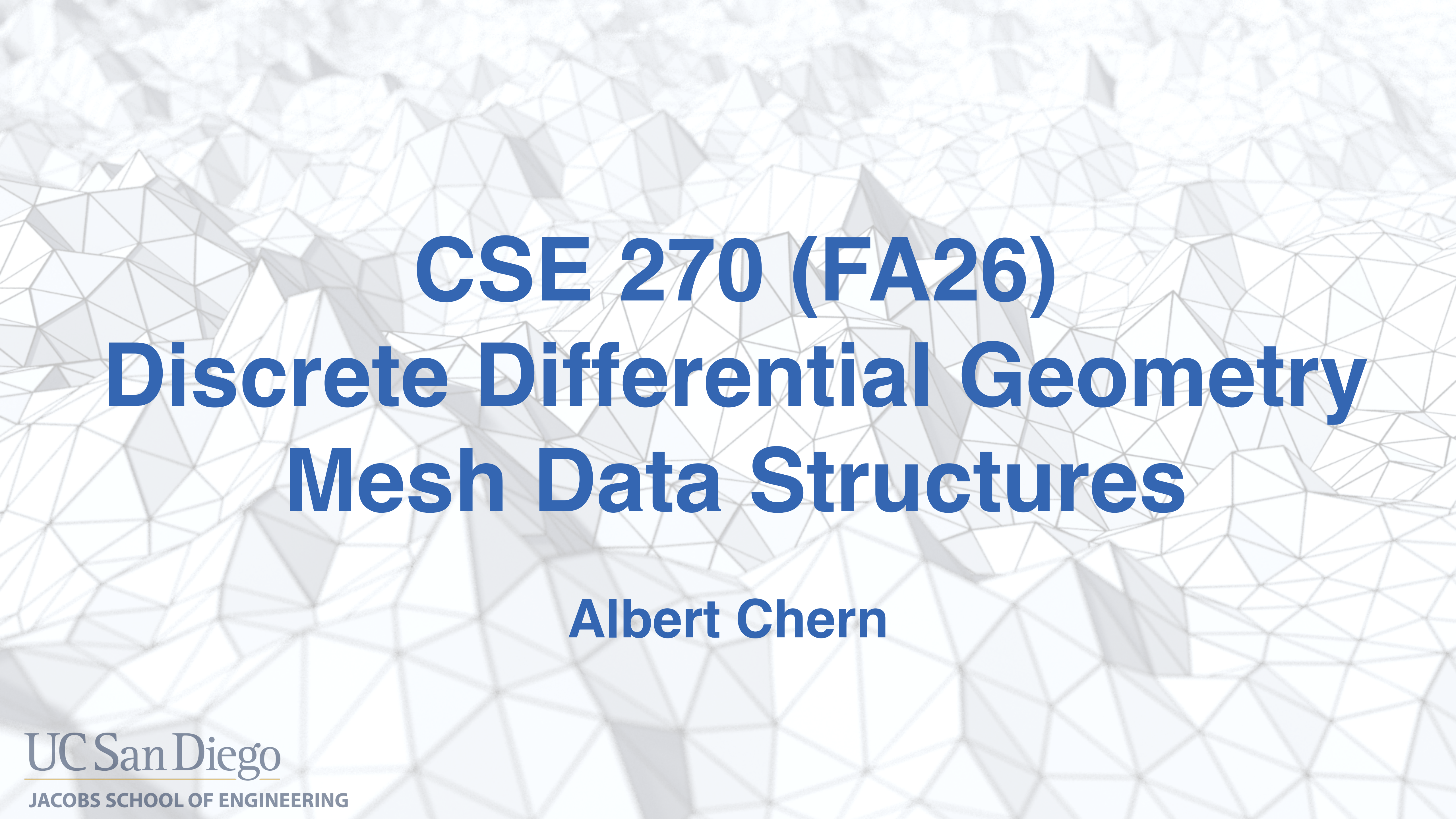




CSE 270 (FA26)

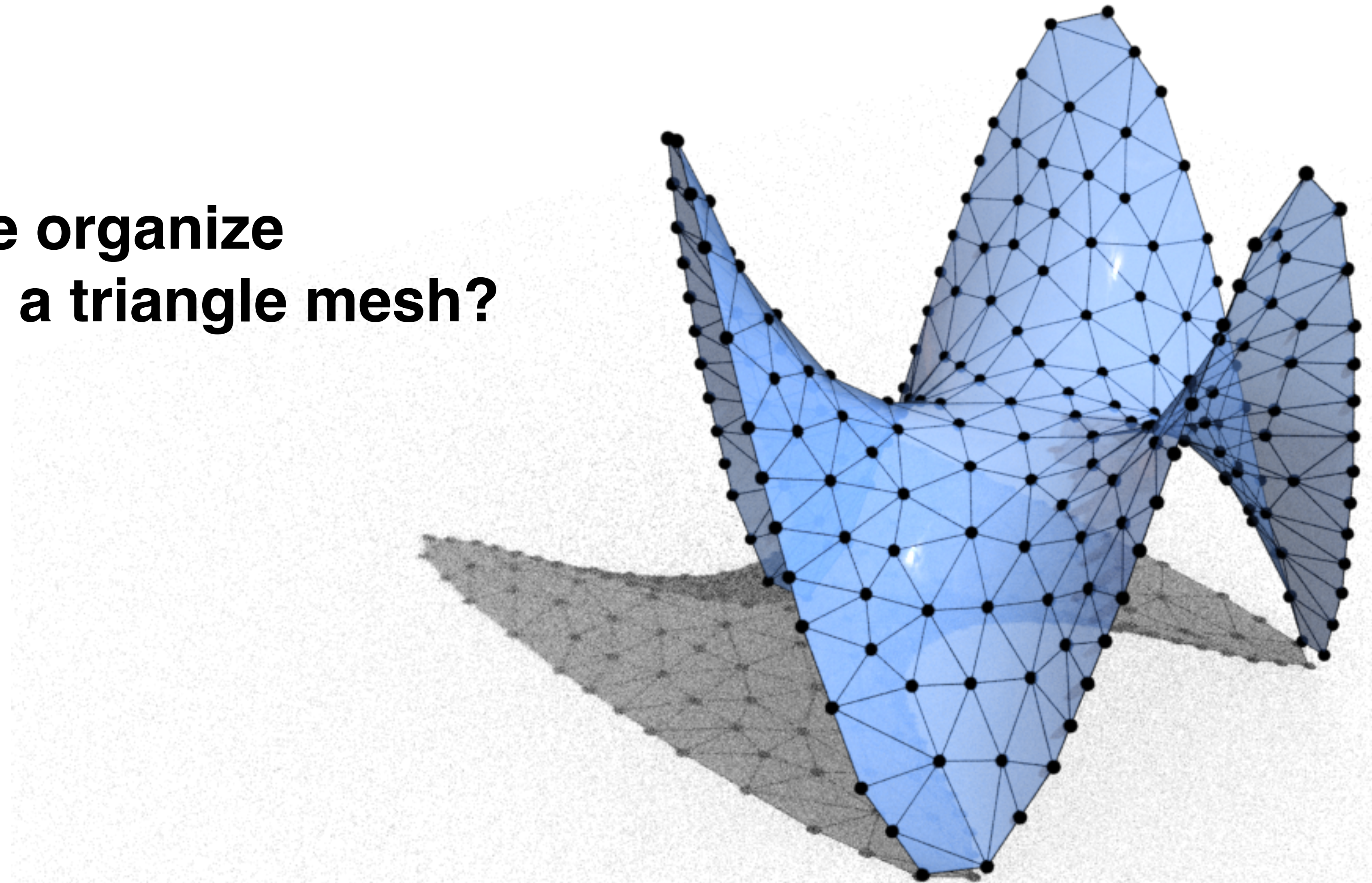
Discrete Differential Geometry

Mesh Data Structures

Albert Chern

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

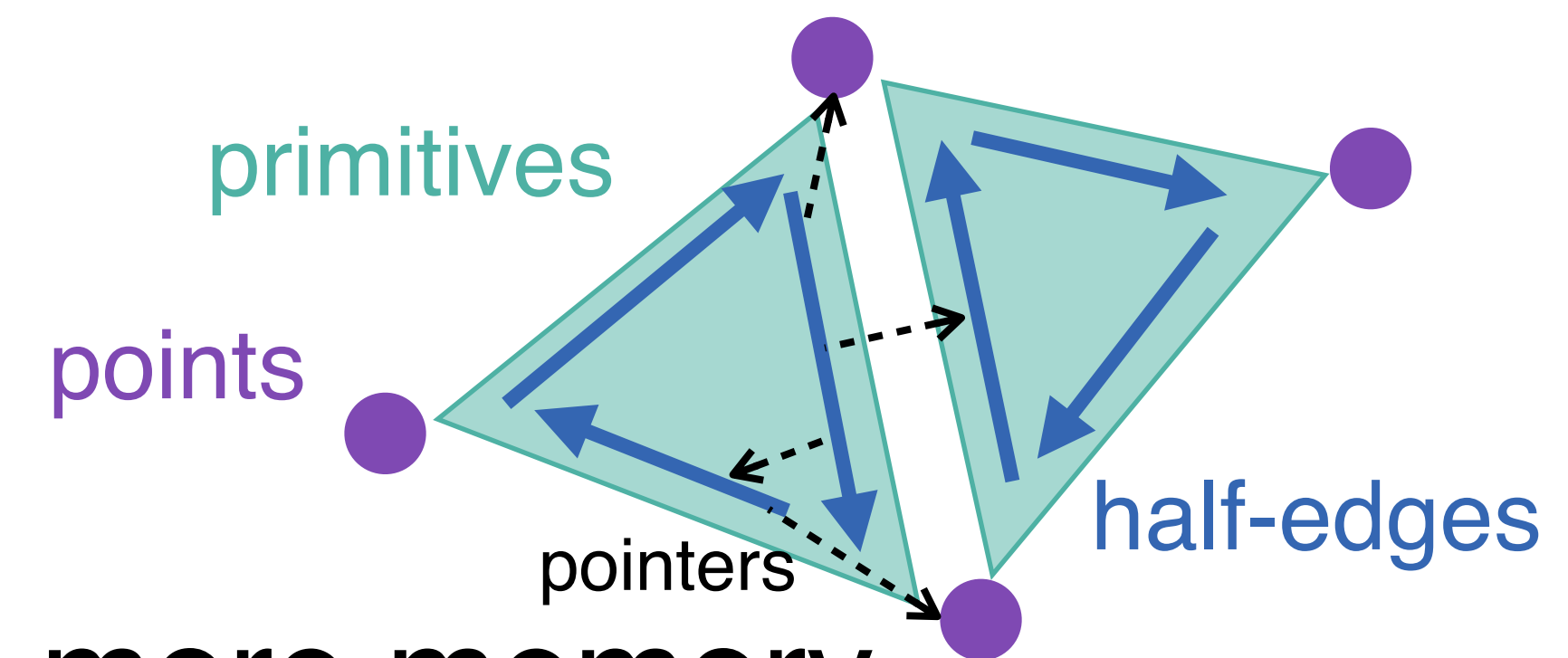
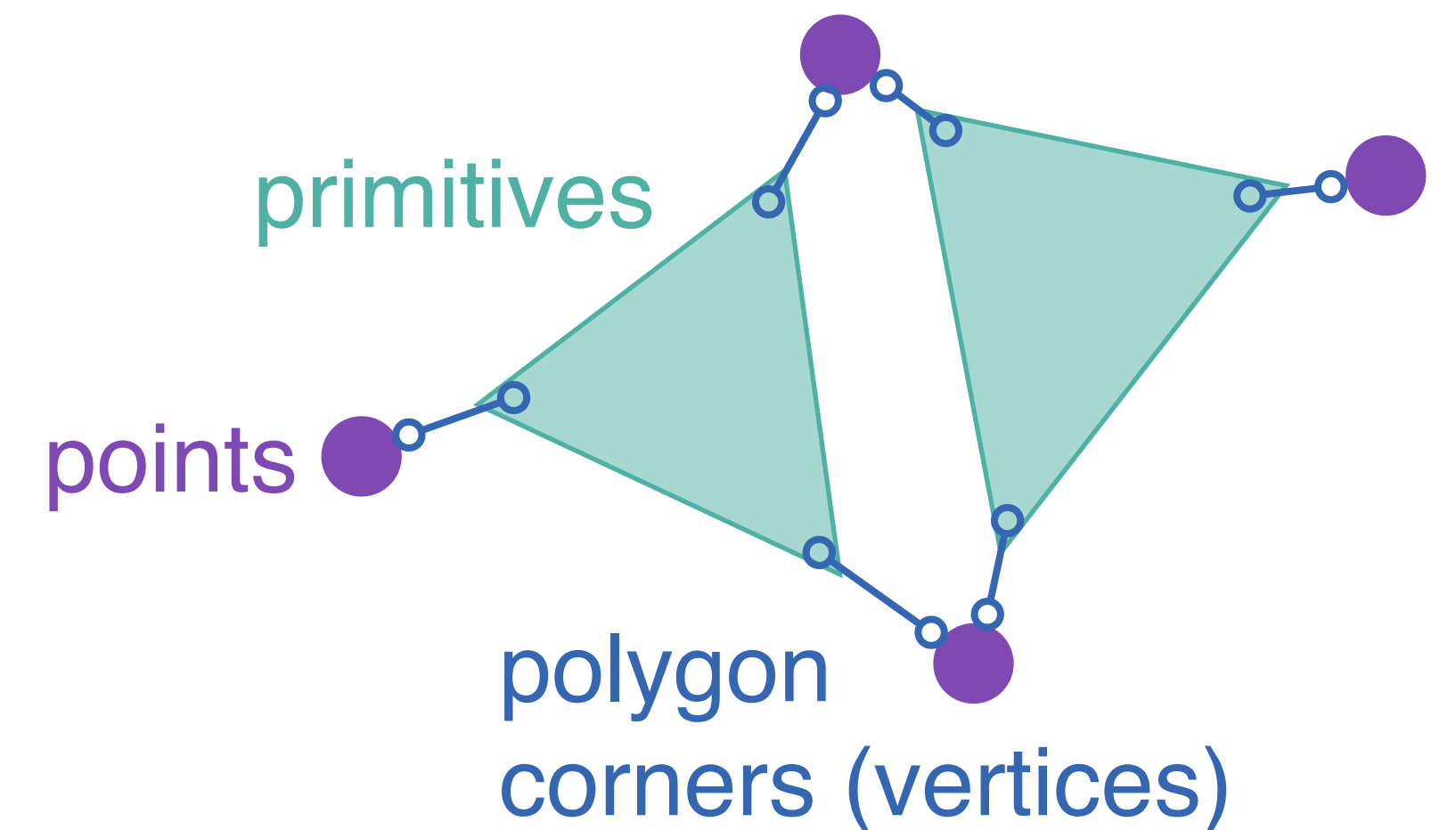
- Two commonly used data structures:

- **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

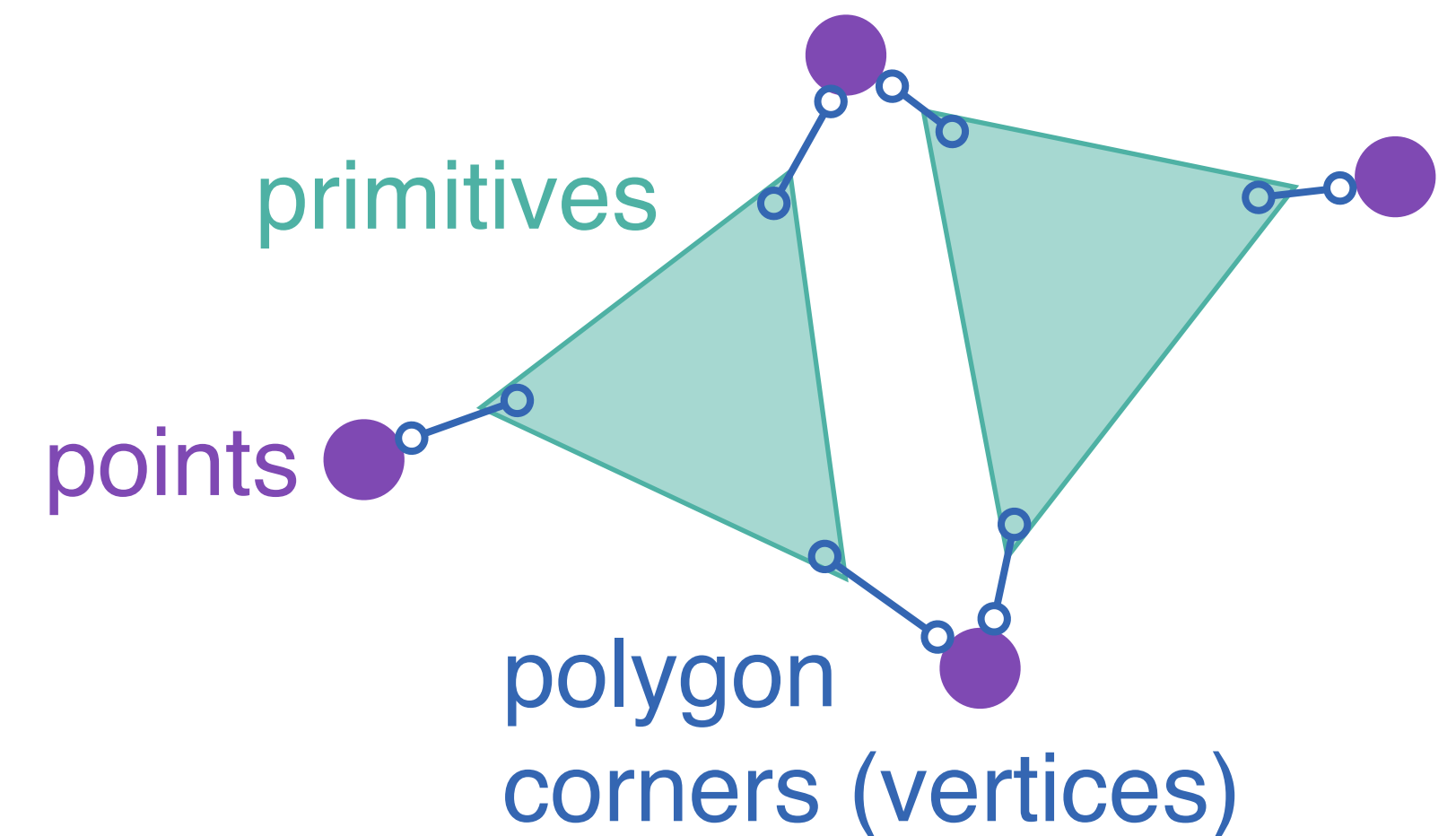


Point-Primitive data structure and attributes

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

Point-Primitive data structure

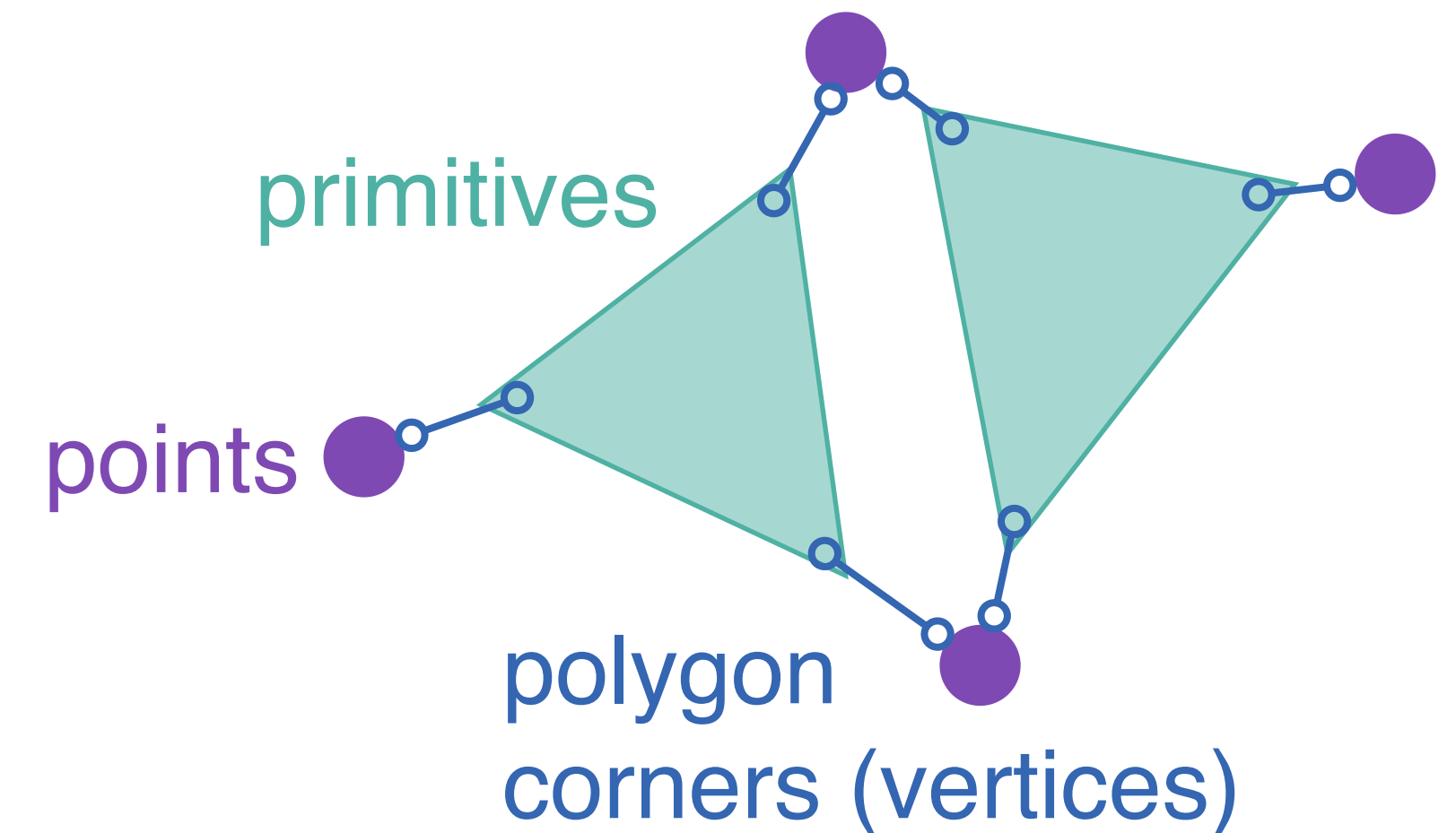
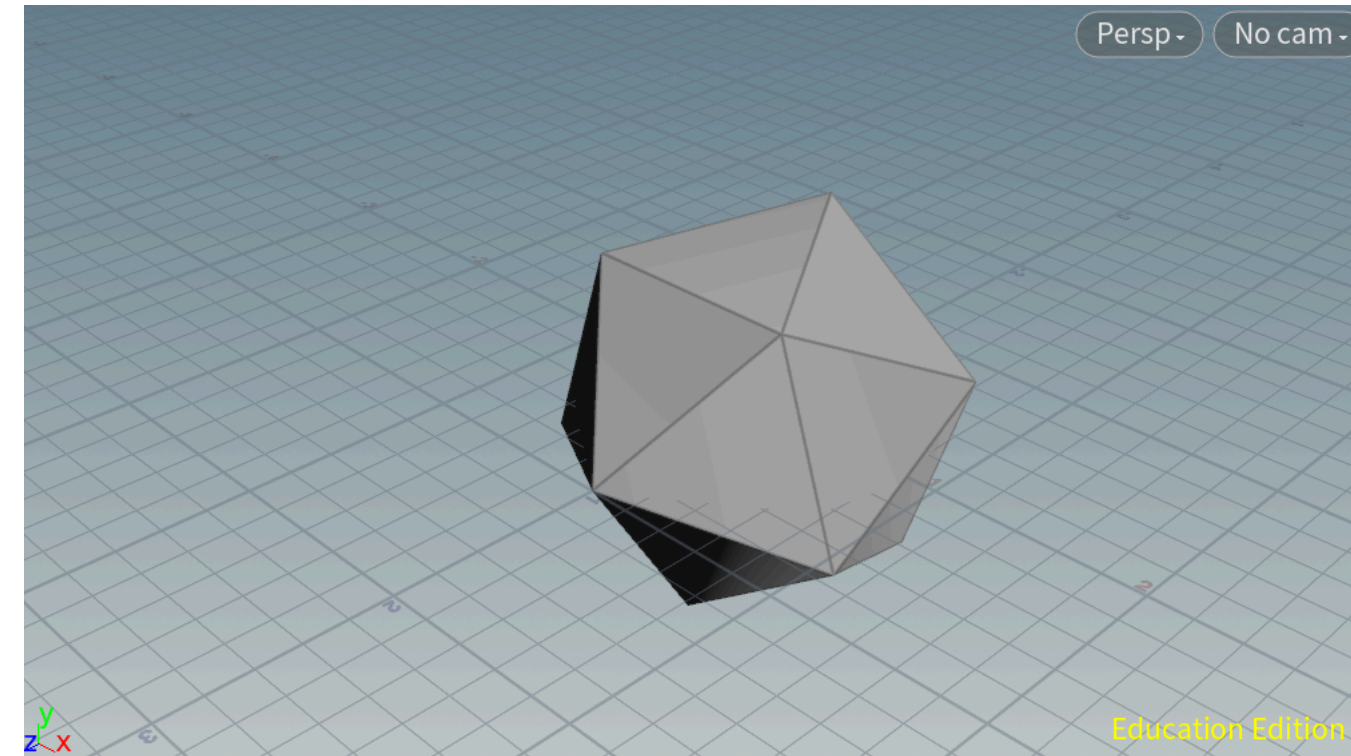
- The mesh is described by
 - ▶ a set $\mathcal{P}$ of points
 - ▶ a set $\mathcal{F}$ of faces (primitives)
 - ▶ a set $\mathcal{V}$ of $\mathcal{F} \times \mathcal{P}$ - tuples
- Each element $v = (f, p)$ is a polygon corner (known as a **vertex** in Houdini);
it indicates that face f is hooked to point p
- These vertices are partially ordered.



Point-primitive representation

- Points

Scene View x Geometry Spreadsheet x +			
obj geo1			
Node: plato...	Group: View Intrinsic		
	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)

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

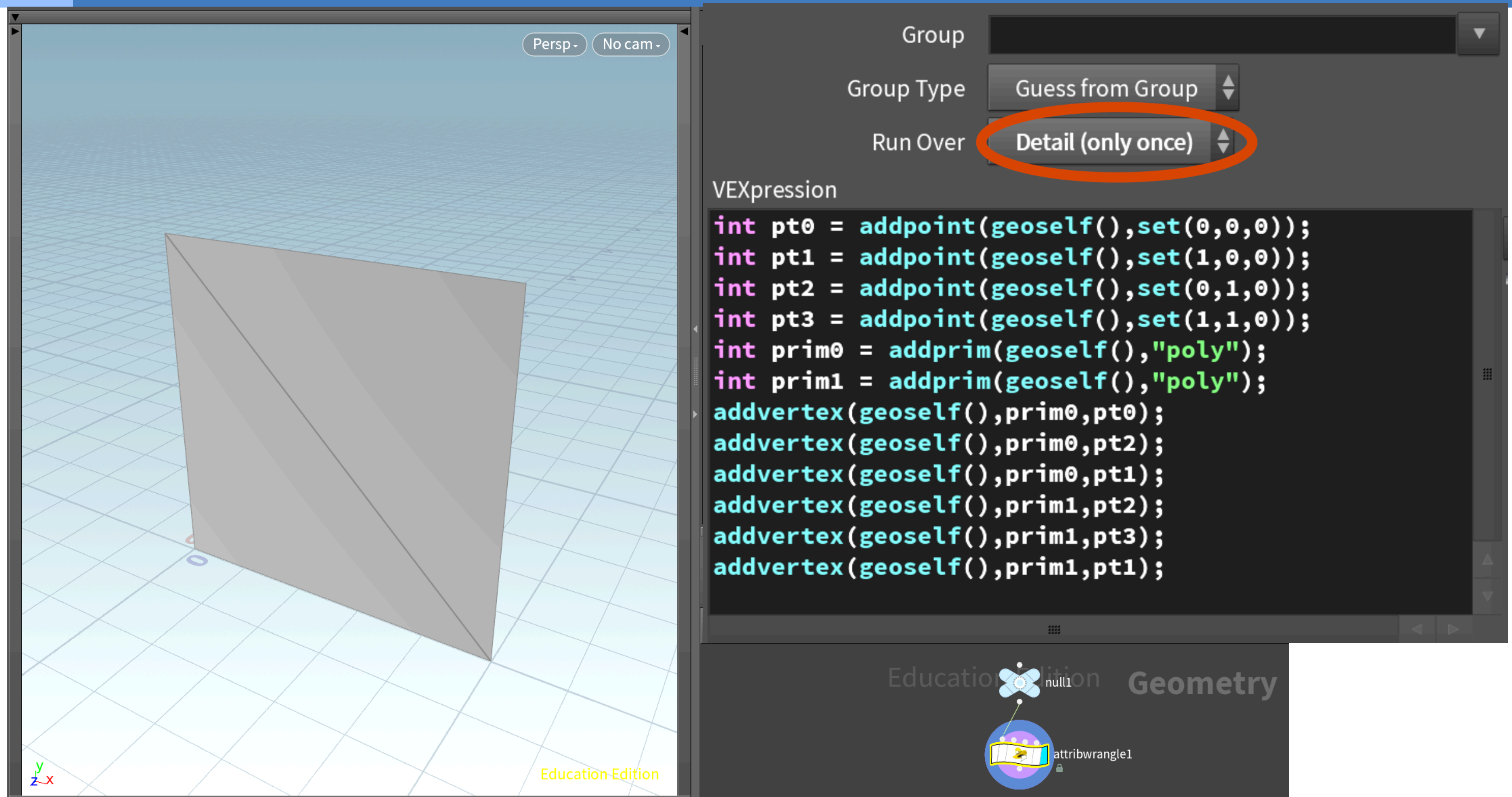
there is also an internal linear indexing of vertices that are more useful

Node: plato...		
	Point Num	
0	0:0	2
1	0:1	1
2	0:2	0
3	1:0	3
4	1:1	2
5	1:2	0
6	2:0	4

- Primitives

Node: plato...	
0	
1	
2	
3	
4	
5	
6	

Create triangle mesh from scratch



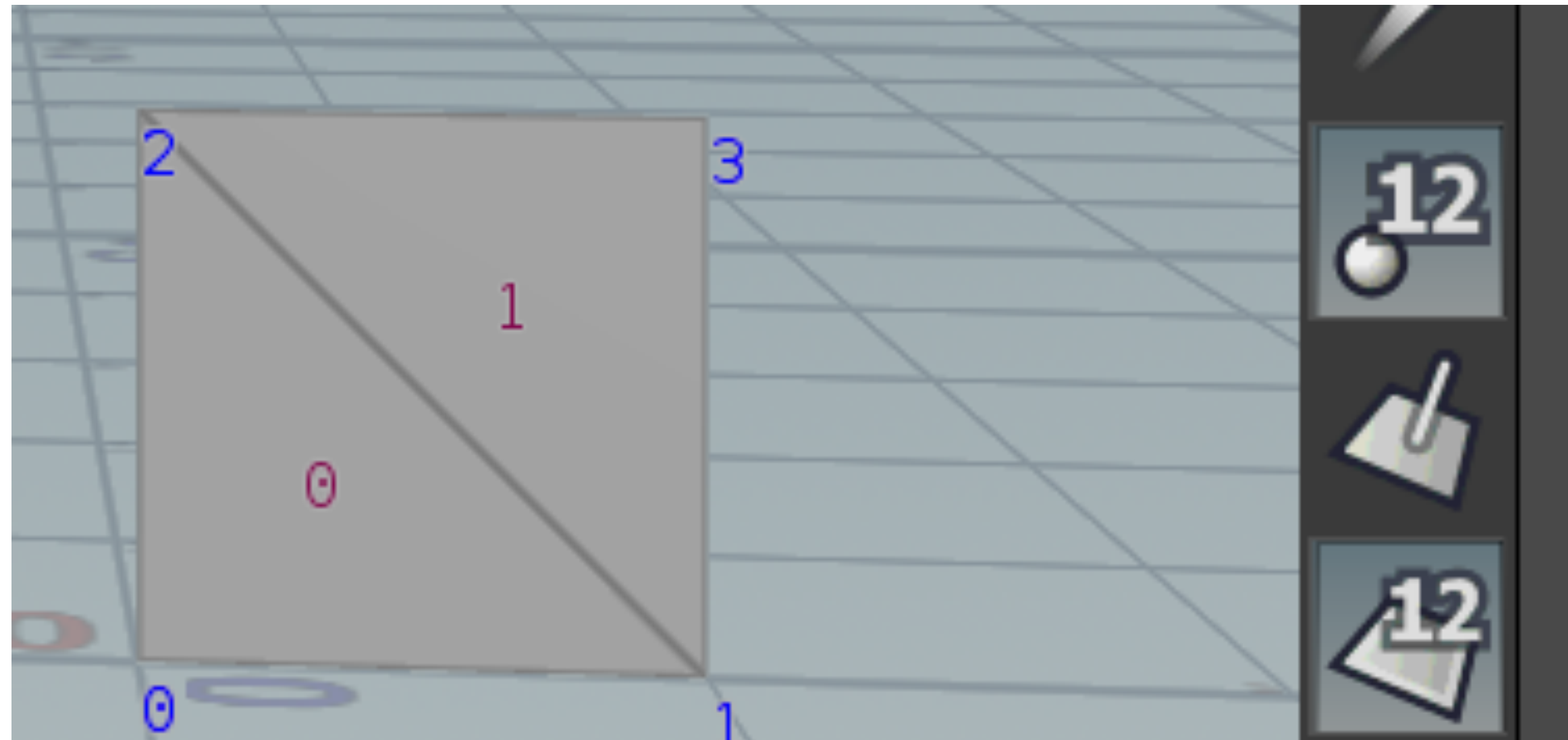
The image displays the Blender 2.79 interface. On the left, the 3D viewport shows a gray triangle mesh on a blue grid floor. The top right of the viewport has 'Persp' and 'No cam' buttons. On the right, the Shader Editor shows the VEX expression for the material. The 'Run Over' dropdown menu is circled in red, showing 'Detail (only once)'. The VEX expression code is as follows:

```
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);
```

At the bottom, the 'Education Edition' logo is visible on the left, and the 'Geometry' node is shown in the bottom right corner of the Shader Editor.

Create triangle mesh from scratch

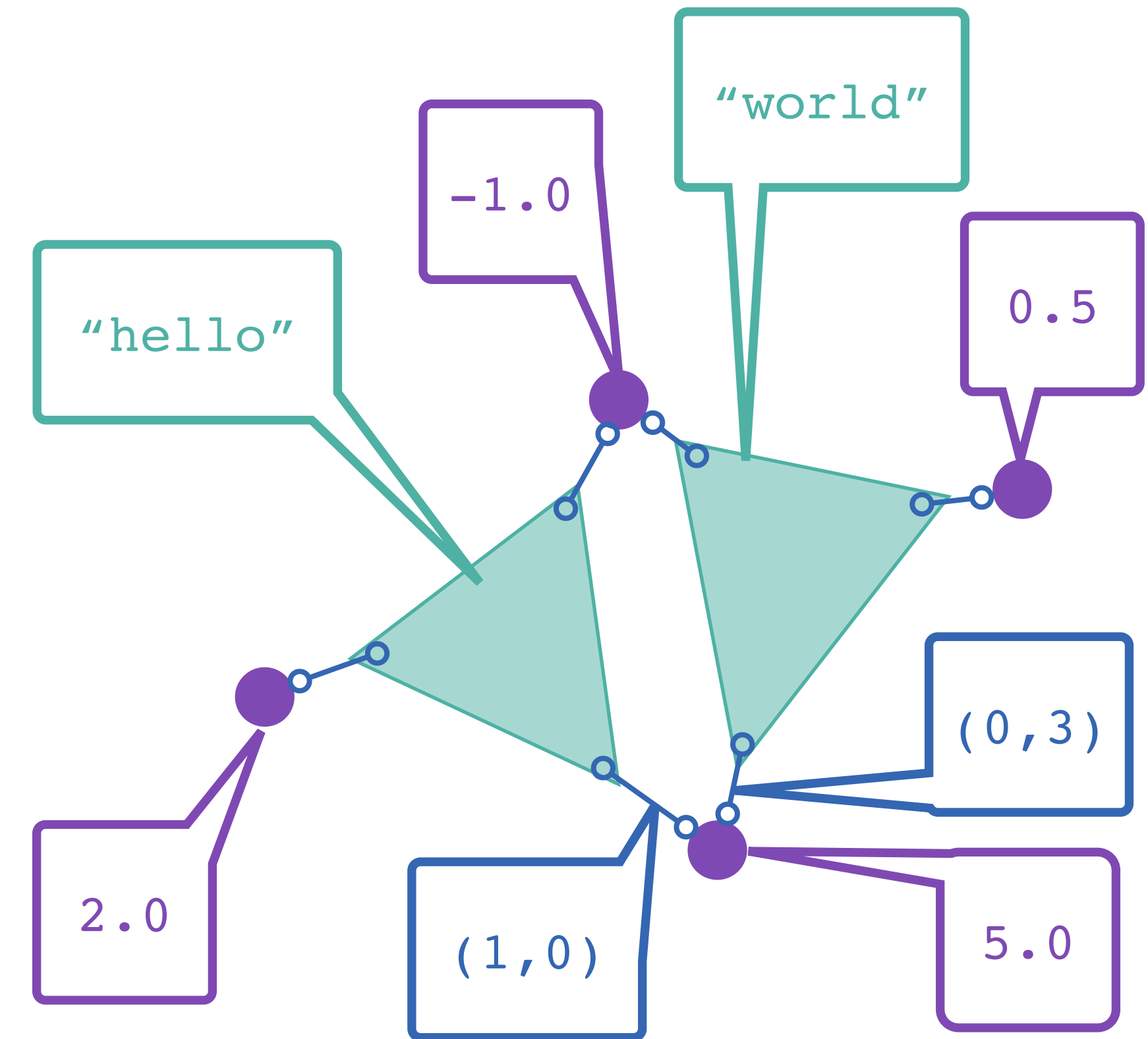
Visualize primnum and ptnum



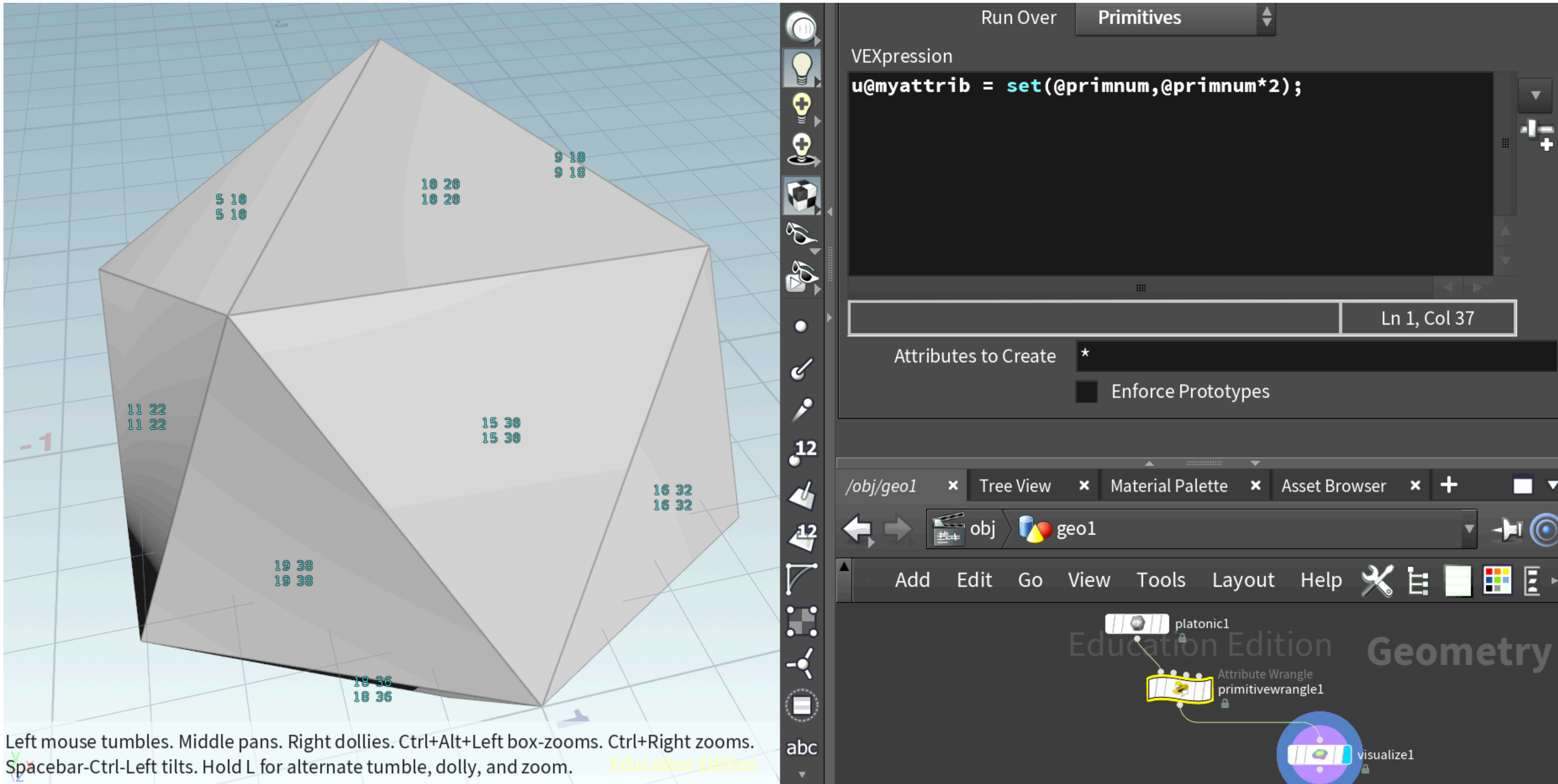
Attributes

- 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



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.x u@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

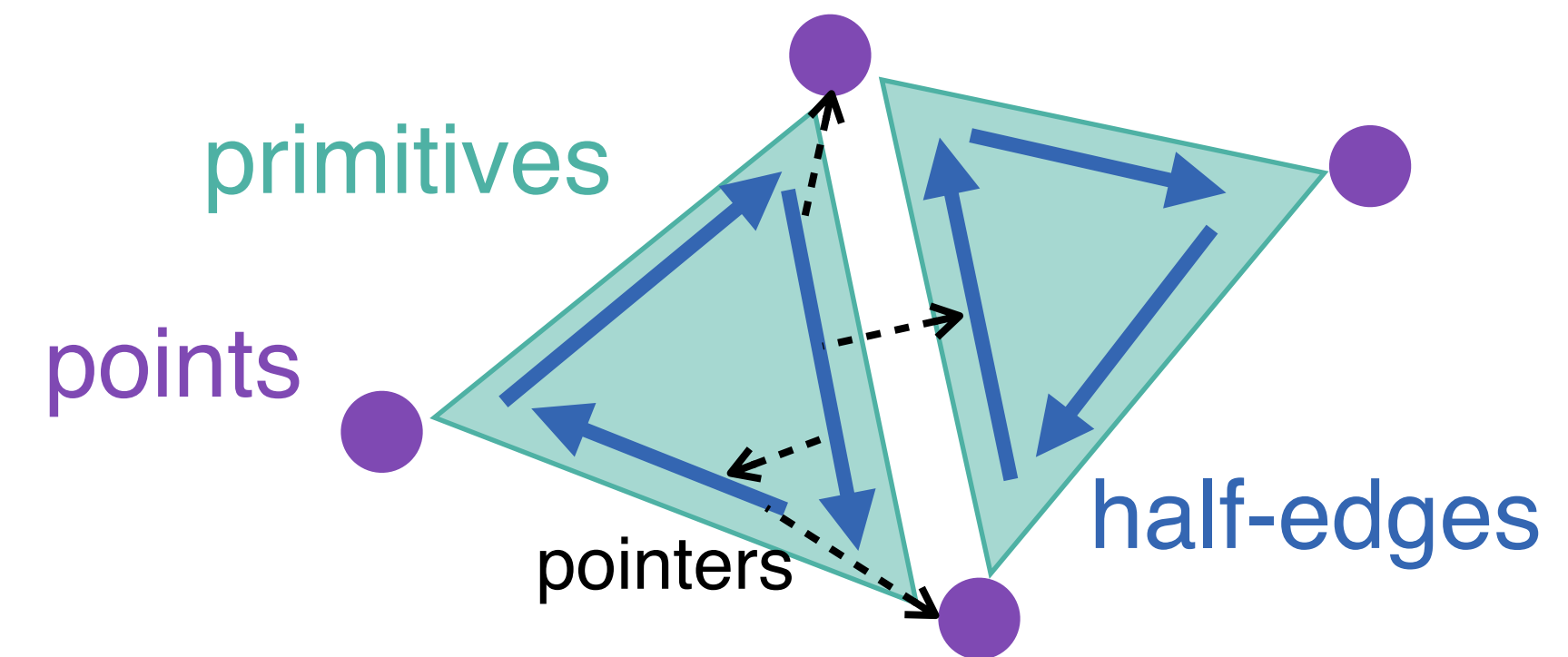
- You can also set attributes to the *current node*
 - ▶ Use the vex functions
`setpointattribvalues`
etc. (See documentation for more info)
- Since vex code runs in parallel, we can't directly read what we just write

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 half-edge/point/face on half-edge/point/face in a specific way.



Half-edge data structure (C++ style)

► Point

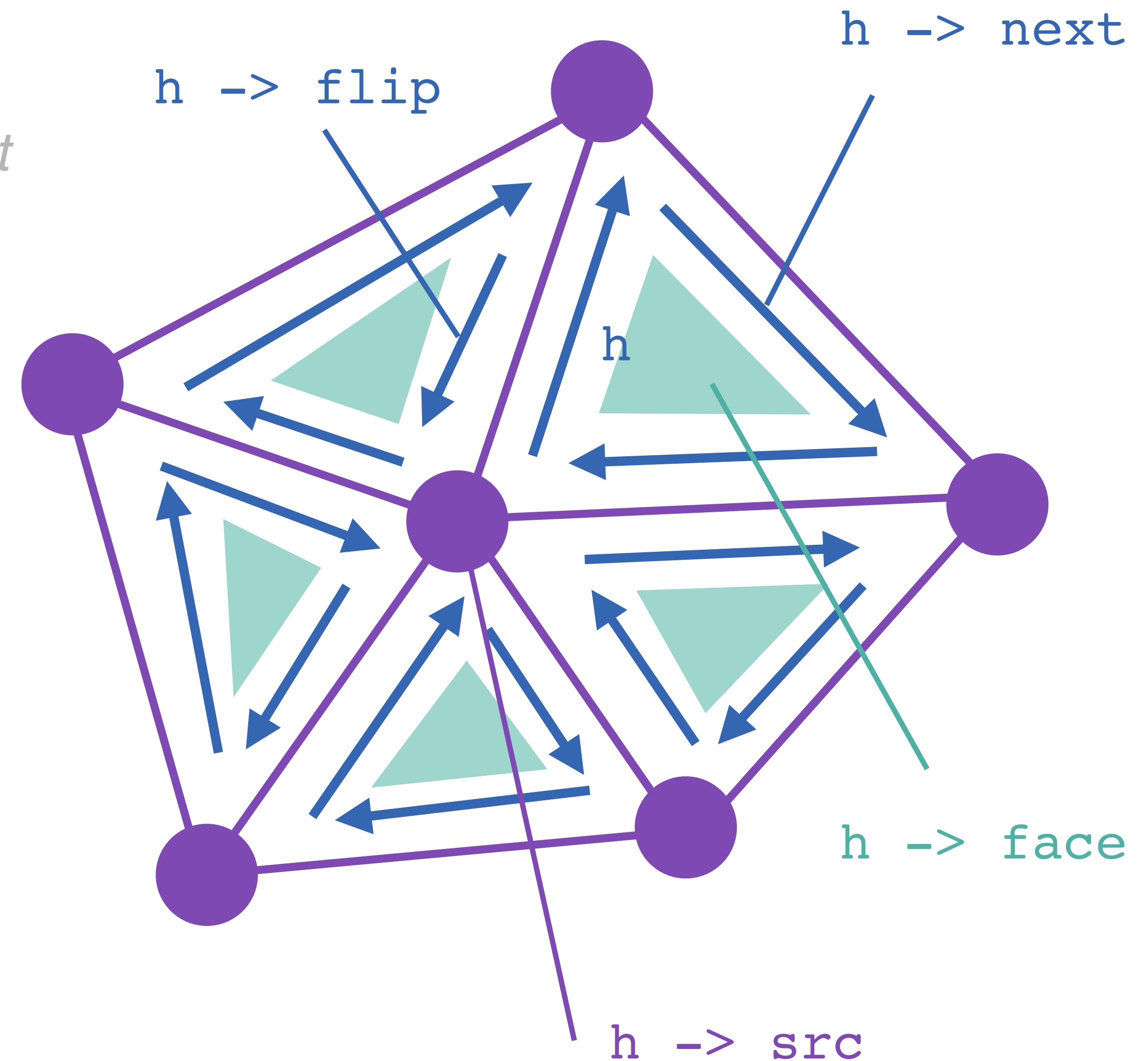
HalfEdge* he; *arbitrary hedge
sourcing from the point*

► HalfEdge

HalfEdge* next;
HalfEdge* flip;
Point* src;
Face* face;

► Face

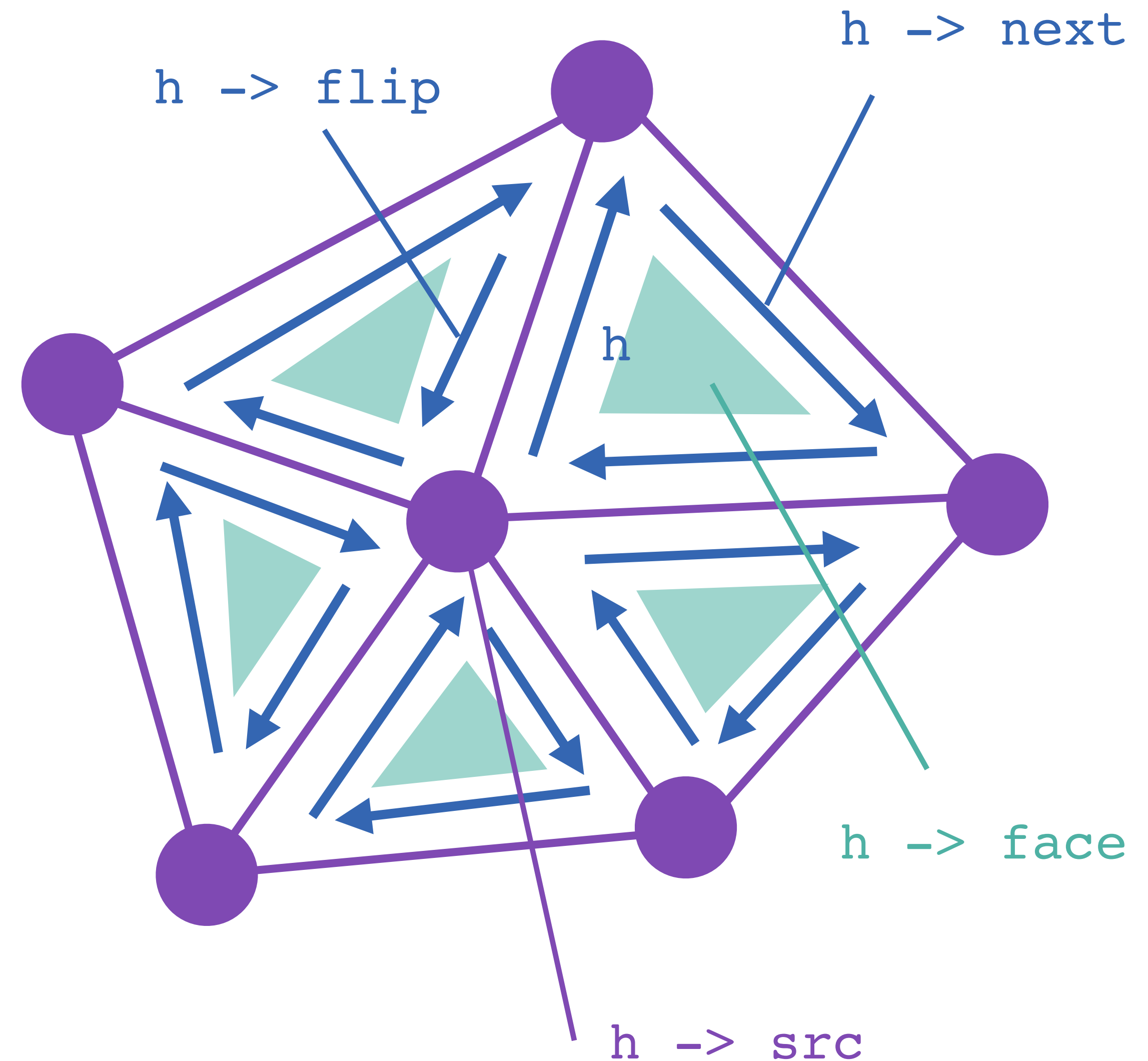
HalfEdge* he; *arbitrary hedge
in the face*



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;
```

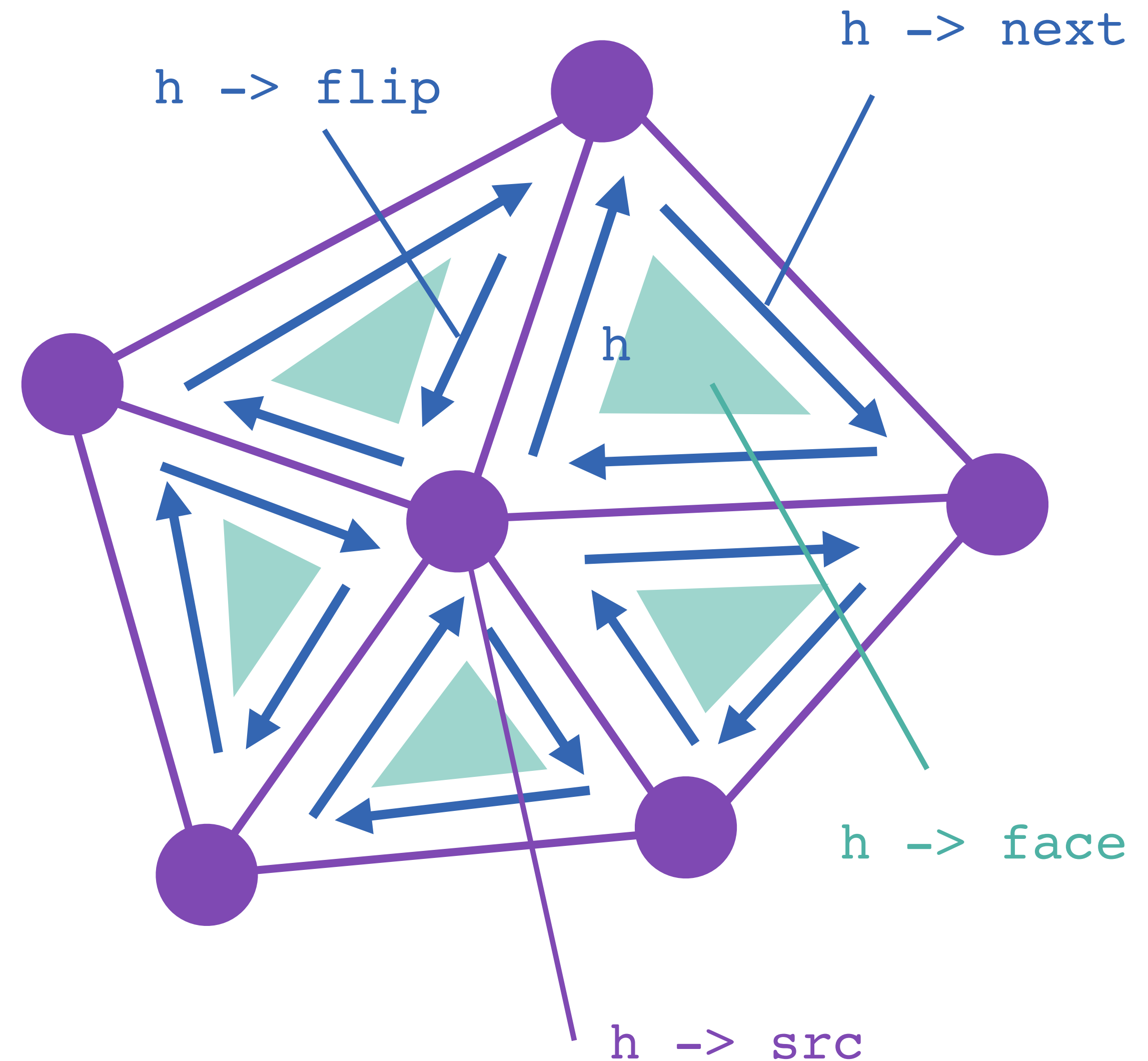
```
HalfEdge* he = he0;
```

```
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)

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

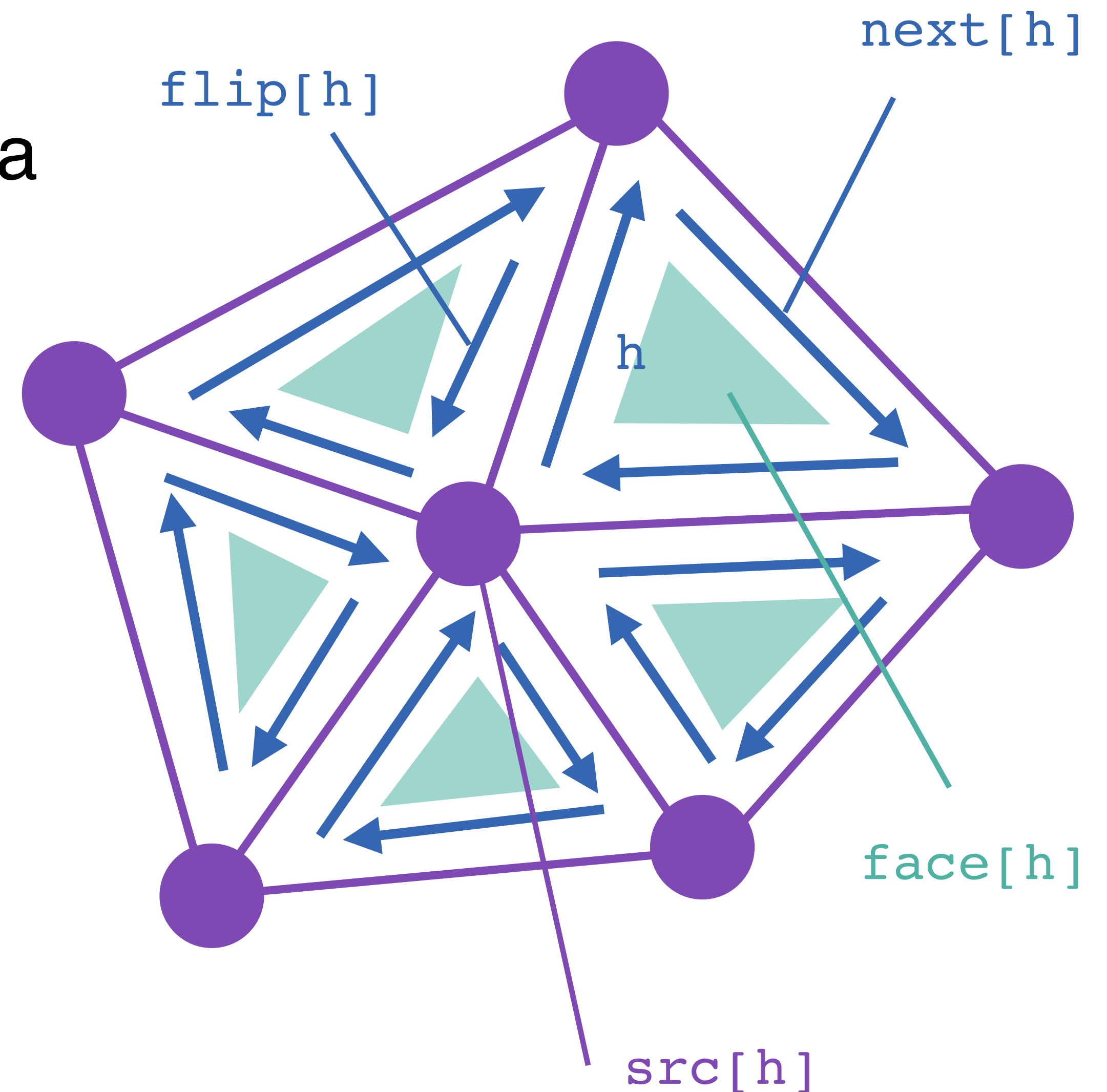
$\text{flip}: \mathcal{H} \rightarrow \mathcal{H}$

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

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

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

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



Hedge pointers as attributes/functions

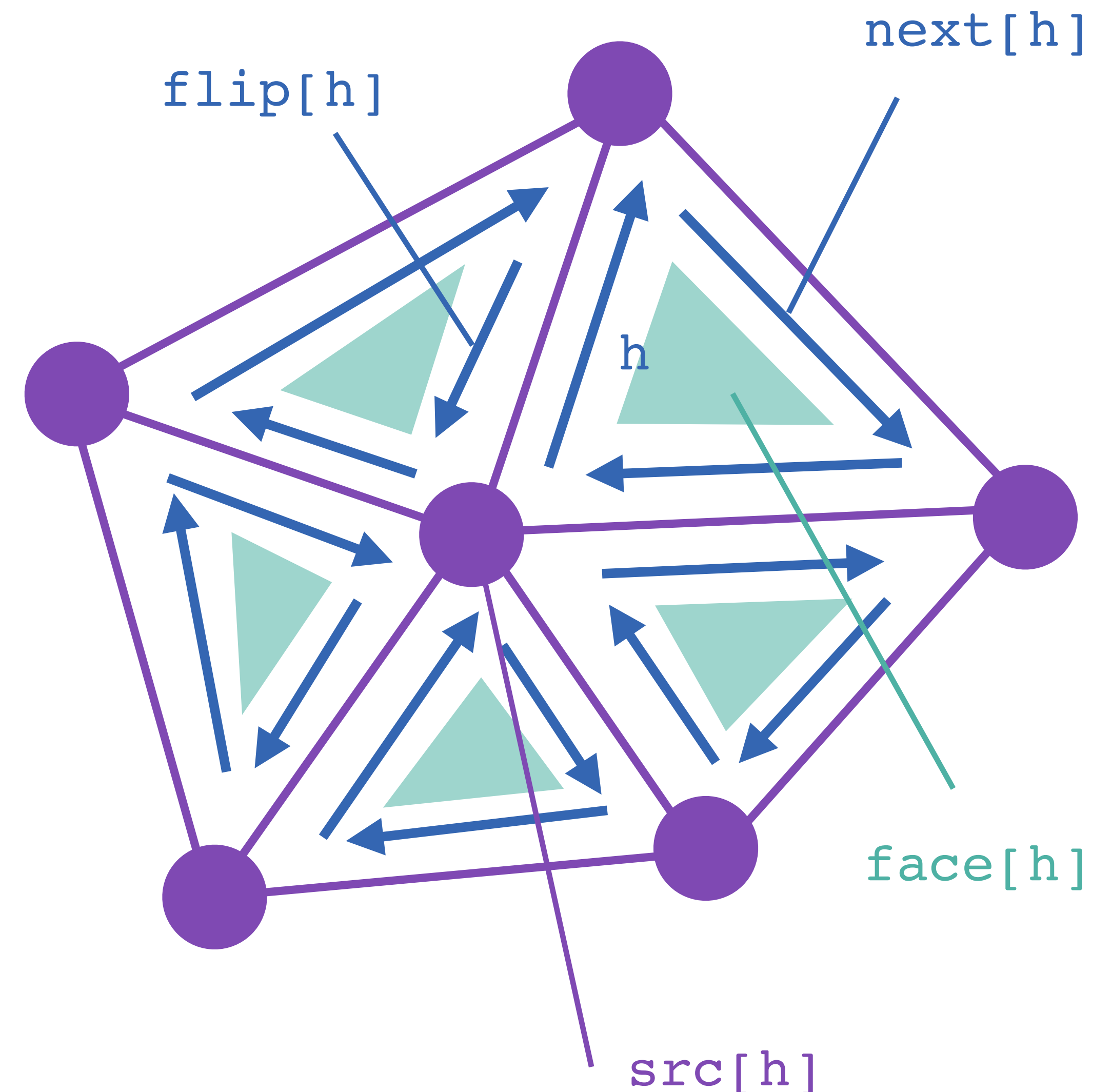
$\text{next}: \mathcal{H} \rightarrow \mathcal{H}$ $\text{face}: \mathcal{H} \rightarrow \mathcal{F}$
 $\text{flip}: \mathcal{H} \rightarrow \mathcal{H}$ $\text{pt_hedge}: \mathcal{P} \rightarrow \mathcal{H}$
 $\text{src}: \mathcal{H} \rightarrow \mathcal{P}$ $\text{face_hedge}: \mathcal{F} \rightarrow \mathcal{H}$

- When $\mathcal{P}, \mathcal{H}, \mathcal{F}$ are **sets of instances** of object classes
 - ▶ Think of these functions as pointer, attributes of the class
- **Parallel computing**: think of *single-instruction multiple data (SIMD)*
 - ▶ Shader language: code snippet run per point/ per face
 - ▶ These hedge functions are stored data available at each pt/face
- In **array processing, vectorization**
 - ▶ Everything in spreadsheets (arrays with semantics & data flow)
 - ▶ These hedge functions are indices of another array cell (pointer within the spreadsheet).

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:

```
Run Over Vertices
VEXpression
i@he = vertexhedge(0,i@vtxnum);
```



In Houdini

► Point

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

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

► HalfEdge

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

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

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

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

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

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

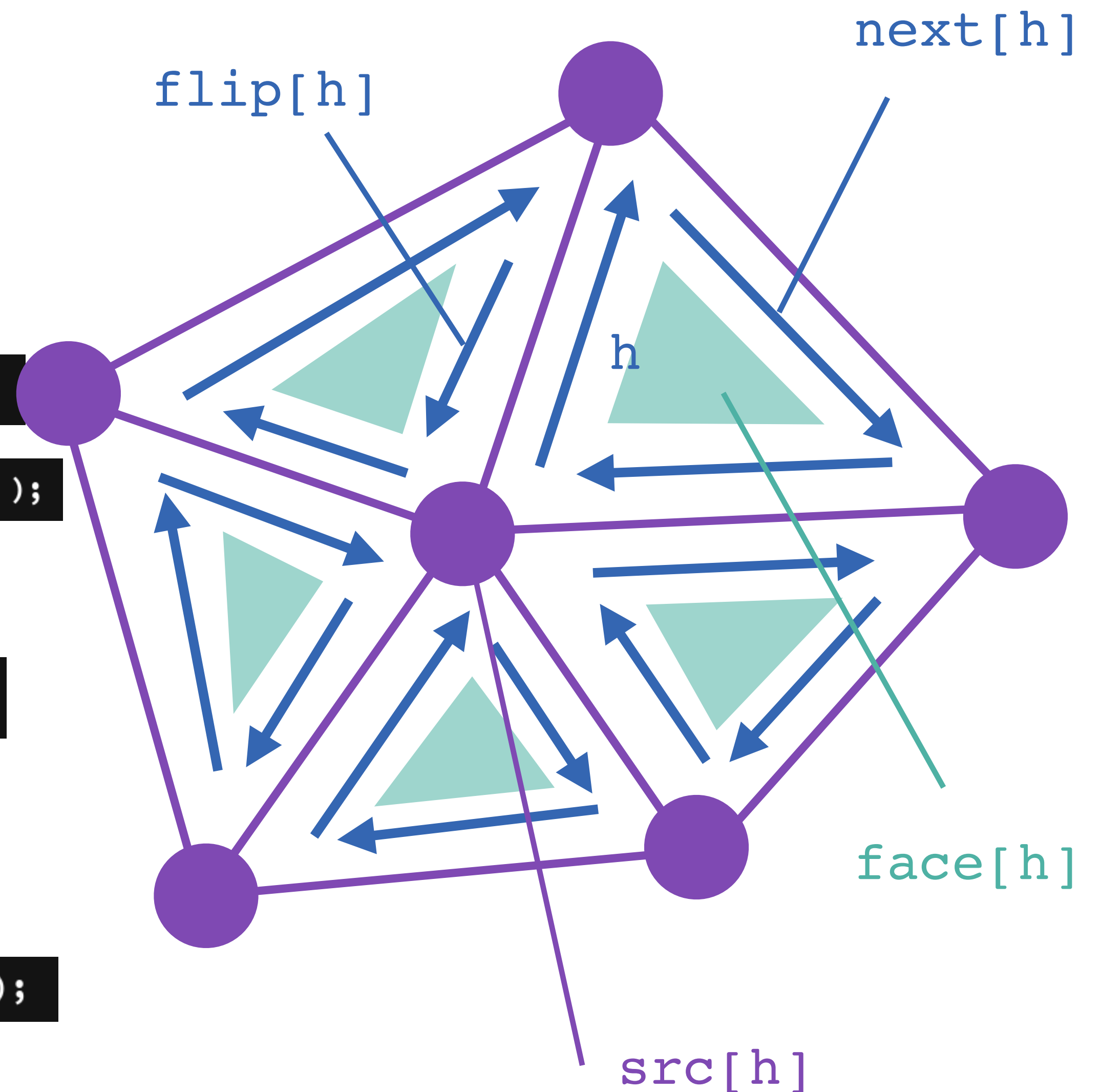
$\text{face}: \mathcal{H} \rightarrow \mathcal{F}$

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

► Face

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

```
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 >

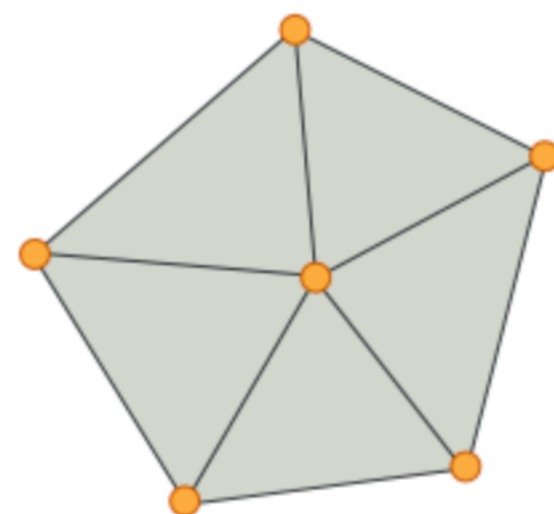
Half-edges

VEX has functions that let you treat edges as unshared per-face “half-edges”.

On this page

- [Overview](#)
- [Equivalence](#)
- [Functions](#)

Overview

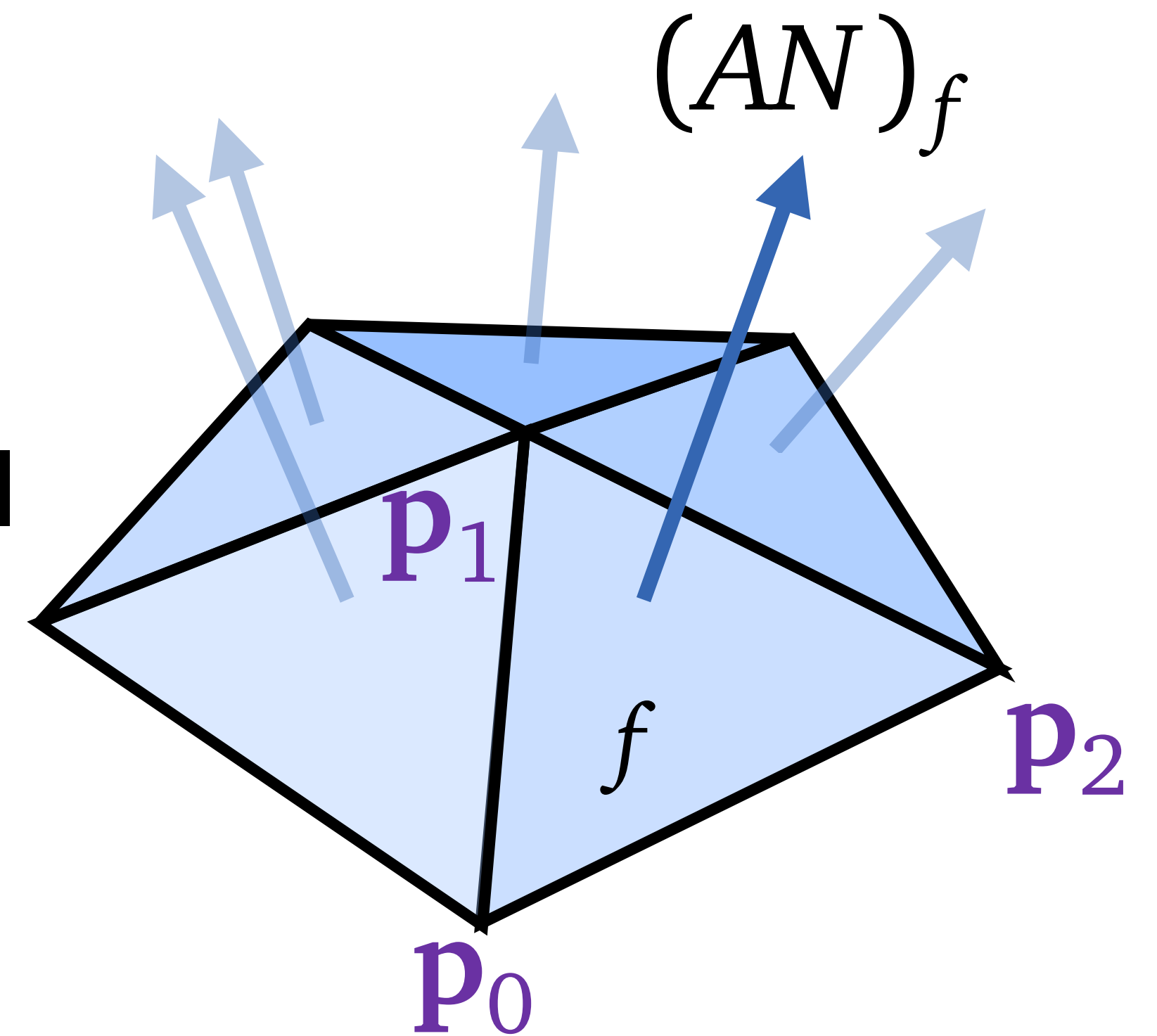


In Houdini, edges are normally treated as *undirected* and *shared* between faces. However, for some tasks (such as simplifying or cleaning geometry) it is useful to view faces as each having their own edges.

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

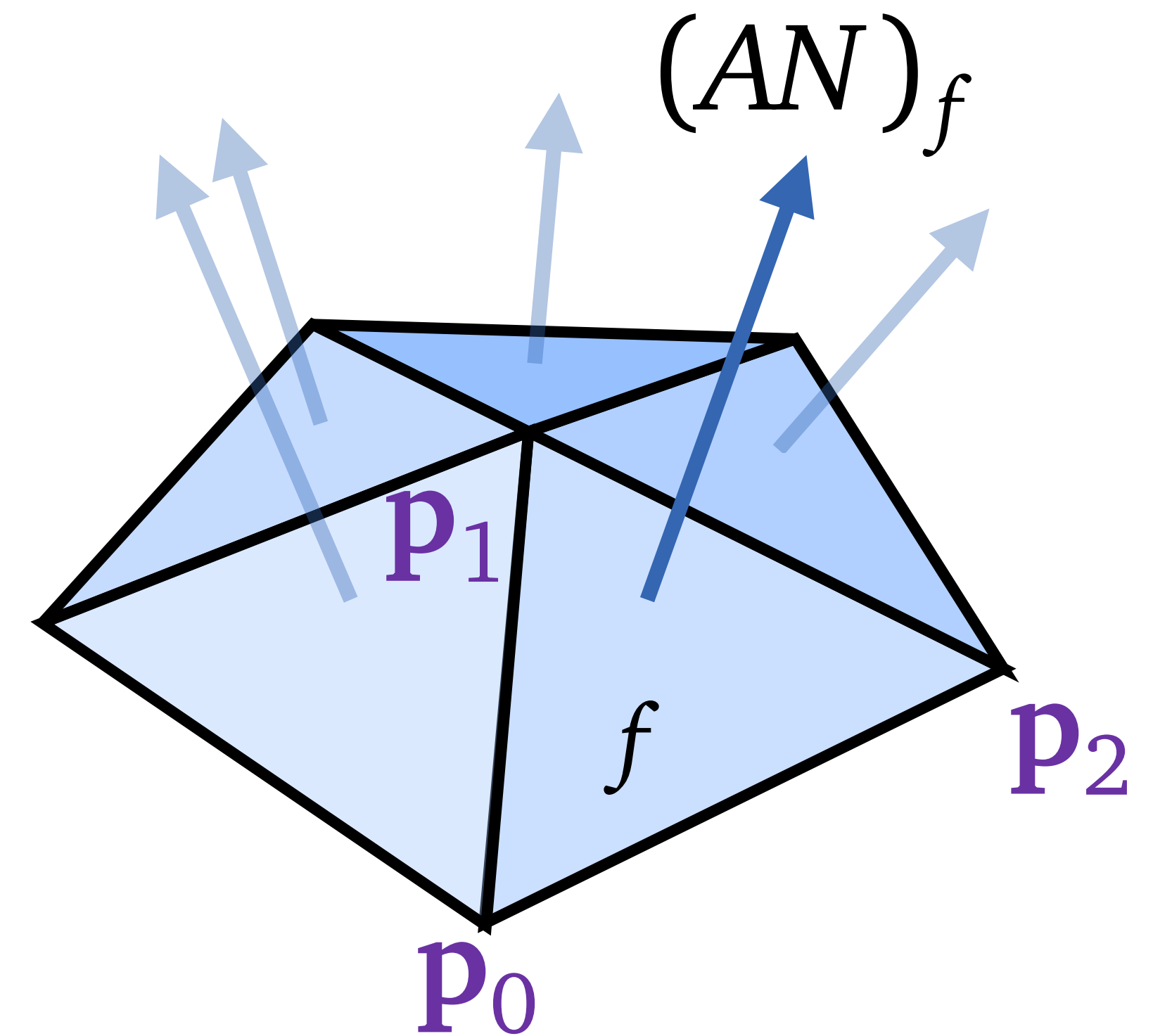
- Given a mesh, compute the face area-normal

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

- 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);
```

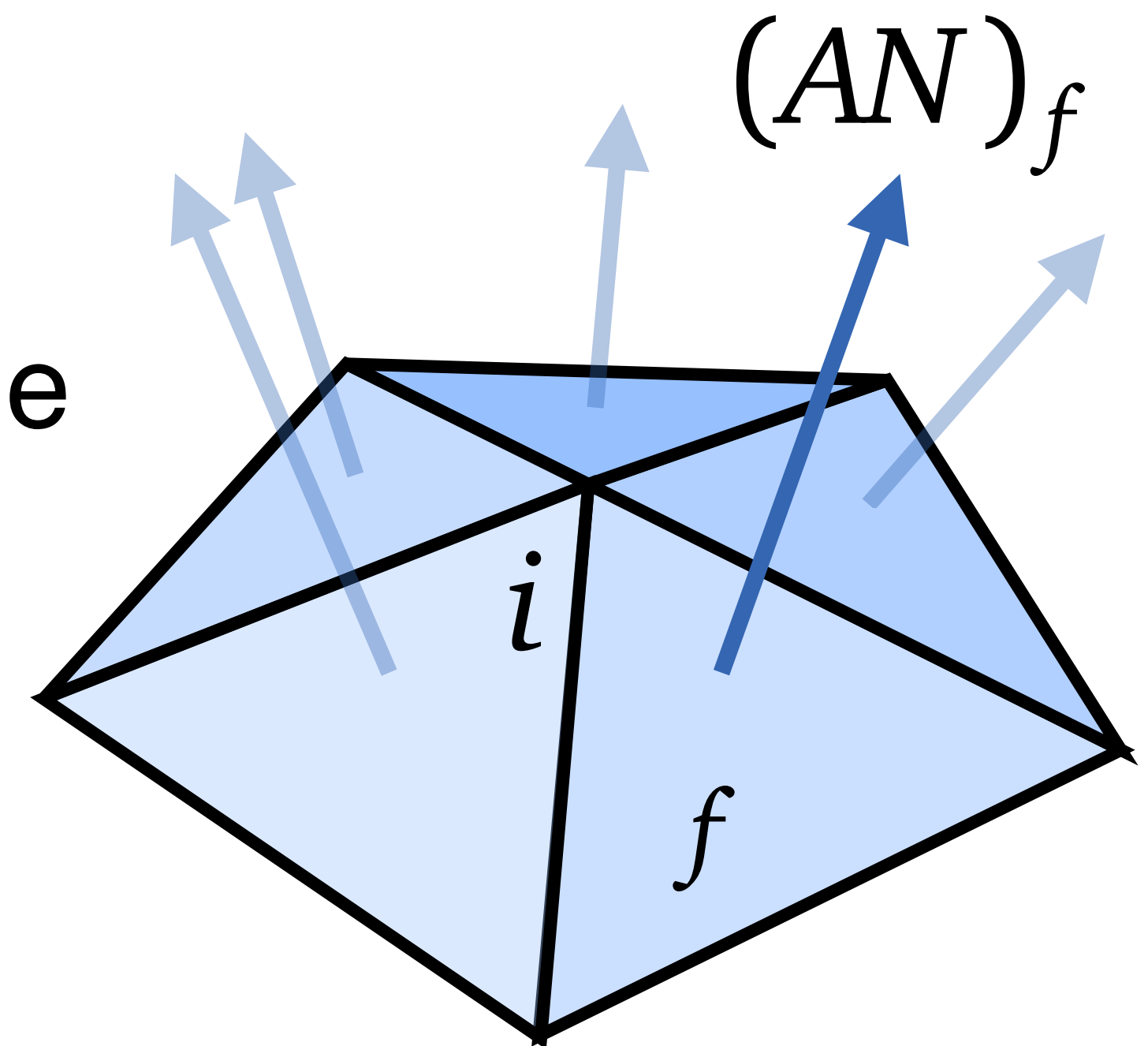
```
v@AreaNormal = 0.5*cross(P[2]-P[0], P[1]- P[0]);  
f@Area = length(v@AreaNormal);  
v@Normal = v@AreaNormal / f@Area;
```



Next example

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

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



Next example

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

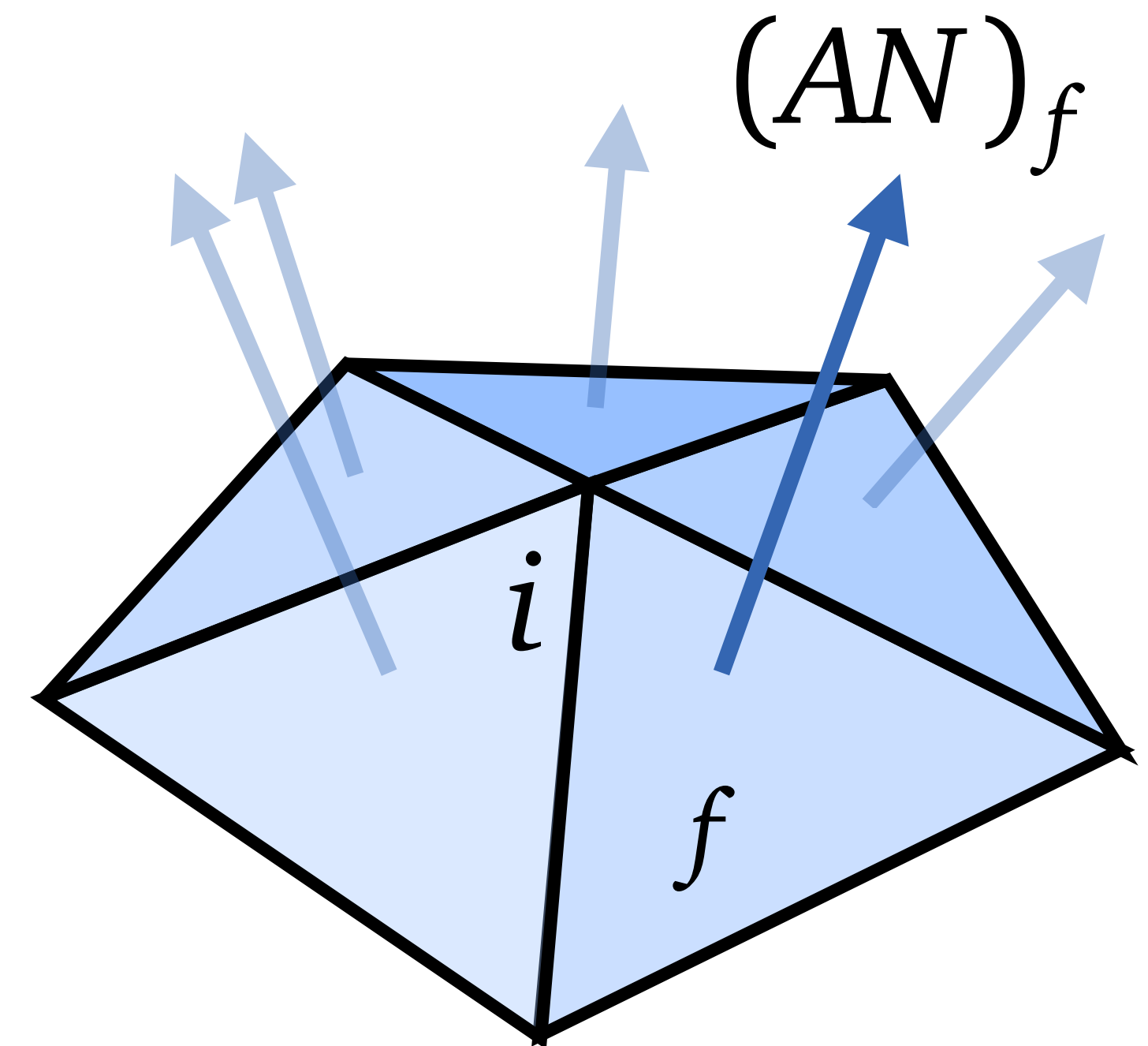
$$N_i = \text{NORMALIZE} \left(\sum_{f \succ i} A_f N_f \right)$$

- 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);
```



Data processing + numerical computing

Fortran (1957) Scientific computing

BLAS / EISPACK / LINPACK (1970s)

Hardware accelerates linear algebra

MATLAB (1979)

High-level linear algebra operators

Dominating scientific computing

LAPACK (1992)

Linear algebra
algorithms

Python (1991)

General purpose
Numeric Python

SciPy (2001)

NumPy (2005)

APL (1960s)

Think of the entire array

Database/SQL (1970-80s)

Relational table.

Think of transformation of data

Microsoft Excel (1980s)

Spreadsheet, data flow

GPU (1990s)

SIMD, graphics operation

GP-GPU (2000s)

More general purpose
parallel computing

Hedge built in MATLAB

```
1 % Script that builds a half edge data structure
2 [V,F,~,~] = readObj('./models/bunny.obj',0); % read mesh
3 [nV,~] = size(V); % number of vertices
4 [nF,~] = size(F); % number of faces
5 vert_id = (1:nV)'; % index set for vertices
6 face_id = (1:nF)'; % index set for faces
7 pp_face_rows = [F(:,1);F(:,2);F(:,3)]; % pp_ = data on half edges
8 pp_face_cols = [F(:,2);F(:,3);F(:,1)]; % represented by "point pairs"
9 pp_face_vals = [face_id; face_id; face_id];
10 pp_face = sparse(pp_face_rows,pp_face_cols,pp_face_vals,nV,nV);
11 pp_ishe = (pp_face~=0); % true if a point pair is a half edge
12 he_ind_in_pp = find(pp_ishe); % from hedge to linear index in pp matrix
13 nH = length(he_ind_in_pp); % number of half edges
14 he_id = (1:nH)'; % index set for half edges
15 pp_he = sparse(nV,nV); pp_he(he_ind_in_pp) = he_id; % he index given pp
16 pp_next_he_vals = [pp_he(sub2ind([nV,nV],F(:,2),F(:,3)))];
17                   pp_he(sub2ind([nV,nV],F(:,3),F(:,1)))];
18                   pp_he(sub2ind([nV,nV],F(:,1),F(:,2)))];
19 pp_next_he = sparse(pp_face_rows,pp_face_cols,pp_next_he_vals,nV,nV);
20 he_face = full(pp_face(he_ind_in_pp));
21 he_next = full(pp_next_he(he_ind_in_pp));
22 [he_src,he_dst] = ind2sub([nV,nV],he_ind_in_pp);
23 he_flip = full(pp_he(sub2ind([nV,nV],he_dst,he_src)));
24 he_isbdy = he_flip==0;
25 he_isint = ~he_isbdy;
26 vert_he = accumarray(he_src,he_id,[nV,1],@min);
27 face_he = accumarray(he_face,he_id,[nF,1],@min);
28 clear('pp_*')
```

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



Compute triangle area (MATLAB)

$$A = \sqrt{s(s-a)(s-b)(s-c)}. \quad s = \frac{1}{2}(a+b+c)$$

```
he_srcP = V(he_src,:);  
he_dstP = V(he_dst,:);  
he_edgevec = he_dstP - he_srcP;  
he_edgelen = sqrt( sum( he_edgevec.^2 ,2) );  
face_l1 = he_edgelen(face_he);  
face_l2 = he_edgelen(he_next(face_he));  
face_l3 = he_edgelen(he_next(he_next(face_he)));  
face_s = 0.5*(face_l1+face_l2+face_l3);  
face_area = sqrt(face_s .* (face_s-face_l1) ...  
    .* (face_s-face_l2) .* (face_s-face_l3));
```

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 $(\mathcal{H}, \text{next}, \text{flip})$ consists of
 - ▶ a finite set $\mathcal{H}$
 - ▶ an invertible map (a.k.a. a permutation) $\text{next}: \mathcal{H} \rightarrow \mathcal{H}$
 - ▶ a fixed-point-free involution $\text{flip}: \mathcal{H} \rightarrow \mathcal{H}$

- Each face is an orbit of next

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

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

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