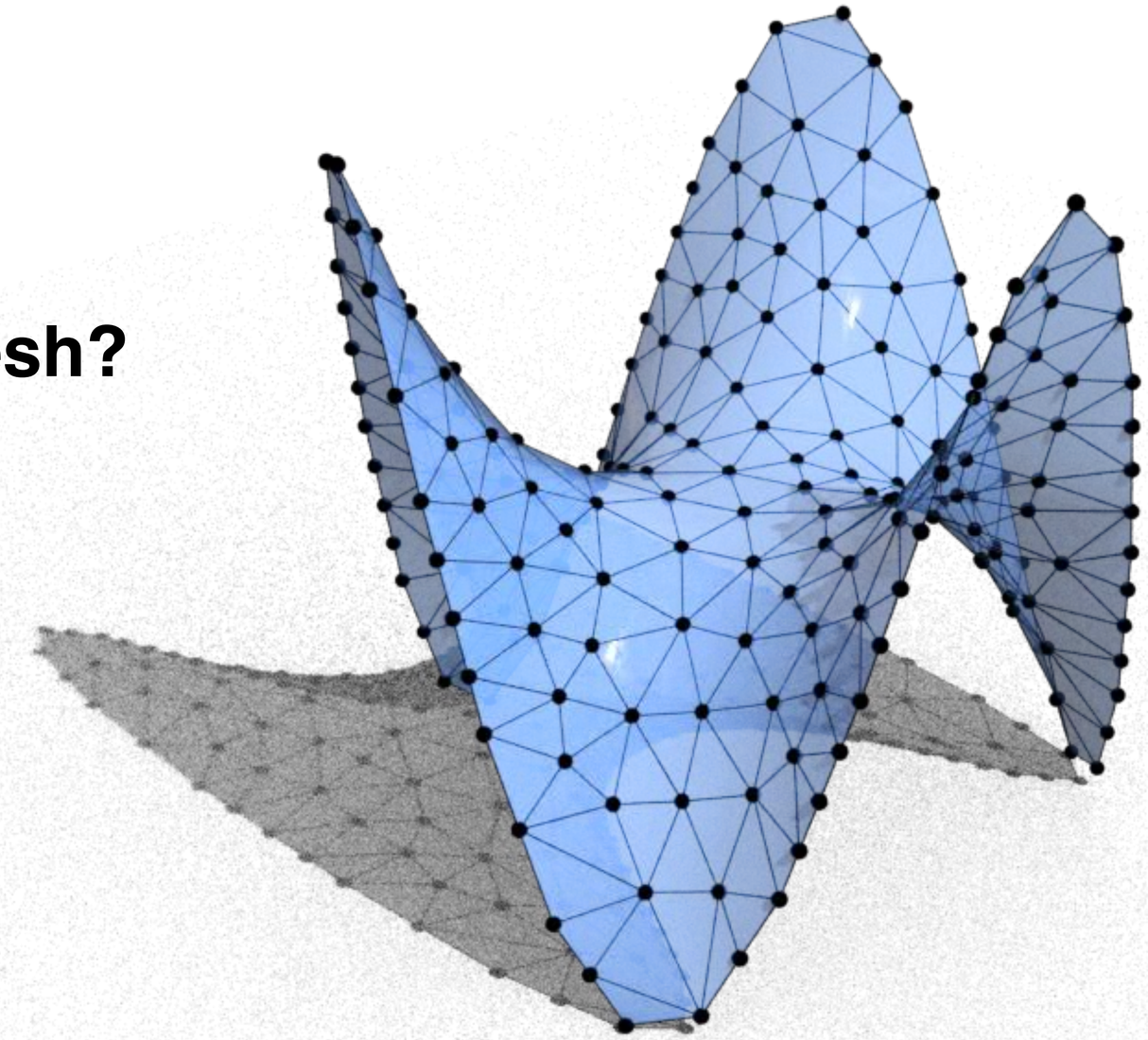
# CSE 270 (WI24)
# Discrete Differential Geometry
# Mesh Data Structures

Albert Chern

UC San Diego
JACOBS SCHOOL OF ENGINEERING

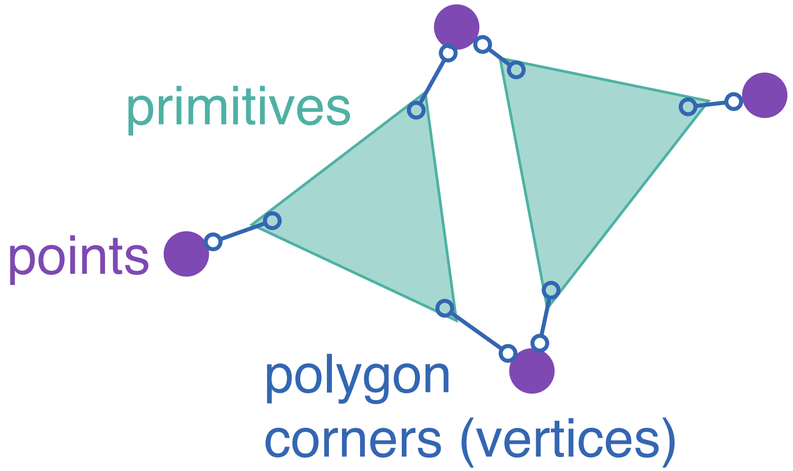
# Description of a triangle surface

**How should we organize information on a triangle mesh?**

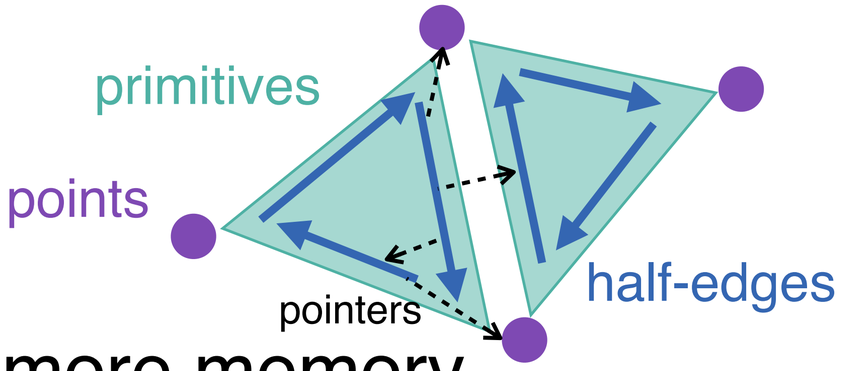


# Accessing points and triangles

- Create, operate, navigate in a triangle mesh
- Two commonly used data structures:
- **Point-primitive data structure**
  - List of points, triangles, and connectivities (polygon corners)
  - Minimalistic and intuitive, hard to query neighbors




- **Half-edge data structure**
  - List of points, half-edges and triangles
  - Useful for querying neighbors, takes a bit more memory



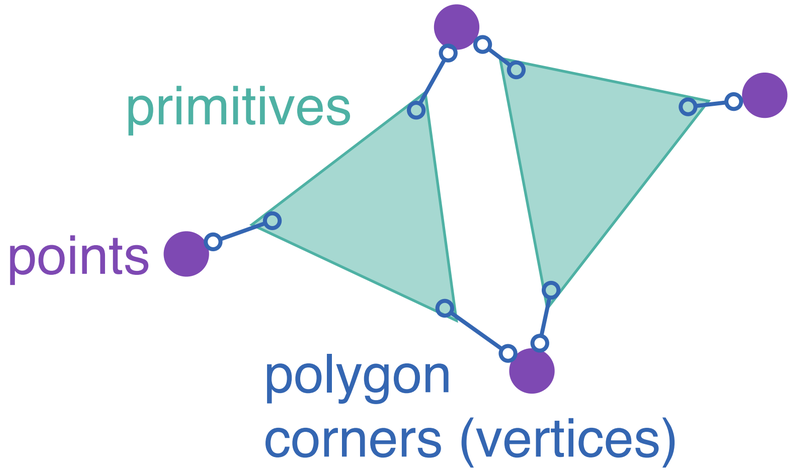
# Point-Primitive data structure and attributes

- Point-primitive data structure, attributes
- Half-edge data structure
- Abstraction of half-edges

# Point–Primitive data structure

- The mesh is described by
  - a set $\mathcal{P}$ of points
  - a set $\mathcal{F}$ of faces (primitives)
  - a set $\mathcal{V}$ of $\mathcal{F} \times \mathcal{P}$ - tuples
- Each element $v = (f, p)$ is a polygon corner (known as a **vertex** in Houdini);

  it indicates that face $f$ is hooked to point $p$
- These vertices are partially ordered.



# Point-primitive representation

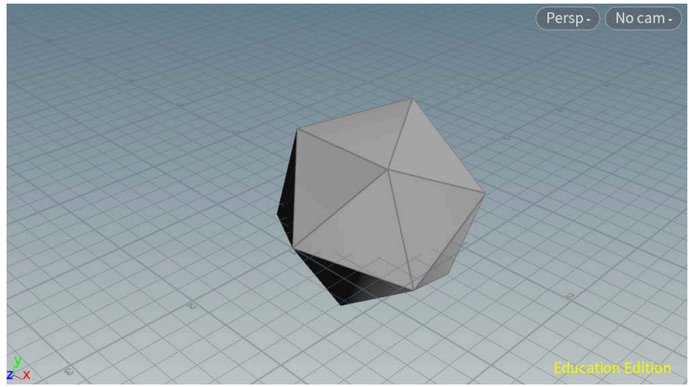
- Points

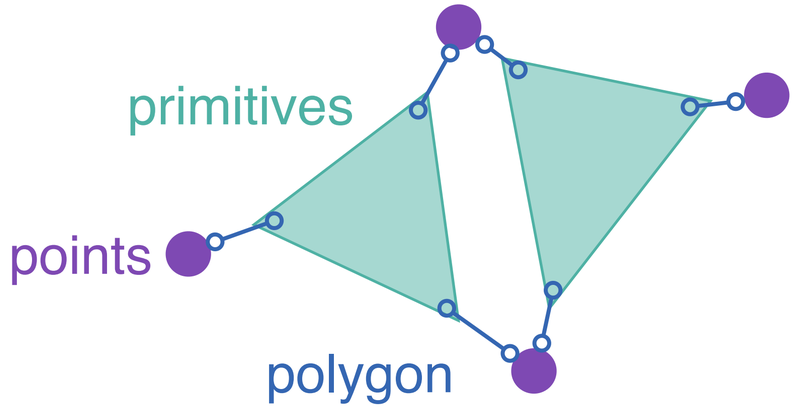
Scene View × Geometry Spreadsheet × +

obj geo1

**Node:** plato… Group: View Intrins

| | P[x] | P[y] | P[z] |
|---|---|---|---|
| 0 | 0.0 | 0.0 | -1.0 |
| 1 | 0.0 | 0.894427 | -0.447213 |
| 2 | 0.850651 | 0.276394 | -0.447213 |
| 3 | 0.525731 | -0.723607 | -0.447213 |
| 4 | -0.525731 | -0.723607 | -0.447213 |
| 5 | -0.850651 | 0.276394 | -0.447213 |
| 6 | -0.525731 | 0.723607 | 0.447213 |
| 7 | -0.850651 | -0.276394 | 0.447213 |
| 8 | 0.0 | -0.894427 | 0.447213 |
| 9 | 0.850651 | -0.276394 | 0.447213 |
| 10 | 0.525731 | 0.723607 | 0.447213 |
| 11 | 0.0 | 0.0 | 1.0 |




- Vertices (polygon corners)

prim

0th,1st or 2nd vertex of the triangle

there is also an internal linear indexing of vertices that are more useful

**Node:** plato… G

| | Point Num |
|---|---|
| 0:0 | 2 |
| 0:1 | 1 |
| 0:2 | 0 |
| 1:0 | 3 |
| 1:1 | 2 |
| 1:2 | 0 |
| 2:0 | 4 |

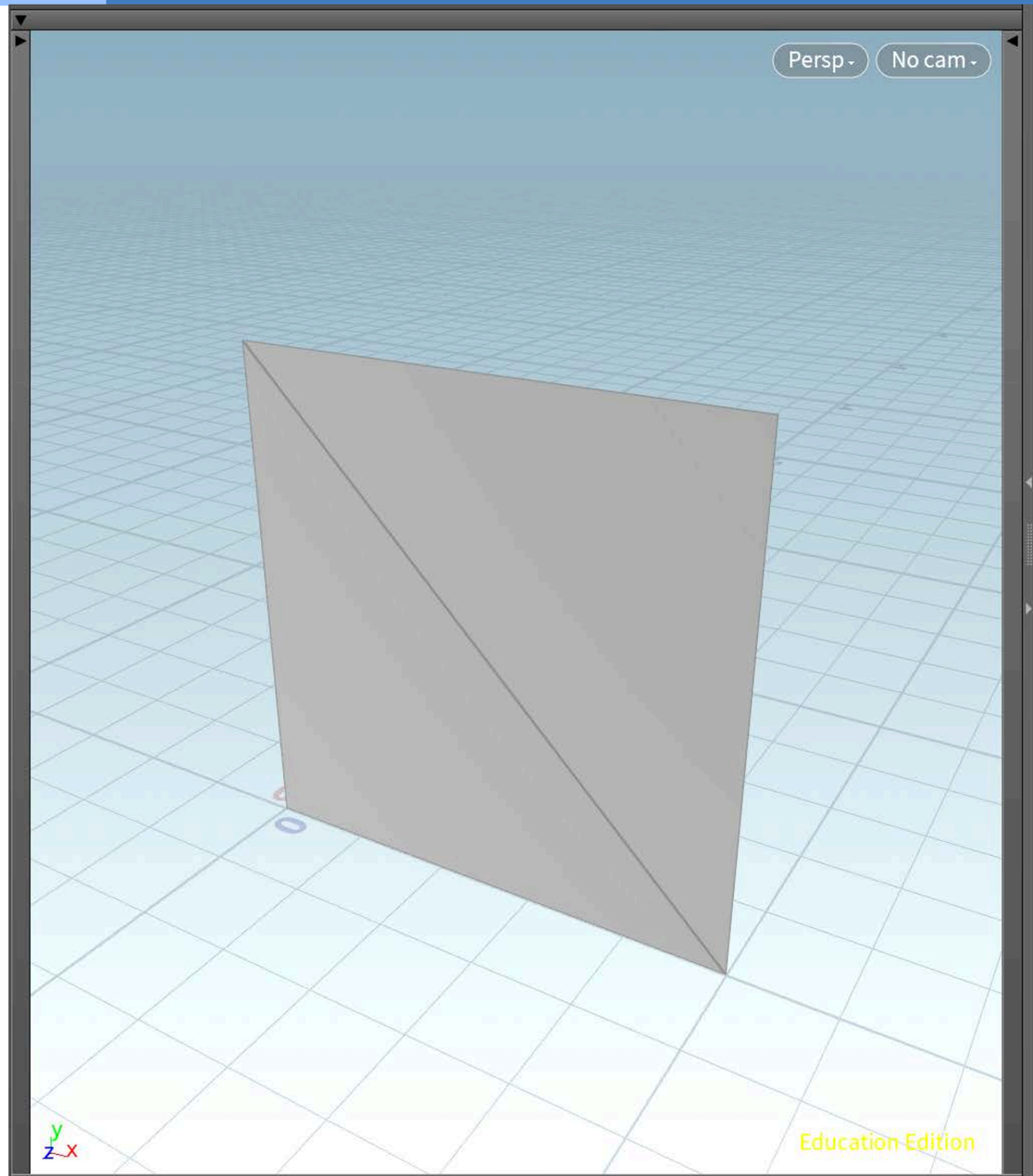
0 1 2 3 4 5 6

- Primitives

**Node:** plato… C

| |
|---|
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

# Create triangle mesh from scratch



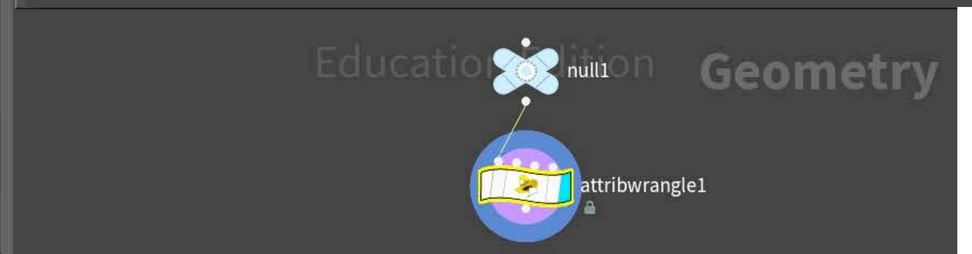

Group

Group Type: Guess from Group

Run Over: Detail (only once)

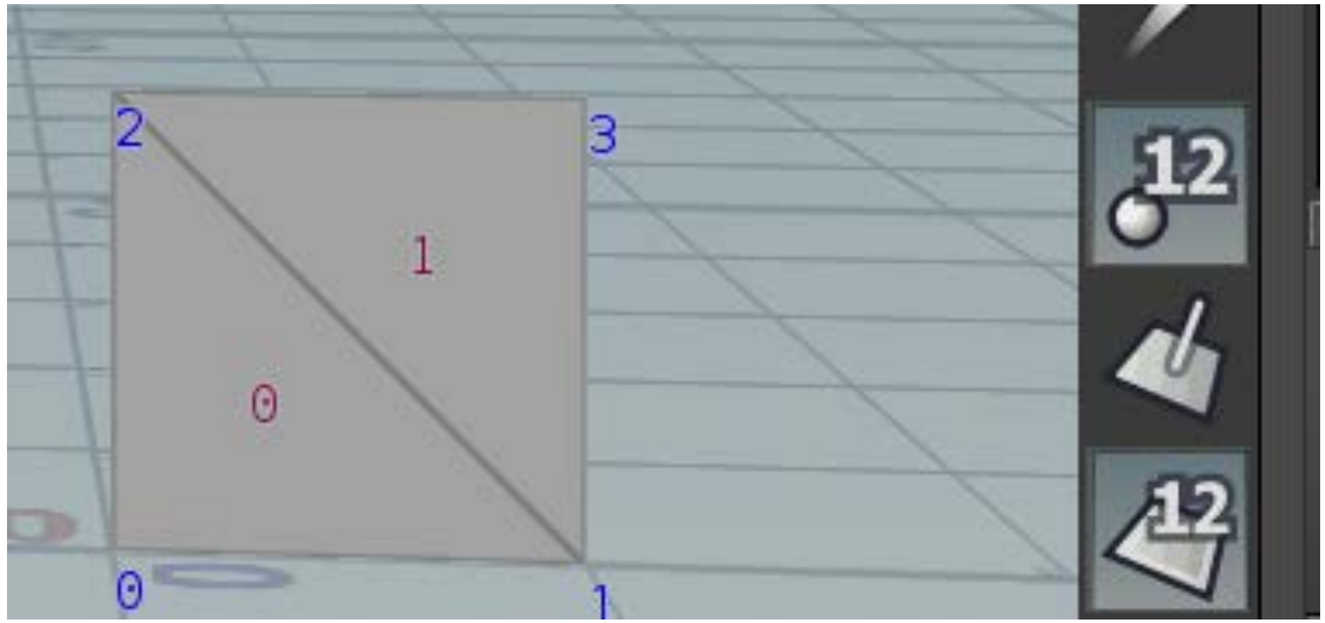
VEXpression

```
int pt0 = addpoint(geoself(),set(0,0,0));
int pt1 = addpoint(geoself(),set(1,0,0));
int pt2 = addpoint(geoself(),set(0,1,0));
int pt3 = addpoint(geoself(),set(1,1,0));
int prim0 = addprim(geoself(),"poly");
int prim1 = addprim(geoself(),"poly");
addvertex(geoself(),prim0,pt0);
addvertex(geoself(),prim0,pt2);
addvertex(geoself(),prim0,pt1);
addvertex(geoself(),prim1,pt2);
addvertex(geoself(),prim1,pt3);
addvertex(geoself(),prim1,pt1);
```



# Create triangle mesh from scratch

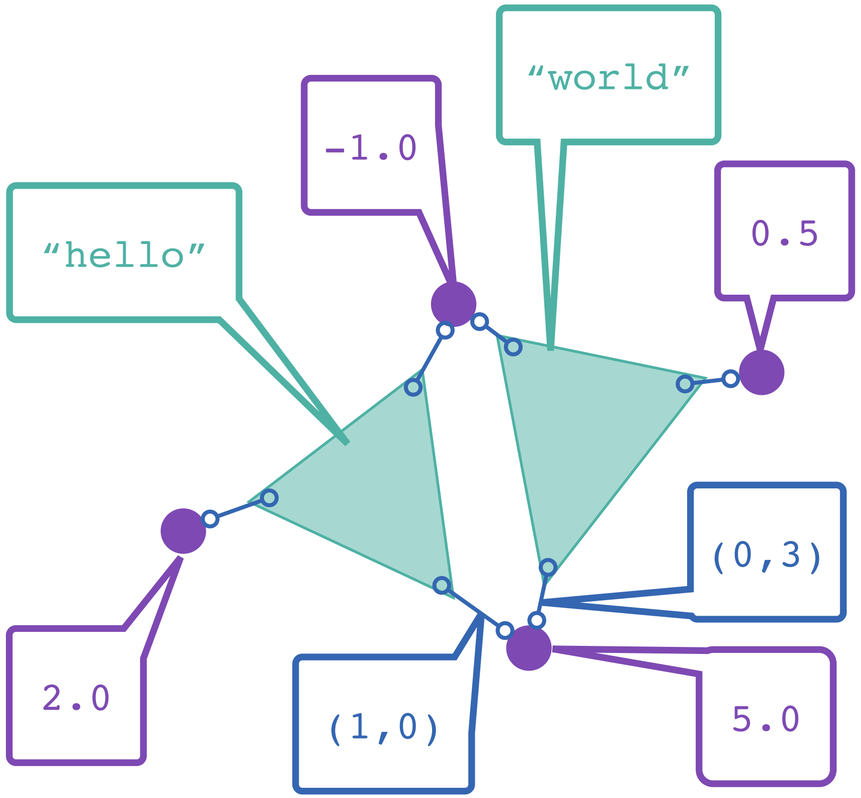
## Visualize primnum and ptnum



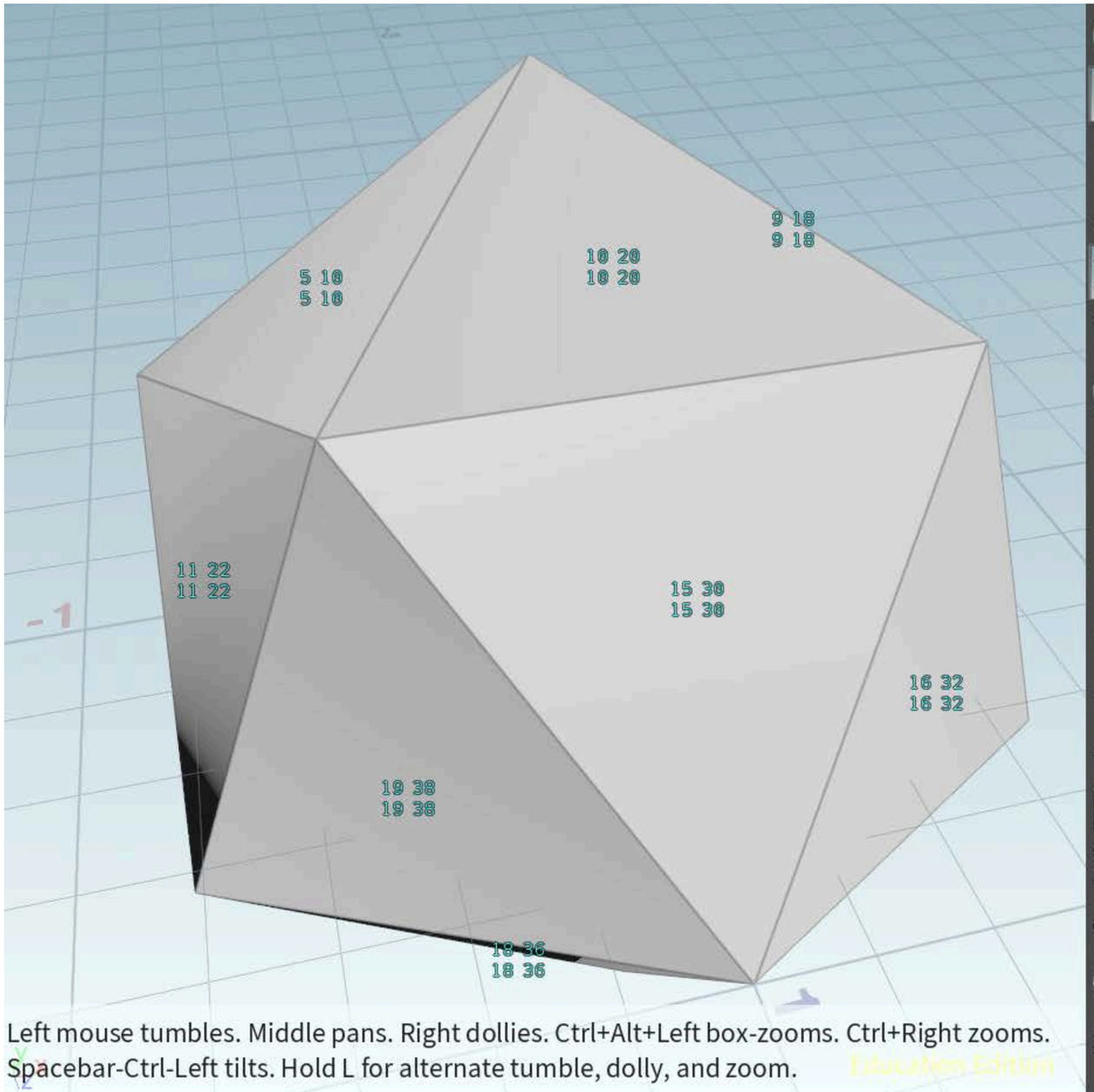
# Attributes

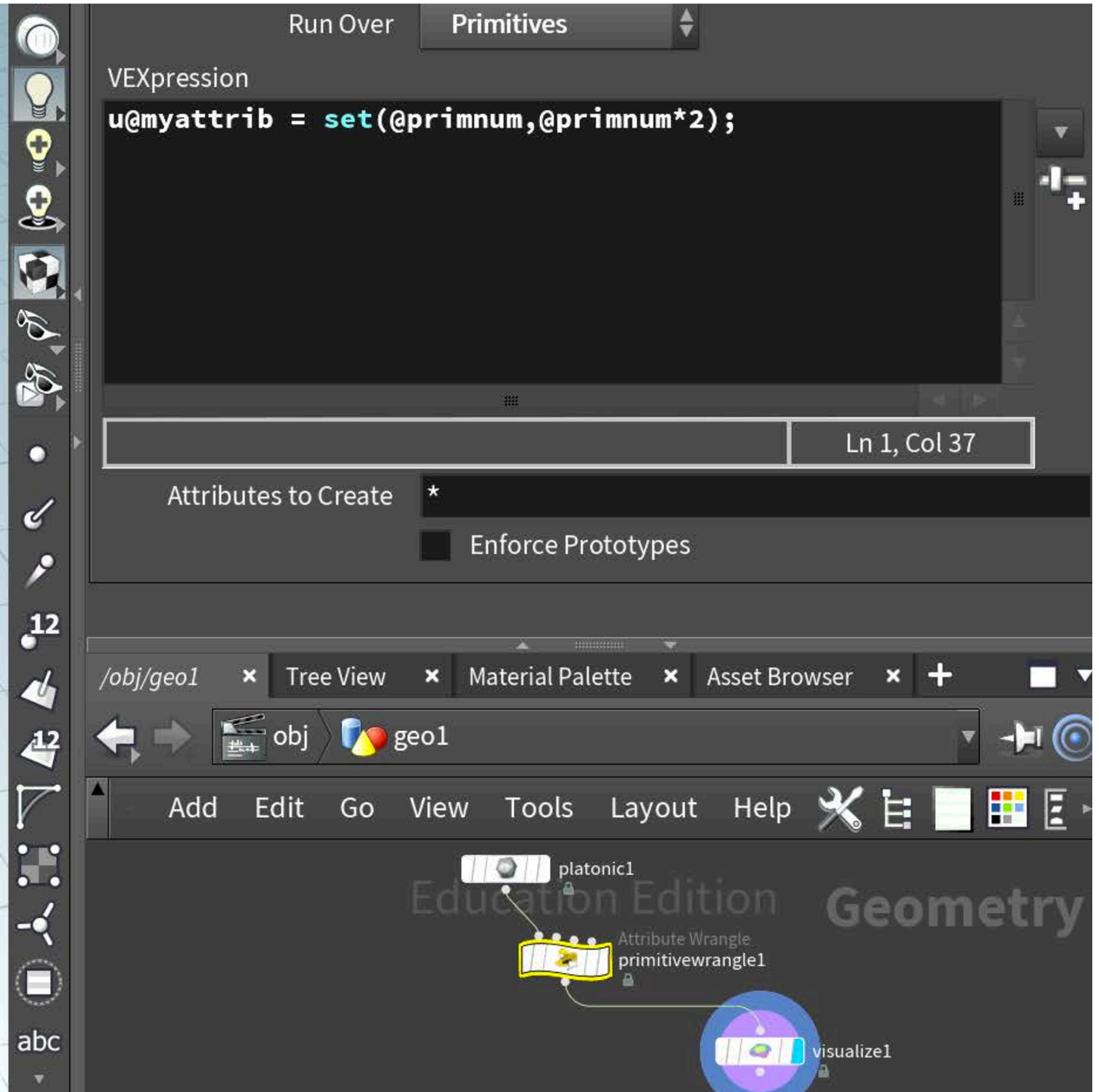
- We can store data on the mesh.  They can be data over points, primitives or vertices.
- A point data is called a point attribute. (similar for primitives and vertices.)
- One can view a point attribute of type T a function:

$$u : \mathcal{P} \rightarrow T$$



# Attributes


9 18
9 18
10 20
10 20
5 10
5 10
11 22
11 22
15 30
15 30
16 32
16 32
19 38
19 38
18 36
18 36
Left mouse tumbles. Middle pans. Right dollies. Ctrl+Alt+Left box-zooms. Ctrl+Right zooms.
Spacebar-Ctrl-Left tilts. Hold L for alternate tumble, dolly, and zoom.


Run Over
Primitives
VEXpression
u@myattrib = set(@primnum,@primnum*2);
Ln 1, Col 37
Attributes to Create
*
Enforce Prototypes
/obj/geo1
Tree View
Material Palette
Asset Browser
obj
geo1
Add
Edit
Go
View
Tools
Layout
Help
platonic1
Attribute Wrangle
primitivewrangle1
visualize1
Education Edition
Geometry

# Attributes

- You can define point-attributes, vertex-attributes, prim-attributes, and detail attributes.
- Use prefix “@” such as @myattrib = …
- It’s better to use type-specific prefix.  Frequently used:

| | Type name | Prefix for @ | component access |
|---|---|---|---|
| Integer | `int` | `i@` | |
| Float | `float` | `f@` | |
| 2D vector | `vector2` | `u@` | `u@myattrib.xu@myattrib.y` |
| 3D vector | `vector` | `v@` | `v@myattrib.x … v@myattrib.z` |
| 4D vector | `vector4` | `p@` | `p@myattrib.x … p@myattrib.w` |
| Array of floats | `float var[]` | `f[]@` | `f[]@myattrib[3]` |
| Array of vectors | `vector var[]` | `v[]@` | `v[]@myattrib[3].y` |

# Attributes

- You can name attributes whatever you want, except
  - You can’t name a point attribute the same name as vertex attribute
  - The following reserved attributes have special meaning
    - v@P: point position
    - v@N: normal vectors
    - v@Cd: color (RGB color data)
    - v@uv: texture coordinate (3D not 2D due to homogeneous coordinates)
    - f@Alpha: opacity
    - i@ptnum: point index in pointwrangle
    - i@vtxnum: vertex (linear) index in vertexwrangle
    - i@primnum: primitive index in primitive wrangle

# Attributes

- You can read attributes from the *previous node*
  - Use the vex functions “point” “vertex” “prim” and “detail”

```
int primindex = 2;
float x = prim(0,"myattrib",primindex);
```

the type to read is inferred from
who catches the return value



# Attributes

- You can also set attributes to the *current node*
  - Use the vex functions

    `setpointattribvalues`

    etc. (See documentation for more info)

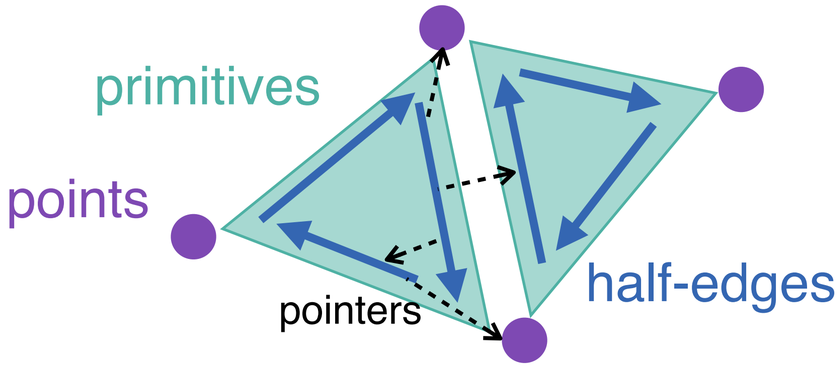
- Since vex code runs in parallel, we can't directly read what we just write

# Half edge data structure

- Point-primitive data structure, attributes
- Half-edge data structure
- Abstraction of half-edges

# Half-edge

- An edge on an oriented surface consists of two half-edges with opposite orientations.
- Each half-edge is incident to a unique face and a unique edge.
- Half edges incident to a face is oriented consistently (“clockwise” in Houdini)
- Half edges are in 1-1 correspondence with polygon corners (the source corner of the half edge)
- In a half-edge data structure, one stores indices of the neighboring hedge/point/face on hedge/point/face in a specific way.



# Half-edge data structure (C++ style)

- Point

  HalfEdge* he; *arbitrary hedge sourcing from the point*

- HalfEdge

  HalfEdge* next;

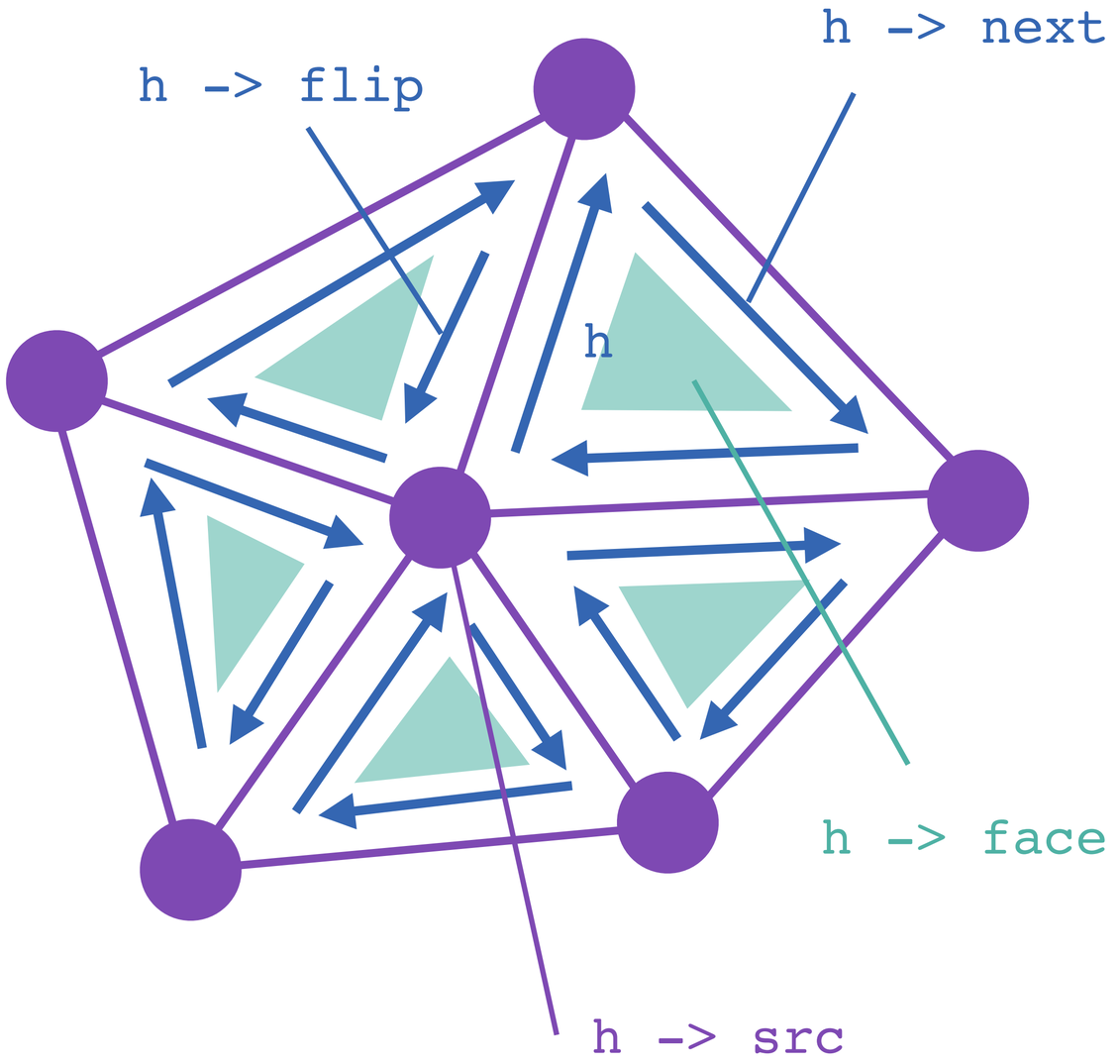
  HalfEdge* flip;

  Point* src;

  Face* face;

- Face

  HalfEdge* he; *arbitrary hedge in the face*

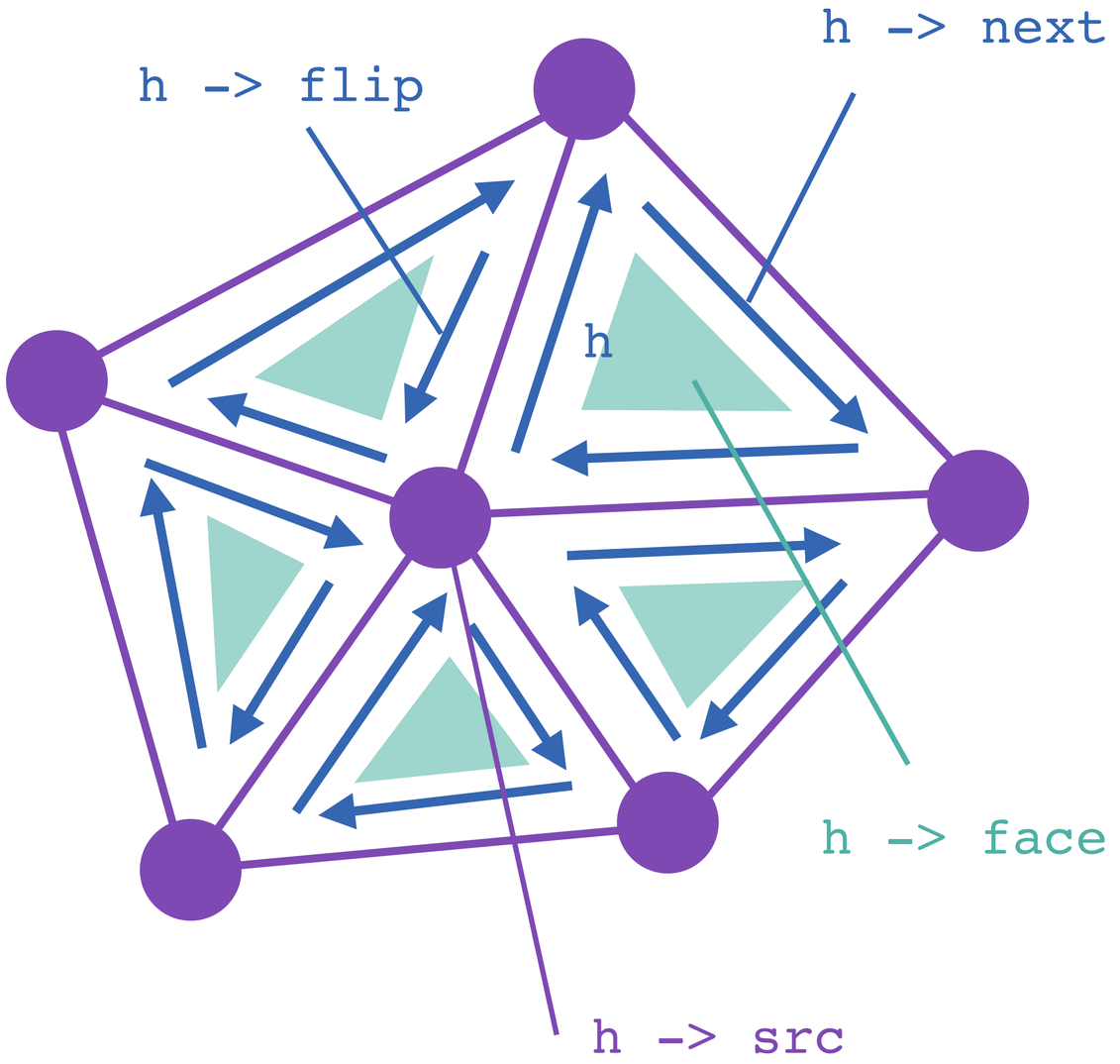


# Half-edge data structure (C++ style)

- Given a face, visit all its incident points

```
Face* face; // given

HalfEdge* he0 = face->he;
HalfEdge* he = he0;
do{
   Visit he->src;
   he = he->next;
} while(he!=he0);
```

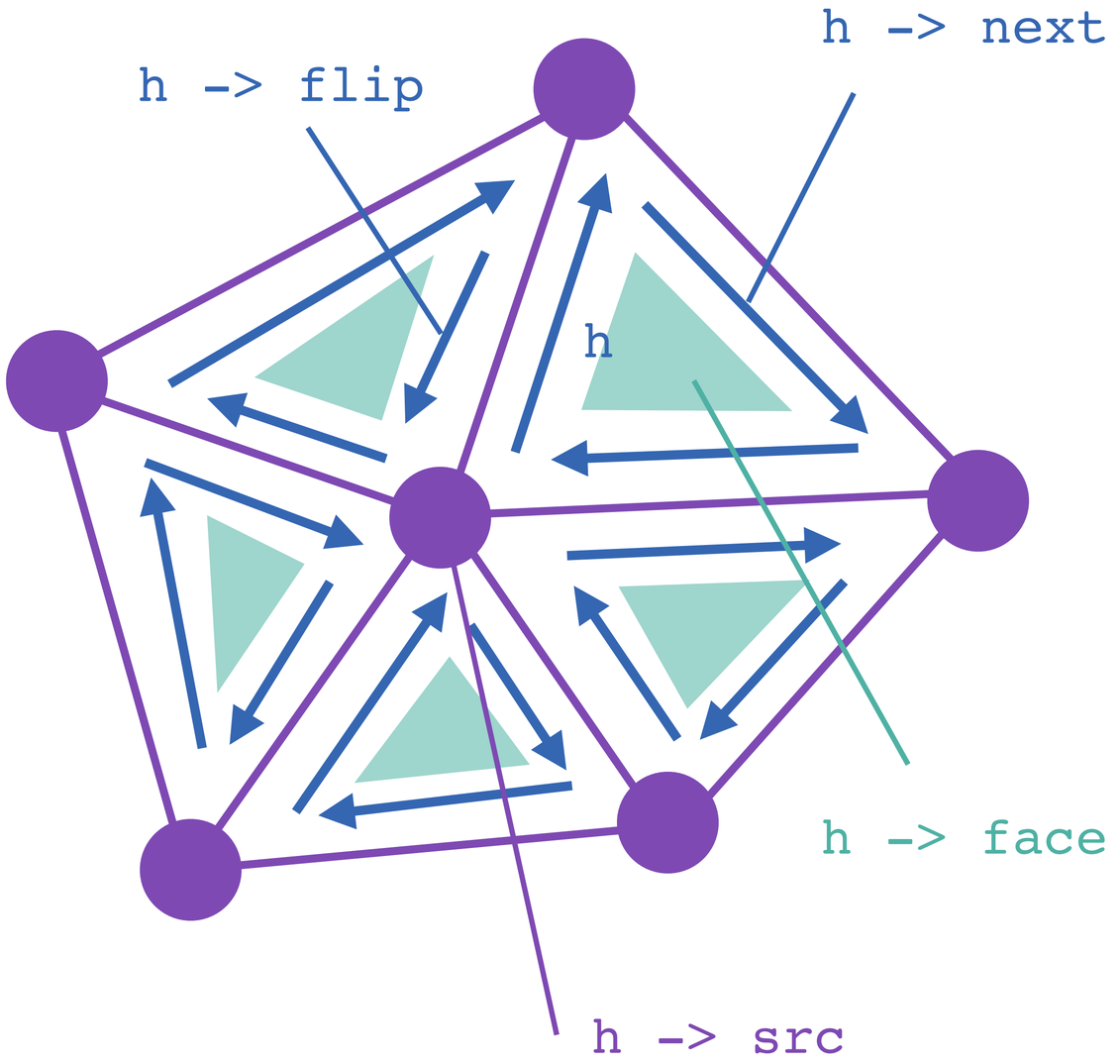


# Half-edge data structure (C++ style)

- Given a point, visit all its neighbor faces

```
Point* pt; // given

HalfEdge* he0 = pt->he;
HalfEdge* he = he0;
do{
   Visit he->face;
   he = he->flip->next;
} while(he!=he0);
```

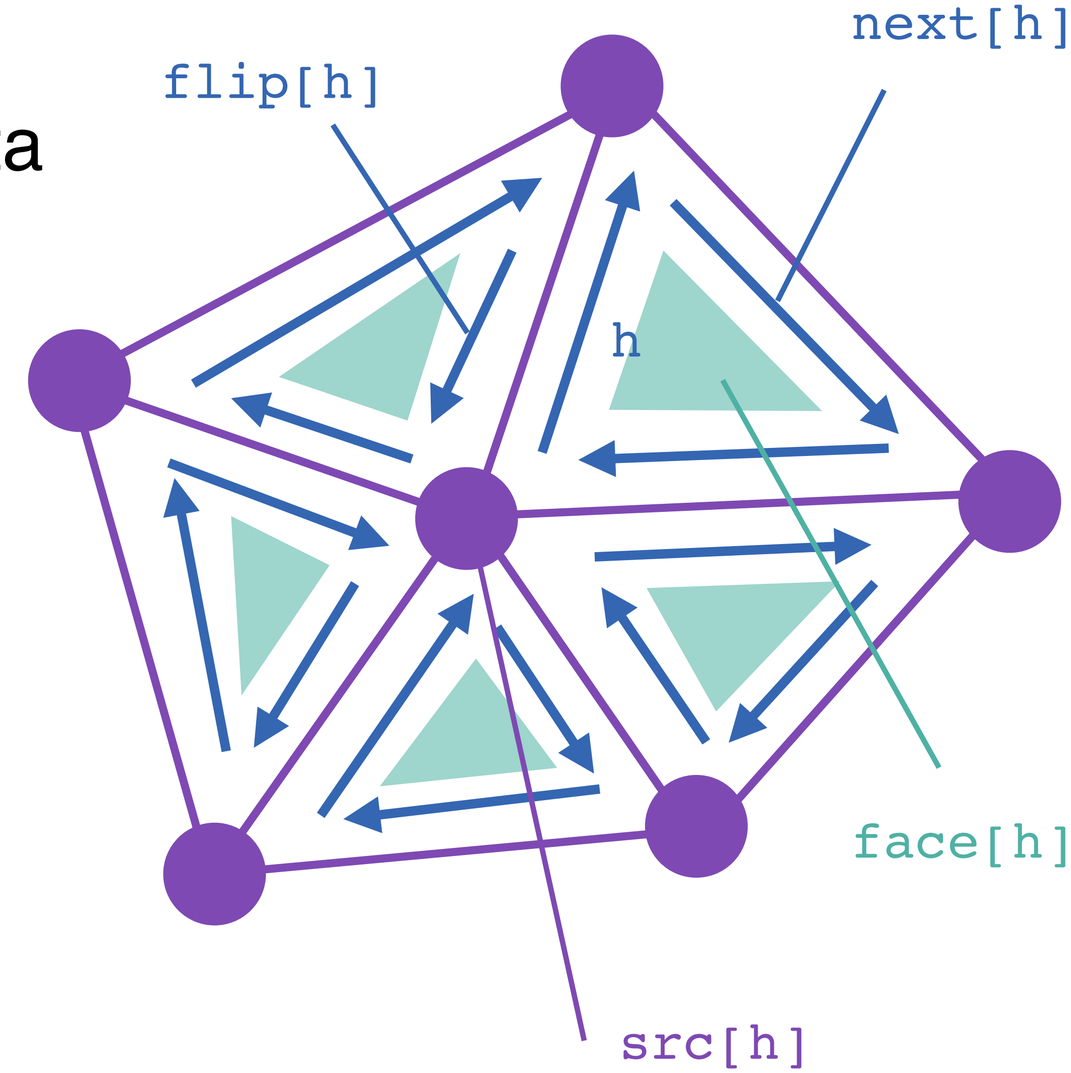


# Hedge pointers as attributes/functions

- Let $\mathcal{P}, \mathcal{H}, \mathcal{F}$ be the list of points, hedges, and faces
- What makes them into a hedge data structure is the following functions (a.k.a. attributes)

$$\texttt{next}: \mathcal{H} \to \mathcal{H}$$
$$\texttt{flip}: \mathcal{H} \to \mathcal{H}$$
$$\texttt{src}: \mathcal{H} \to \mathcal{P}$$
$$\texttt{face}: \mathcal{H} \to \mathcal{F}$$

$$\texttt{pt_hedge}: \mathcal{P} \to \mathcal{H}$$
$$\texttt{face_hedge}: \mathcal{F} \to \mathcal{H}$$



# Hedge built in MATLAB

```
% Script that builds a half edge data structure
[V,F,~,~] = readObj('./models/bunny.obj',0); % read mesh
[nV,~] = size(V);  % number of vertices
[nF,~] = size(F);  % number of faces
vert_id = (1:nV)'; % index set for vertices
face_id = (1:nF)'; % index set for faces
pp_face_rows = [F(:,1);F(:,2);F(:,3)]; % pp_ = data on half edges
pp_face_cols = [F(:,2);F(:,3);F(:,1)]; % represented by "point pairs"
pp_face_vals = [face_id; face_id; face_id];
pp_face = sparse(pp_face_rows,pp_face_cols,pp_face_vals,nV,nV);
pp_ishe = (pp_face~=0); % true if a point pair is a half edge
he_ind_in_pp = find(pp_ishe); % from hedge to linear index in pp matrix
nH = length(he_ind_in_pp); % number of half edges
he_id = (1:nH)'; % index set for half edges
pp_he = sparse(nV,nV); pp_he(he_ind_in_pp) = he_id; % he index given pp
pp_next_he_vals = [pp_he(sub2ind([nV,nV],F(:,2),F(:,3)));
                   pp_he(sub2ind([nV,nV],F(:,3),F(:,1)));
                   pp_he(sub2ind([nV,nV],F(:,1),F(:,2)))];
pp_next_he = sparse(pp_face_rows,pp_face_cols,pp_next_he_vals,nV,nV);
he_face = full(pp_face(he_ind_in_pp));
he_next = full(pp_next_he(he_ind_in_pp));
[he_src,he_dst] = ind2sub([nV,nV],he_ind_in_pp);
he_flip = full(pp_he(sub2ind([nV,nV],he_dst,he_src)));
he_isbdy = he_flip==0;
he_isint = ~he_isbdy;
vert_he = accumarray(he_src,he_id,[nV,1],@min);
face_he = accumarray(he_face,he_id,[nF,1],@min);
clear('pp_*')
```

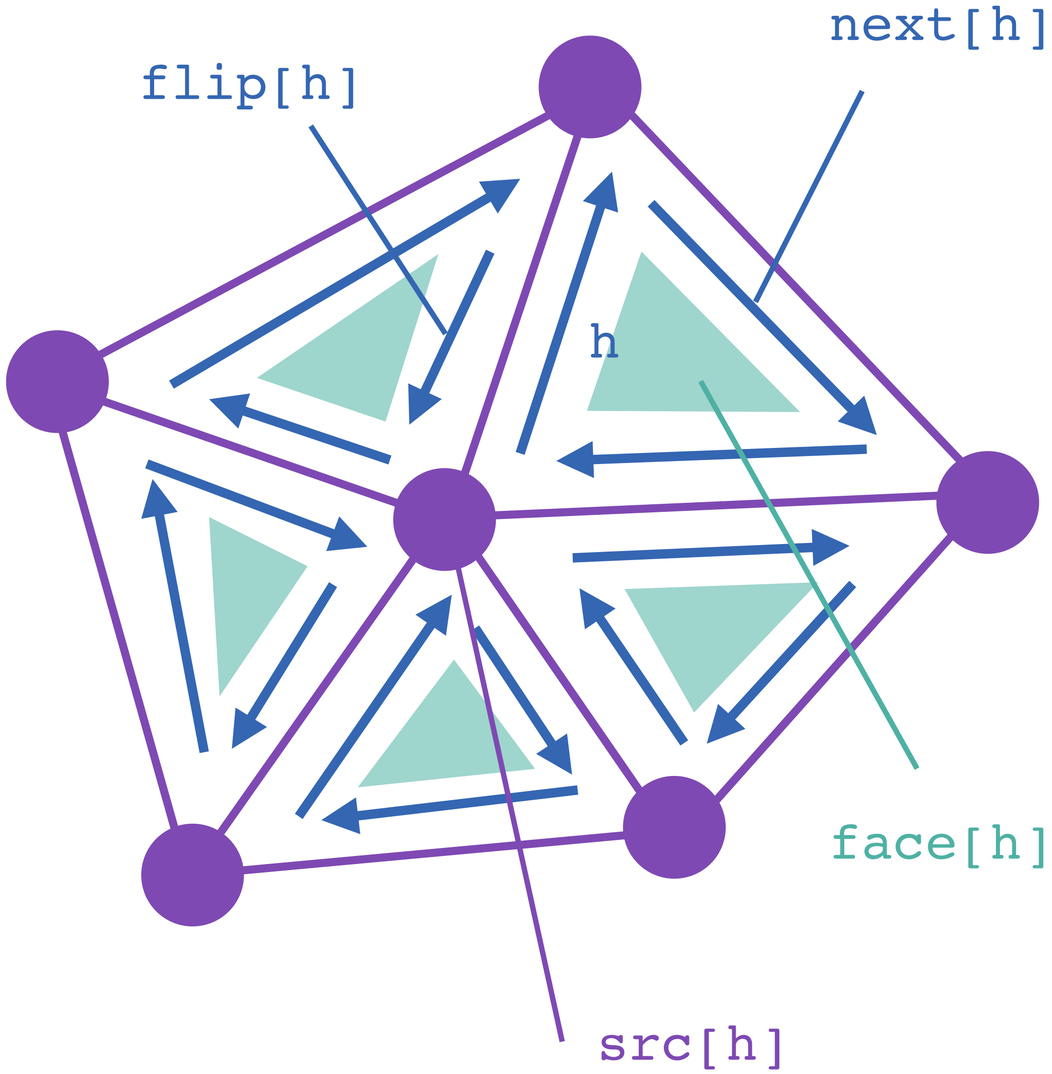
$\mathbf{V} \in \mathbb{R}^{|V|\times 3}$ given pt-prim data

$\mathbf{F} \in \mathbb{Z}^{|F|\times 3}$

# In Houdini

- Every hedge is uniquely associated with a vertex (polygon corner)
- Store attributes on hedge and run over all hedges are equivalent to doing them in vertex wrangle
- Index for hedge is:



# In Houdini

- Point

$\texttt{pt_hedge}: \mathcal{P} \rightarrow \mathcal{H}$

```
int he_index = pointhedge( 0, pt_index );
```

- HalfEdge

$\texttt{next}: \mathcal{H} \rightarrow \mathcal{H}$

```
int he_index2 = hedge_next( 0, he_index );
```

$\texttt{flip}: \mathcal{H} \rightarrow \mathcal{H}$

```
int he_index2 = hedge_nextequiv( 0, he_index );
```

$\texttt{src}: \mathcal{H} \rightarrow \mathcal{P}$

```
int ptindex = hedge_srcpoint( 0, he_index );
```

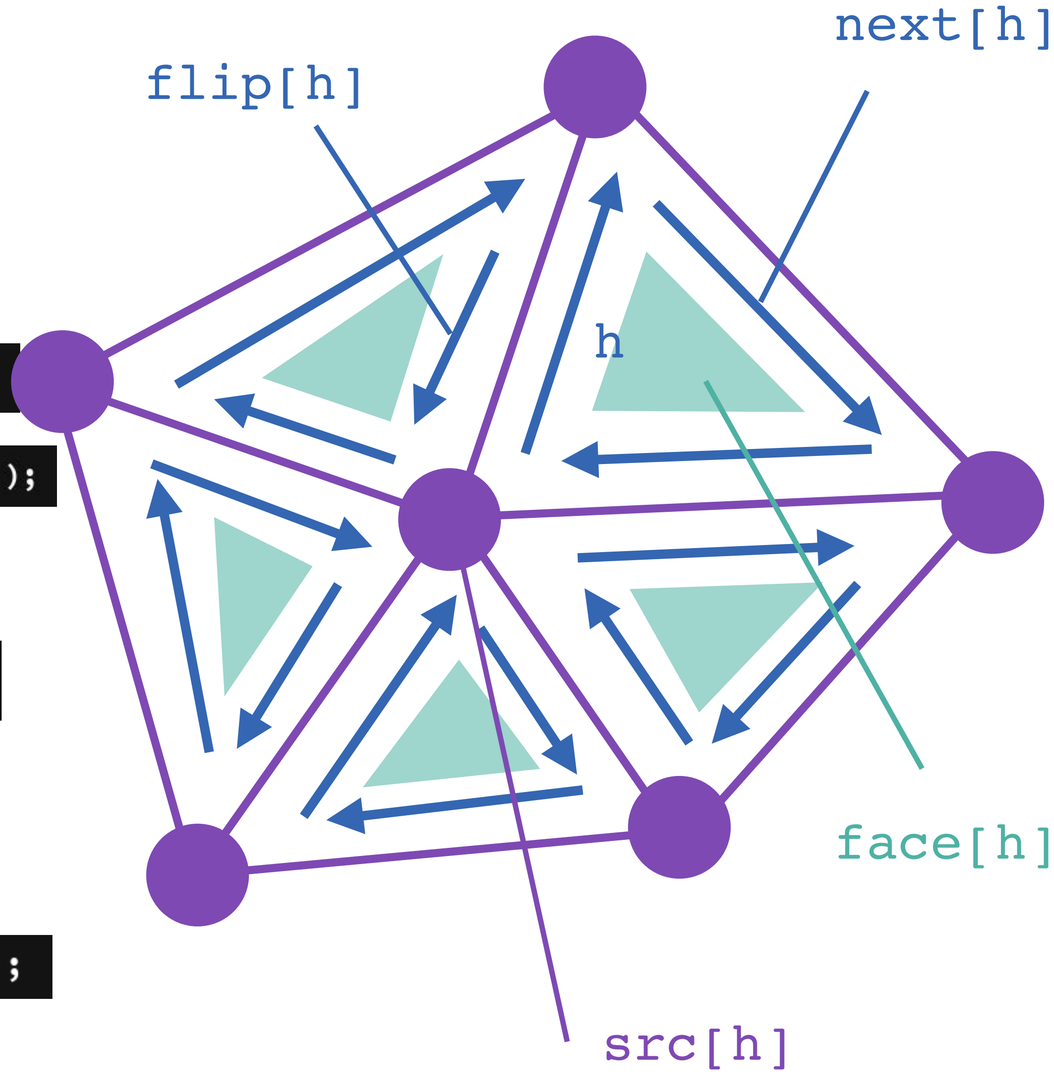
$\texttt{face}: \mathcal{H} \rightarrow \mathcal{F}$

```
int prim_index = hedge_prim( 0, he_index );
```

- Face

$\texttt{face_hedge}: \mathcal{F} \rightarrow \mathcal{H}$

```
int he_index = primhedge( 0, prim_index );
```



# In Houdini

For more information, see
https://www.sidefx.com/docs/houdini/vex/halfedges

Houdini 19.0 > VEX > Half-edges >

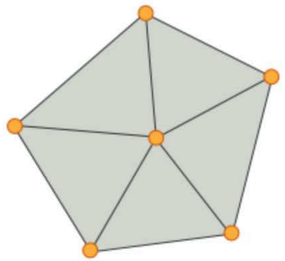
## Half-edges

*VEX has functions that let you treat edges as unshared per-face "half-edges".*

**On this page**


- Overview
- Equivalence
- Functions


### Overview

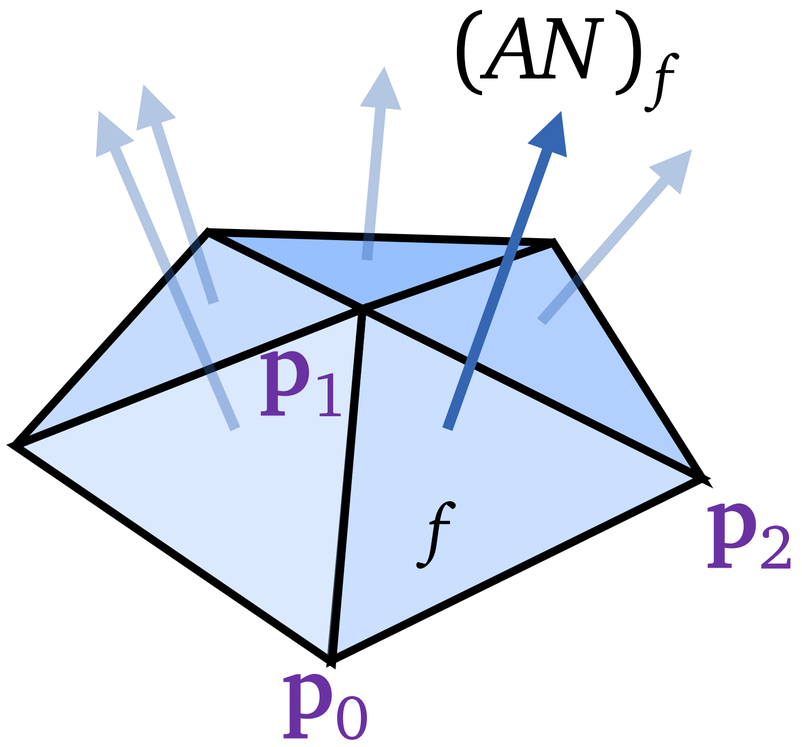


In Houdini, edges are normally treated as *undirected* and *shared* between faces. However, for some tasks (such as simplifying or cleaning geometry) it is useful to view faces as each having their own edges.

# Example

- Given a mesh, compute the face area-normal

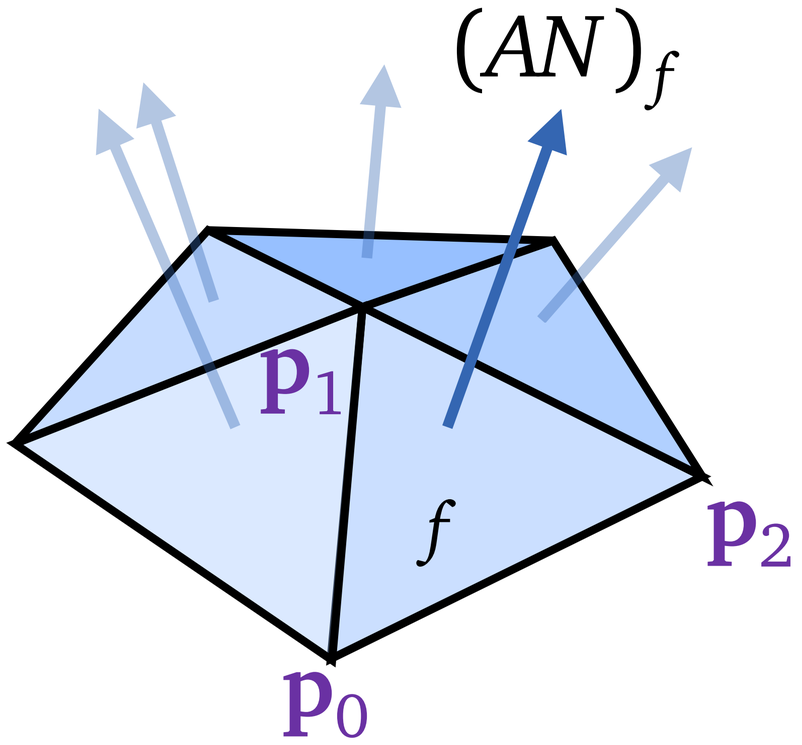
$$(AN)_f = \frac{1}{2}(\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$$



# Example

- Given a mesh, compute the face area-normal

$$(AN)_f = \frac{1}{2}(\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$$




- Run over primitives

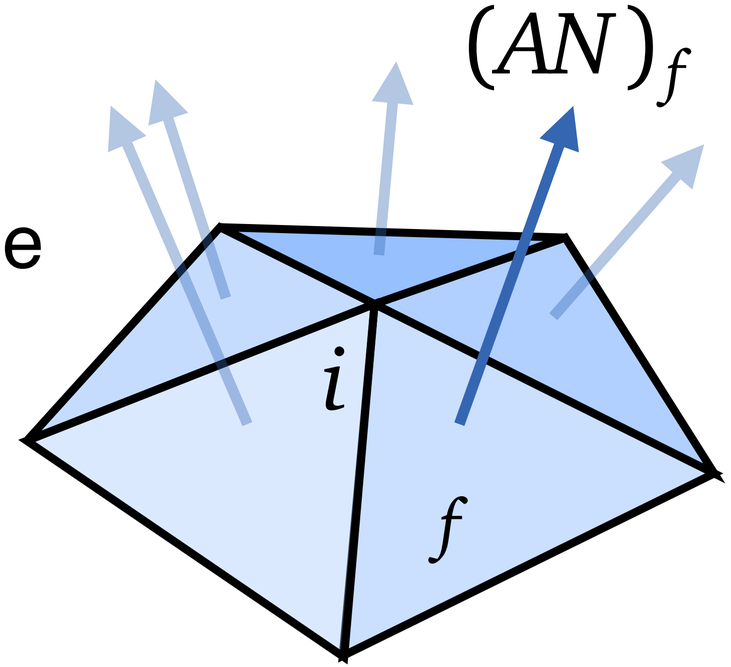
```
int he0 = primhedge(0,i@primnum);
int he = he0;
vector P[]; // point positions
int k = 0;
do{
  P[k] = point(0,"P",hedge_srcpoint(0,he));
  k++;
  he = hedge_next( 0 , he );
}while(he!=he0);

v@AreaNormal = 0.5*cross(P[2]-P[0], P[1]- P[0]);
f@Area = length(v@AreaNormal);
v@Normal = v@AreaNormal / f@Area;
```

# Next example

- Define point normal by area-weighted average of face normals

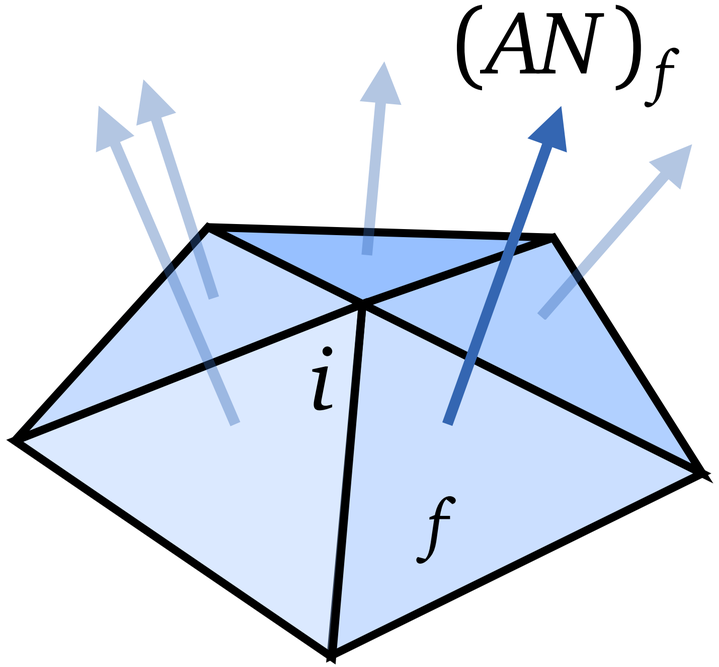
$$N_i = \textsc{Normalize}\left(\sum_{f \succ i} A_f N_f\right)$$



# Next example

- Define point normal by area-weighted average of face normals

$$N_i = \text{NORMALIZE}\left(\sum_{f \succ i} A_f N_f\right)$$




- Run over points

```
int he0 = pointhedge(0,i@ptnum);
int he = he0;
vector cum_area_normal = set(0,0,0);

do{
  int face = hedge_prim(0,he);
  cum_area_normal += prim(0,"AreaNormal",face);
  he = hedge_next(0,hedge_nextequiv(0,he));
}while( he != he0 );

v@PointNormal = normalize(cum_area_normal);
```

# Abstraction of Half Edges

- Point-primitive data structure, attributes
- Half-edge data structure
- Abstraction of half-edges

# Oriented polygon Meshes

- Without loss of generality, every oriented polygon manifold mesh is closed.  (Fill up the boundary component by a polygon face.)

- These closed oriented polygon meshes can be described just by the collection of half edges and two operators (“next” and “flip”).

# Minimalistic half edge surface

- An oriented surface mesh $(\mathcal{H}, \texttt{next}, \texttt{flip})$ consists of
  - a finite set $\mathcal{H}$
  - an invertible map (a.k.a. a permutation) $\texttt{next} : \mathcal{H} \to \mathcal{H}$
  - a fixed-point-free involution $\texttt{flip} : \mathcal{H} \to \mathcal{H}$

- Each face is an orbit of $\texttt{next}$

$$\mathcal{F} := \mathcal{H} / \langle \texttt{next} \rangle$$

- Each point is an orbit of $\texttt{next} \circ \texttt{flip}$

$$\mathcal{P} := \mathcal{H} / \langle \texttt{next} \circ \texttt{flip} \rangle$$