

Triangle meshes 2

CS 4620 Lecture 3

Practical encoding of meshes

- **OBJ file format**

- widely used format for polygon meshes
- supports the usual attributes: position, normal, texture coordinate
- allows triangles or polygons (only triangles and quads widely supported)
- particularly flexible about controlling continuity of attributes
- comes with a crude mechanism for adding materials

- **Demo**

- simple file with one triangle
- effects of normals and texture coords
- exploration of continuity and discontinuity

Simple computations with meshes

- **Smoothing**

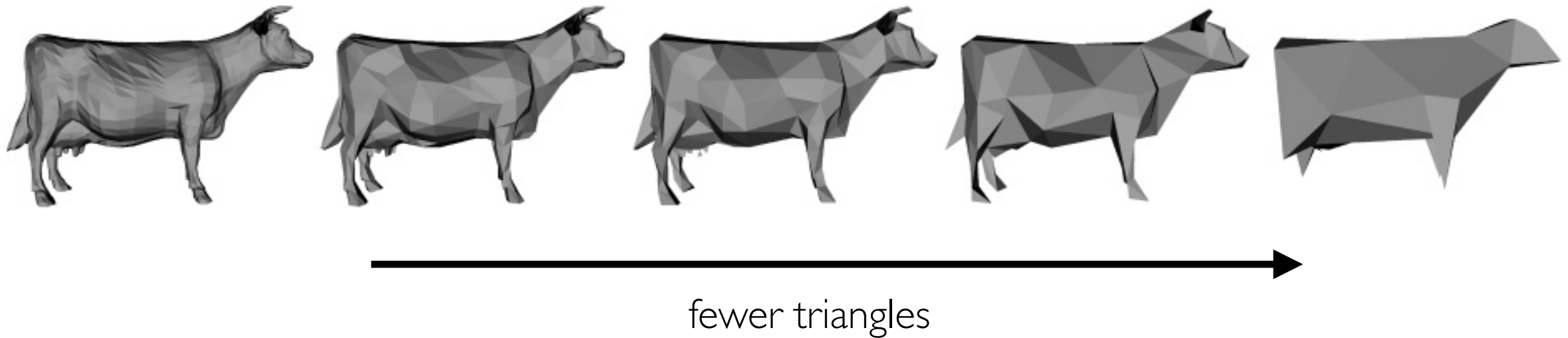
- Idea 1: move each vertex to the average of all neighboring vertices
- Idea 2: move each vertex *partway towards* the avg. of its neighbors
- there are many fancier ways to do this but with similar flavor

- **Computing normals**

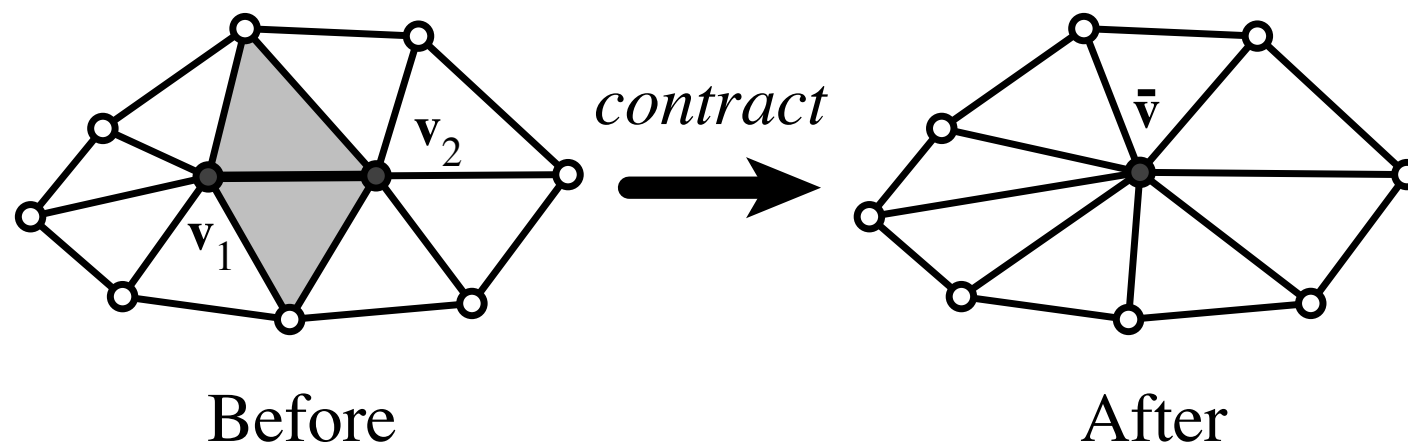
- Idea 1: faces already have normals; just use those.
- Idea 2: set normal @ each vertex to the average of the neighboring triangles' normals
- Idea 3: ...to a *weighted* avg. of the neighboring triangles' normals
 - weight by area
 - weight by angle

Ops. that change mesh topology

- **Mesh simplification**



– popular approach based on edge-collapse operations:



[Garland & Heckbert SIGGRAPH 97]

Queries on meshes

- **For face, find all:**

- vertices
- edges
- neighboring faces

- **For vertex, find all:** ←

- incident edges
- incident triangles
- neighboring vertices

useful for smoothing/normal operations,
if you want to compute them one vertex
at a time (all at once is easier!)

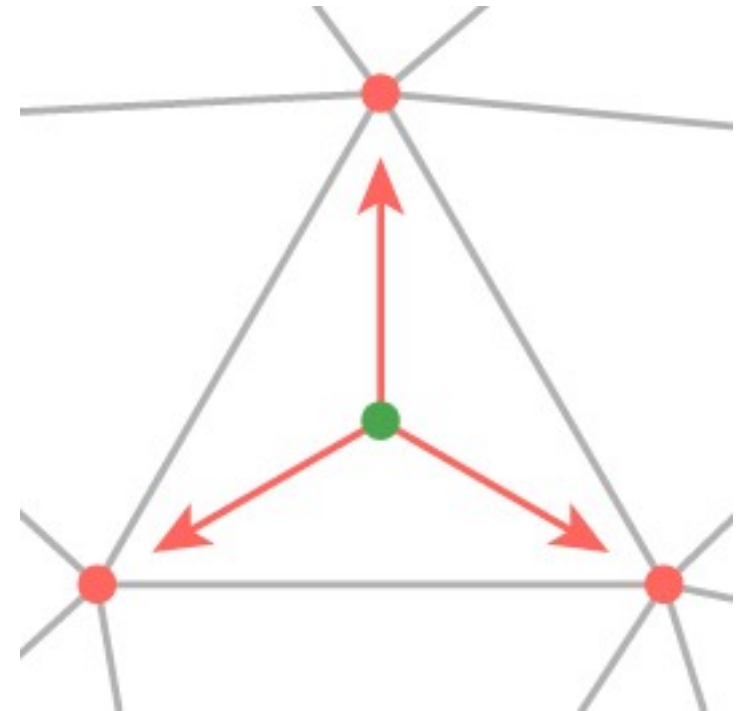
- **For edge, find:**

- two adjacent faces
- two adjacent vertices

most of these ops. required to implement
edge-collapse-based simplification

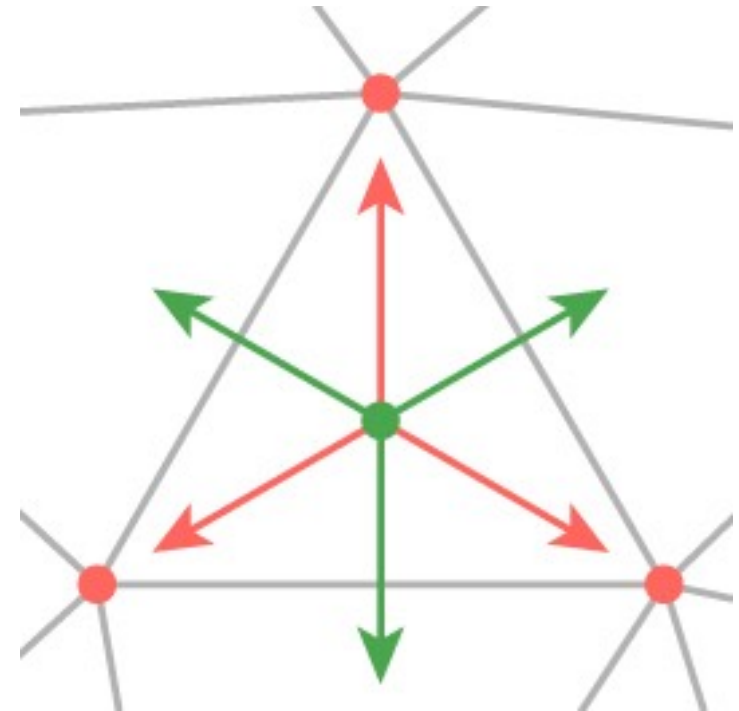
Triangle neighbor structure

- **Extension to indexed triangle set**
- **Triangle points to its three neighboring triangles**
- **Vertex points to a single neighboring triangle**
- **Can now enumerate triangles around a vertex**



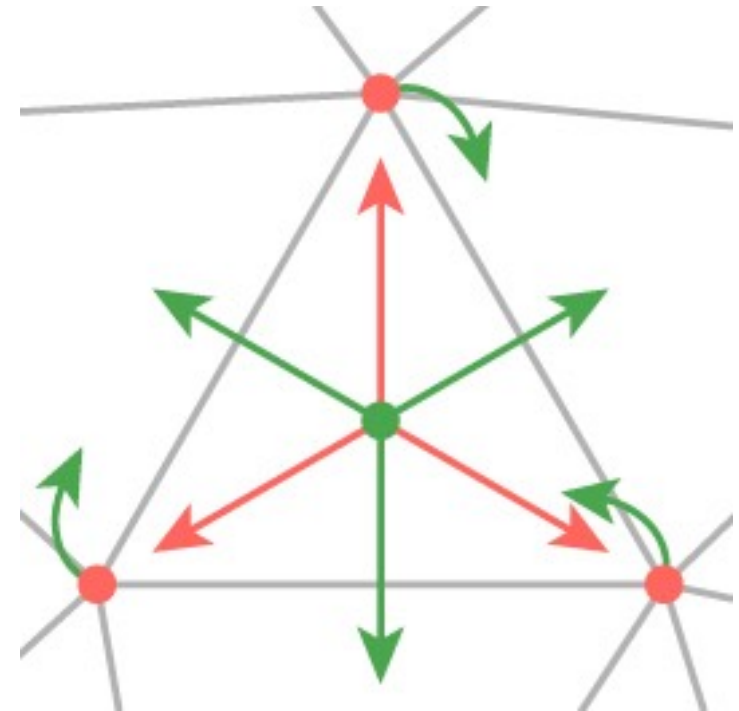
Triangle neighbor structure

- **Extension to indexed triangle set**
- **Triangle points to its three neighboring triangles**
- **Vertex points to a single neighboring triangle**
- **Can now enumerate triangles around a vertex**



Triangle neighbor structure

- **Extension to indexed triangle set**
- **Triangle points to its three neighboring triangles**
- **Vertex points to a single neighboring triangle**
- **Can now enumerate triangles around a vertex**



Triangle neighbor structure

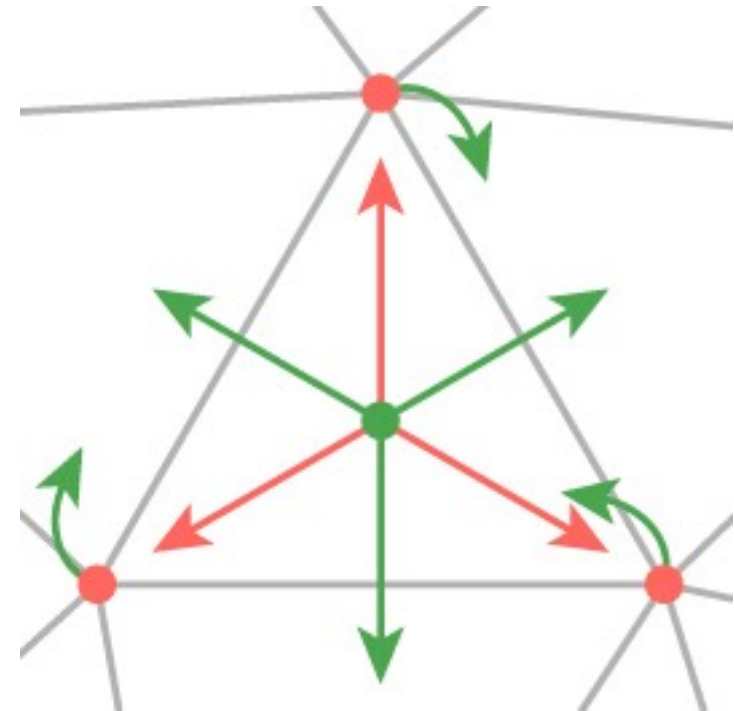
```
Triangle {  
    Triangle nbr[3];  
    Vertex vertex[3];  
}
```

```
// t.neighbor[i] is adjacent  
// across the edge from i to i+1
```

```
Vertex {  
    // ... per-vertex data ...  
    Triangle t; // any adjacent tri  
}
```

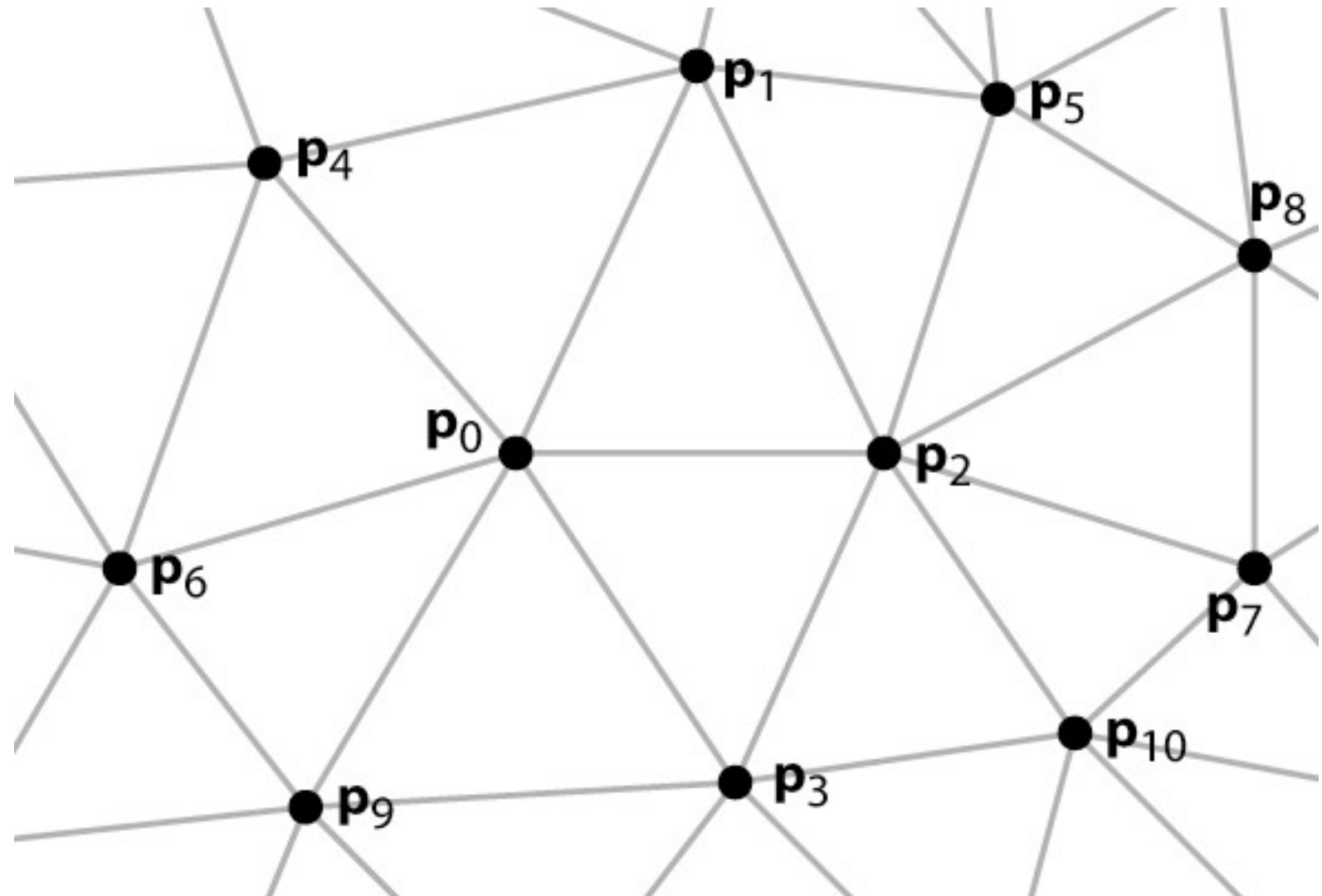
```
// ... or ...
```

```
Mesh {  
    // ... per-vertex data ...  
    int tInd[nt][3]; // vertex indices  
    int tNbr[nt][3]; // indices of neighbor triangles  
    int vTri[nv]; // index of any adjacent triangle  
}
```



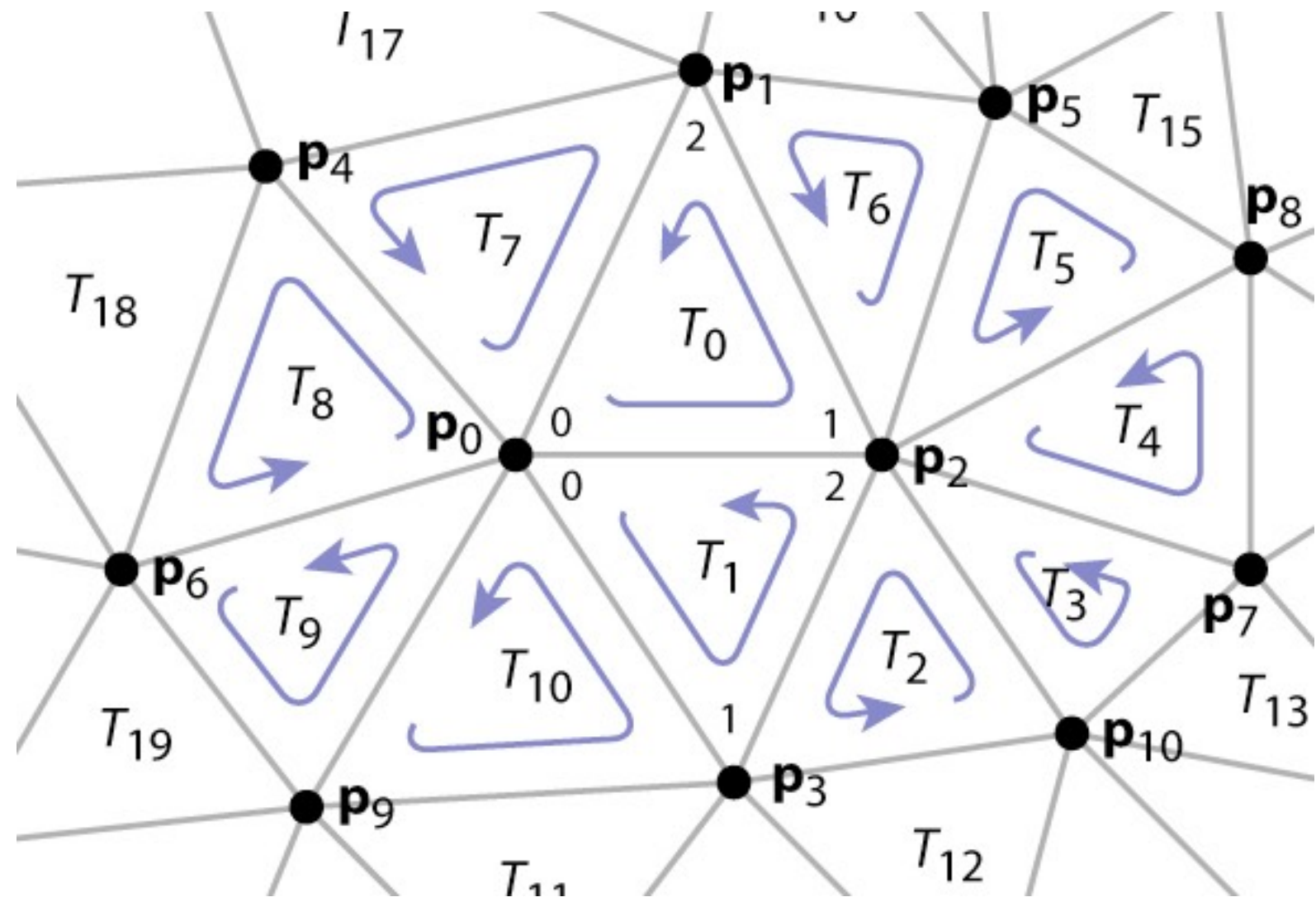
Triangle neighbor structure

vTri[0]	0	tNbr[0]	1, 6, 7
vTri[1]	6	tNbr[1]	10, 2, 0
vTri[2]	1	tNbr[2]	3, 1, 12
vTri[3]	1	tNbr[3]	2, 13, 4
	⋮		⋮
		tInd[0]	0, 2, 1
		tInd[1]	0, 3, 2
		tInd[2]	10, 2, 3
		tInd[3]	2, 10, 7
			⋮



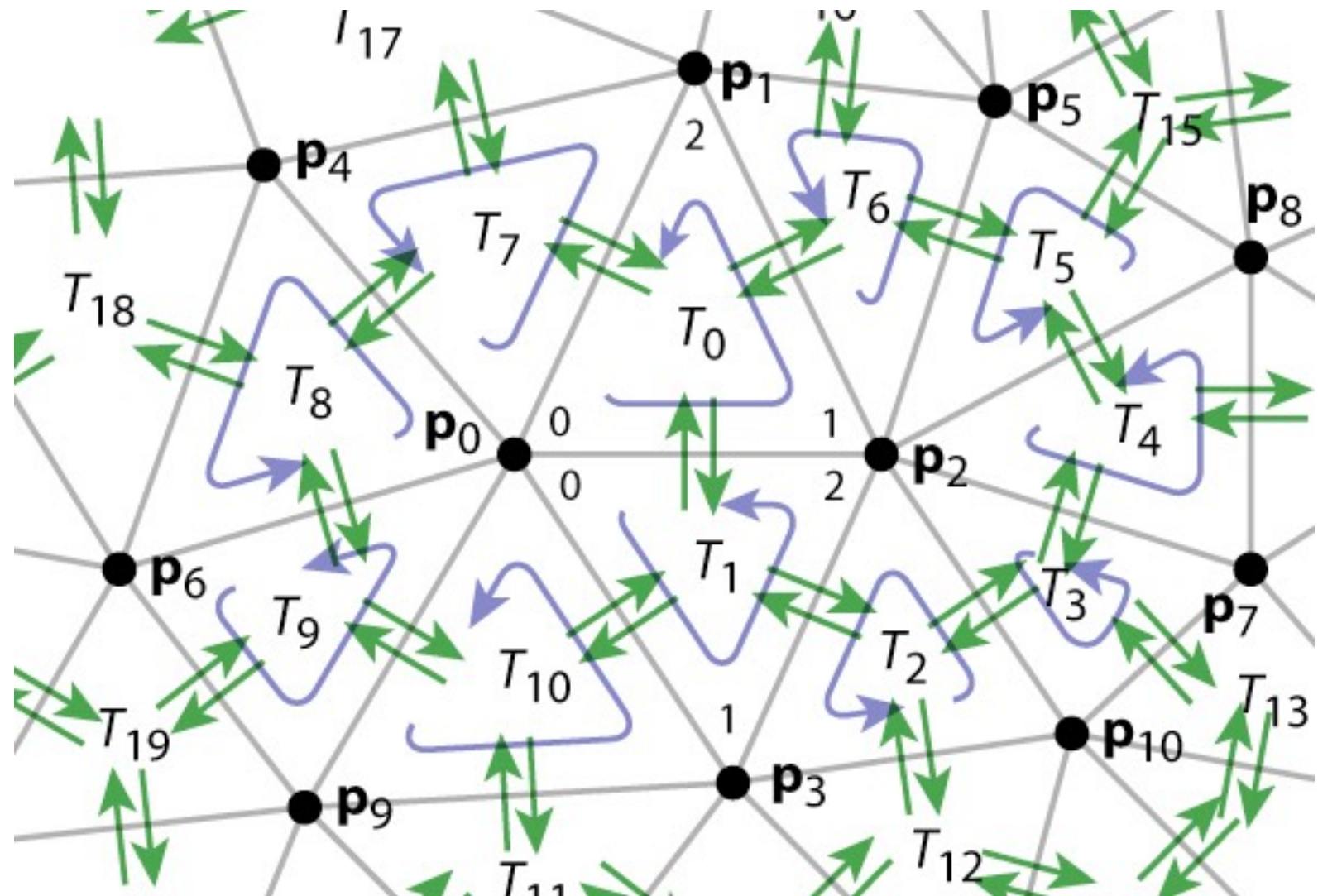
Triangle neighbor structure

vTri[0]	0	tNbr[0]	1, 6, 7
vTri[1]	6	tNbr[1]	10, 2, 0
vTri[2]	1	tNbr[2]	3, 1, 12
vTri[3]	1	tNbr[3]	2, 13, 4
	⋮		⋮
		tInd[0]	0, 2, 1
		tInd[1]	0, 3, 2
		tInd[2]	10, 2, 3
		tInd[3]	2, 10, 7
			⋮



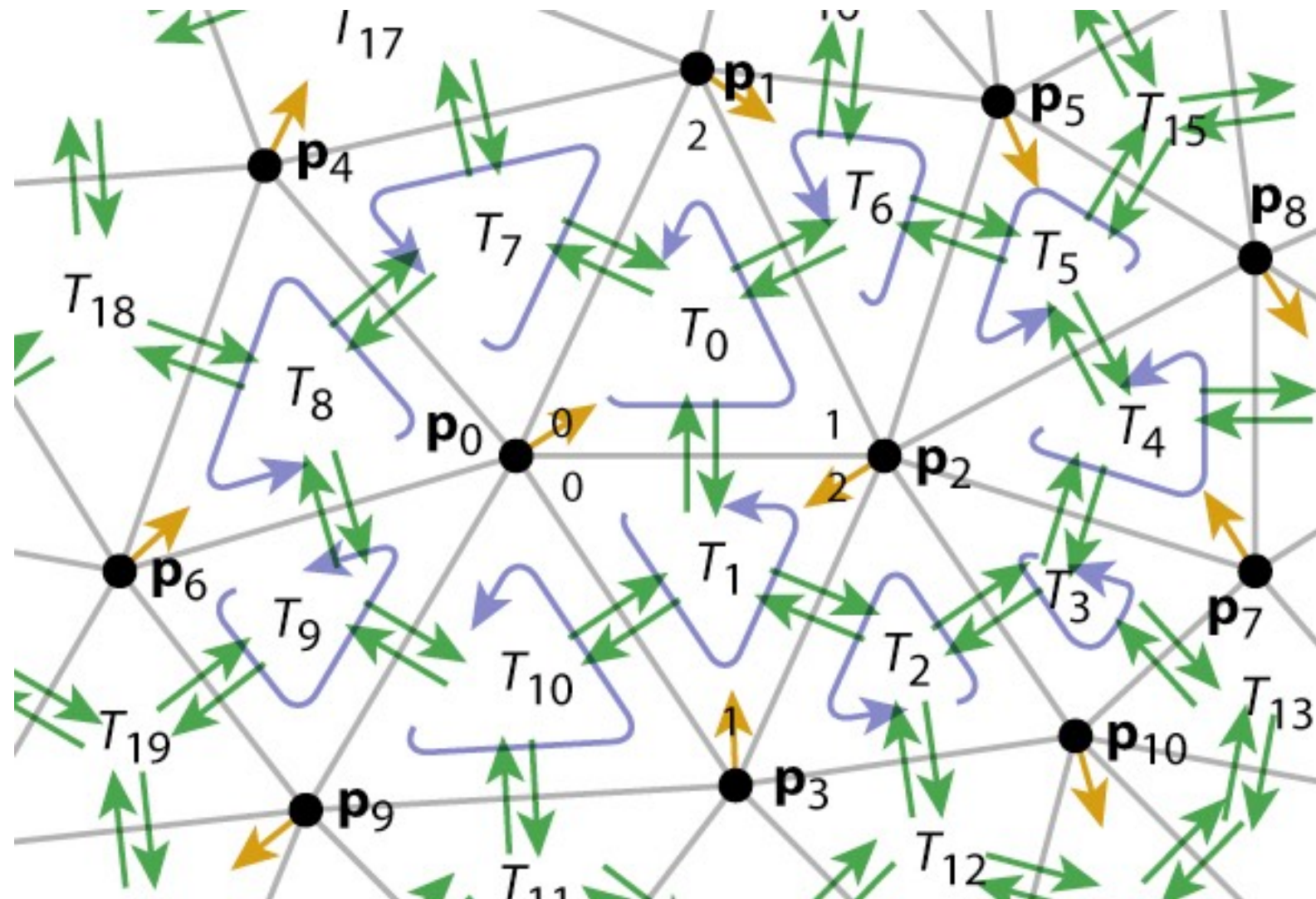
Triangle neighbor structure

vTri[0]	0	tNbr[0]	1, 6, 7
vTri[1]	6	tNbr[1]	10, 2, 0
vTri[2]	1	tNbr[2]	3, 1, 12
vTri[3]	1	tNbr[3]	2, 13, 4
	⋮		⋮
		tInd[0]	0, 2, 1
		tInd[1]	0, 3, 2
		tInd[2]	10, 2, 3
		tInd[3]	2, 10, 7
			⋮



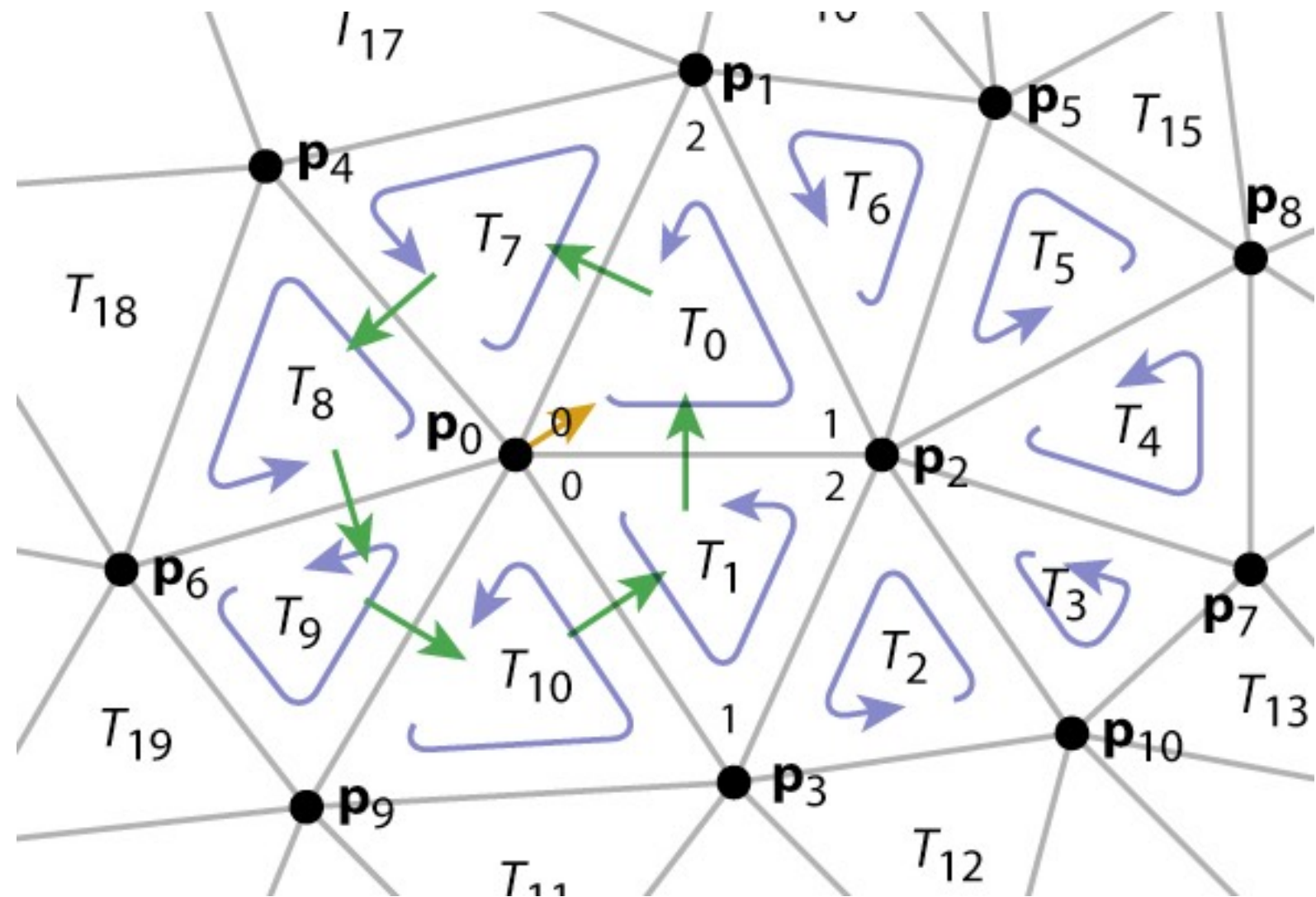
Triangle neighbor structure

$vTri[0]$	0	$tNbr[0]$	1, 6, 7	$tInd[0]$	0, 2, 1
$vTri[1]$	6	$tNbr[1]$	10, 2, 0	$tInd[1]$	0, 3, 2
$vTri[2]$	1	$tNbr[2]$	3, 1, 12	$tInd[2]$	10, 2, 3
$vTri[3]$	1	$tNbr[3]$	2, 13, 4	$tInd[3]$	2, 10, 7
	\vdots		\vdots		\vdots



Triangle neighbor structure

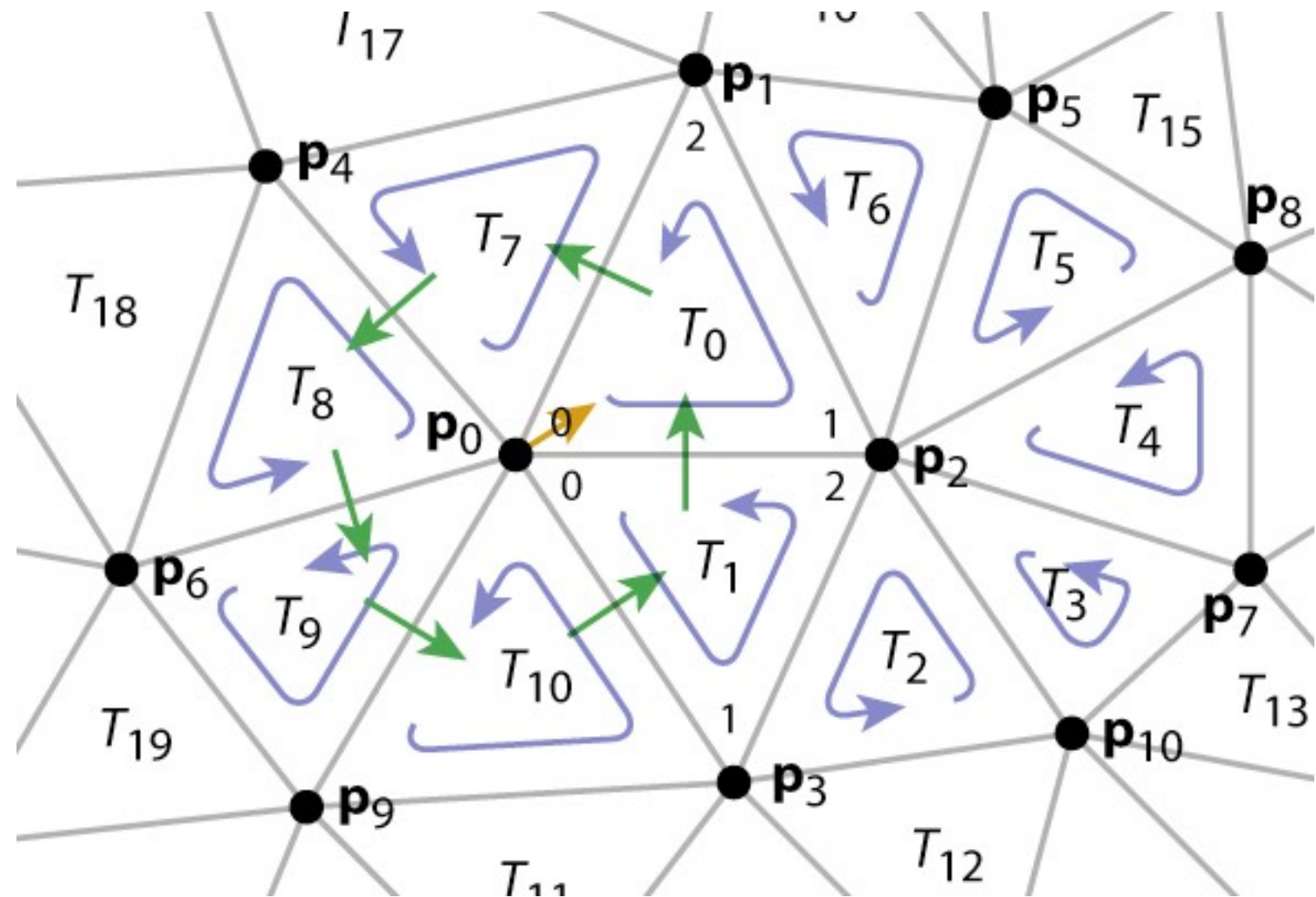
vTri[0]	0	tNbr[0]	1, 6, 7
vTri[1]	6	tNbr[1]	10, 2, 0
vTri[2]	1	tNbr[2]	3, 1, 12
vTri[3]	1	tNbr[3]	2, 13, 4
	⋮		⋮
		tInd[0]	0, 2, 1
		tInd[1]	0, 3, 2
		tInd[2]	10, 2, 3
		tInd[3]	2, 10, 7
			⋮



Triangle neighbor structure

```
TrianglesOfVertex(v) {
  t = v.t;
  do {
    find t.vertex[i] == v;
    t = t.nbr[pred(i)];
  } while (t != v.t);
}
```

```
pred(i) = (i+2) % 3;
succ(i) = (i+1) % 3;
```



Triangle neighbor structure

- **indexed mesh was 36 bytes per vertex**
- **add an array of triples of indices (per triangle)**
 - $\text{int}[n_T][3]$: about 24 bytes per vertex
 - 2 triangles per vertex (on average)
 - (3 indices x 4 bytes) per triangle
- **add an array of representative triangle per vertex**
 - $\text{int}[n_V]$: 4 bytes per vertex
- **total storage: 64 bytes per vertex**
 - still not as much as separate triangles

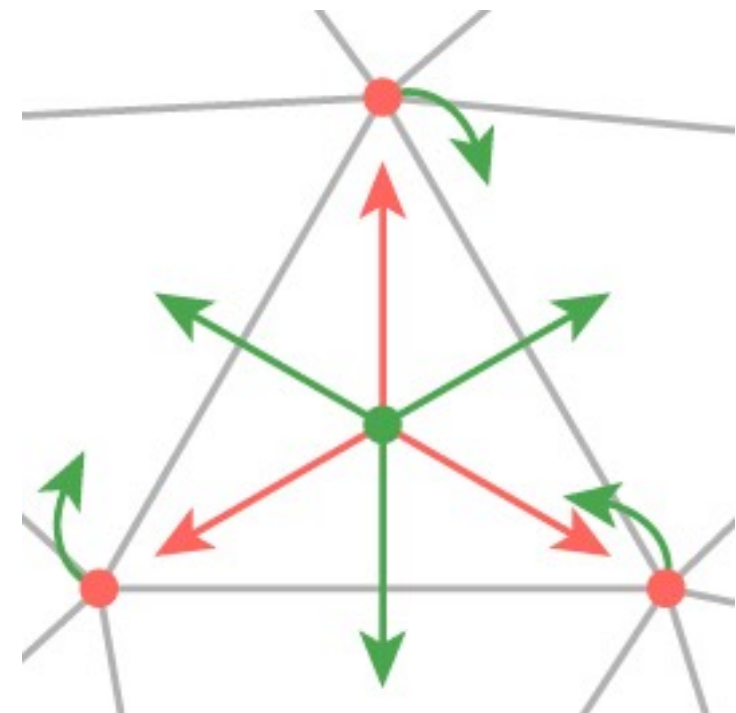
Triangle neighbor structure—refined

```
Triangle {  
    Edge nbr[3];  
    Vertex vertex[3];  
}
```

```
// if t.nbr[i].i == j  
// then t.nbr[i].t.nbr[j].t == t
```

```
Edge {  
    // the i-th edge of triangle t  
    Triangle t;  
    int i; // in {0,1,2}  
    // in practice t and i share 32 bits  
}
```

```
Vertex {  
    // ... per-vertex data ...  
    Edge e; // any edge leaving vertex  
}
```



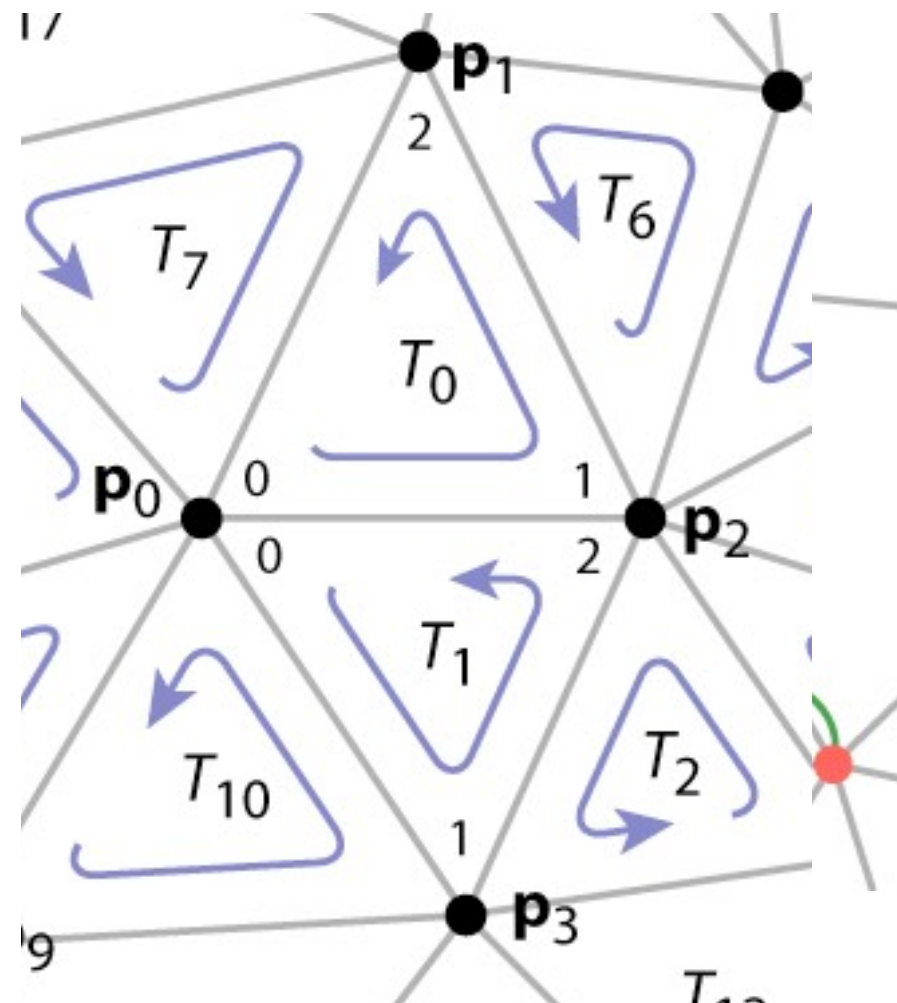
$T_0.\text{nbr}[0] = \{ T_1, 2 \}$

$T_1.\text{nbr}[2] = \{ T_0, 0 \}$

$V_0.e = \{ T_1, 0 \}$

Triangle neighbor structure—refined

```
Triangle {  
    Edge nbr[3];  
    Vertex vertex[3];  
}  
  
// if t.nbr[i].i == j  
// then t.nbr[i].t.nbr[j].t == t  
  
Edge {  
    // the i-th edge of triangle t  
    Triangle t;  
    int i; // in {0,1,2}  
    // in practice t and i share 32 bits  
}  
  
Vertex {  
    // ... per-vertex data ...  
    Edge e; // any edge leaving vertex  
}
```

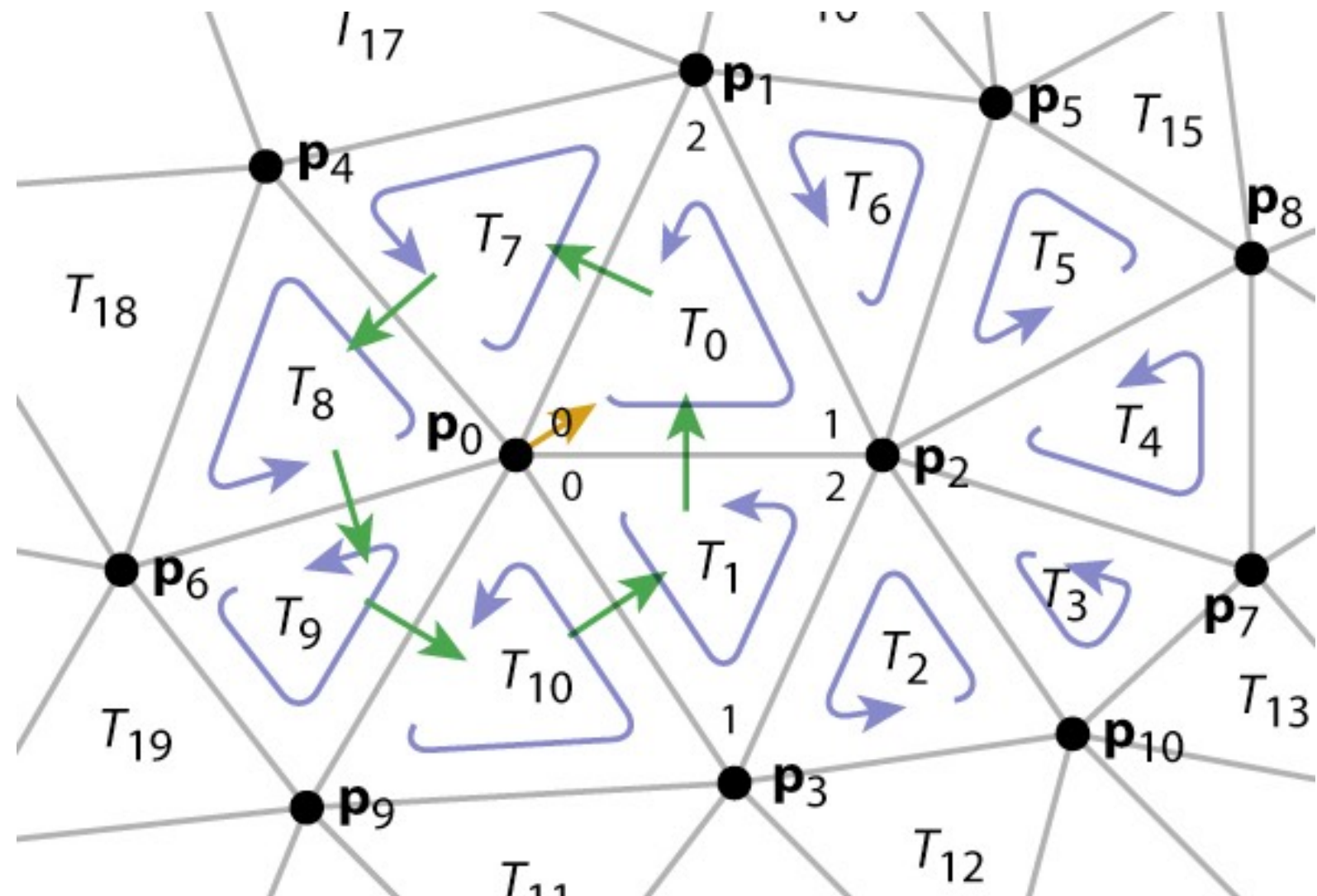


$T_0.\text{nbr}[0] = \{ T_1, 2 \}$
 $T_1.\text{nbr}[2] = \{ T_0, 0 \}$
 $V_0.e = \{ T_1, 0 \}$

Triangle neighbor structure

```
TrianglesOfVertex(v) {
  {t, i} = v.e;
  do {
    {t, i} = t.nbr[pred(i)];
  } while (t != v.t);
}
```

```
pred(i) = (i+2) % 3;
succ(i) = (i+1) % 3;
```



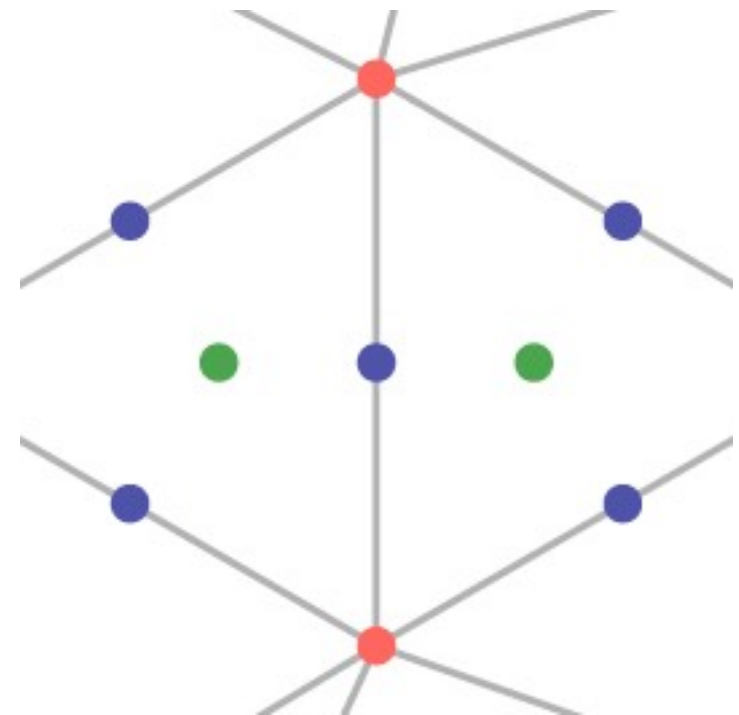
$T_0.nbr[0] = \{ T_1, 2 \}$

$T_1.nbr[2] = \{ T_0, 0 \}$

$V_0.e = \{ T_1, 0 \}$

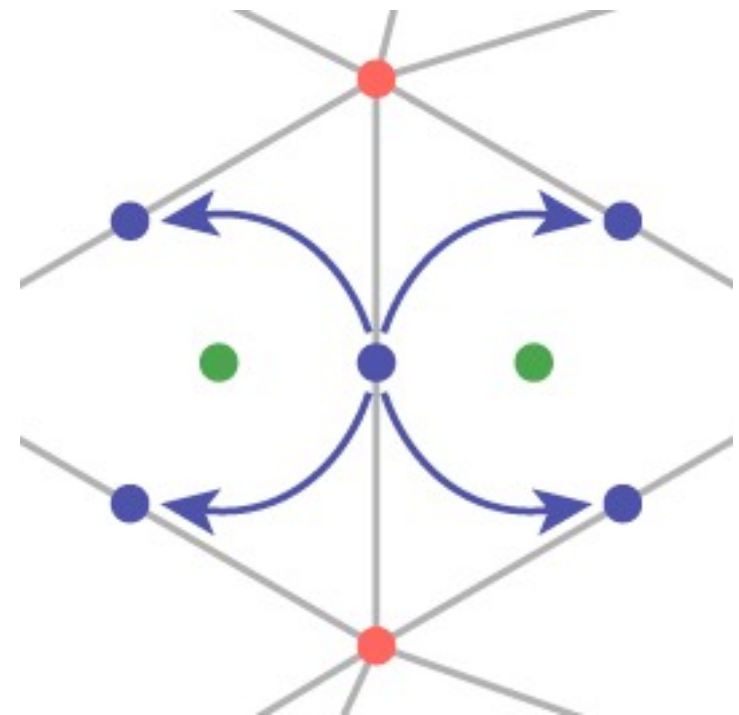
Winged-edge mesh

- **Edge-centric rather than face-centric**
 - therefore also works for polygon meshes
- **Each (oriented) edge points to:**
 - left and right forward edges
 - left and right backward edges
 - front and back vertices
 - left and right faces
- **Each face or vertex points to one edge**



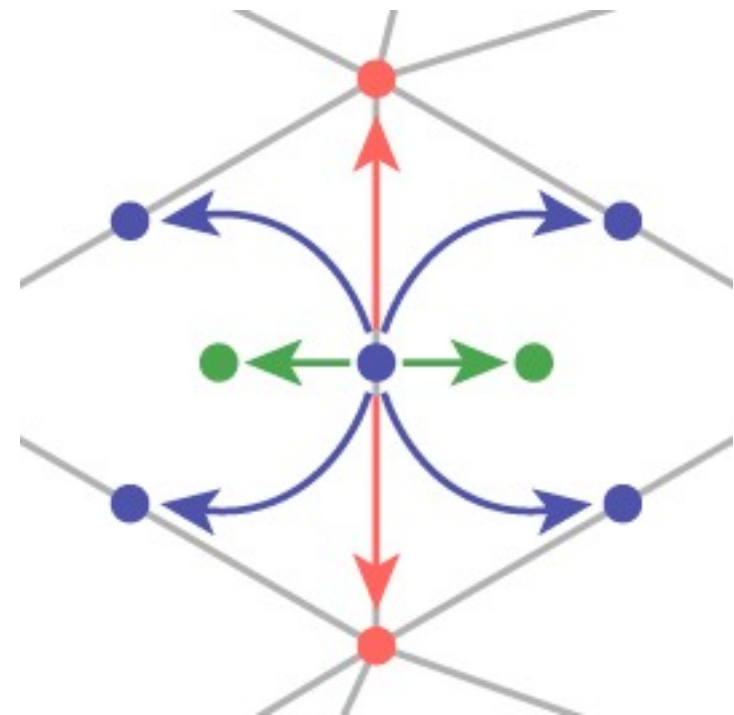
Winged-edge mesh

- **Edge-centric rather than face-centric**
 - therefore also works for polygon meshes
- **Each (oriented) edge points to:**
 - left and right forward edges
 - left and right backward edges
 - front and back vertices
 - left and right faces
- **Each face or vertex points to one edge**



Winged-edge mesh

- **Edge-centric rather than face-centric**
 - therefore also works for polygon meshes
- **Each (oriented) edge points to:**
 - left and right forward edges
 - left and right backward edges
 - front and back vertices
 - left and right faces
- **Each face or vertex points to one edge**

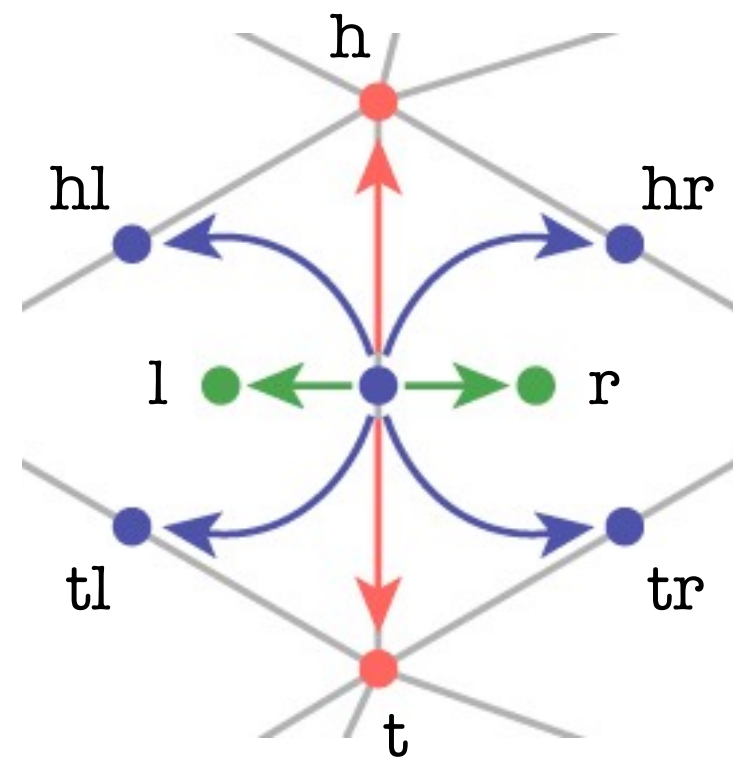


Winged-edge mesh

```
Edge {  
  Edge hl, hr, tl, tr;  
  Vertex h, t;  
  Face l, r;  
}
```

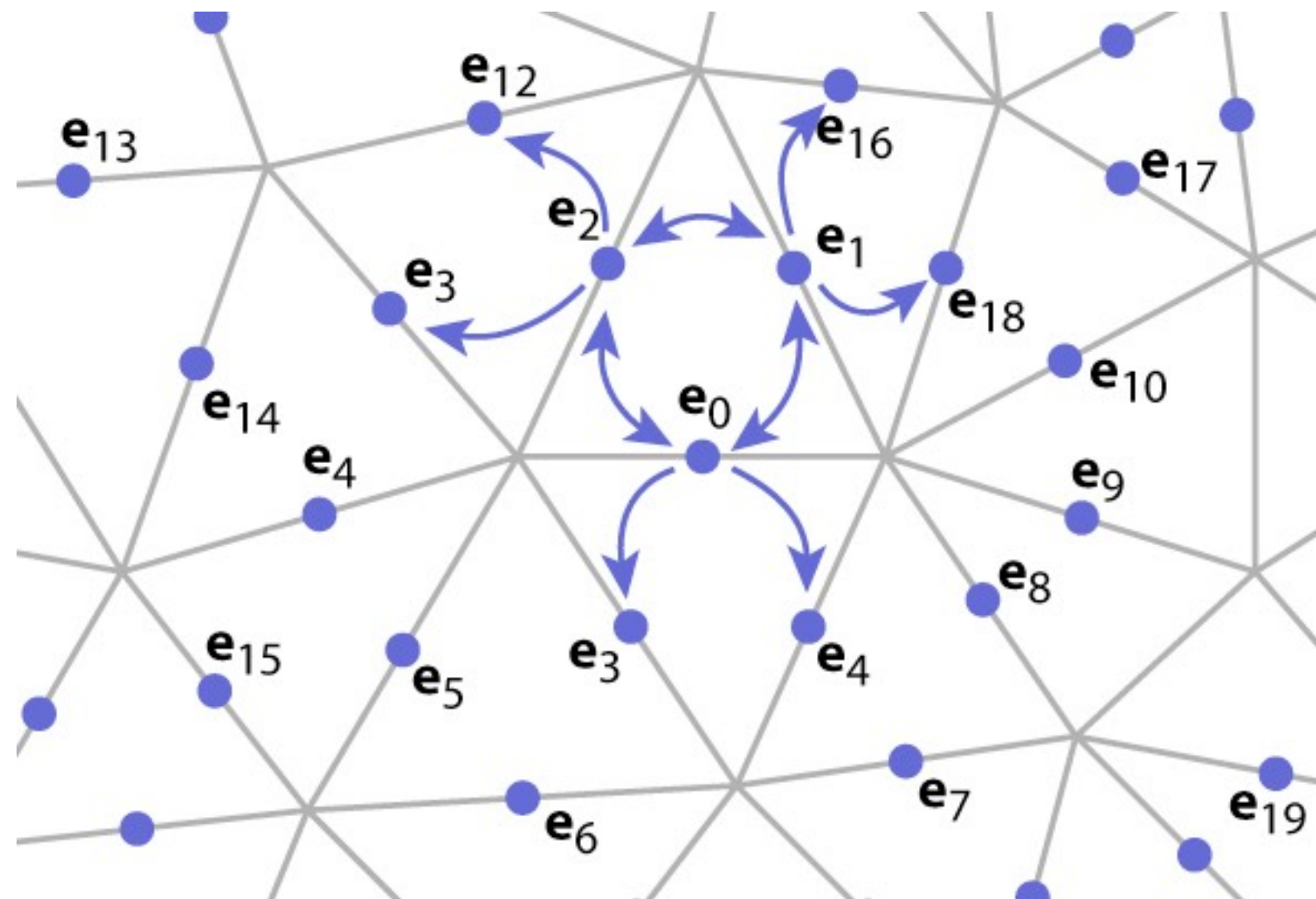
```
Face {  
  // per-face data  
  Edge e; // any adjacent edge  
}
```

```
Vertex {  
  // per-vertex data  
  Edge e; // any incident edge  
}
```



Winged-edge structure

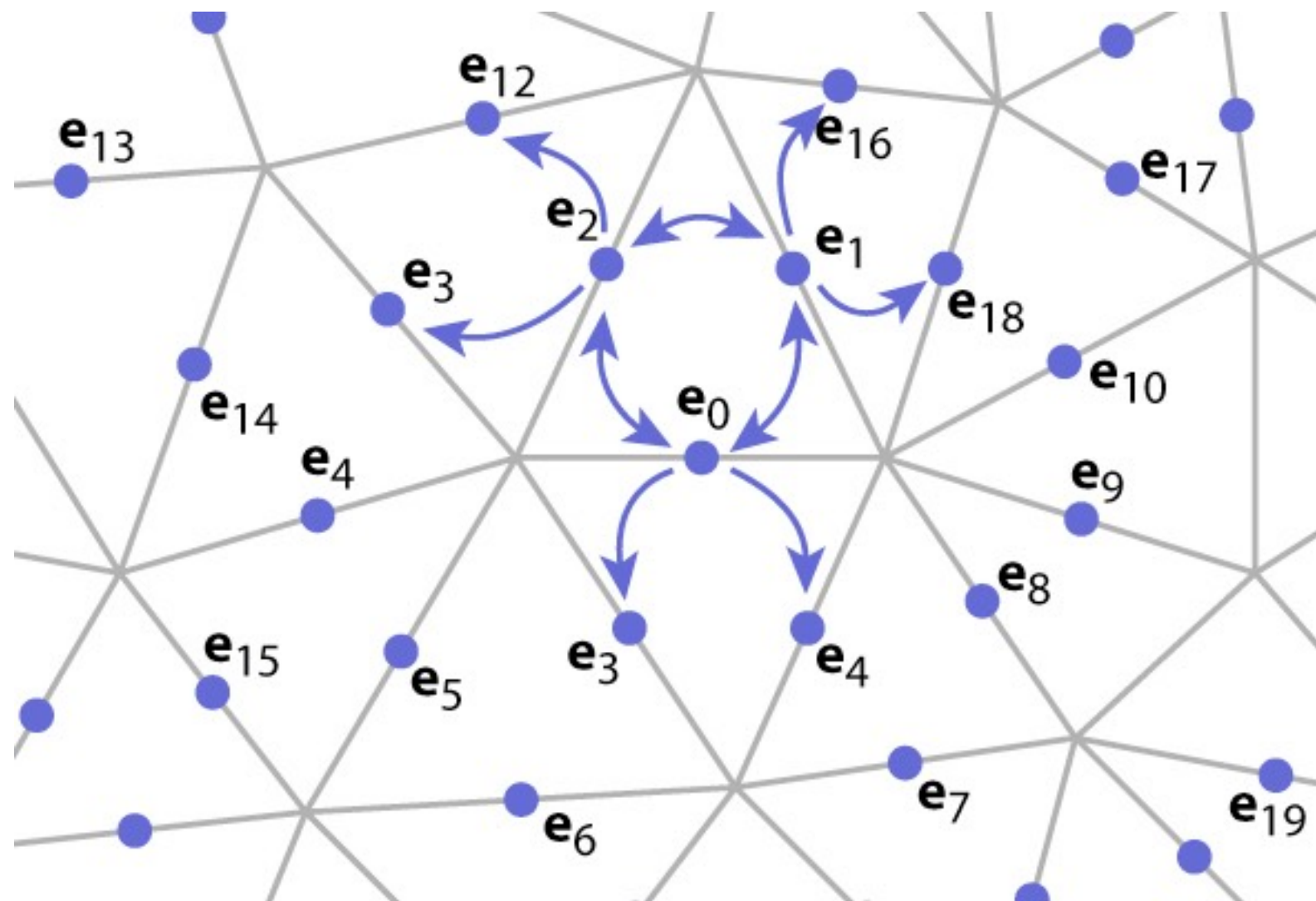
	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
	⋮			



Winged-edge structure

```
EdgesOfFace(f) {
  e = f.e;
  do {
    if (e.l == f)
      e = e.hl;
    else
      e = e.tr;
  } while (e != f.e);
}
```

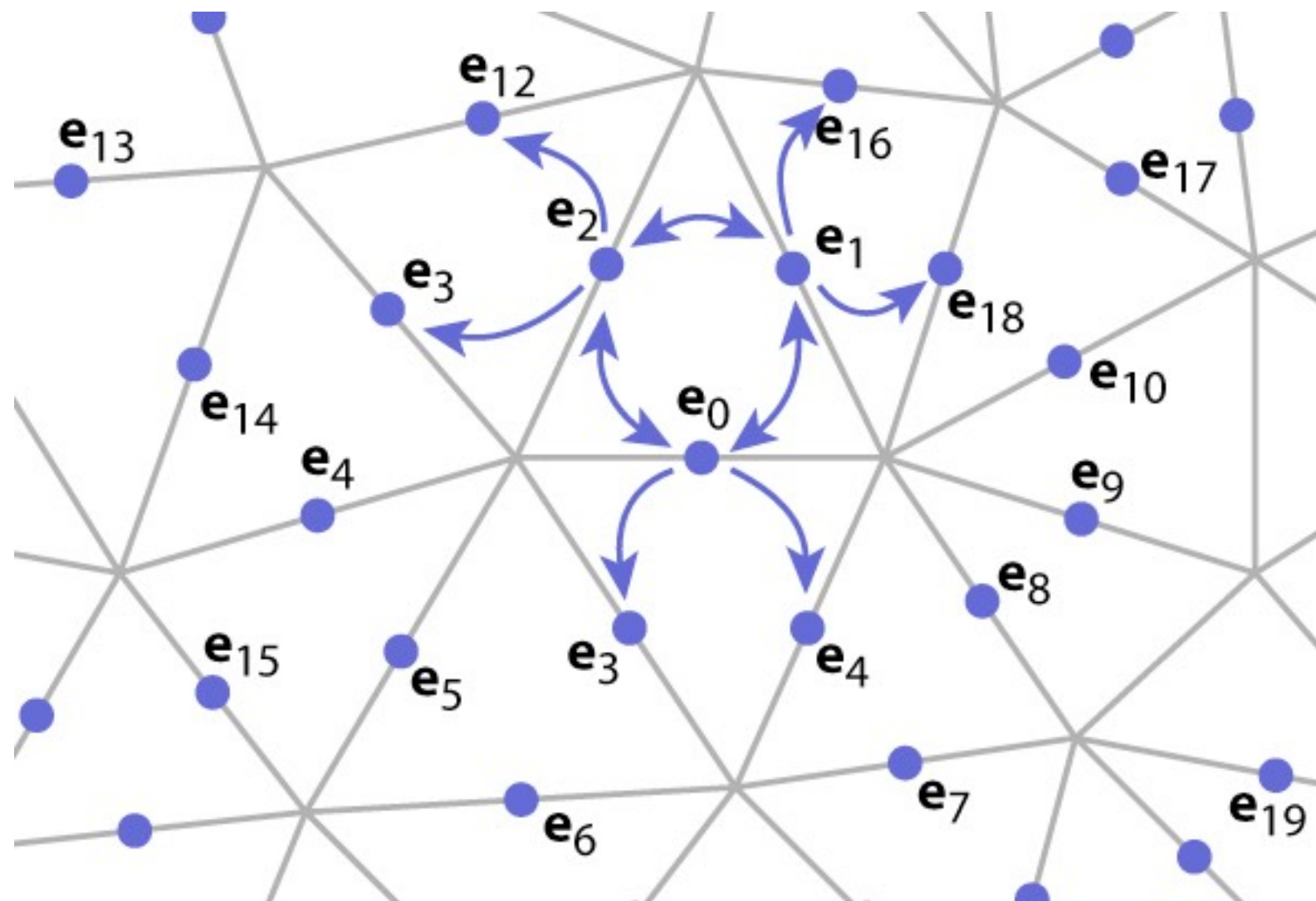
	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
	⋮			



Winged-edge structure

```
EdgesOfVertex(v) {
  e = v.e;
  do {
    if (e.t == v)
      e = e.tl;
    else
      e = e.hr;
  } while (e != v.e);
}
```

	hl	hr	tl	tr
edge[0]	1	4	2	3
edge[1]	18	0	16	2
edge[2]	12	1	3	0
	⋮			

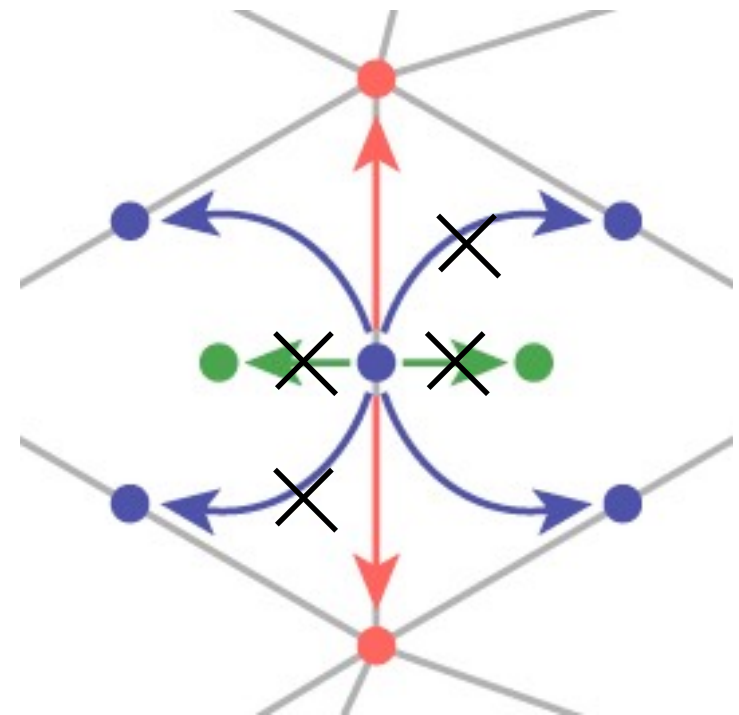


Winged-edge structure

- **array of vertex positions: 12 bytes/vert**
- **array of 8-tuples of indices (per edge)**
 - head/tail left/right edges + head/tail verts + left/right tris
 - $\text{int}[n_E][8]$: about 96 bytes per vertex
 - 3 edges per vertex (on average)
 - (8 indices x 4 bytes) per edge
- **add a representative edge per vertex**
 - $\text{int}[n_V]$: 4 bytes per vertex
- **total storage: 112 bytes per vertex**
 - but it is cleaner and generalizes to polygon meshes

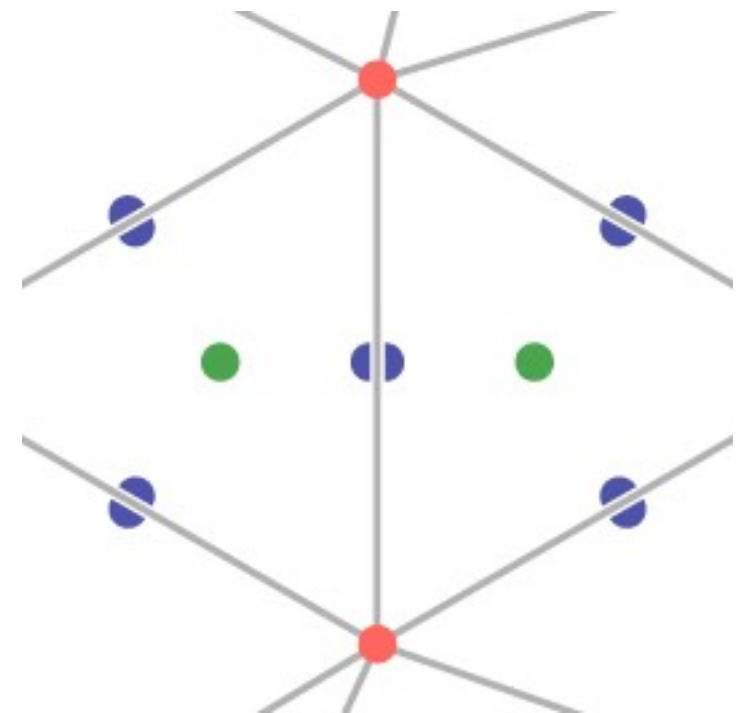
Winged-edge optimizations

- **Omit faces if not needed**
- **Omit one edge pointer on each side**
 - results in one-way traversal



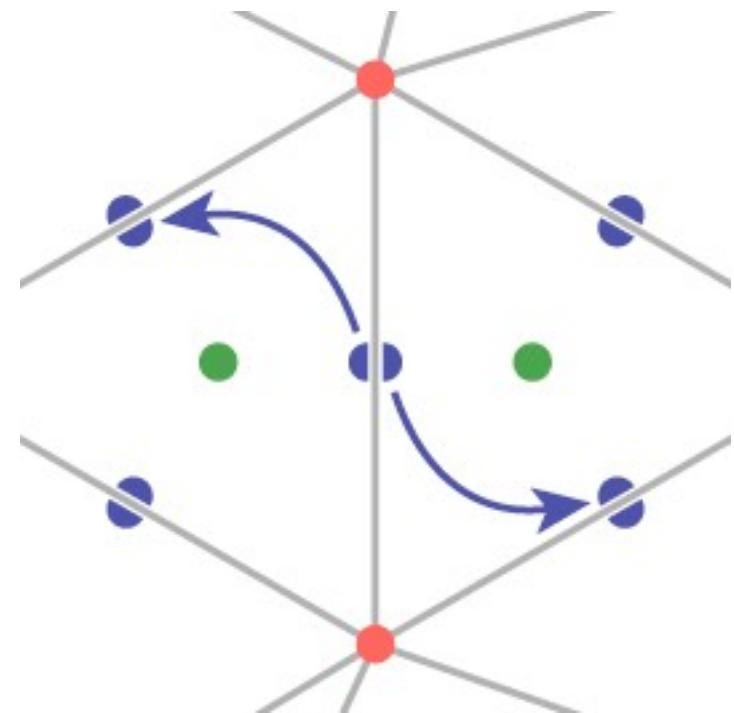
Half-edge structure

- **Simplifies, cleans up winged edge**
 - still works for polygon meshes
- **Each half-edge points to:**
 - next edge (left forward)
 - next vertex (front)
 - the face (left)
 - the opposite half-edge
- **Each face or vertex points to one half-edge**



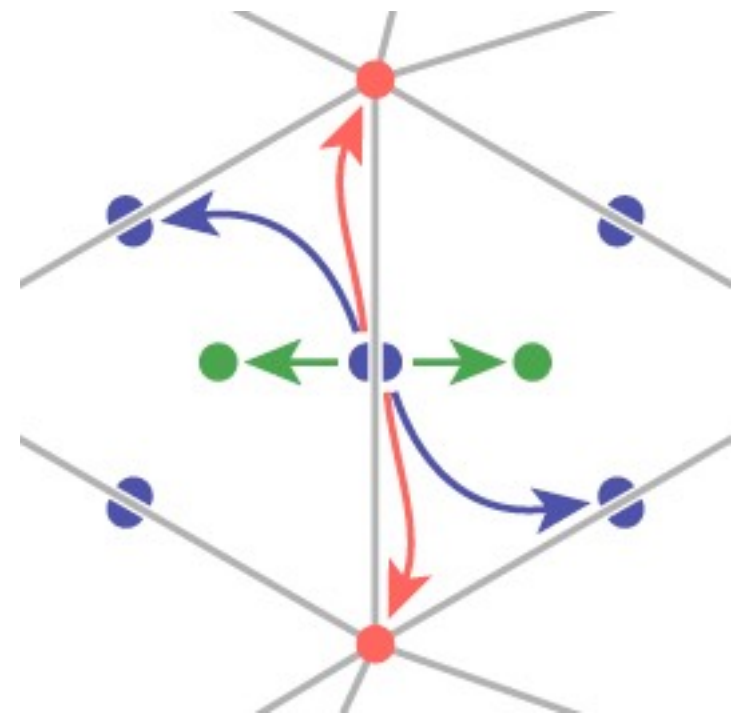
Half-edge structure

- **Simplifies, cleans up winged edge**
 - still works for polygon meshes
- **Each half-edge points to:**
 - next edge (left forward)
 - next vertex (front)
 - the face (left)
 - the opposite half-edge
- **Each face or vertex points to one half-edge**



Half-edge structure

- **Simplifies, cleans up winged edge**
 - still works for polygon meshes
- **Each half-edge points to:**
 - next edge (left forward)
 - next vertex (front)
 - the face (left)
 - the opposite half-edge
- **Each face or vertex points to one half-edge**

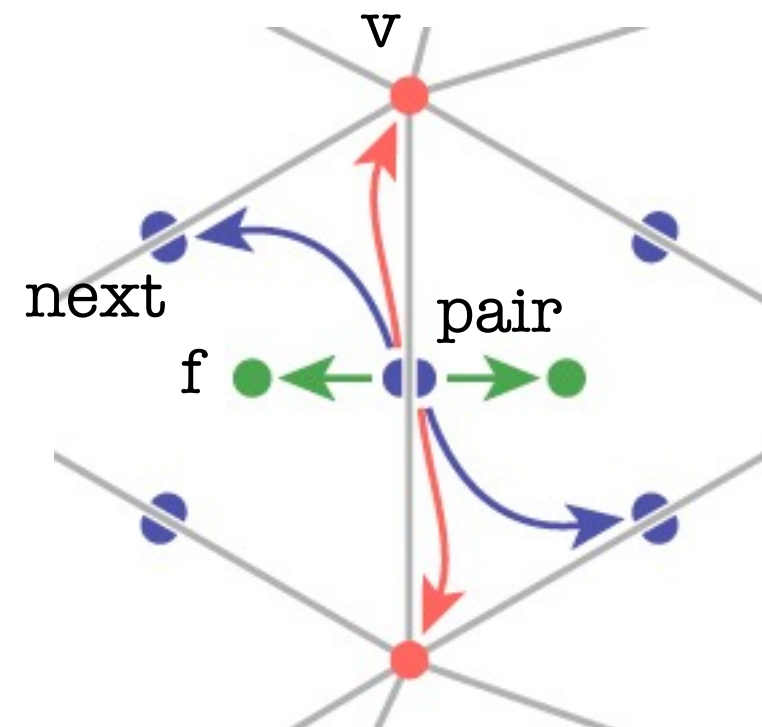


Half-edge structure

```
HEdge {  
    HEdge pair, next;  
    Vertex v;  
    Face f;  
}
```

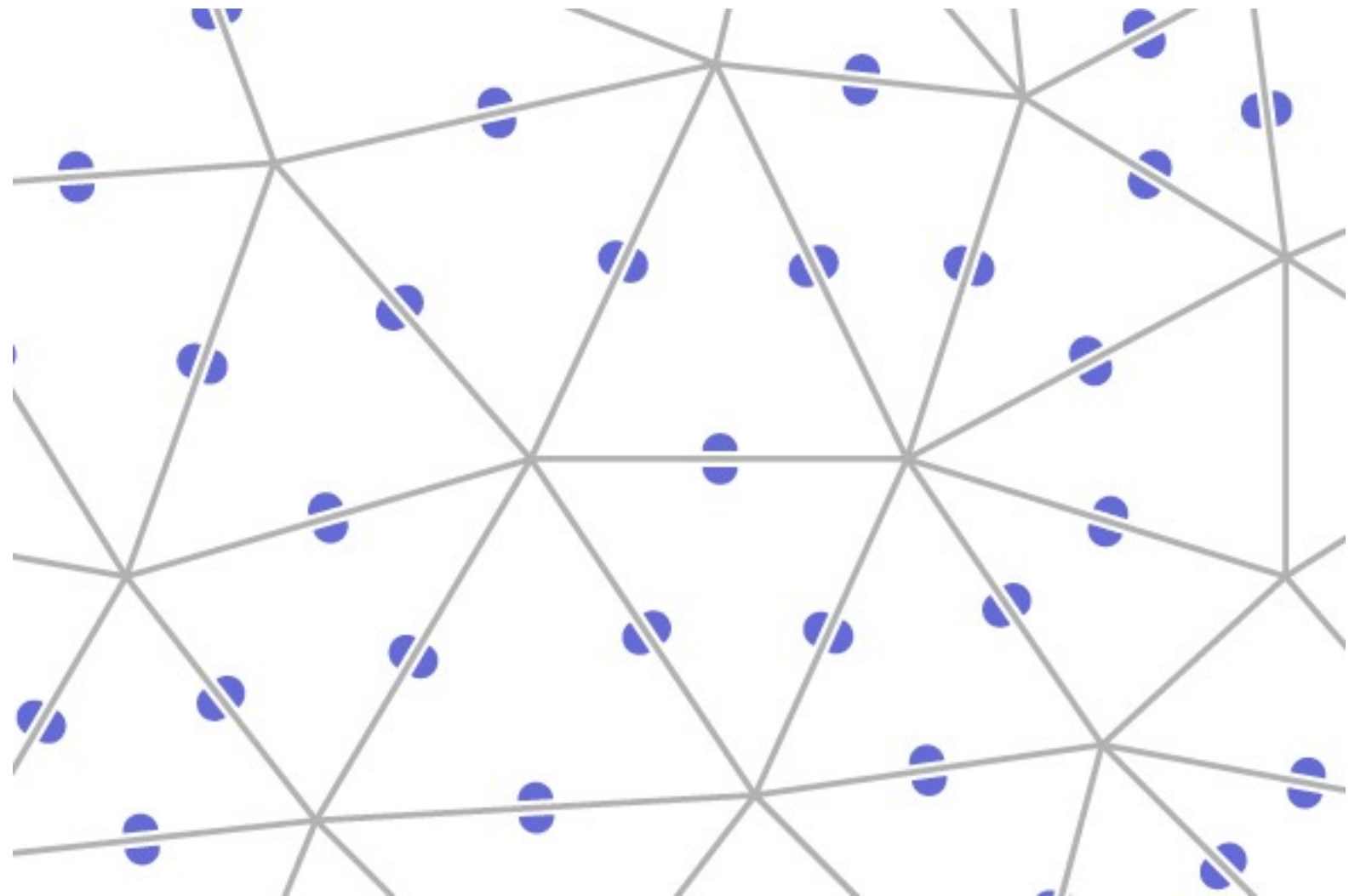
```
Face {  
    // per-face data  
    HEdge h; // any adjacent h-edge  
}
```

```
Vertex {  
    // per-vertex data  
    HEdge h; // any incident h-edge  
}
```



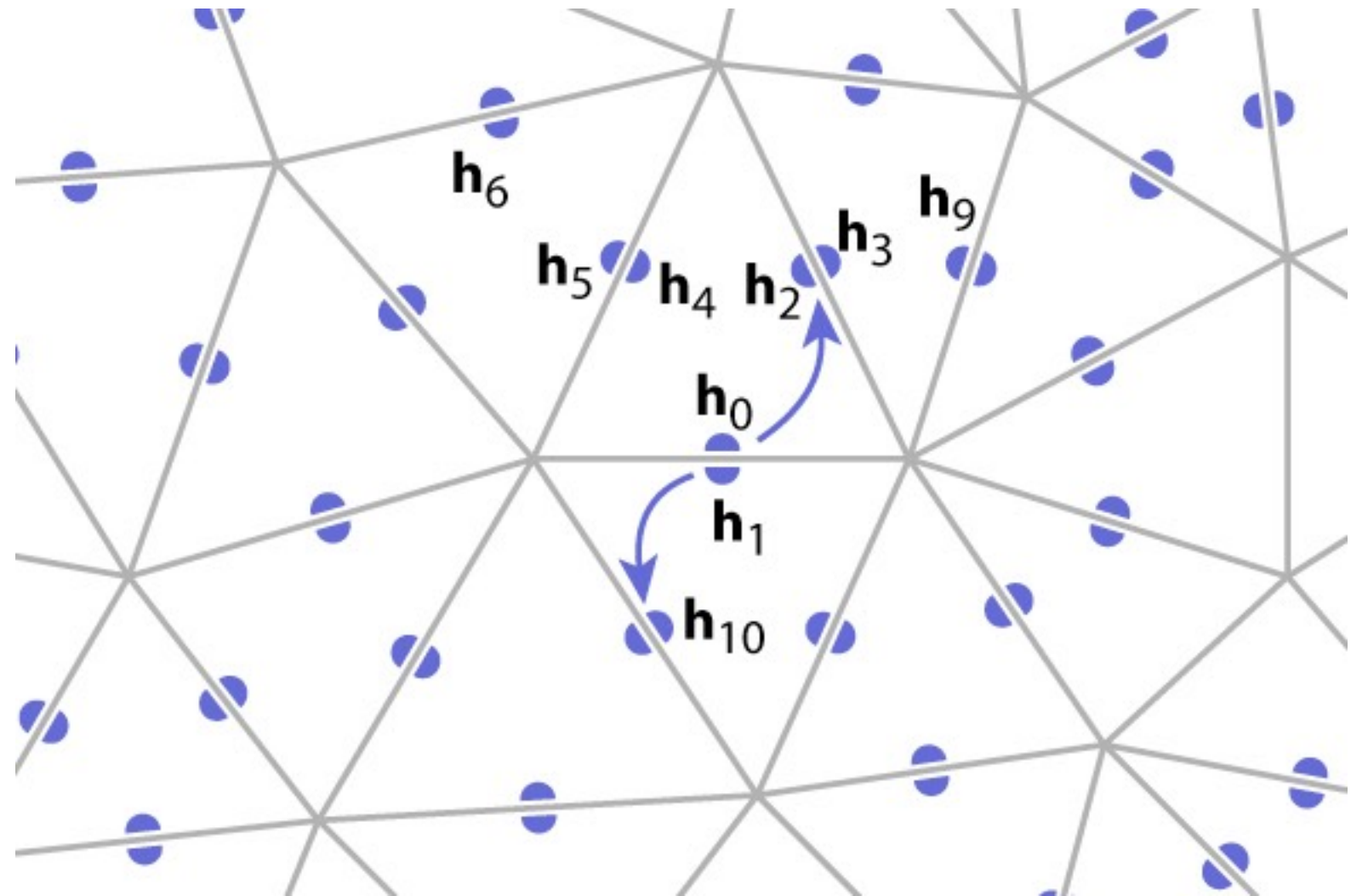
Half-edge structure

	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



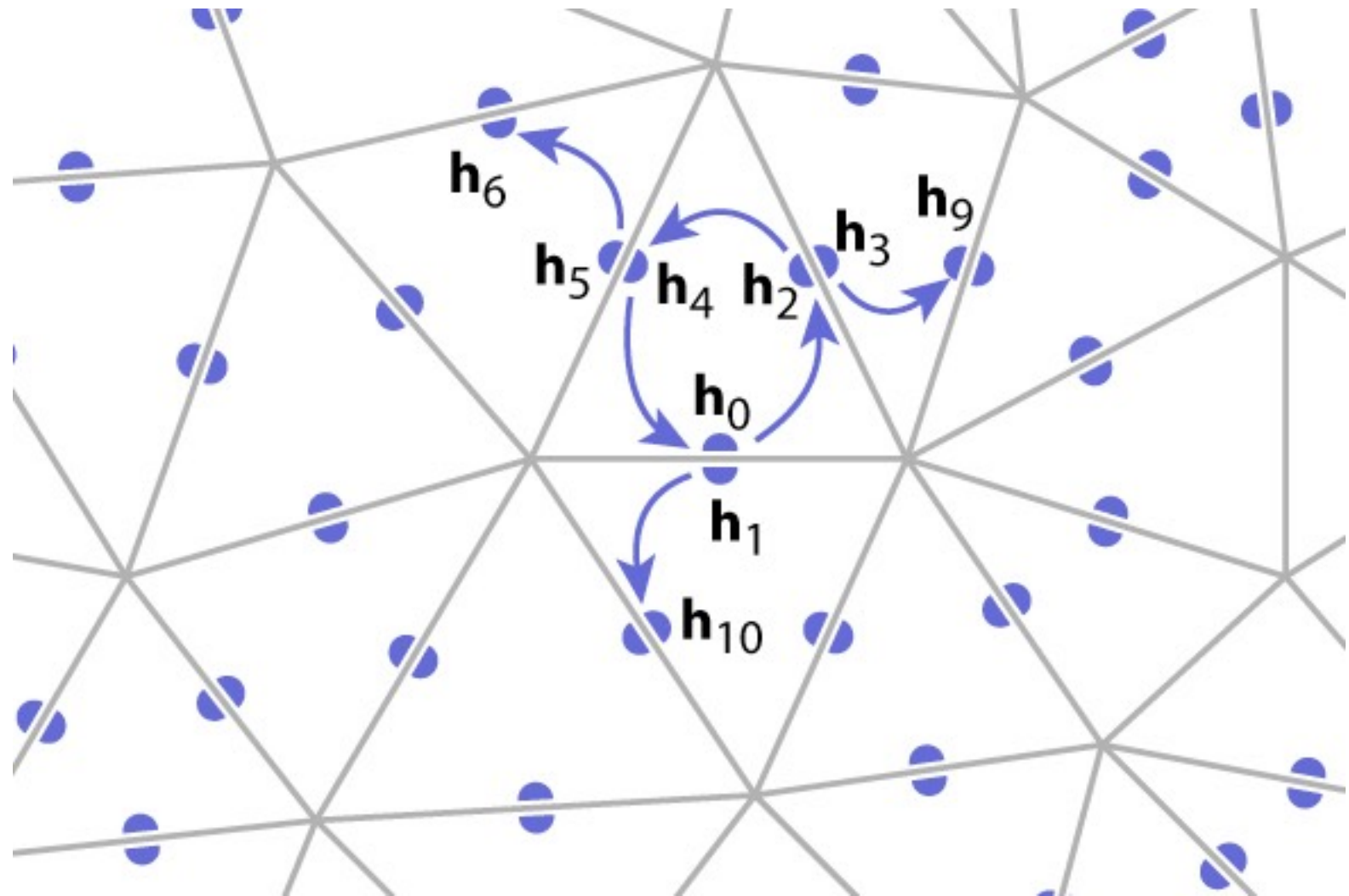
Half-edge structure

	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-edge structure

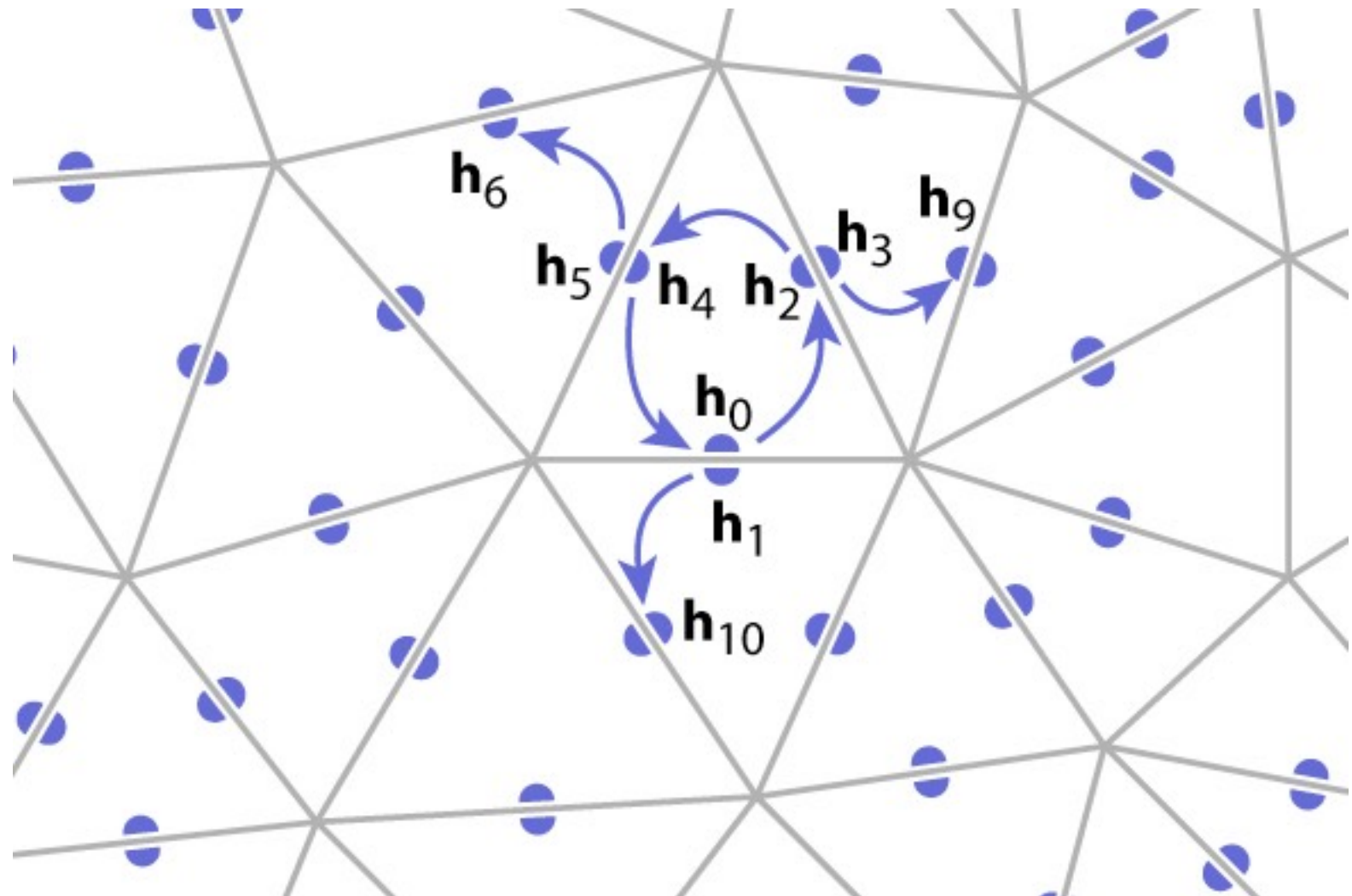
	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-edge structure

```
EdgesOfFace(f) {  
  h = f.h;  
  do {  
    h = h.next;  
  } while (h != f.h);  
}
```

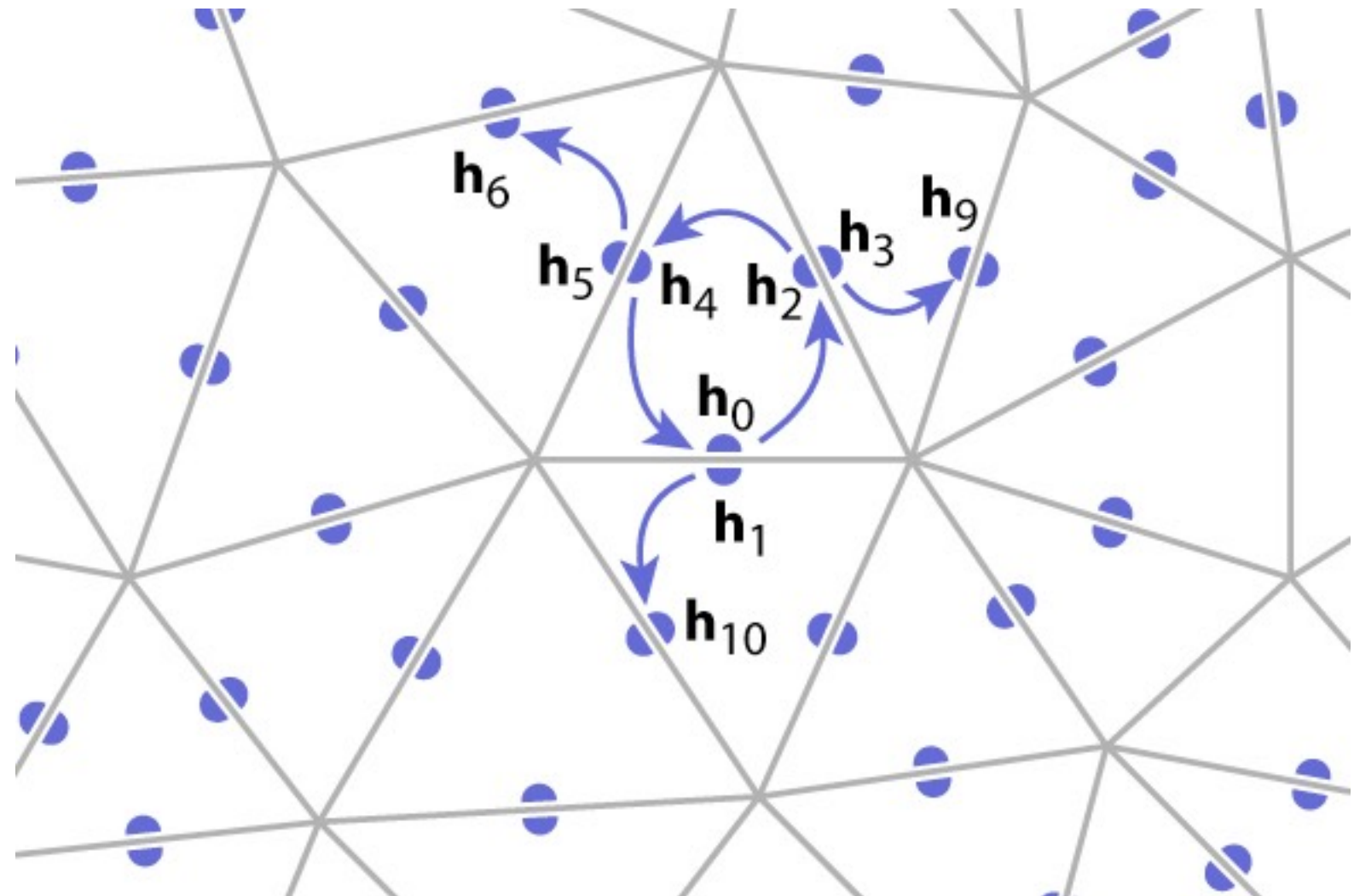
	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-edge structure

```
EdgesOfVertex(v) {  
  h = v.h;  
  do {  
    h = h.next.pair;  
  } while (h != v.h);  
}
```

	pair	next
hedge[0]	1	2
hedge[1]	0	10
hedge[2]	3	4
hedge[3]	2	9
hedge[4]	5	0
hedge[5]	4	6
	⋮	



Half-edge structure

- **array of vertex positions: 12 bytes/vert**
- **array of 4-tuples of indices (per h-edge)**
 - next, pair h-edges + head vert + left tri
 - $\text{int}[2n_E][4]$: about 96 bytes per vertex
 - 6 h-edges per vertex (on average)
 - (4 indices x 4 bytes) per h-edge
- **add a representative h-edge per vertex**
 - $\text{int}[n_V]$: 4 bytes per vertex
- **total storage: 112 bytes per vertex**

Half-edge optimizations

- **Omit faces if not needed**
- **Use implicit pair pointers**
 - they are allocated in pairs
 - they are even and odd in an array
- **Result: 2 indices per HEdge**
 - HEdges are 48 bytes/vertex
 - total 64 bytes/vertex (same as triangle neighbor)

