vertex can be sorted.
- An edge is on the boundary of two faces.

Topological Polyhedra

<u>Note</u>:

Given a topological polygon P, and given an edge $e \in P$ that only occurs once on P:

For any vertices $v_1, v_2 \in P$ there is a path from v_1 to v_2 that doesn't pass through e.



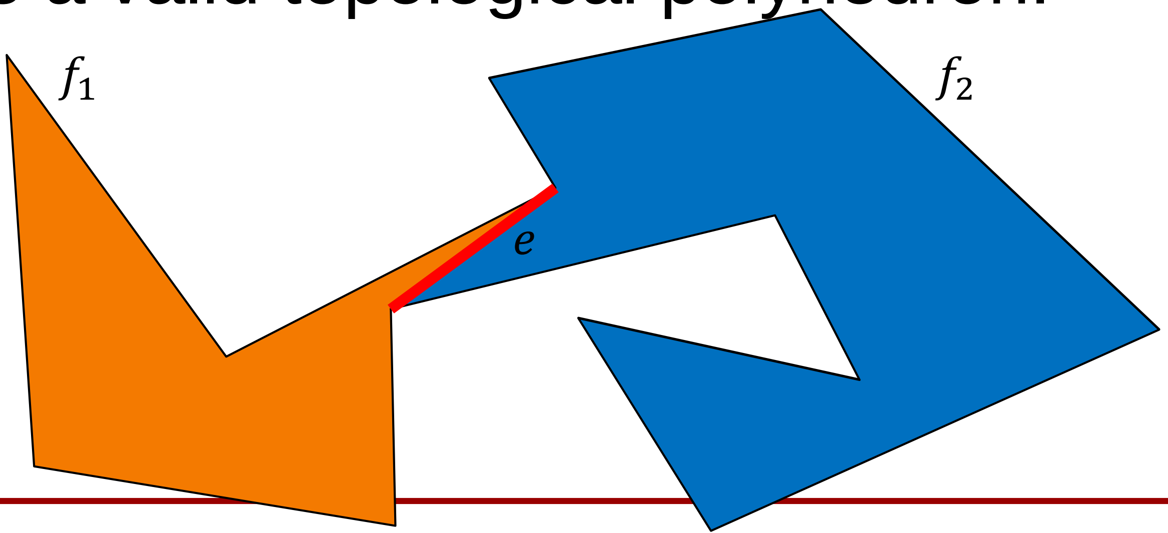
Topological Polyhedra

<u>Claim</u>:

If f_1 and f_2 are distinct faces of a topological polyhedron which share an edge e, then:

- replacing f_1 and f_2 with $f_1 \cup f_2$, and
- removing e from the edge list,

we still have a valid topological polyhedron.



Topological Polyhedra

Proof (CW Complex):

The edges/vertices of $f_1 \cup f_2$ are in the complex.

*Since the intersection $f_1 \cap f_2$ is connected and the interiors of f_1 and f_2 have disk-topology, the interior of $f_1 \cup f_2$ also has disk-topology.

*This is just a sketch of the proof.

Topological Polyhedra

Proof (CW Complex):

The boundary of $f_1 \cup f_2$ is connected.

- Let $v \in e$ be an end-point.
- For $v_1, v_2 \in f_1 \cup f_2$, there is a curve connecting v to each v_i that does not contain the edge e.
- Concatenating the two curves we connect v_1 to v_2 along the boundary of $f_1 \cup f_2$.

Topological Polyhedra

Proof (Manifold):

The smaller polyhedron still passes through all the vertices.

The edge e is removed and all other edges remain adjacent to a face.

Topological Polyhedra

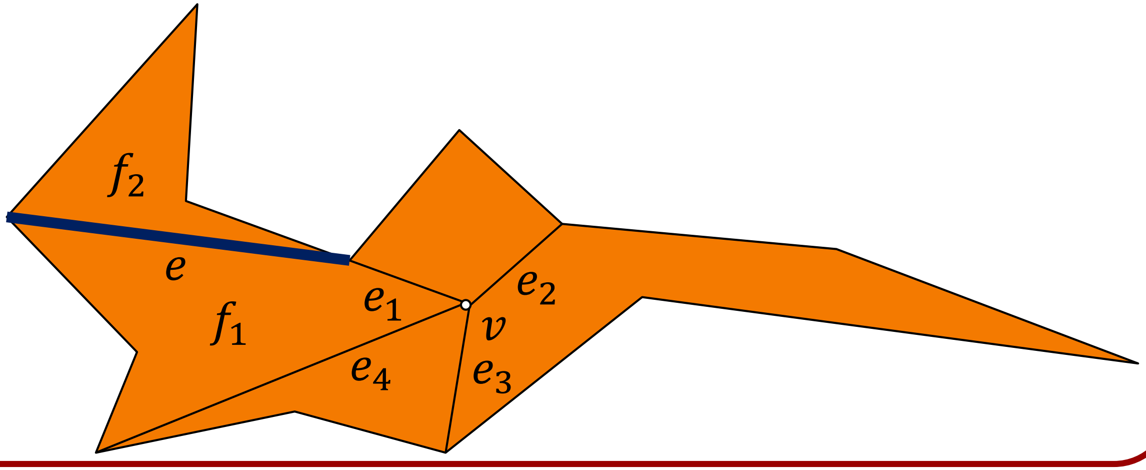
Proof (Manifold Edges):

The old edges still have only two faces on them (or one face twice).

Topological Polyhedra

Proof (Manifold Vertices):

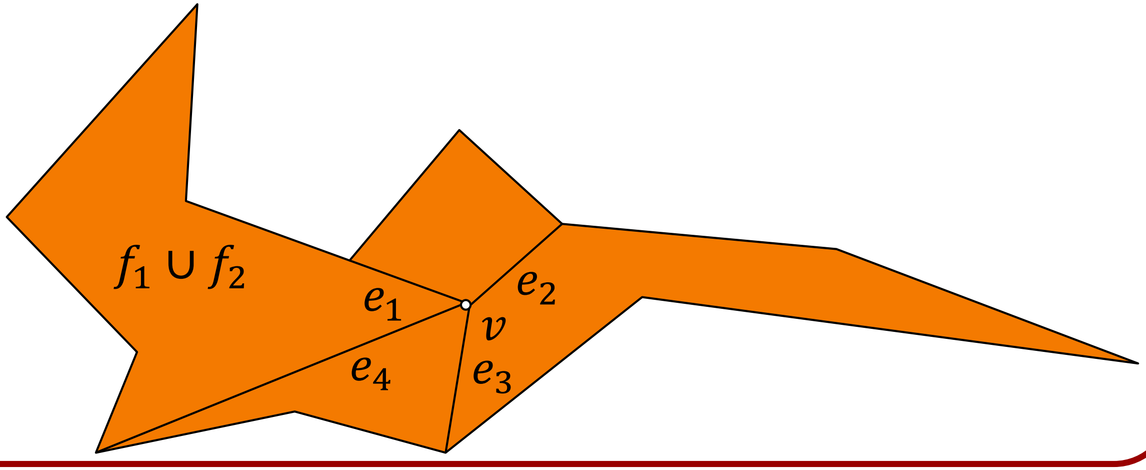
If $v \notin e$, we can use the old edge ordering.



Topological Polyhedra

Proof (Manifold Vertices):

If $v \notin e$, we can use the old edge ordering.



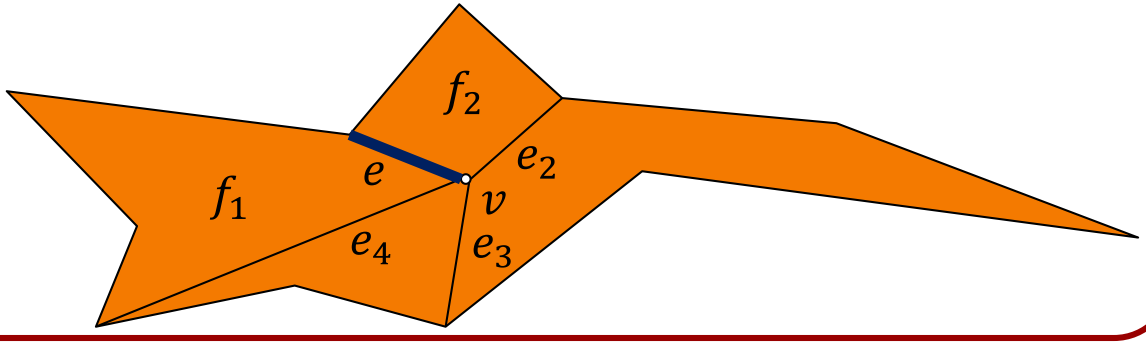
Topological Polyhedra

Proof (Manifold Vertices):

If $v \notin e$, we can use the old edge ordering.

If $v \in e$ let $\{e_1, e_2, \ldots, e_k\}$ be the old ordered edges around v, shifted so that $e_1 = e$.

Then e_k and e_2 are consecutive edges on $f_1 \cup f_2$ so $\{e_2, \ldots, e_k\}$ is a valid ordering.



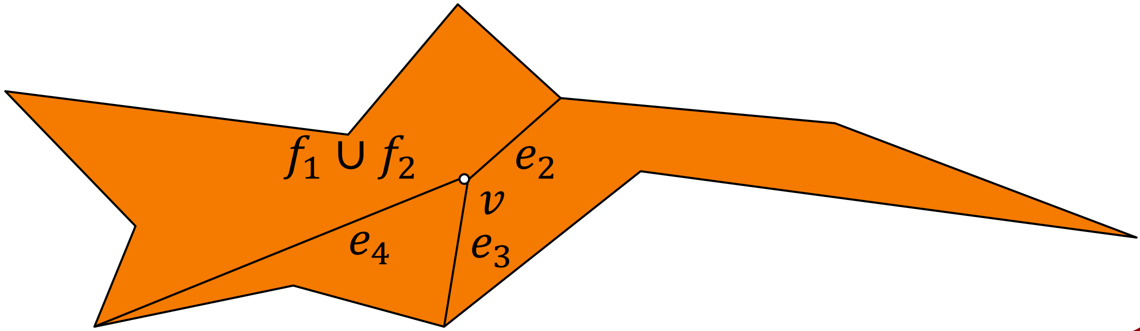
Topological Polyhedra

Proof (Manifold Vertices):

If $v \notin e$, we can use the old edge ordering.

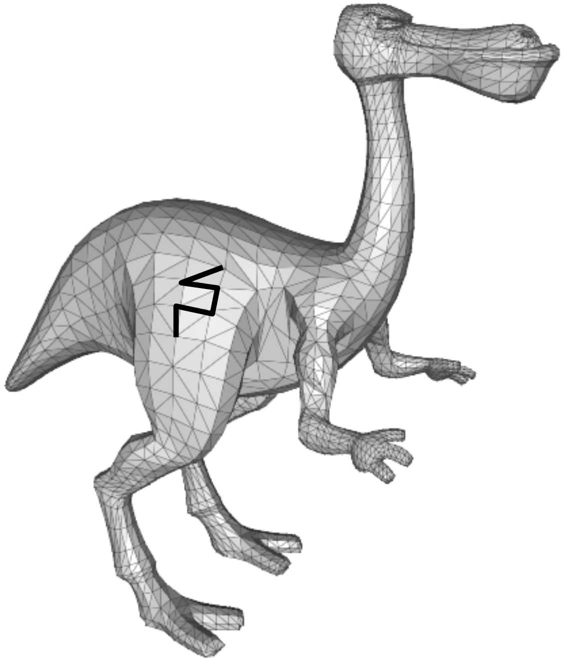
If $v \in e$ let $\{e_1, e_2, \ldots, e_k\}$ be the old ordered edges around v, shifted so that $e_1 = e$.

Then e_k and e_2 are consecutive edges on $f_1 \cup f_2$ so $\{e_2, \ldots, e_k\}$ is a valid ordering.



Curves

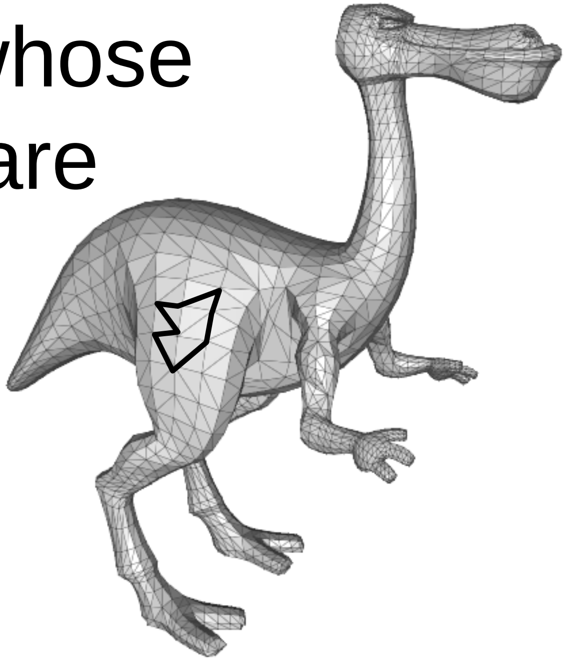
A (connected) *curve* on a topological polyhedron is a list of edges such that the ending vertex of one edge is the starting vertex of the next.



Curves

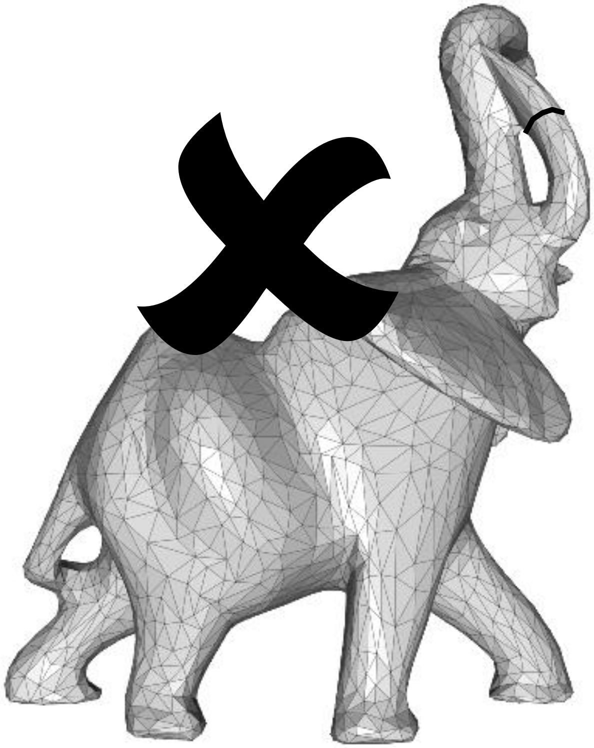
A (connected) *curve* on a topological polyhedron is a list of edges such that the ending vertex of one edge is the starting vertex of the next.

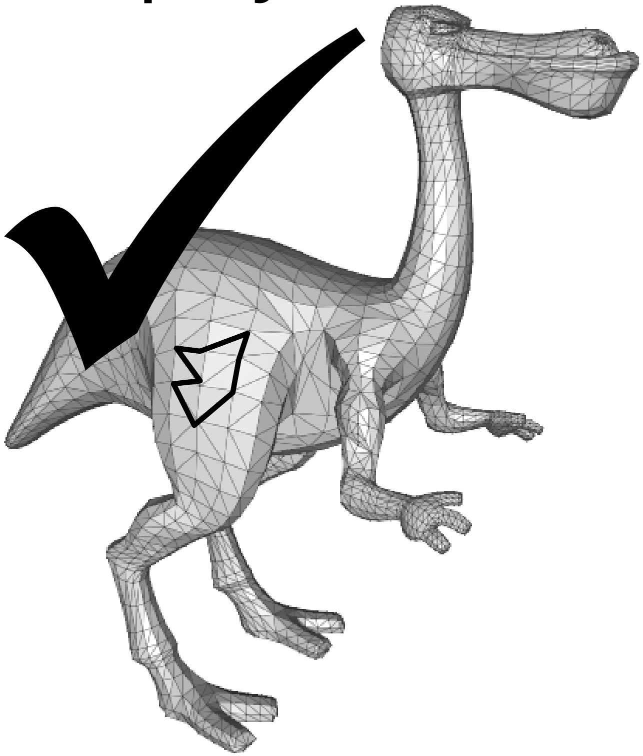
A *closed curve* is a curve whose starting and ending points are the same.



Genus-0 Polyhedra

A polyhedron is *genus-0* (or *simply connected*) if every non-trivial closed curve disconnects the faces of the polyhedron.

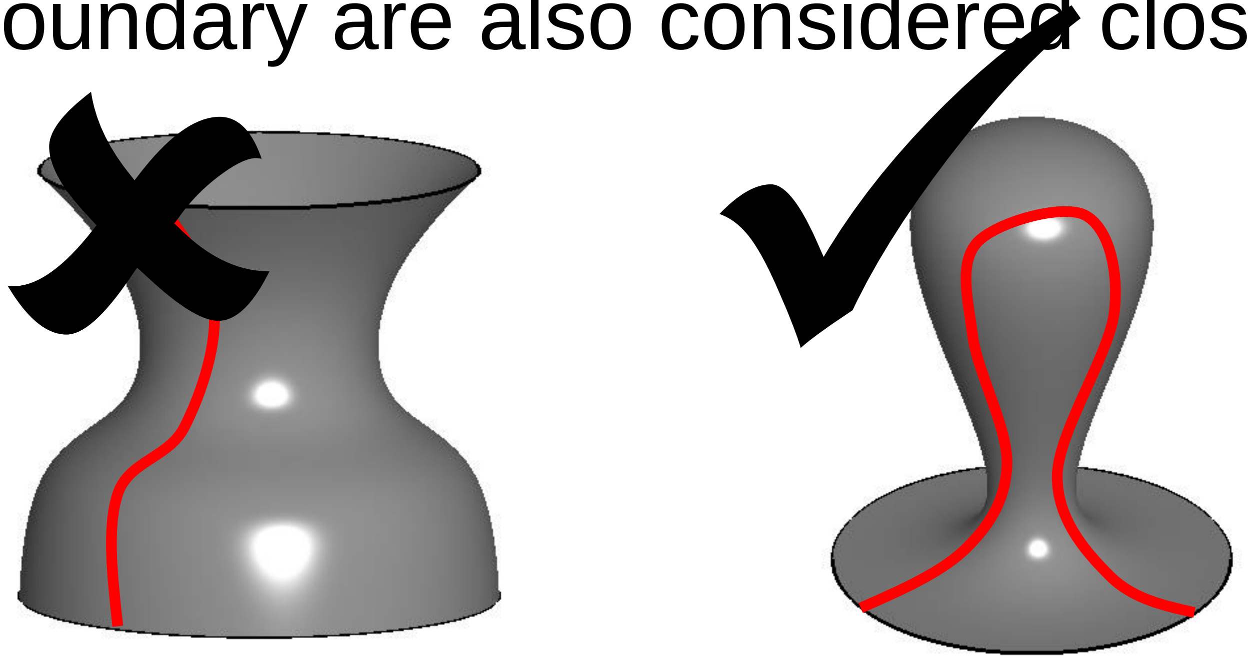




Genus-0 Polyhedra

Aside:

The definition can be extended to surfaces with boundary if curves that start and end at the boundary are also considered closed.



Genus-0 Polyhedra

Equivalently, given a topological polyhedron P we can define the dual graph $P^* = (V^*, E^*)$.

⇒ A curve $C \subset E$ corresponds to a set of dual edges $C^* \subset E^*$ of the dual.

⇒ P is genus-0 if removing C^* disconnects P^*.

Genus-0 Polyhedra

1. There is a continuous map from a polytope to a sphere.
   (e.g. Put the center of mass at the origin and normalize the positions.)

2. By the Jordan Curve Theorem the sphere is genus-zero.

<u>One Can Show</u>:

$\Rightarrow$ The polytope must also be genus-0.

Euler's Formula

For a genus-0 polyhedron P, the number of vertices, $|V|$, the number of edges, $|E|$, and the number of faces, $|F|$, satisfy:

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

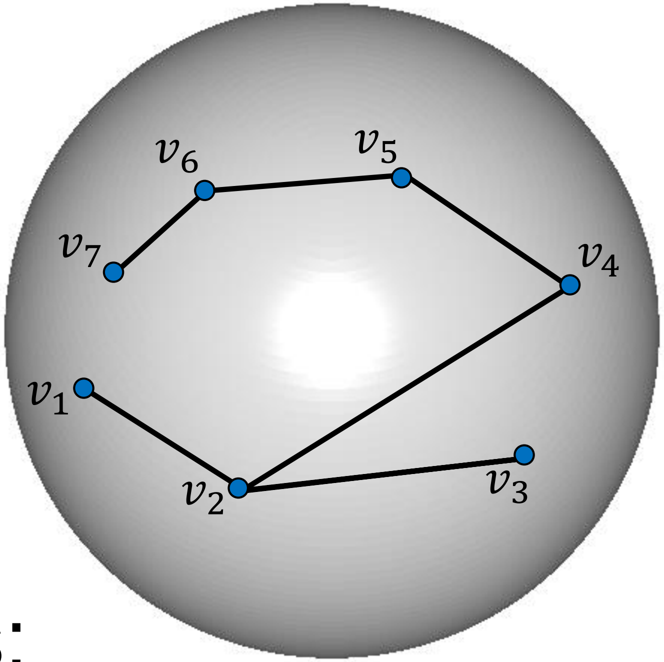
Euler's Formula (by Induction on $|F|$)

<u>Base case</u>: $|F| = 1$

We have:

- $V = \{v_1, \ldots, v_n\}$

*The edges on the boundary of the face form a connected tree (otherwise there is a closed loop and the interior of the face is disconnected).



Then there are $n - 1$ edges:

$$|V| - |E| + |F| = n - (n - 1) + 1 = 2$$

*This is just a sketch of the proof.

Euler's Formula (by Induction on $|F|$)

Induction: Assume true for $|F| = n - 1$

Find $e \in E$ shared by two distinct faces.

- If no such e exists, then all faces are adjacent to themselves, which contradicts the assumption that the polyhedron is connected.

Euler's Formula (by Induction on $|F|$)

Induction: Assume true for $|F| = n - 1$

Find $e \in E$ shared by two distinct faces.

- Remove e and merge the two adjoining faces, f_1 and f_2.

Claim:

The new polyhedron, P', is still genus-0.

Euler's Formula (by Induction on $|F|$)

Proof (P' is genus-zero):

Let C be a non-trivial curve on P'.

$\Rightarrow C$ is a non-trivial curve on P with $e \notin C$.

$\Rightarrow f_1$ and f_2 are in the same component.

$\Rightarrow C$ disconnects $f_1 \cup f_2$ from a face g on P.

$\Rightarrow C$ disconnects $f_1 \cup f_2$ from g in P'.

Euler's Formula (by Induction on $|F|$)

Induction: Assume true for $|F| = n - 1$

Find $e \in E$ shared by two distinct faces.

Remove e and merge the two adjoining faces.

P' is genus-0 with $|E| - 1$ edges, $|F| - 1$ faces, and $|V|$ vertices.

By the induction hypothesis we have:

$$|V| - (|E| - 1) + (|F| - 1) = 2$$

$$\Updownarrow$$

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

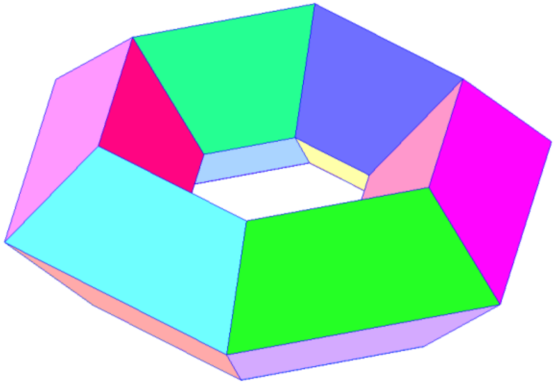
Euler's Formula

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

More Generally:

If a polygon mesh is genus-g (g is the number of handles) then:

$$|V| - |E| + |F| = 2 - 2g.$$



$|V| = 24, |E| = 48, |F| = 24$

[Wikipedia: Toroidal Polyhedron]

Euler's Formula

Implication:

The number of faces and edges is linear in the number of vertices.

Euler's Formula

Proof:

Assume all faces are triangles.
(Triangulating only increase $|F|$ and $|E|$.)

Since each edge is shared by two triangles:

$$|E| = 3|F|/2$$

Using Euler's Formula:

$$|V| - |E| + |F| = 2$$
$$\Updownarrow$$
$$|F| = 2|V| - 4 \quad \text{and} \quad |E| = 3|V| - 6$$

Outline

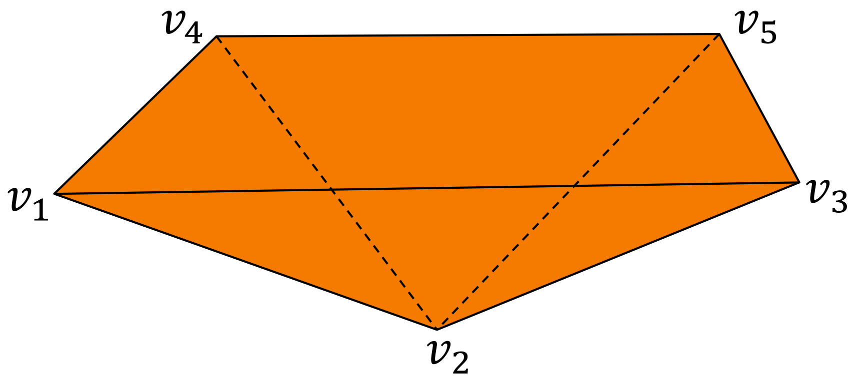
- Polyhedra
- (Oriented) Mesh Representation
  - Face-vertex data-structure
  - Winged-edge data-structure

(Oriented) Mesh Representation

Face-Vertex Lists:

Most often (e.g. ply, obj, etc. formats) polygon meshes are represented using vertex and face lists:

- **Vertex Entry**: (x, y, z) coordinates.
- **Face Entry**: Count and CCW indices of the vertices.



(Oriented) Mesh Representation

Face-Vertex Lists:

Most often (e.g. ply, obj, etc. formats) polygon meshes are represented using vertex and face lists:

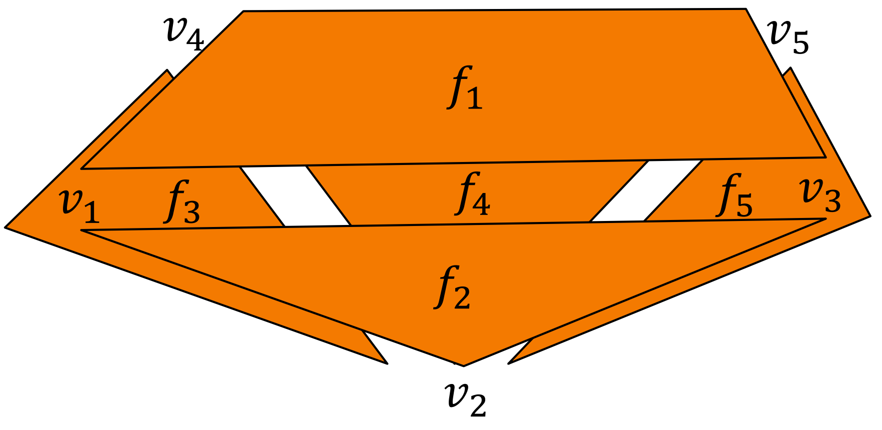
- **Vertex Entry**: (x, y, z) coordinates.
- **Face Entry**: Count and CCW indices of the vertices.

Vertex List

Id	x	y	z
1	-1	-1	0
2	0	0	-1
3	1	-1	0
4	-1	1	0
5	1	1	0

Face List

Id	#	Indices			
1	4	1	3	5	4
2	3	1	2	3	
3	3	4	2	1	
4	3	5	2	4	
5	3	3	2	5	



(Oriented) Mesh Representation

<u>Face-Vertex Lists</u>:

Most often (e.g. ply, obj, etc. formats) polygon meshes are represented using vertex and face lists:

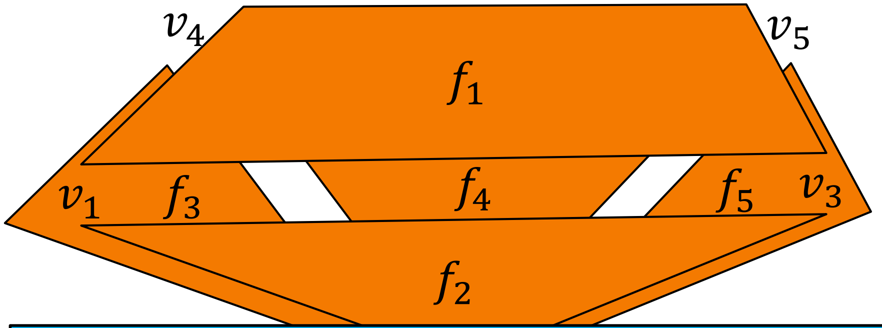
- **Vertex Entry**: (x, y, z) coordinates.
- **Face Entry**: Count and CCW indices of the vertices.

Vertex List

Id	x	y	z
1	-1	-1	0
2	0	0	-1
3	1	-1	0
4	-1	1	0
5	1	1	0

Face List

Id	#	Indices			
1	4	1	3	5	4
2	3	1	2	3	
3	3	4	2	1	
4	3	5	2	4	
5	3	3	2	5	



<u>Limitation</u>:

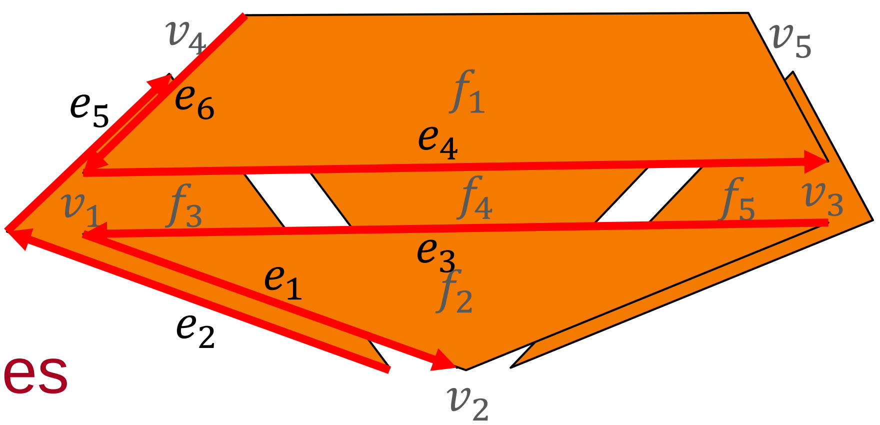
- Variable sized rows
- No explicit connectivity

(Oriented) Mesh Representation

Winged-Edge List:

Common representation for connectivity querying, represented using vertex, half-edge, and face lists:

- **Vertex Entry**:
  - » (x, y, z) coordinates
  - » Outgoing h.e. index
- **Face Entry**:
  - » h.e. index
- **Half-Edge Entry**:
  - » in/out wing h.e. indices
  - » opposite h.e. index
  - » end vertex index
  - » face index



Vertex List

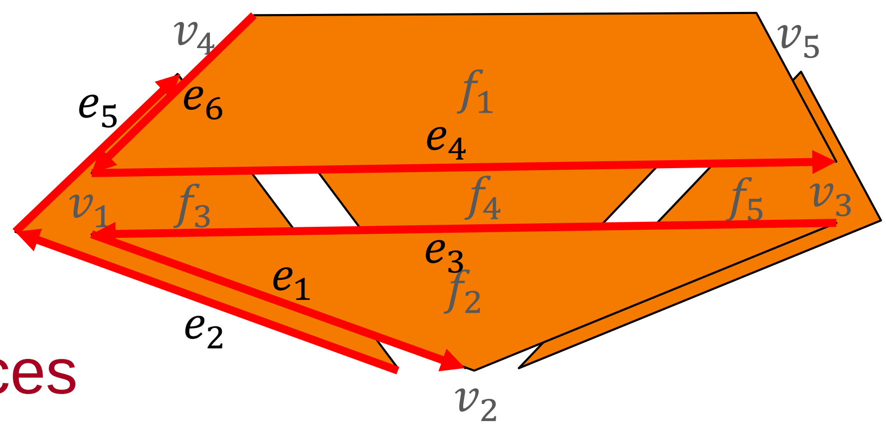
Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

) Mesh Representation

e List:

presentation for connectivity querying, using vertex, half-edge, and face lists:

- **Vertex Entry**:
  - (x, y, z) coordinates
  - Outgoing h.e. index
- **Face Entry**:
  - h.e. index
- **Half-Edge Entry**:
  - in/out wing h.e. indices
  - opposite h.e. index
  - end vertex index
  - face index



Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

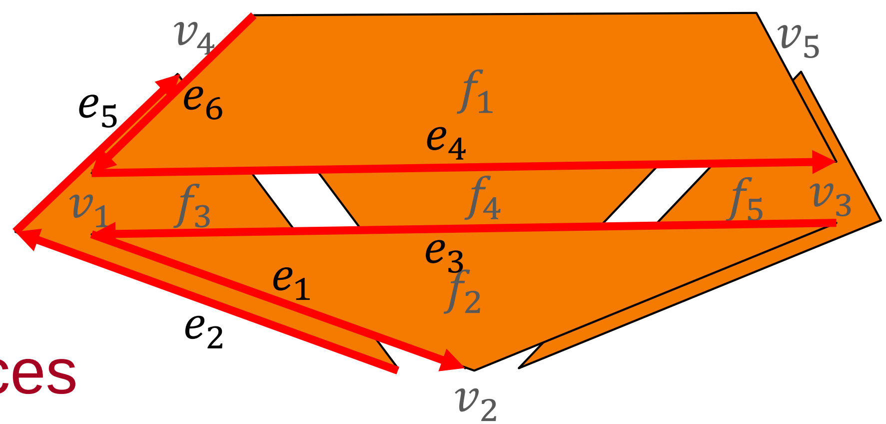
Face List

Id	h
1	4
2	3
3	5
4	…
5	…

…n Representation

…ation for connectivity querying,
…ertex, half-edge, and face lists:

- **Vertex Entry**:
  - (x, y, z) coordinates
  - Outgoing h.e. index
- **Face Entry**:
  - h.e. index
- **Half-Edge Entry**:
  - in/out wing h.e. indices
  - opposite h.e. index
  - end vertex index
  - face index



tion

ity querying,

nd face lists:

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

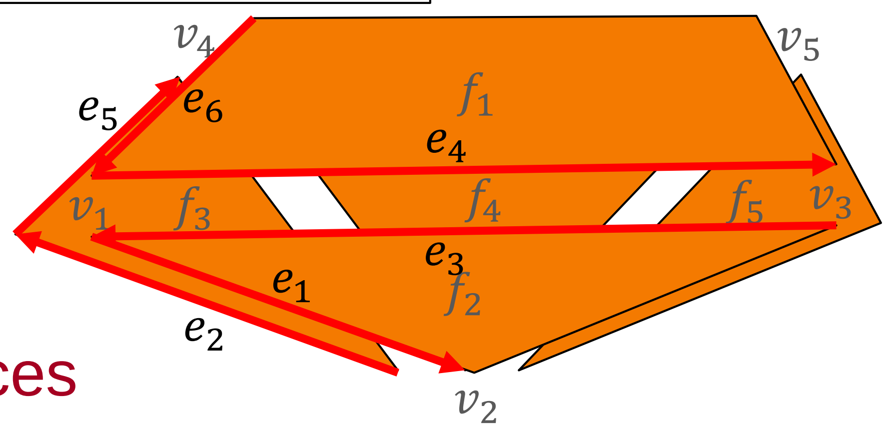
Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	...	4	1	1
…					

  - » Outgoing h.e. index
- ◦ **Face Entry**:
  - » h.e. index
- ◦ **Half-Edge Entry**:
  - » in/out wing h.e. indices
  - » opposite h.e. index
  - » end vertex index
  - » face index



Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

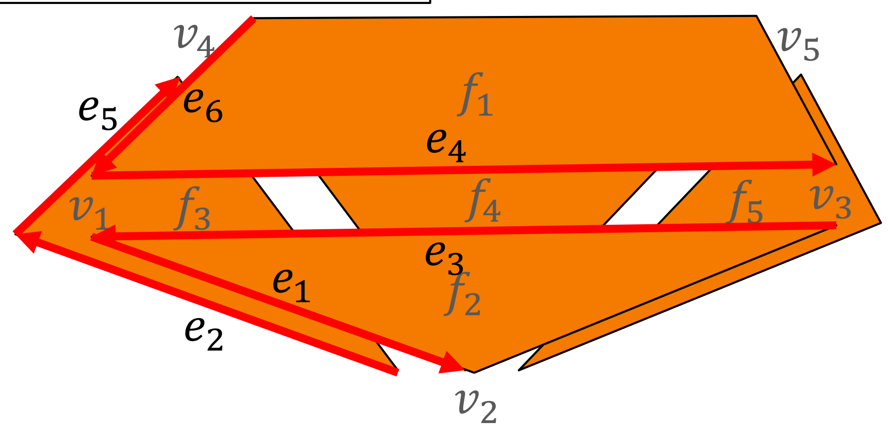
Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

…tion

…ity querying,

…nd face lists:

- » Outgoing h.e. index
- ◦ **Face Entry**:
  - » h.e. index
- ◦ **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1:

tion

ity querying,
nd face lists:

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

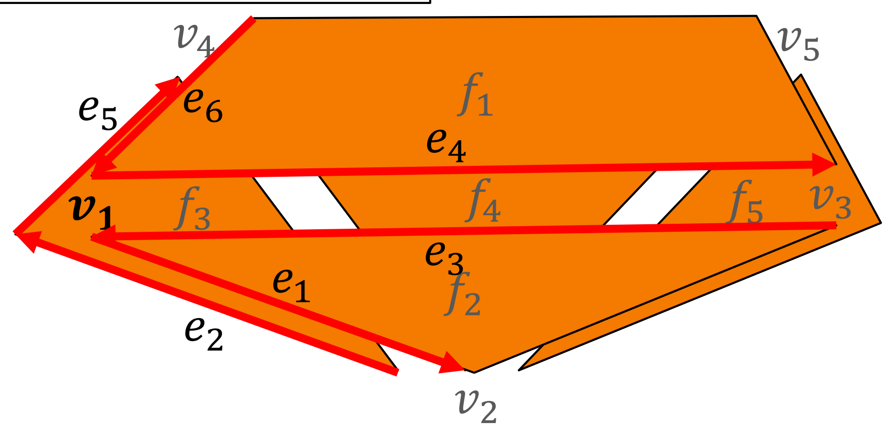
Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

- » Outgoing h.e. index
- ○ **Face Entry**:
  - » h.e. index
- ○ **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1:

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	**3**	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

tion

ity querying,

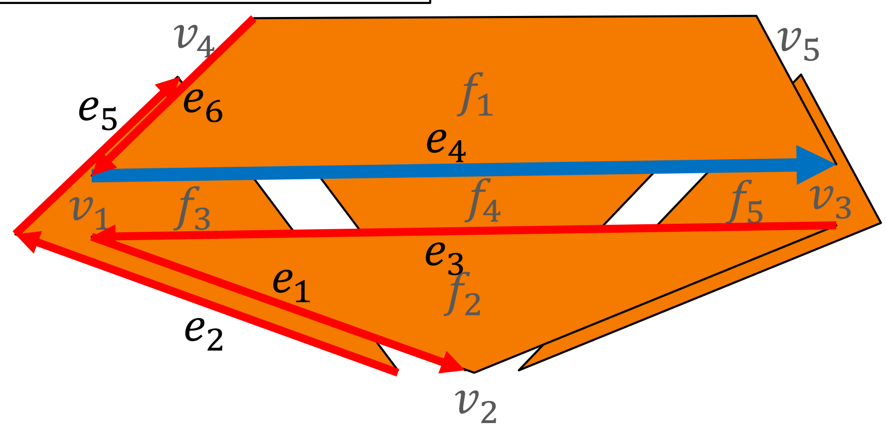
nd face lists:

» Outgoing h.e. index

◦ **Face Entry**:

» h.e. index

◦ **Half-Edge Entry**:



Example:

Find CCW vertices around v_1: v_3

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	**6**	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

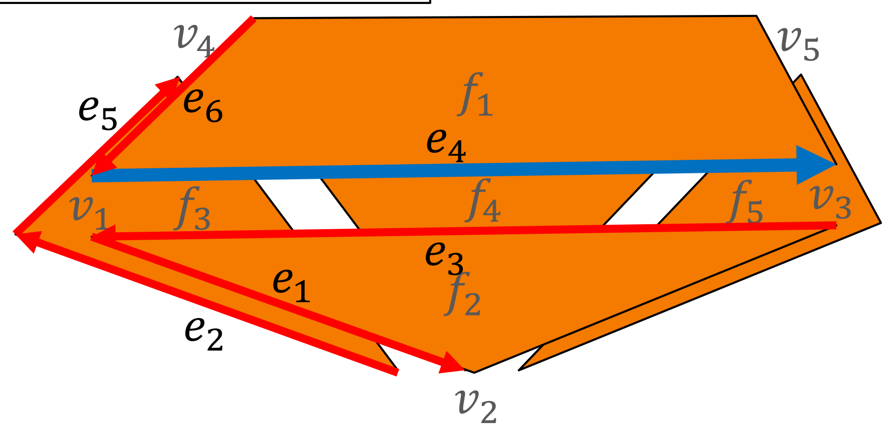
…tion

…ity querying,

…nd face lists:

   » Outgoing h.e. index

- **Face Entry**:
   » h.e. index
- **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1: v_3

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

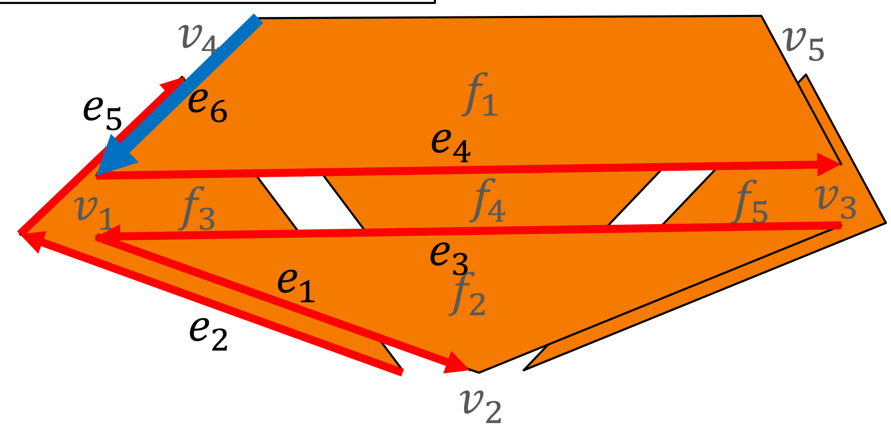
Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

…tion

…ity querying,

…nd face lists:

- » Outgoing h.e. index
- ○ **Face Entry**:
  - » h.e. index
- ○ **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1: v_3

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

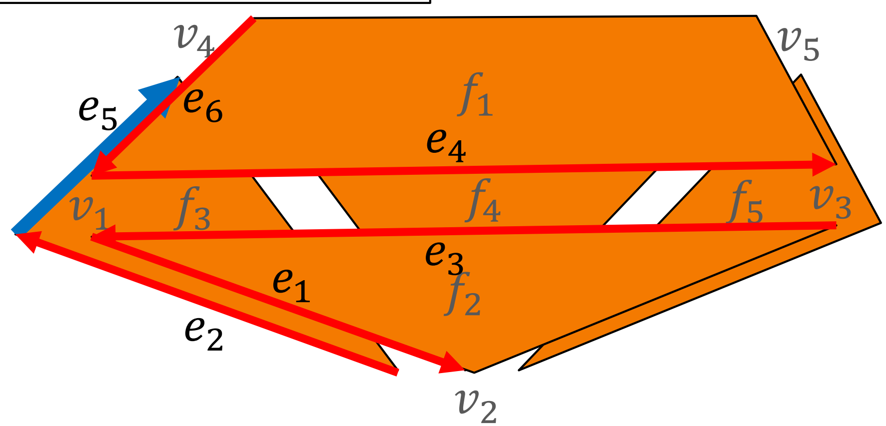
Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

tion

ity querying,

nd face lists:

- » Outgoing h.e. index
- ○ **Face Entry**:
  - » h.e. index
- ○ **Half-Edge Entry**:



Example:

Find CCW vertices around v_1: v_3, v_4

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

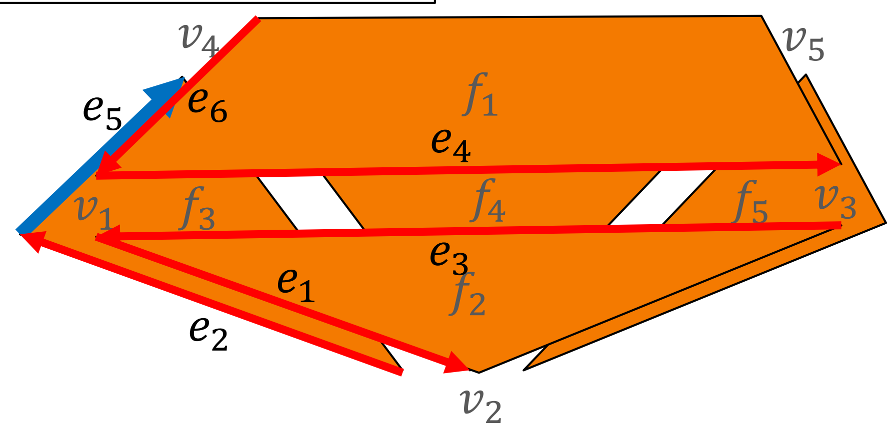
tion

ity querying,

nd face lists:

» Outgoing h.e. index

- **Face Entry**:
  » h.e. index
- **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1: v_3, v_4

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	...	4	1	1
…					

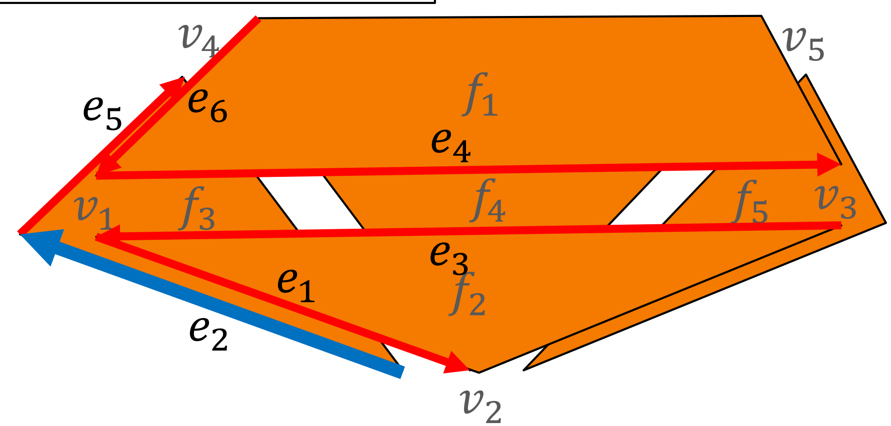
tion

ity querying,

nd face lists:

» Outgoing h.e. index

- **Face Entry**:
  » h.e. index
- **Half-Edge Entry**:



Example:

Find CCW vertices around v_1: v_3, v_4

tion

ity querying,

nd face lists:

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

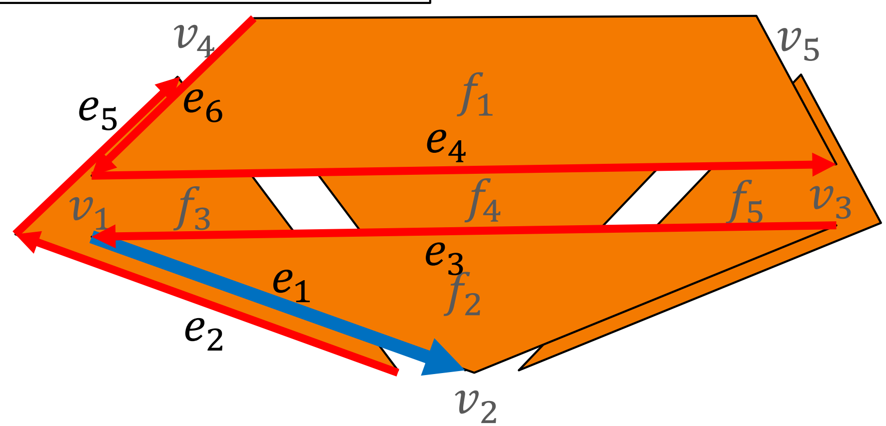
Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	...	4	1	1
…					

  - » Outgoing h.e. index
- ○ **Face Entry**:
  - » h.e. index
- ○ **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1: v_3, v_4, v_2

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	**3**	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

tion

ity querying,

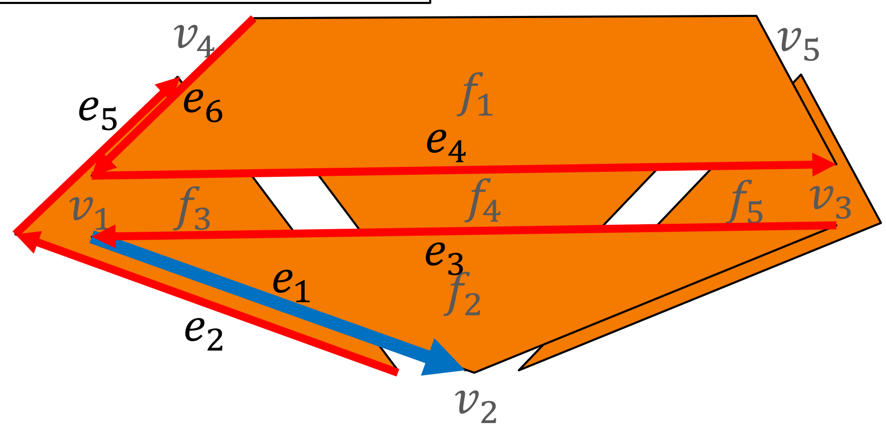
nd face lists:

» Outgoing h.e. index

- **Face Entry**:

  » h.e. index

- **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1: v_3, v_4, v_2

tion

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

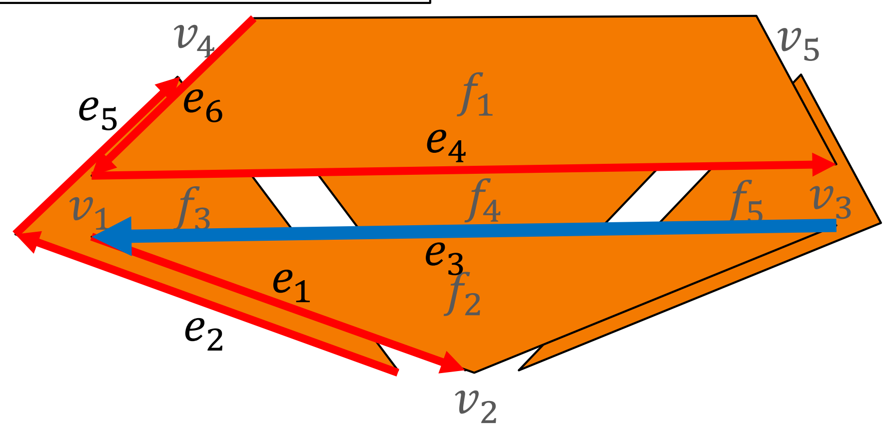
Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	**4**	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	…	4	1	1
…					

ity querying,

nd face lists:

   - » Outgoing h.e. index
- **Face Entry**:
   - » h.e. index
- **Half-Edge Entry**:



Example:

Find CCW vertices around v_1: v_3, v_4, v_2

Vertex List

Id	x	y	z	h
1	-1	-1	0	4
2	0	0	-1	2
3	1	-1	0	3
4	-1	1	0	6
5	1	1	0	…

Face List

Id	h
1	4
2	3
3	5
4	…
5	…

Half-Edge List

Id	o	w_i	w_o	v	f
1	2	3	…	2	2
2	1	…	5	1	3
3	4	…	1	1	2
4	3	6	…	3	1
5	6	2	…	4	3
6	5	...	4	1	1

tion

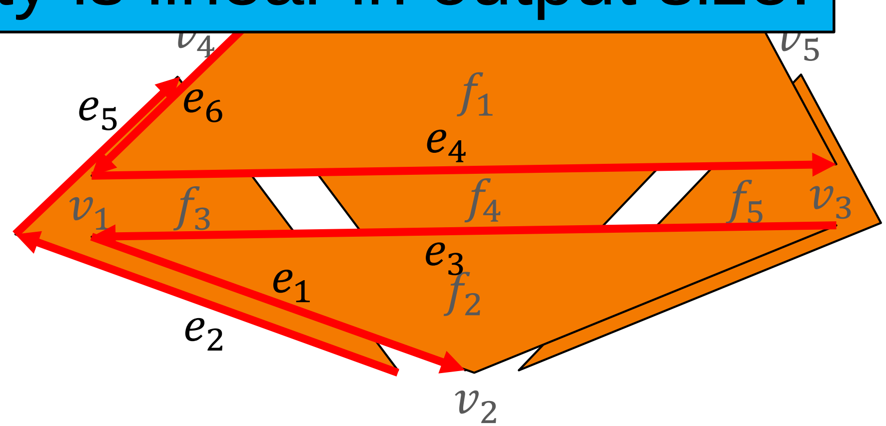
ity querying,

ınd face lists:

Computational complexity is linear in output size.

» Outgoing h.e. index

- **Face Entry**:
  » h.e. index
- **Half-Edge Entry**:



<u>Example</u>:

Find CCW vertices around v_1: v_3, v_4, v_2

(Oriented) Mesh Representation

Goal:

Given a face-vertex representation of a mesh (V,F), convert it to a winged-edge representation (_V,_E,_F).

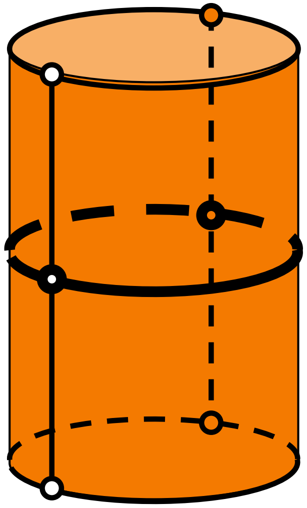
(Oriented) Mesh Representation

Goal:

Given a face-vertex representation of a mesh (V,F), convert it to a winged-edge representation (_V,_E,_F).

Warning:

The following discussion assumes that in a mesh, a (directed) edge is uniquely determined by its starting and ending vertices.



This does not have to be true.

(Oriented) Mesh Representation

```
GenerateHalfEdge( V , F , _V , _E , _F )
    _V.resize( v.size() ) , _F.resize( F.size() )
    for( i=0 ; i<V.size() ; i++ ) _V[i].p = V[i]

    unordered_map< VertexPair , int > eMap
    ConstructEdgeToFaceMap( F, eMap )

    _E.resize( eMap.size() )

    SetHalfEdgeIndices( eMap , _V , _E , _F )
```

Assuming that:

- The VertexPair object defines a hashing function

(Oriented) Mesh Representation

```
ConstructEdgeToFaceMap( F , eMap )
    for( f=0 ; f<F.size() ; f++ )
        for( v=0 ; v<F[f].size() ; v++ )
            VertexPair key( F[f][v] , F[f][v+1] )
            eMap[key] = f
```

Assuming that:

- Indexing is modulo the face size

(Oriented) Mesh Representation

```
SetHalfEdgeIndices( eMap , _V , _E , _F )
    int e = 0
    for( iter i=eMap.begin() ; i!=eMap.end() ; i++ , e++ )
        int v1 = i.key.first , v2 = i.key.second , f = i.value
        _E[e].v = v2 , _E[e].f = f
        _V[v1].he = _F[f].he = i.value = e
    for( f=0 ; f<F.size() ; f++ ) for( v=0 ; v<F[f].size() ; v++ )
            VertexPair  key( F[f][v  ] , F[f][v+1] )
            VertexPair oKey( F[f][v+1] , F[f][v  ] )
            VertexPair nKey( F[f][v+1] , F[f][v+2] )
            _E[ eMap[ key ] ].opposite = eMap[ oKey ]
            _E[ eMap[ key ] ].next = eMap[ nKey ]
            _E[ eMap[ nKey ] ].previous = eMap[ key ]
```