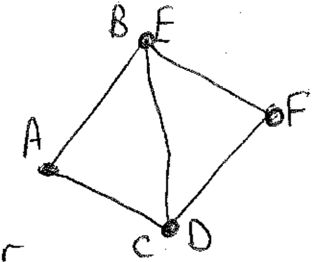
* Objects in graphics (and many other areas) are stored as triangles

* independent



* easy to render
* hard to do real processing with
* need to know our neighbors

(A, B, C), (D, E, F), ...

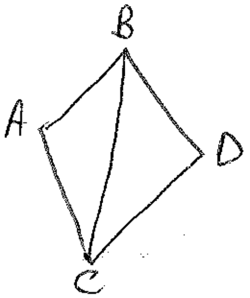
coordinates

each triangle = 9 floats

9n storage

*Better → share vertices

* triangles are vertex indices



A: ~
B: (, ,)
C: - - -
D: ~

list pts

(A B C)
(C B D)

list tris

indices

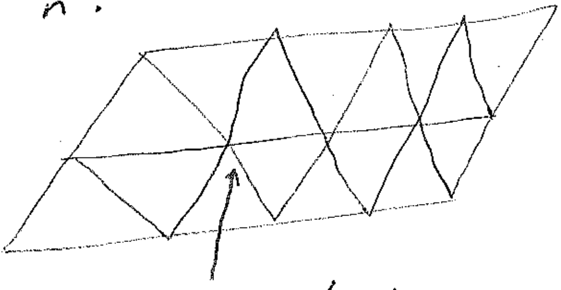
* Store
vector <vec3> pts;
vector <ivec3> tris;

$3n + 3v$

$\approx 4.5n$

about half the storage

v vs n?

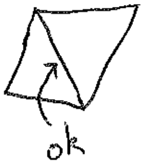


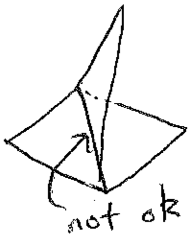
six tris/vertex
three vert/tri
$n \approx 2v$

* good representation for many general uses

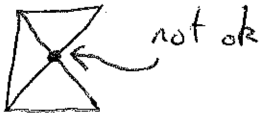
Manifold

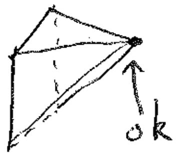
1. every edge has <u>two</u> triangles

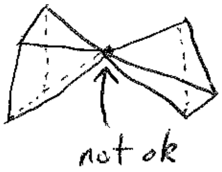




2. every vertex has a full loop of triangles





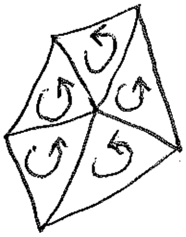


Manifold with boundary

1. edges have <u>1 or 2</u> triangles
2. every vertex has one edge-connected set of tris

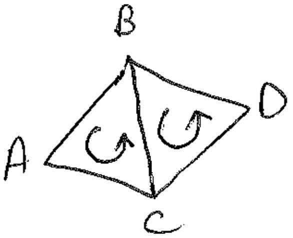
Orientation

triangle vertices listed in consistent order (usually CCW)



* note that the direction on each side of an edge is <u>reversed</u>

(ACB) or (CBA) or (BAC)



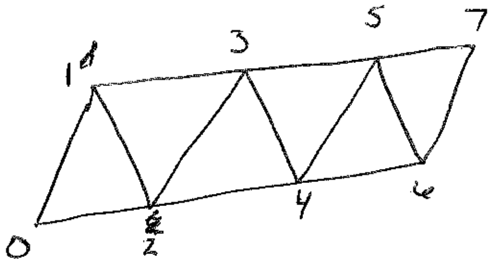
so (ACB), (BCD) is ok here.

(ABC) (BCD) is bad

wrong order

* oriented ~~Manifold~~ may not be possible if not manifold or non-orientable surface (Mobius strip)

* strips



* most of the benifit even for small strips

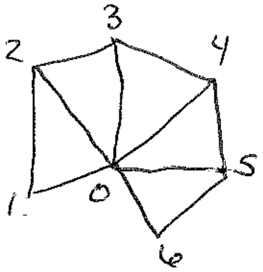
* comp~~ressions~~ storage

n triangles → $n+2$ points $\approx n$

(instead of $3n$)

$\rightarrow \left(1 + \frac{3}{2}\right) n \approx 2.5 n$

fans



Same storage size as fan, but rather limited generally to no more than 6 tri.

* opengl supports both

* mostly ~~only~~ only for storage, rendering

find neighbor

from

	T	E	V
T	TT	TE	TV easy
E	ET	EE	EV
V	VT	VE	VV

* do you need edges?

* which queries do you need?

* constant time lookup

Can store these things directly:

Tri:
    vertex v[3]   ← index or pointer to vertex
    edge e[3]

* array of Tri, Edge, Vertex

Edge:
    Vertex^{v}[2]
    Tri^{t}[2]

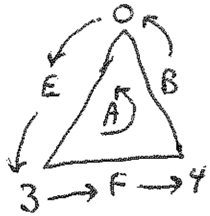
* <u>overkill.</u>

Vertex:
    Tri* t;
    Edge* e;   ← annoying - dynamic size

* only store adj triangles

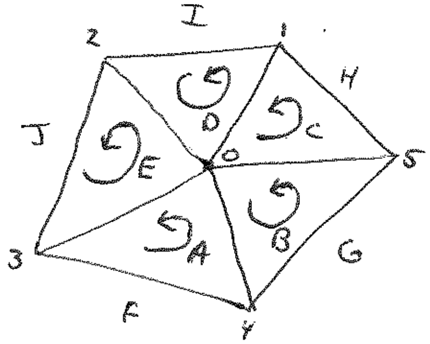
* each vertex stores arbitry triangle

eg. A:
0 34
BEF



Tri:
vertex [3] → TV
tri [3] → TT

vertex
any tri



* TT in CCW order

0: C
1: D
2: D
3: A
4: B
5: C
⋮

A: 0 34 ; BEF
B: 4 5 0 ; AGC
C: 1 0 5 ; HDB
D: 0 1 2 ; ECI
E: 3 0 2 ; JAD
⋮

* <u>no edges</u>

* TT → stored
* TV → stored
* VT → walk pointers

eg.

0 → <u>C</u> first tri

C → 1 0 5 → H D B → <u>D</u> lookup v index, use to choose next T

D → 0 1 2 → E C I → <u>E</u>

E → 3 0 2 → J A D → <u>A</u>

A → 0 34 → B E F → <u>B</u>

B → 4 5 0 → A G C → C → stop (started on C)

* for VV, output <u>next vertex</u> in triangle while doing this traversal

⑤

Half-edge structure

halfedge:

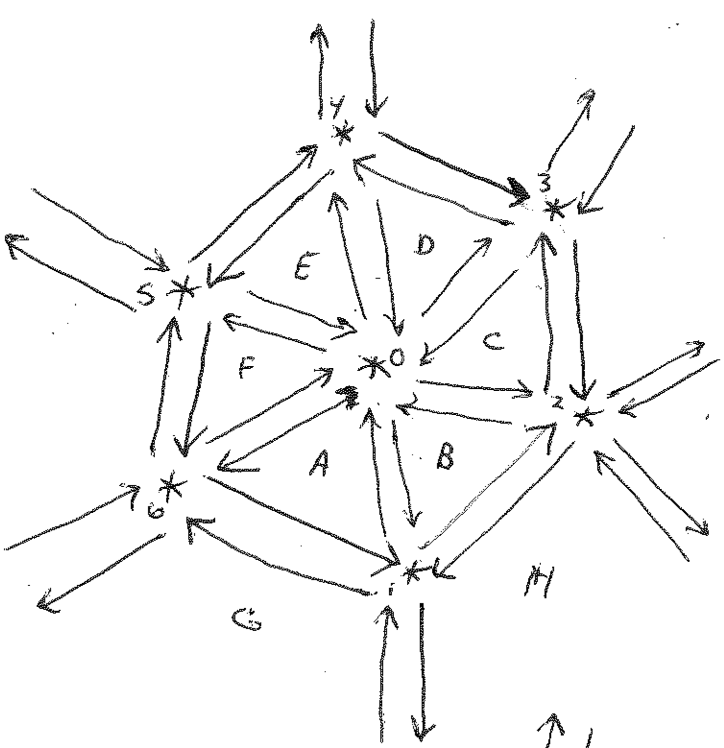
  halfedge next, prev, pair;

  ~~triangle~~ Face cell;

~~triangle~~:

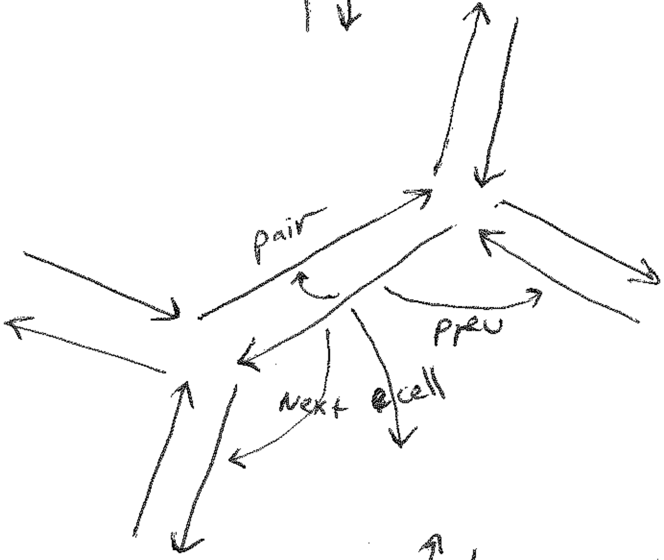
  Face:

    halfedge e;



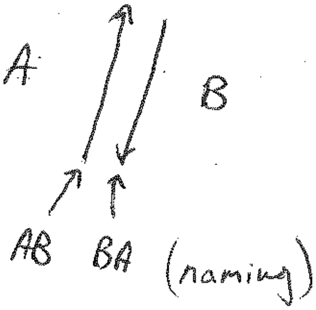
vertex:

  halfedge e;



Can answer all queries efficiently

eg: ~~vertex~~ VT



$$\begin{tikzcd}
O \arrow[r, "e"'] & BA \arrow[r, "prev"'] \arrow[d, "cell"] & BC \arrow[r, "pair"'] & CB \arrow[r] \arrow[d, "cell"] & {} \\
& \underline{B} & & \underline{C} &
\end{tikzcd}$$

$$\begin{tikzcd}
\cdots CB \arrow[r, "prev"'] & CD \arrow[r, "pair"'] & DC \arrow[r, "prev"'] \arrow[d, "cell"] & DE \arrow[r, "pair"'] & ED \arrow[r, "prev"'] \arrow[d, "cell"] & EF \cdots \\
& & \underline{D} & & \underline{E} &
\end{tikzcd}$$

$$\begin{tikzcd}
EF \arrow[r, "pair"'] & FE \arrow[r, "prev"'] \arrow[d, "cell"] & FA \arrow[r, "pair"'] & AF \arrow[r, "prev"'] \arrow[d, "cell"] & AB \arrow[r, "pair"'] & \boxed{BA} \arrow[r] & \text{stop} \\
& \underline{F} & & \underline{A} & & \text{started here.} &
\end{tikzcd}$$

* walk around a vertex

TT: walk a face

$$\begin{tikzcd}
A \arrow[r, "e"'] & AB \arrow[r, "next"'] \arrow[d, "pair"] & AF \arrow[r, "next"'] \arrow[d, "pair"] & AG \arrow[r, "next"'] \arrow[d, "pair"] & \boxed{AB} \text{ stop} \\
 & BA \arrow[d, "cell"] & FA \arrow[d, "cell"] & GA \arrow[d, "cell"] & \\
 & \underline{B} & \underline{F} & \underline{G} &
\end{tikzcd}$$

* half-edge works well with any size polygons, does not require triangles

* since half-edges come in pairs, put pairs together. pair index can be computed. $0 \rightleftarrows 1 \quad 2 \rightleftarrows 3$ etc.

* can also do vertex splitting and merging, etc. not too difficult to update structure, but must be very careful about ordering and filling in all pointers.