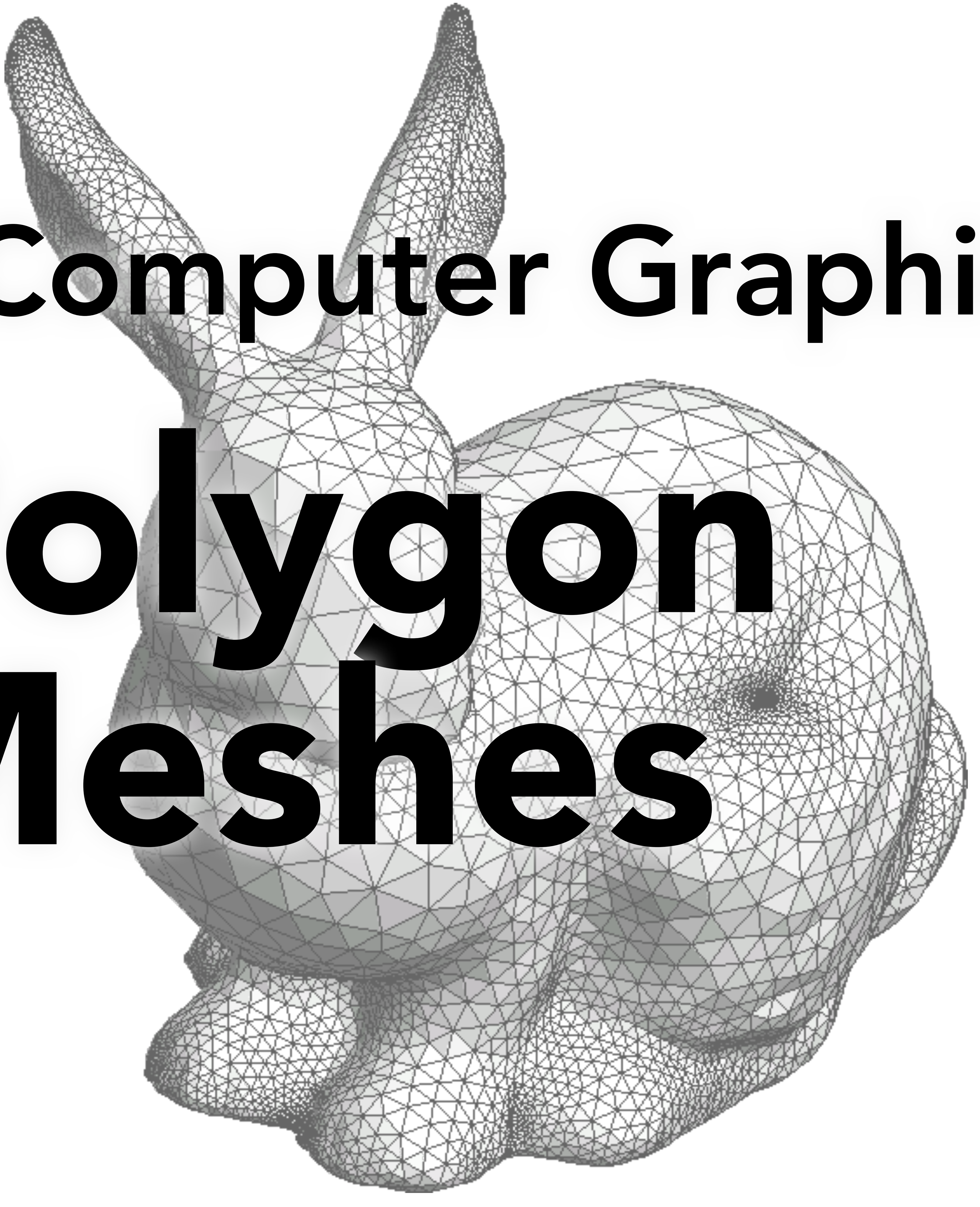


COL781: Computer Graphics

16. Polygon Meshes

Announcements

Assignment 1 due on Moodle tonight at midnight!

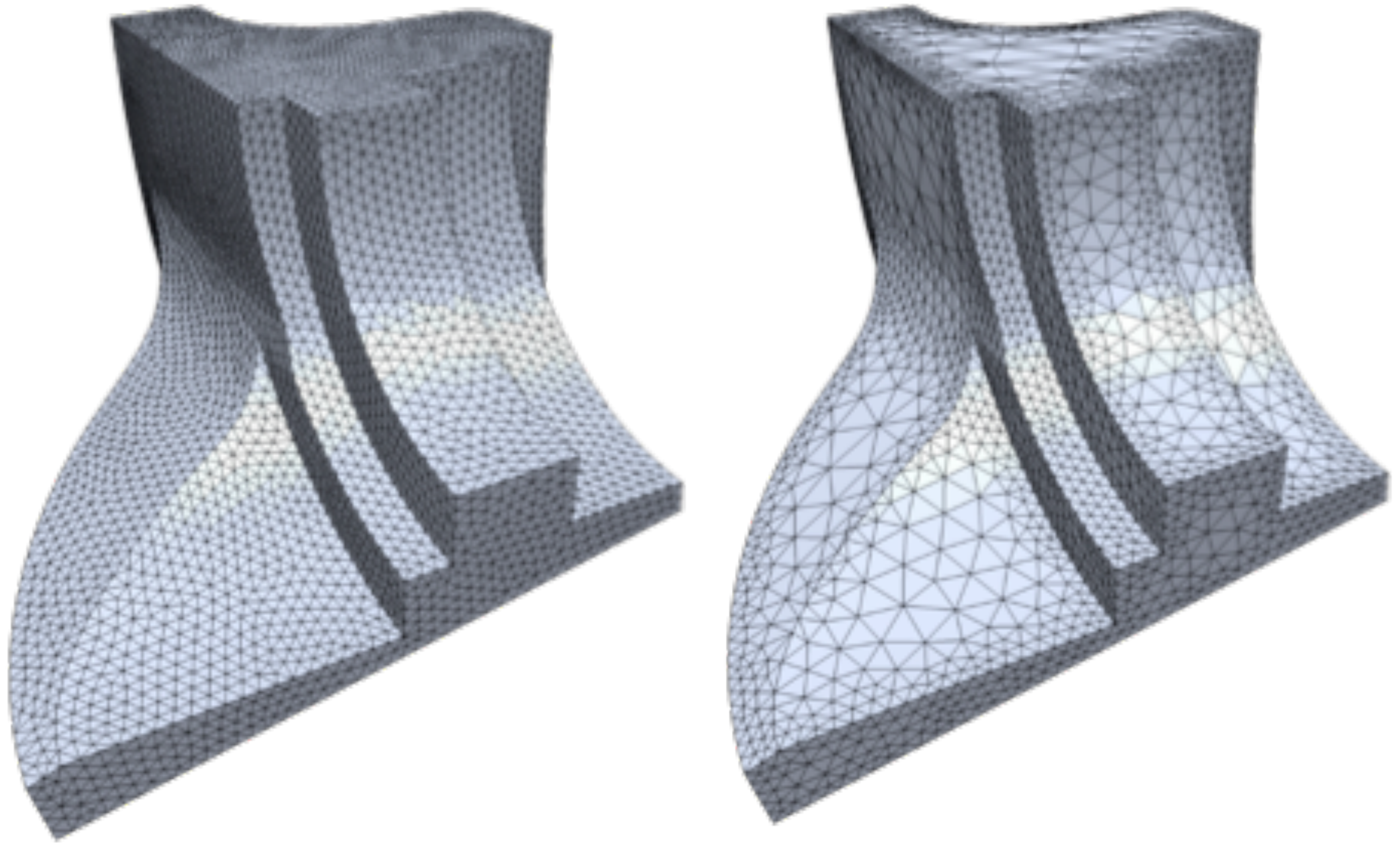
Assignment 2 has been (partially) posted: mesh processing

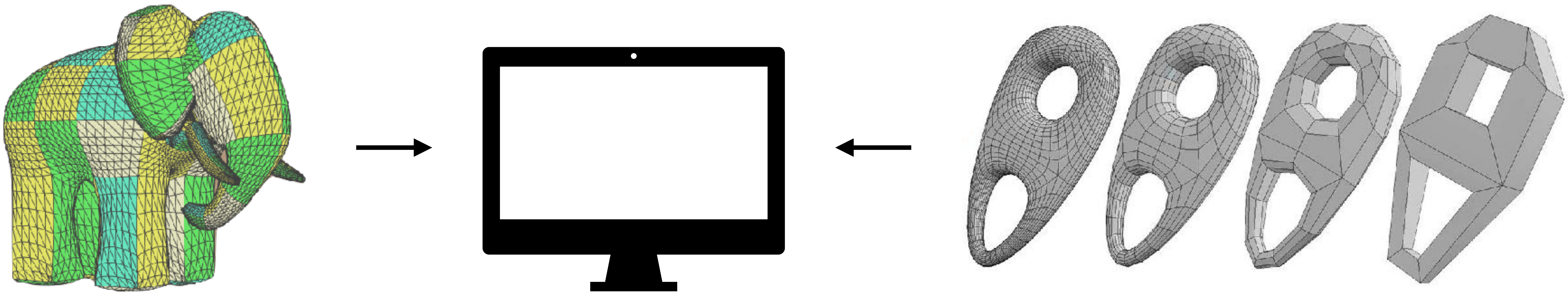
Friday class needs to be rescheduled. Friday 5pm or 6pm? Saturday 11am?

Polygon meshes

A good "lowest common denominator" for surface geometry

- Efficient to render
- Flexible for topology, refinement, etc.

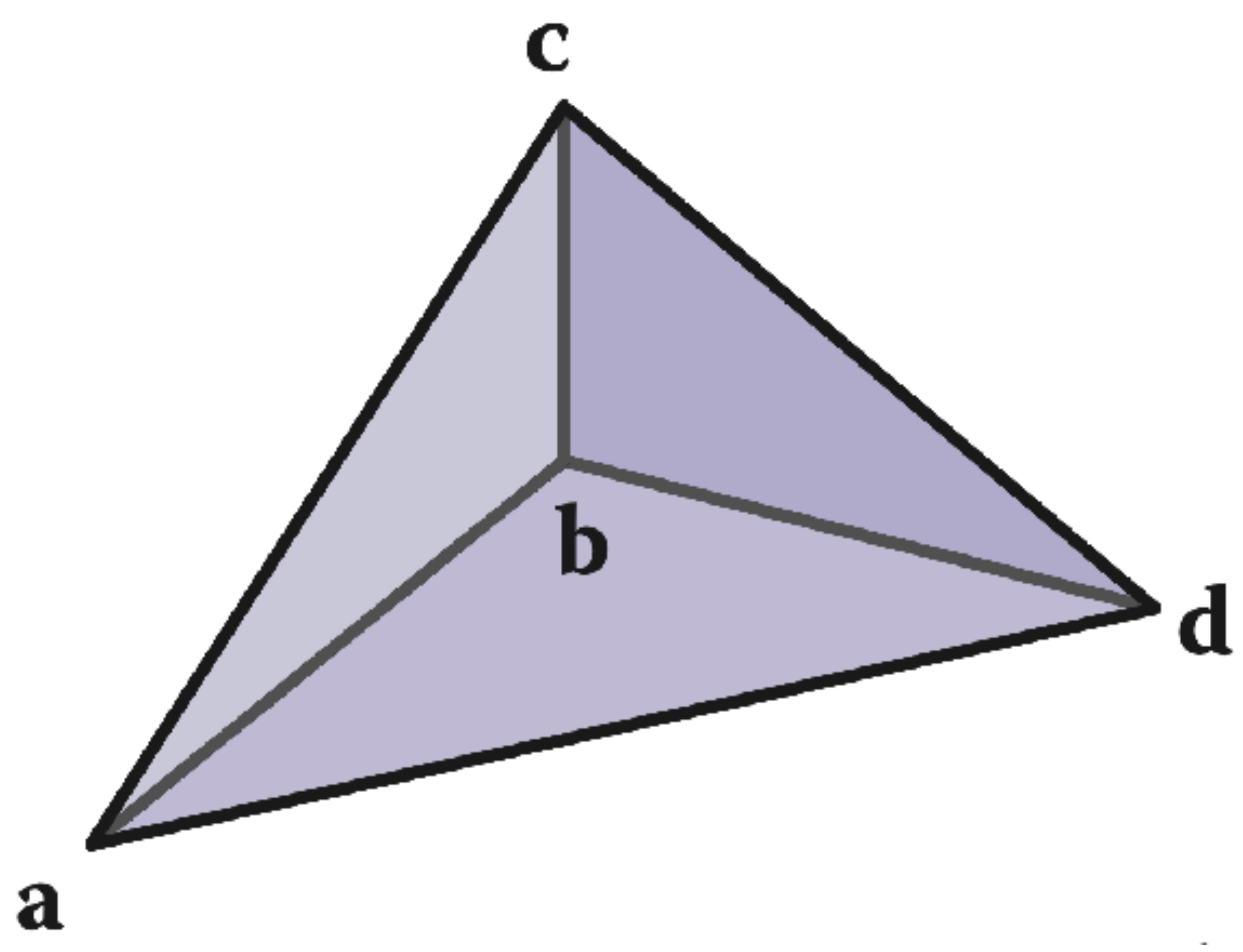




How to store a mesh?

"Polygon soup"

#	*Vertex* 0	*Vertex* 1	*Vertex* 2
0	(a_x, a_y, a_z)	(b_x, b_y, b_z)	(c_x, c_y, c_z)
1	(b_x, b_y, b_z)	(d_x, d_y, d_z)	(c_x, c_y, c_z)
2	(a_x, a_y, a_z)	(d_x, d_y, d_z)	(b_x, b_y, b_z)



Redundant storage of vertices

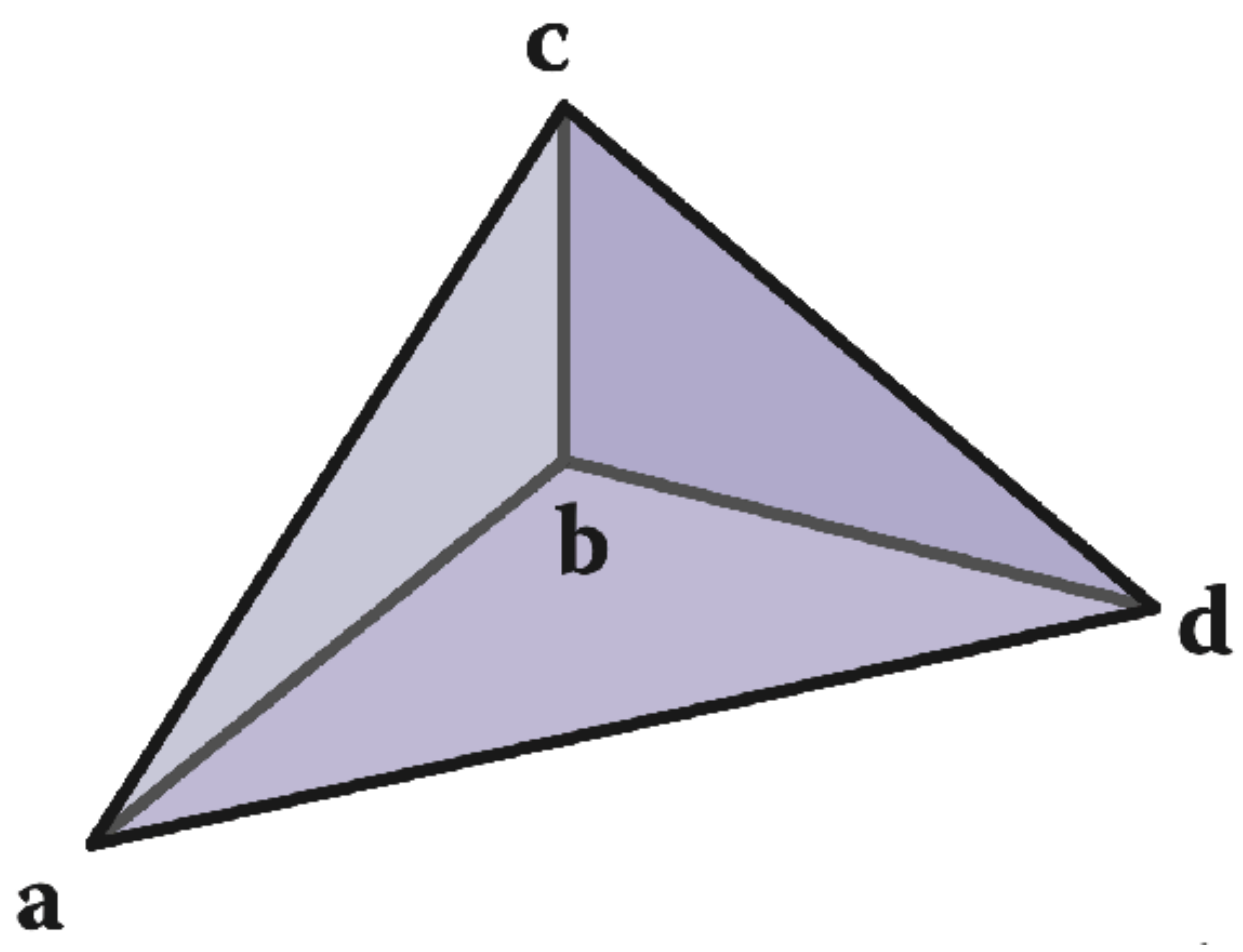
No connectivity information

How to store a mesh?

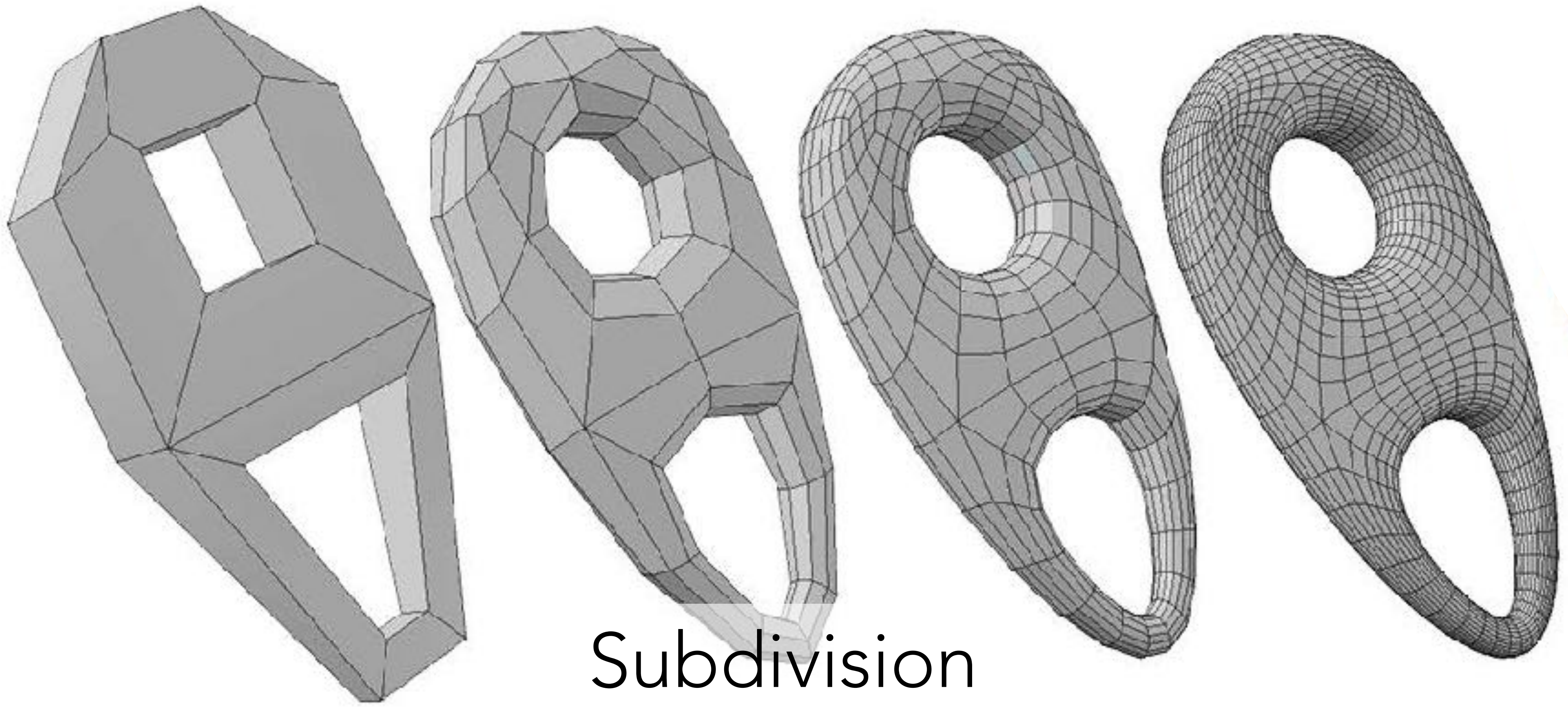
Indexed polygons

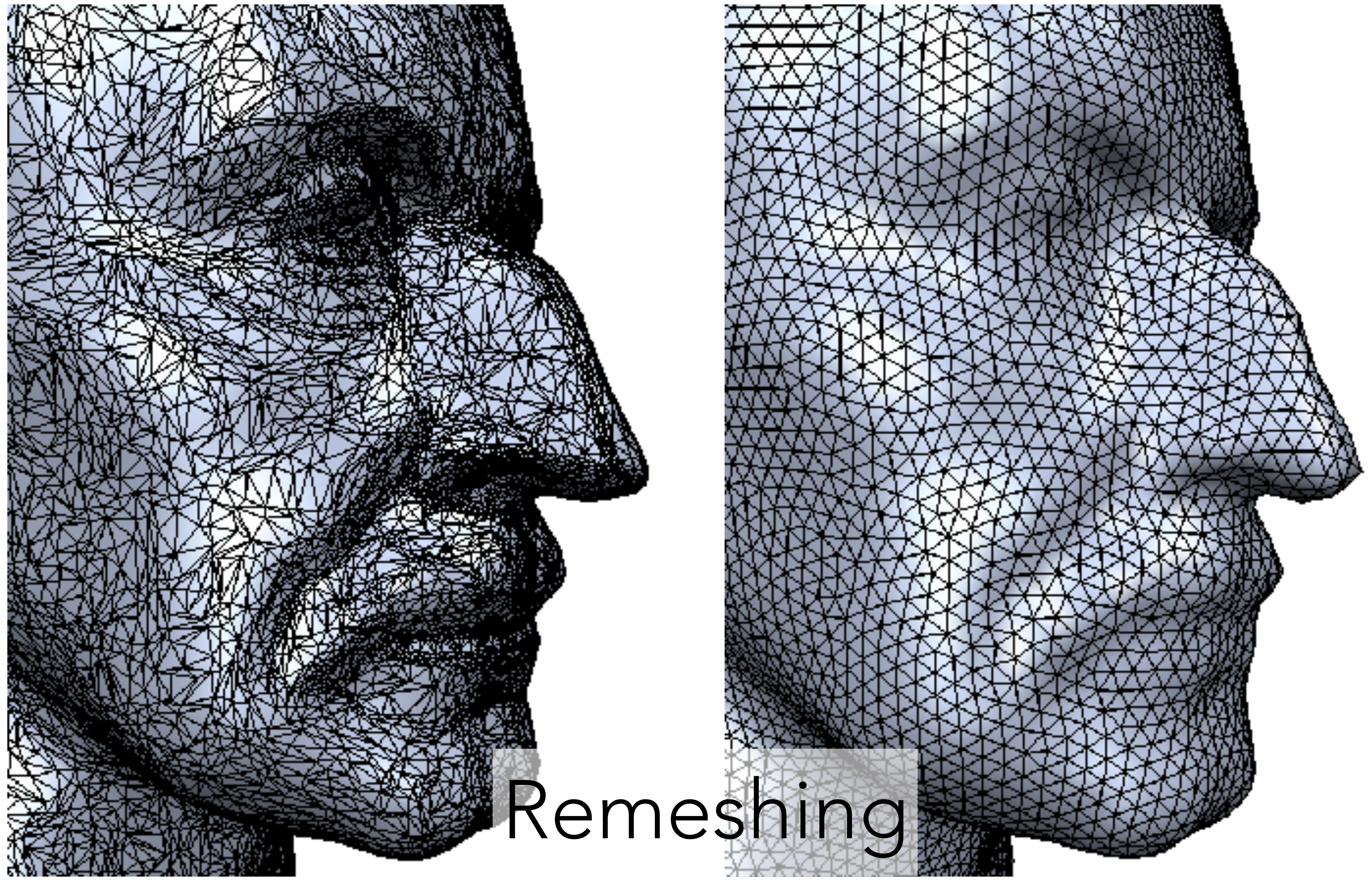
Triangles	
#	*Vertices*
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

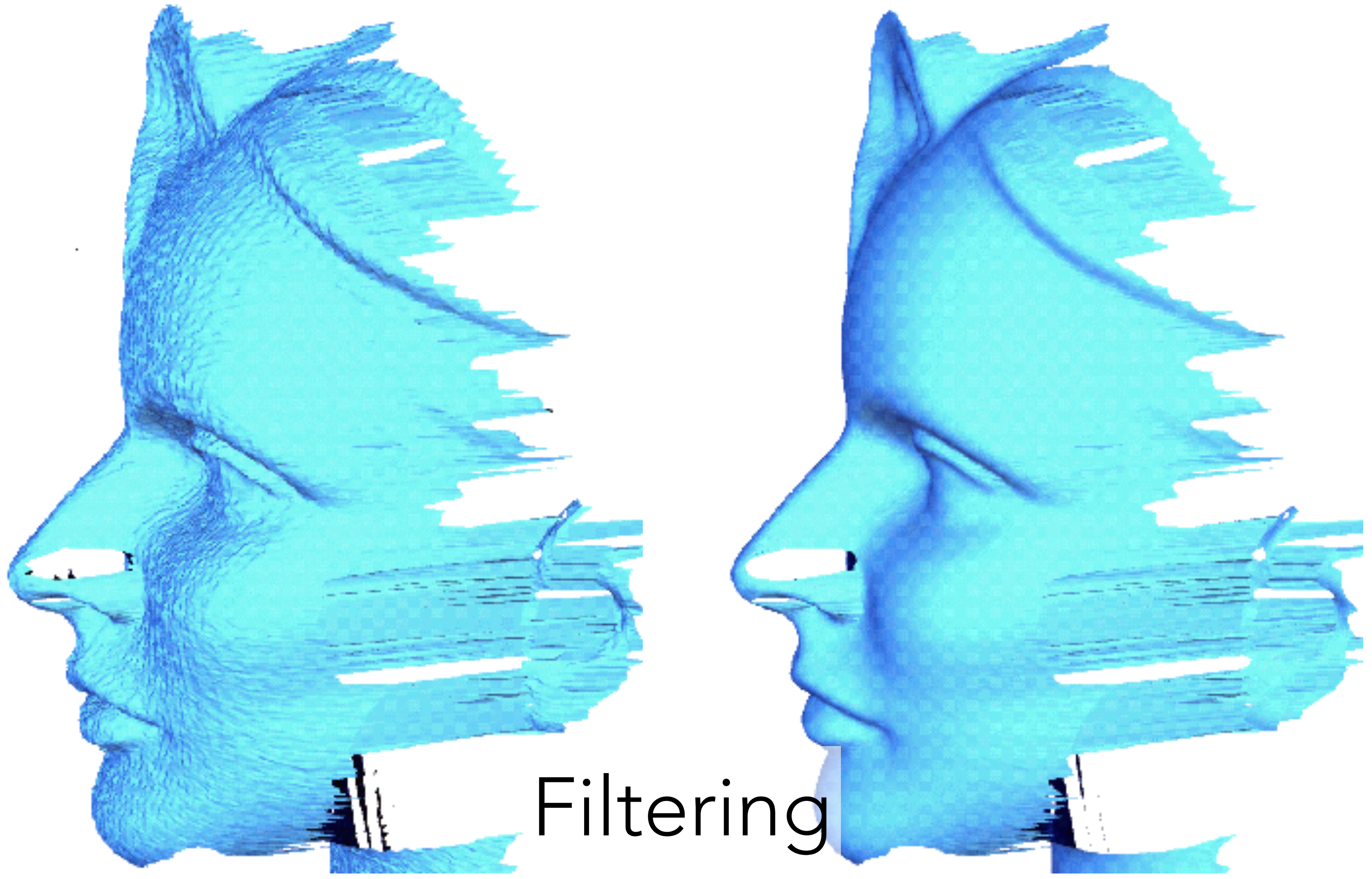
Vertices	
#	*Position*
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

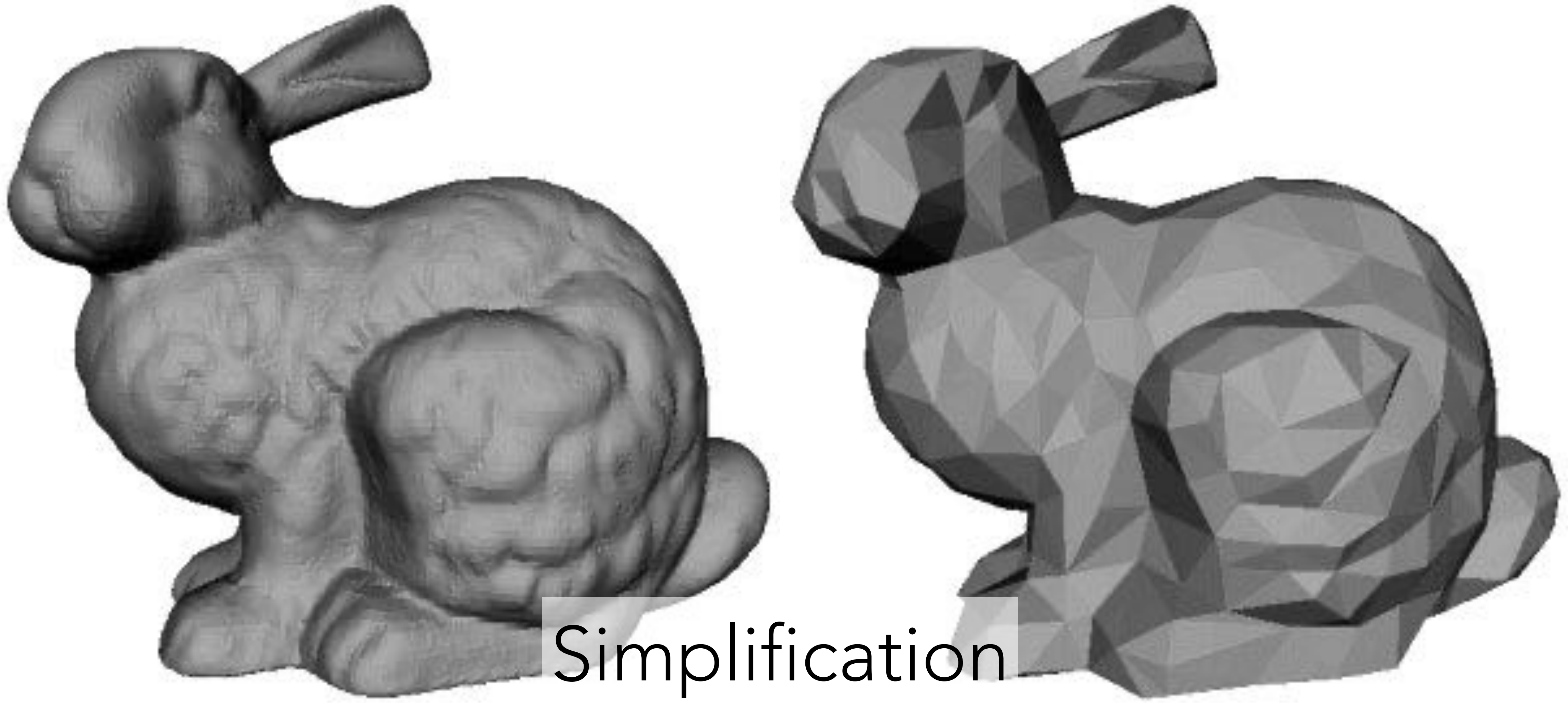


How do we find all the polygons adjacent to a given vertex?
Or which polygon is across from a given edge of a polygon?


Subdivision

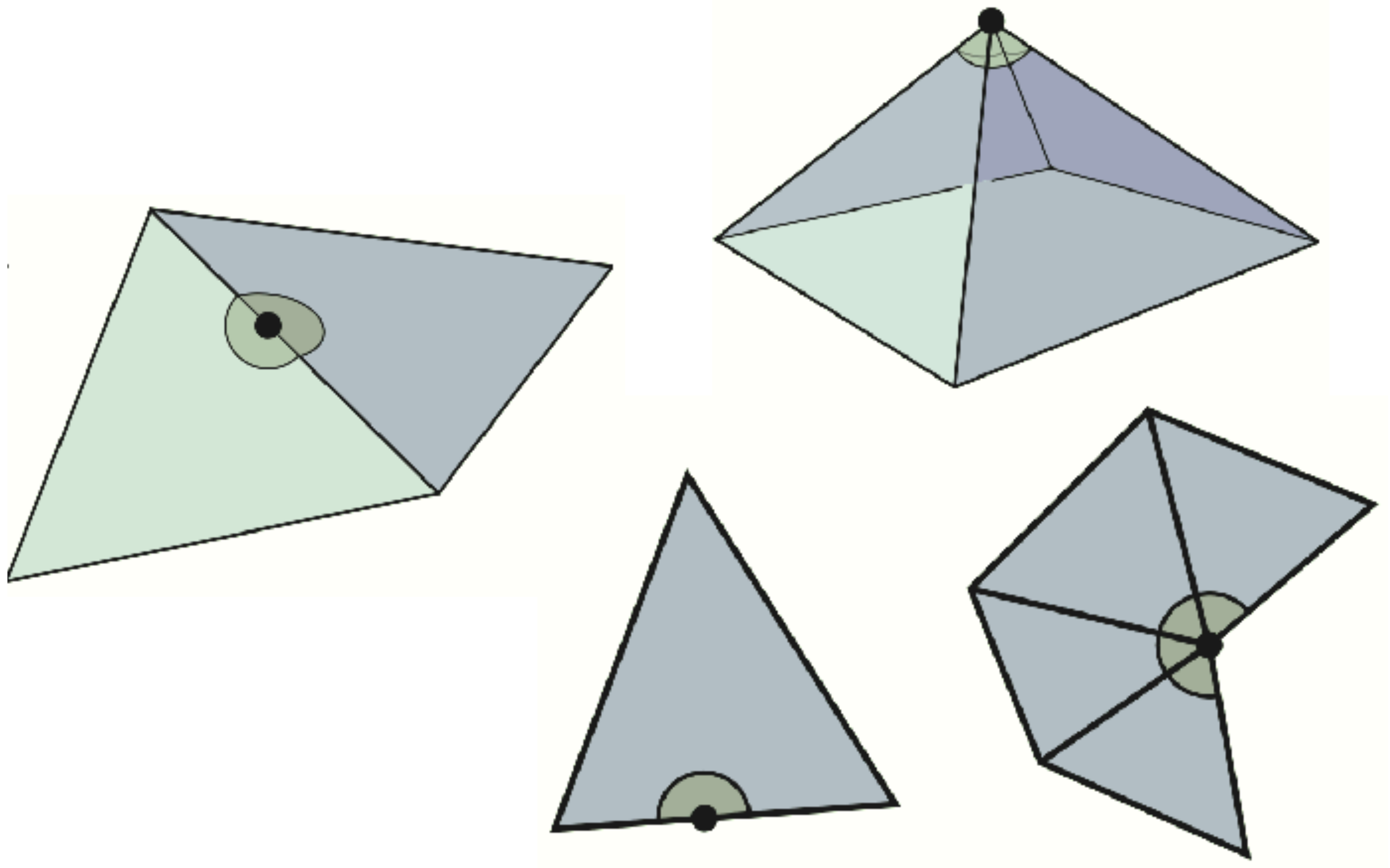

Remeshing


Filtering

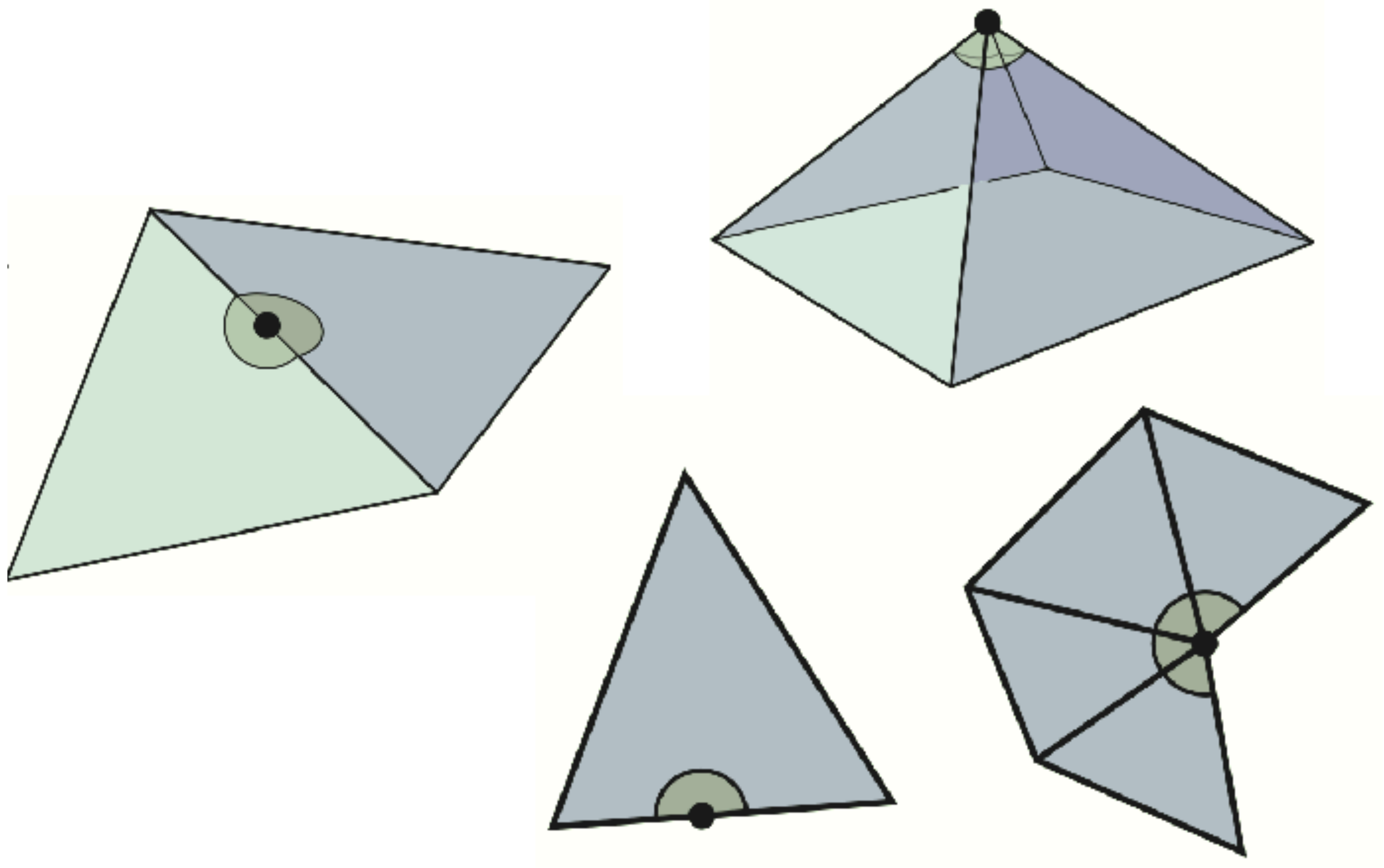

Simplification

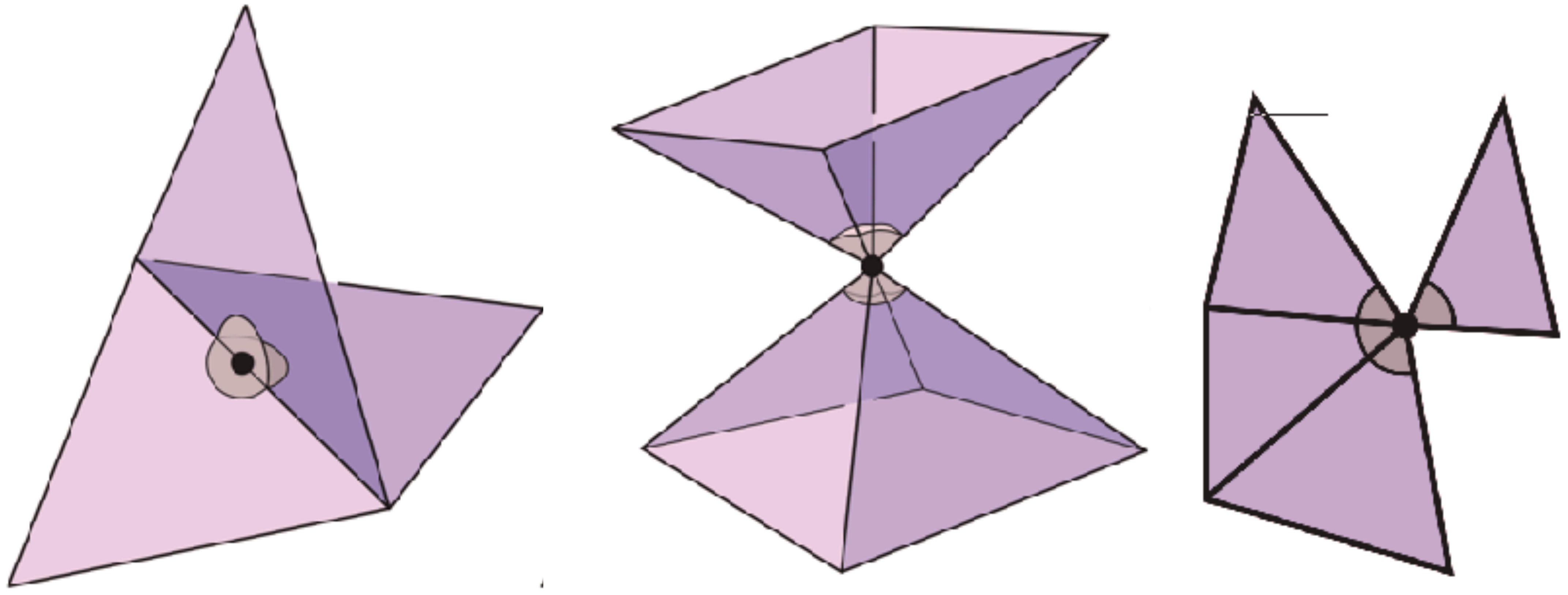
We need a data structure that allows us to quickly access neighbours:

- for each degree-k **vertex**: k adjacent vertices, edges, faces
- for each **edge**: 2 adjacent faces
- for each n-gonal **face**: n adjacent vertices, edges, faces



An arbitrary polygon mesh can have weirder neighbourhoods…

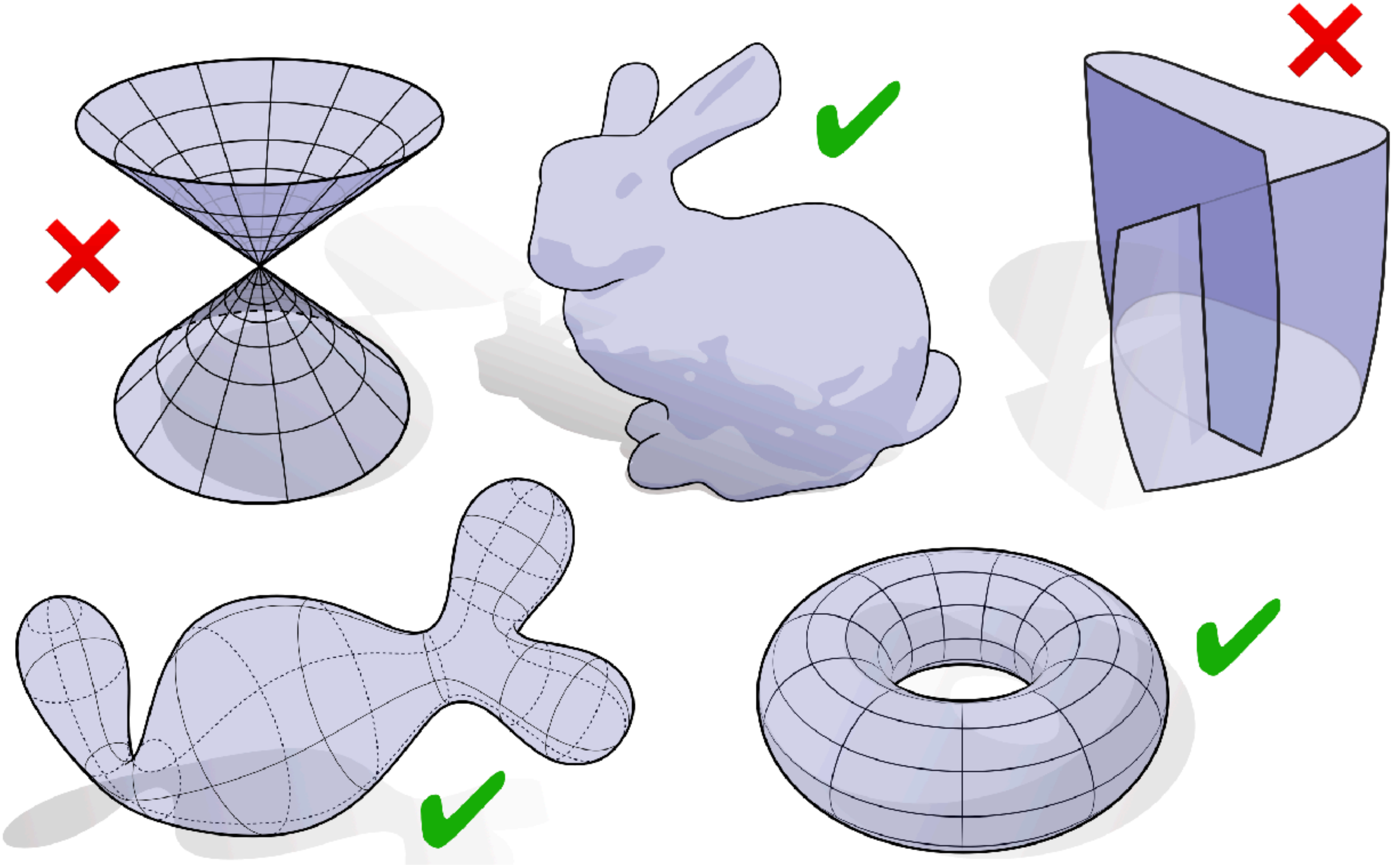




But we usually don’t want such meshes!

How do we formalize this?

A 2D **manifold** is a set of points such that the neighbourhood of every point is topologically a 2D disk.



Puzzle:

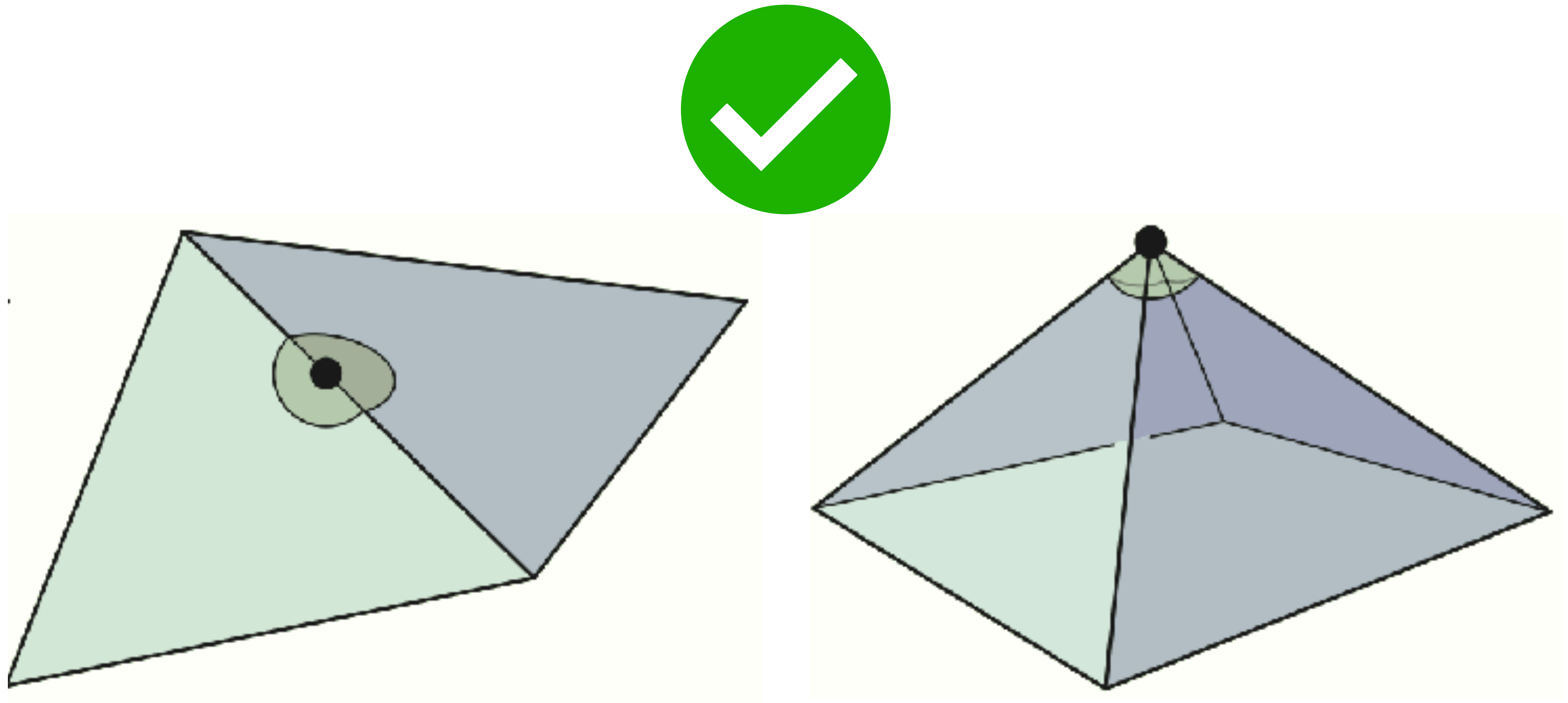
What conditions do you need to impose on a polygon mesh so that it represents a manifold surface?

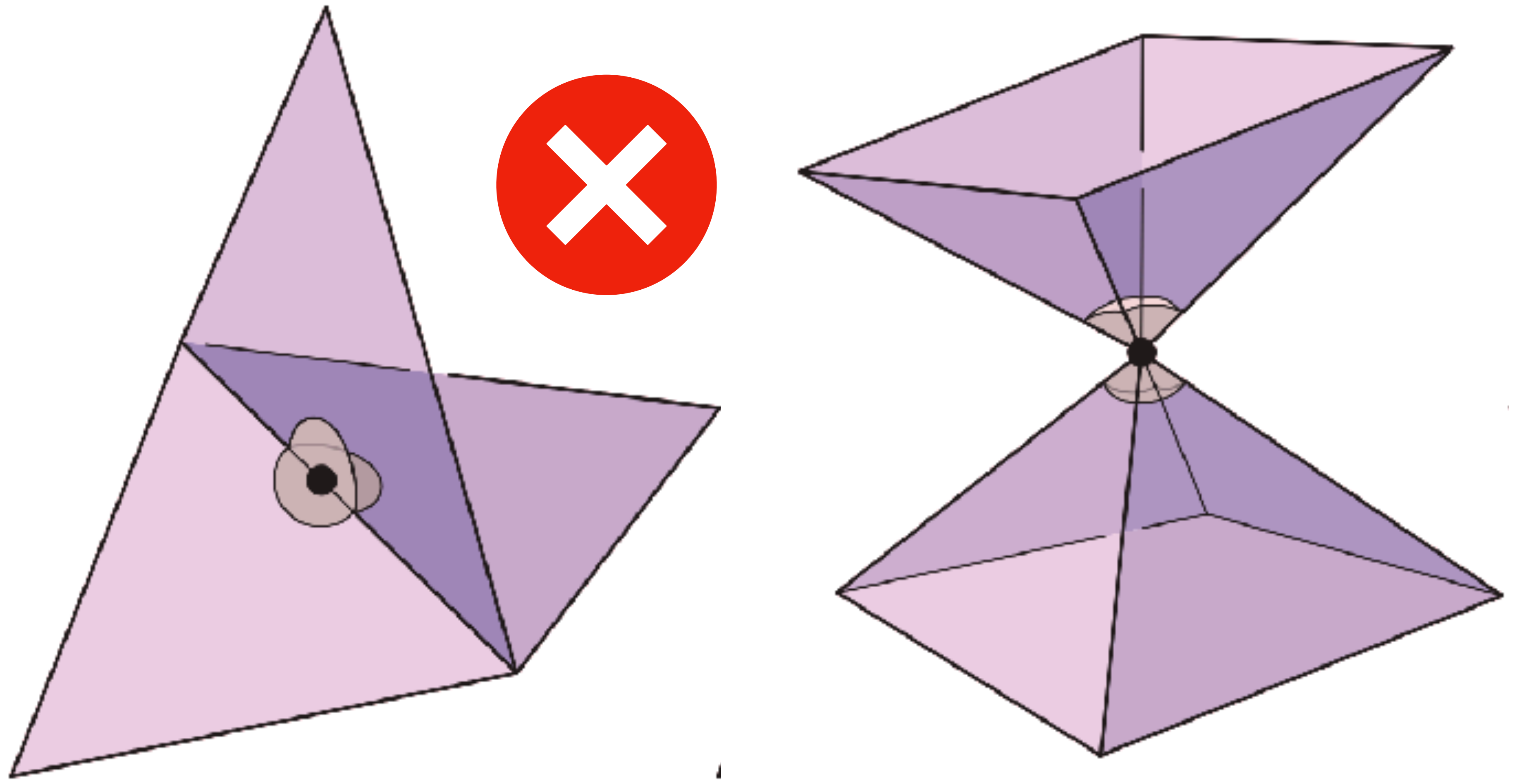
Assume the mesh itself has no self-intersections; points that are adjacent in space are connected in the mesh.

Manifold meshes

A polygon mesh is a manifold only if:

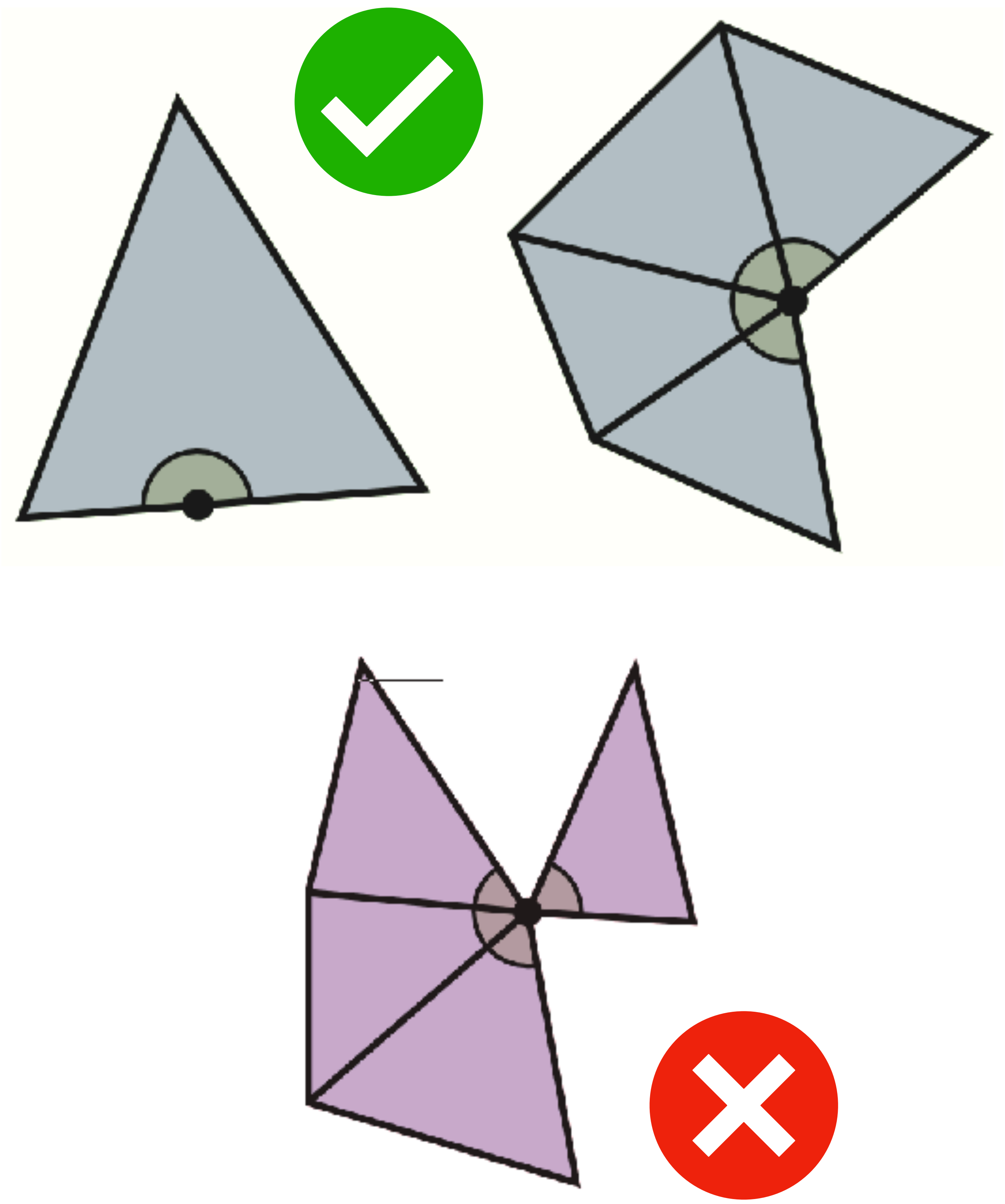
- Every edge has exactly 2 adjacent faces
- Every vertex has adjacent faces and edges in a single ring





A polygon mesh is a manifold with boundary if:

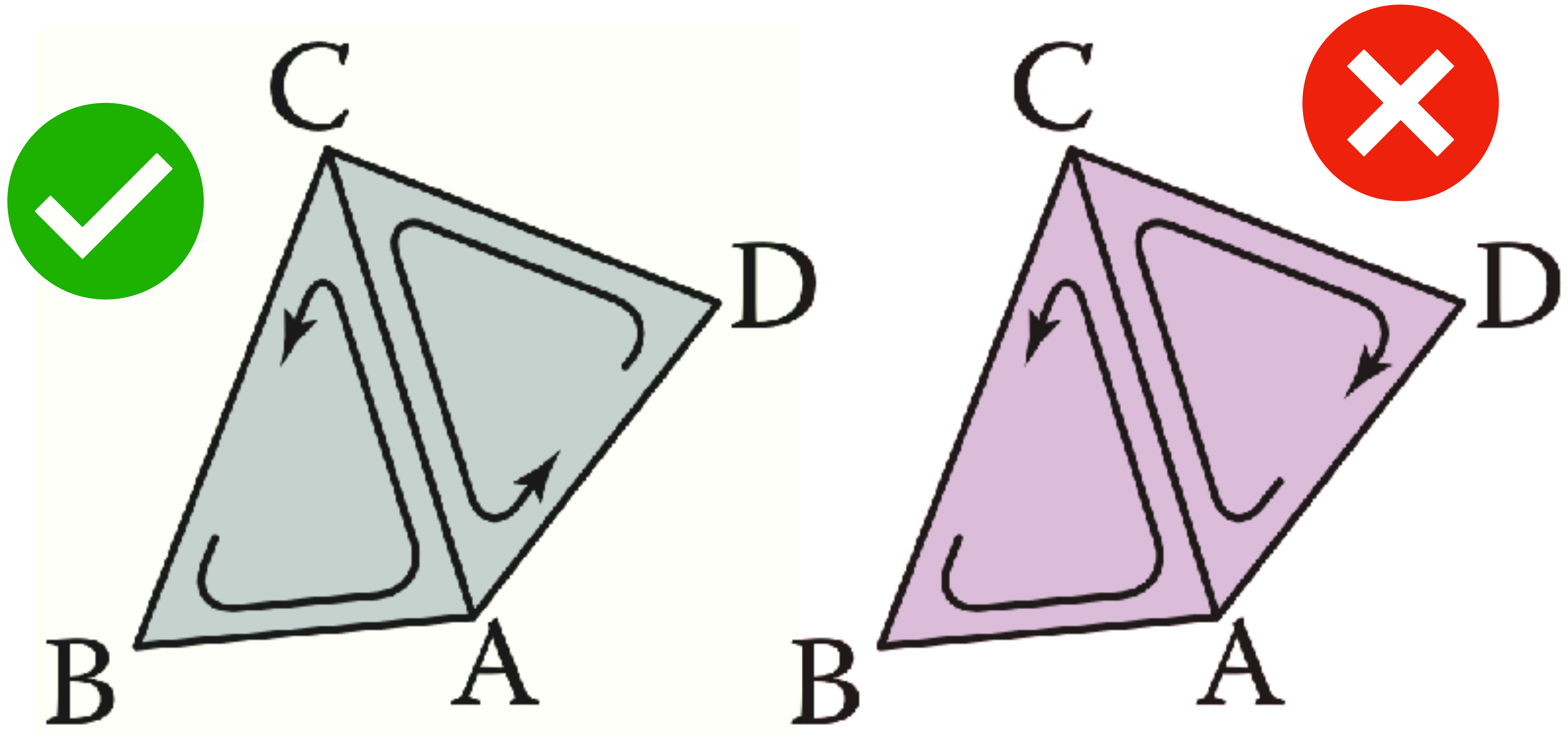
- Boundary edges have only 1 adjacent face
- Boundary vertices have adjacent faces and edges in a single chain
- Other edges and vertices are manifold



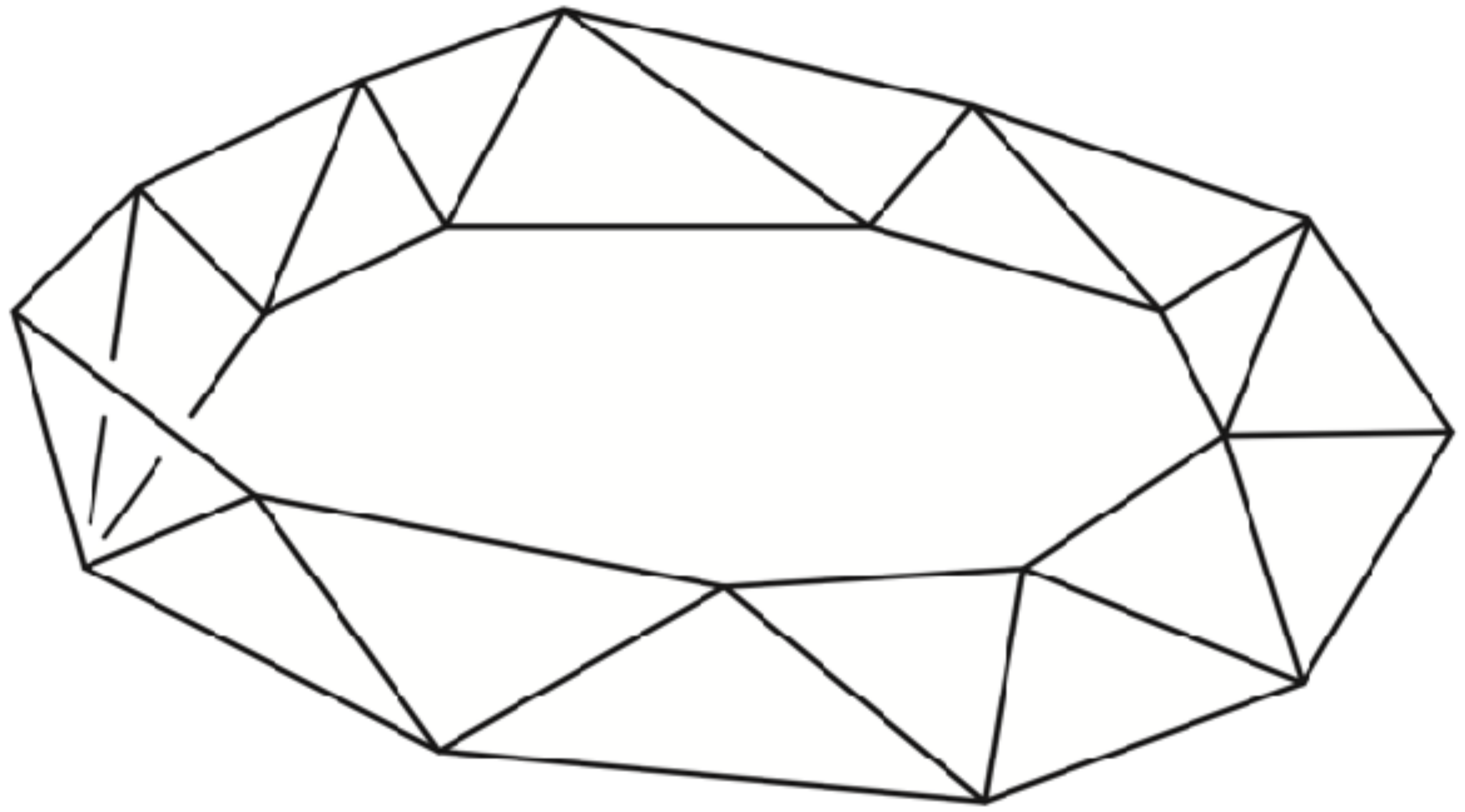
Orientation consistency

Adjacent faces should all be oriented in the same direction, so they agree on “front” and “back”

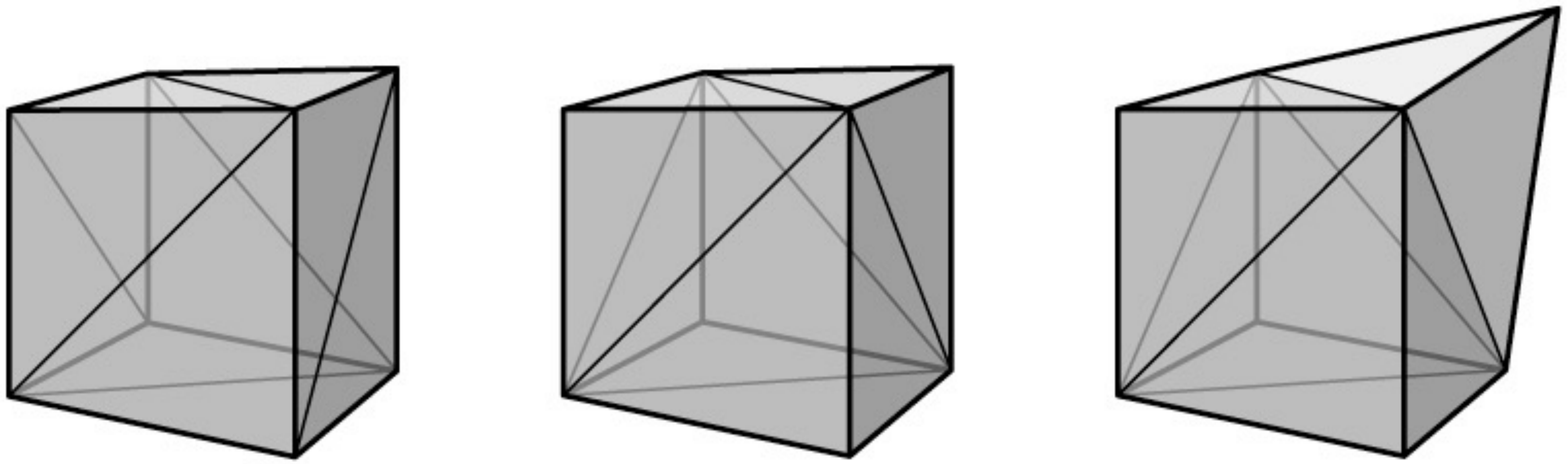
- Each edge should be traversed in opposite directions by adjacent faces



Not all manifolds can be oriented consistently:



All these requirements are purely about connectivity, not geometry! Can be verified discretely by checking only indices, no floating-point arithmetic.

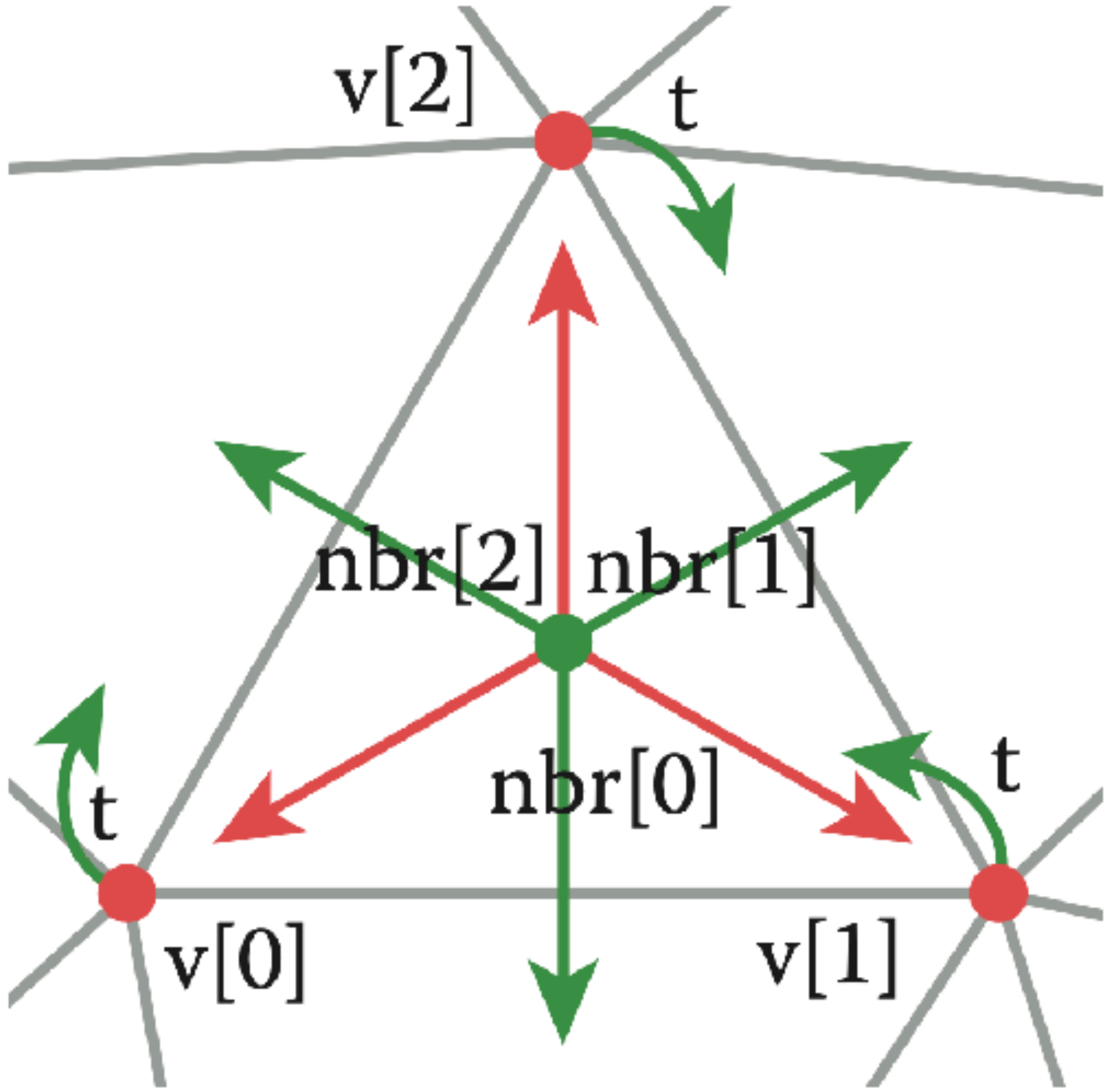


Same geometry, different connectivity | Original mesh | Same connectivity, different geometry

Today: data structures to efficiently store and look up connectivity

Triangle neighbour data structure

```
Vertex {
    Point position;
    Triangle *triangle;
}

Triangle {
    Vertex *vertices[3];
    Triangle *neighbors[3];
}
```



vTri

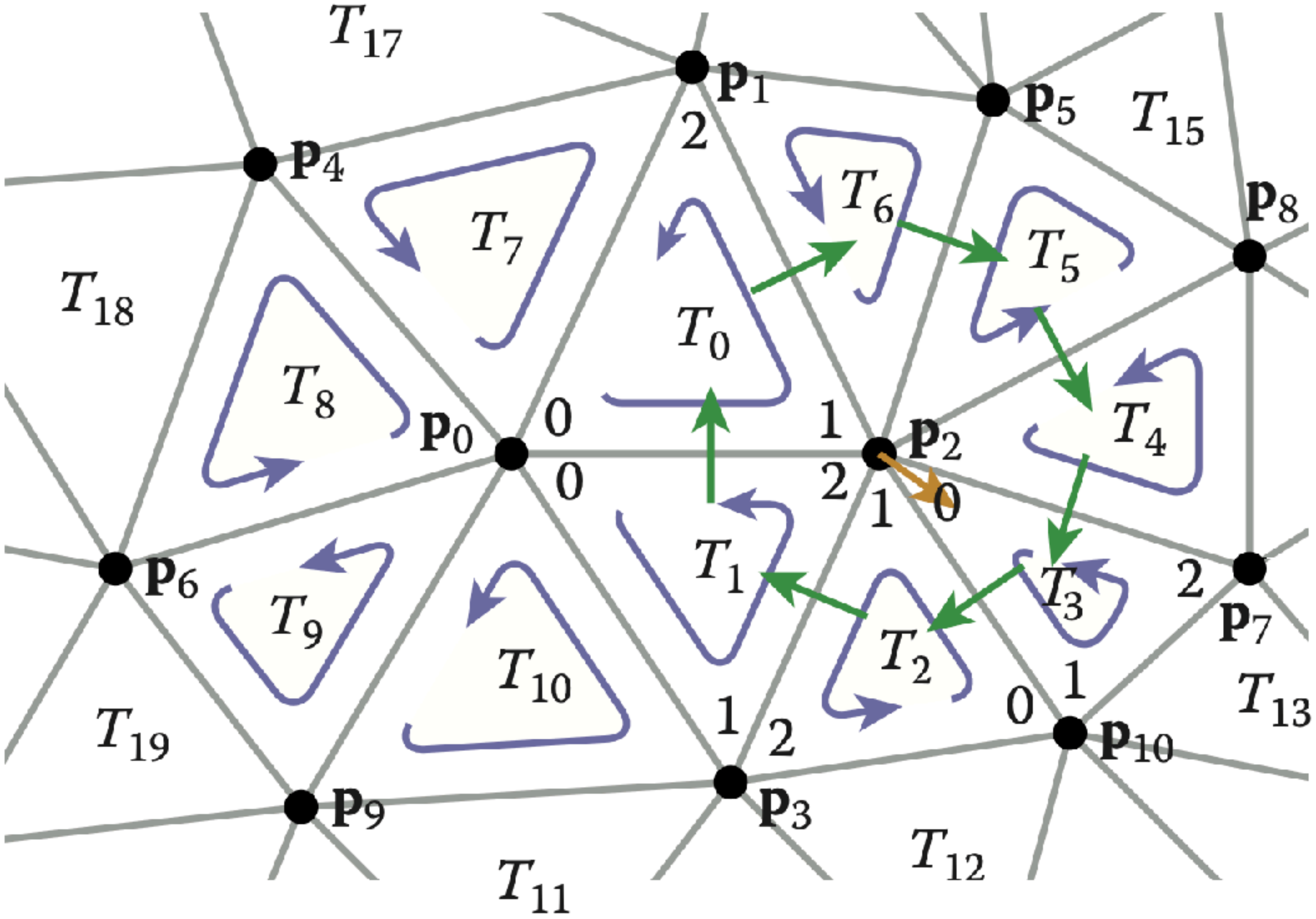
[0]	0
[1]	6
[2]	3
[3]	1
	⋮

tNbr

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

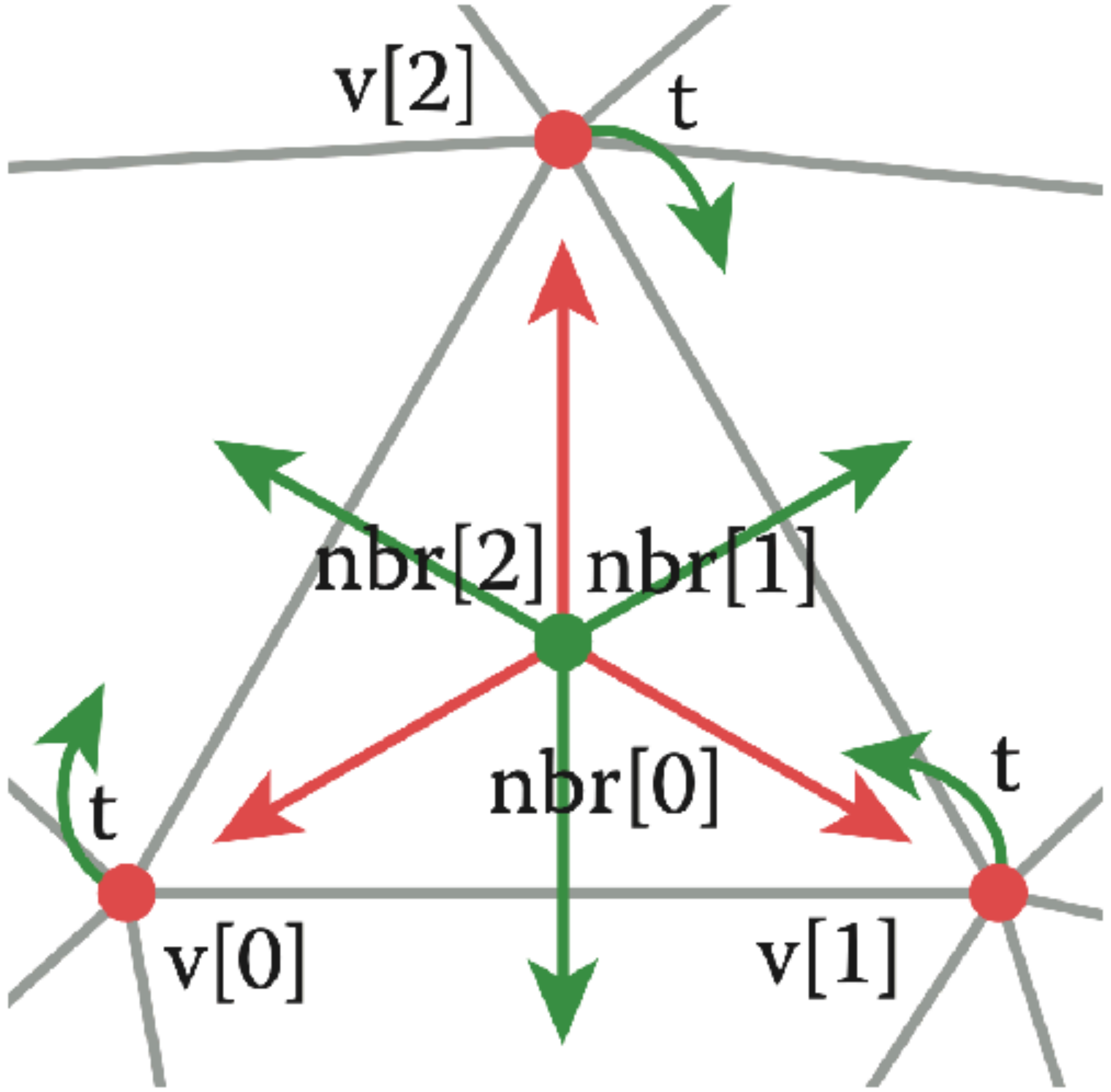
tInd

[0]	0, 2, 1
[1]	0, 3, 2
[2]	10, 2, 3
[3]	2,10,7
	⋮

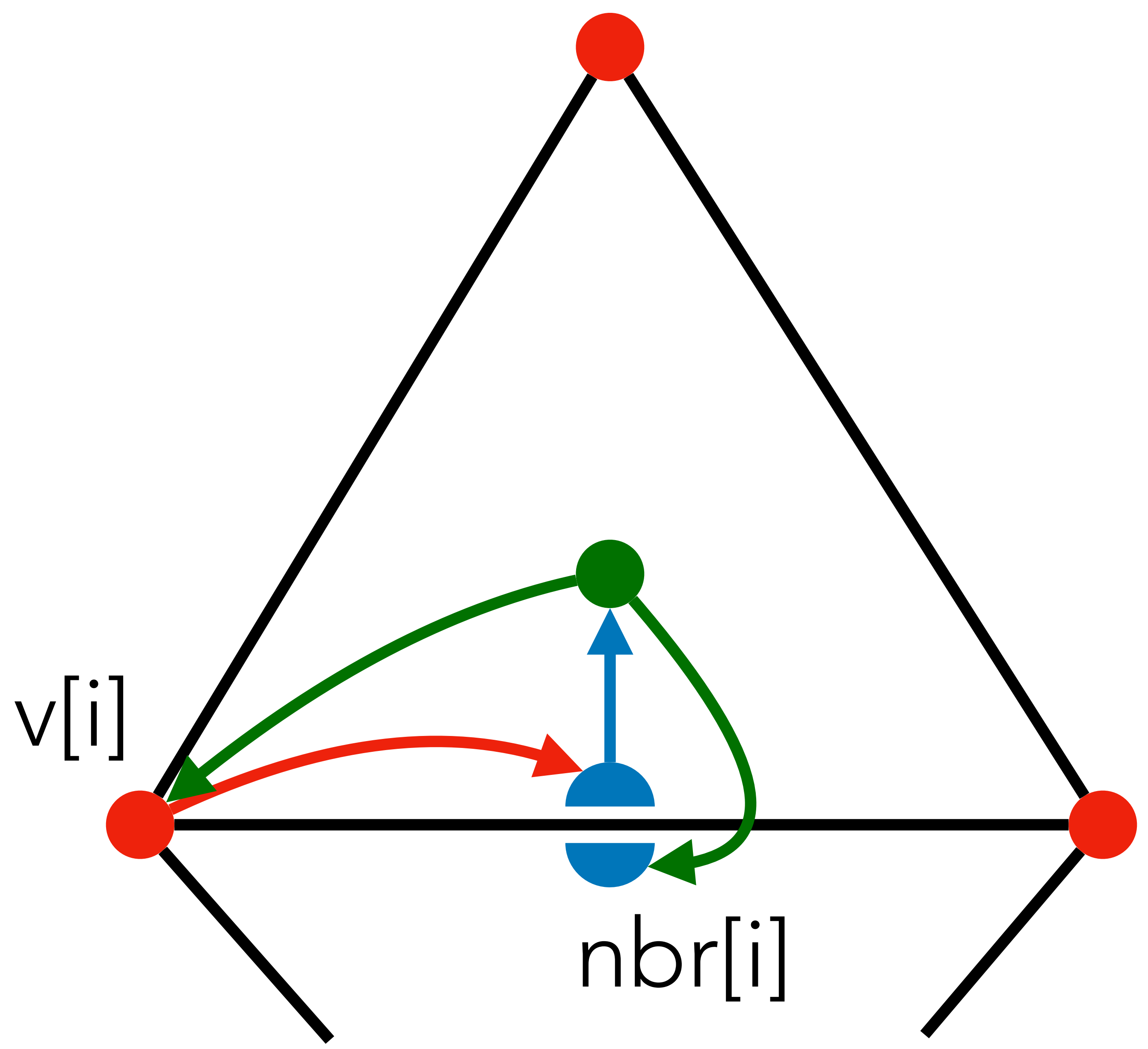


Example: Traverse all triangles adjacent to a vertex.

```
Triangle* t = v->triangle;
do {
    // do something with t
    int i = 《index of v in t->vertices》 ;
    t = t->neighbors[i];
} while (t != v->triangle);
```

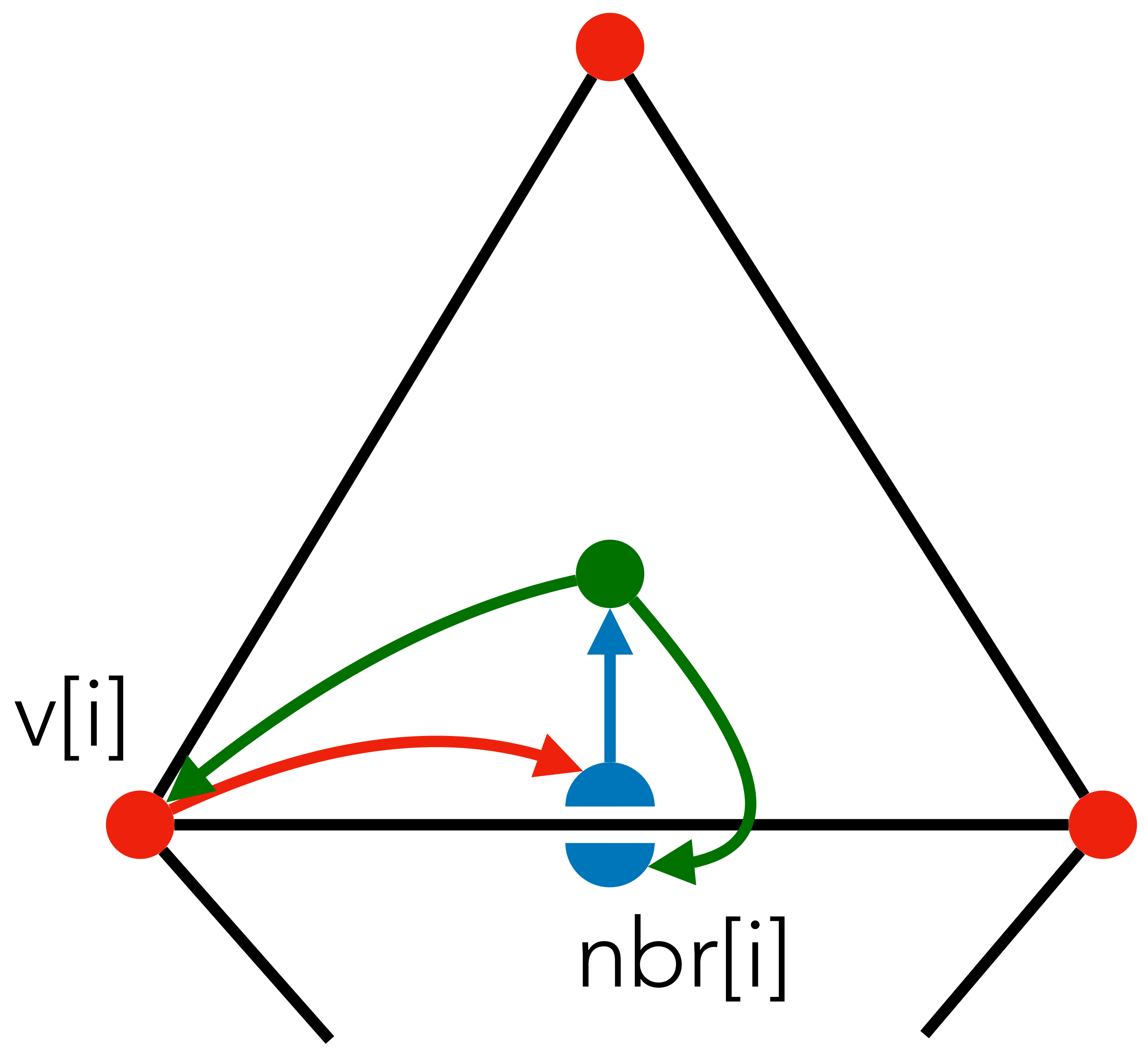


```
Vertex {
    Point position;
    Edge *edge;
}

Edge {
    Triangle *triangle;
    int index;
}

Triangle {
    Vertex *vertices[3];
    Edge *neighbors[3];
}
```



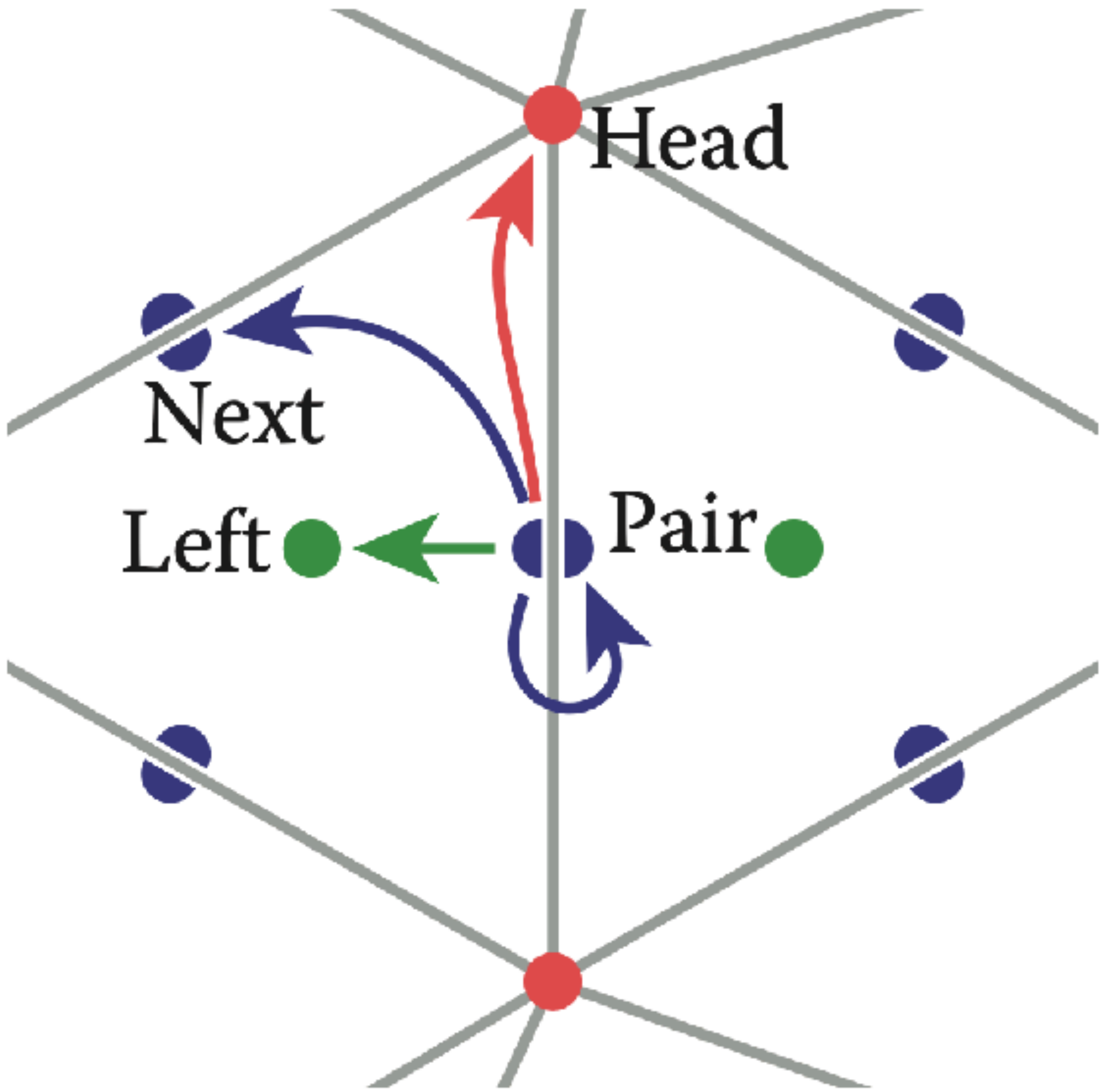
Example: Traverse all triangles adjacent to a vertex.

```
Edge* e = v->edge;
Triangle* t = e->triangle;
int i = e->index;
do {
    // do something with t
    e = t->neighbors[i];
    t = e->triangle;
    i = (e->index+1) mod 3;
} while (t != v->edge->triangle);
```

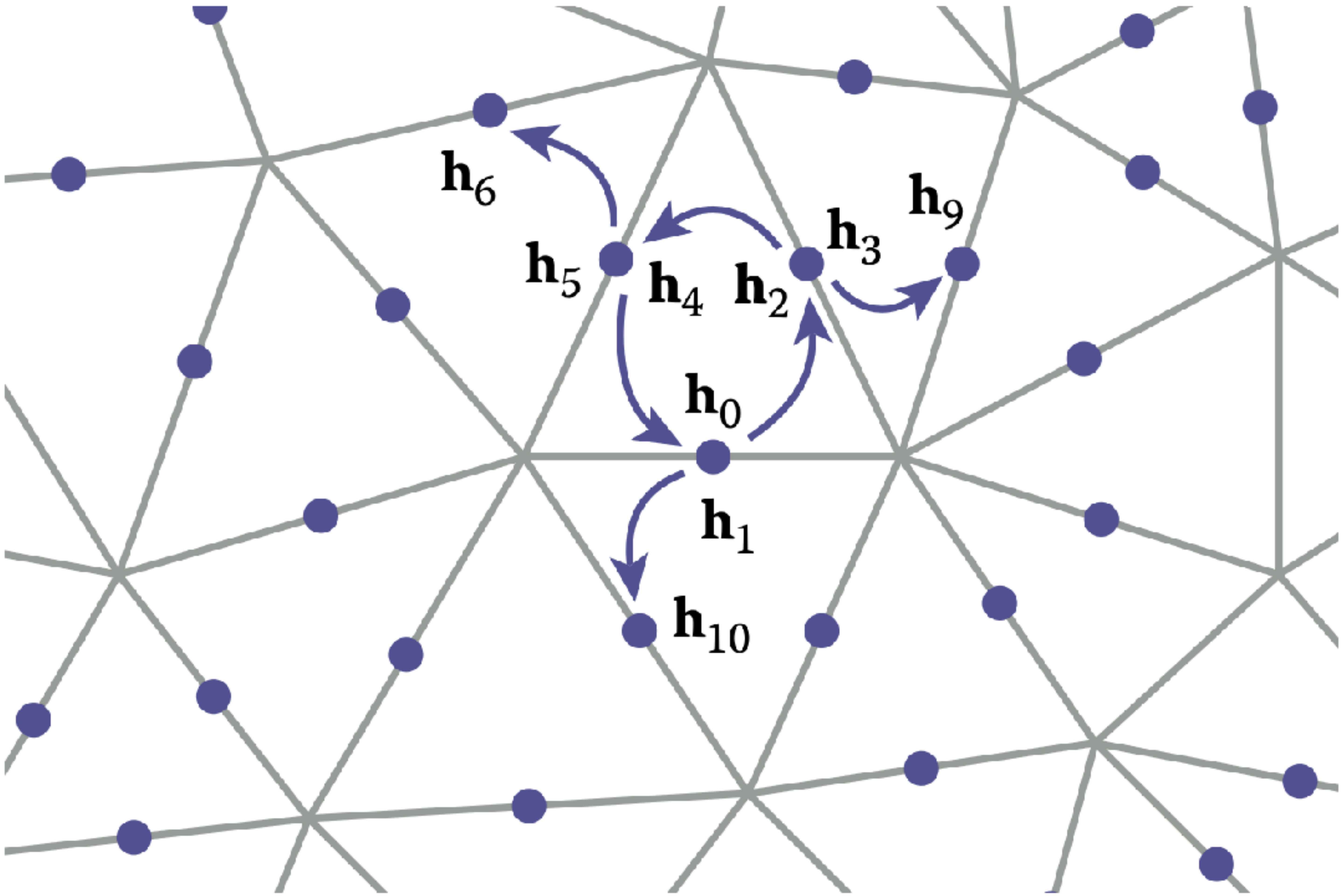


Half-edge data structure

```
HalfEdge {
    HalfEdge *pair, *next;
    Vertex *head;
    Face *left;
}

Vertex {
   HalfEdge *halfEdge;
}

Face {
   HalfEdge *halfEdge;
}
```



	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
	⋮	

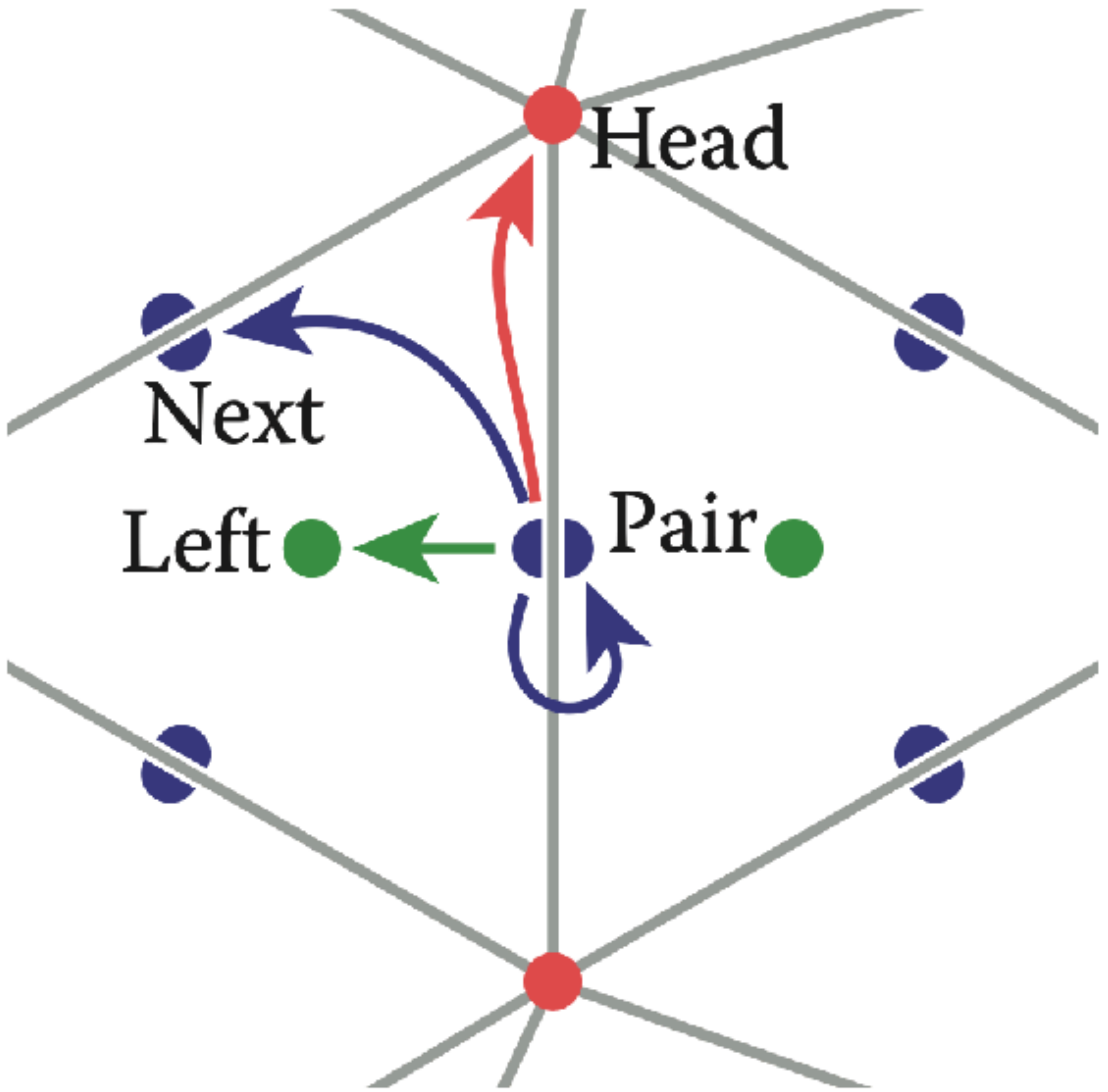


Example 1: Traverse all vertices of a face.

```
HalfEdge *h = f->halfEdge;
do {
    // do something with h->head;
    h = h->next;
} while (h != f->halfEdge);
```

Example 2: Traverse all faces adjacent to a vertex.

```
HalfEdge *h = v->halfEdge;
do {
    // do something with h->left;
    h = h->next->pair;
} while (h != v->halfEdge);
```



Practice problem

Using a half-edge representation of a triangle mesh, write (pseudo)code to find the "bending angle" at a given edge, i.e. the angle between the normals of the adjacent faces.

This is 180° minus the better-known **dihedral angle**.