Triangle Meshes

CS451

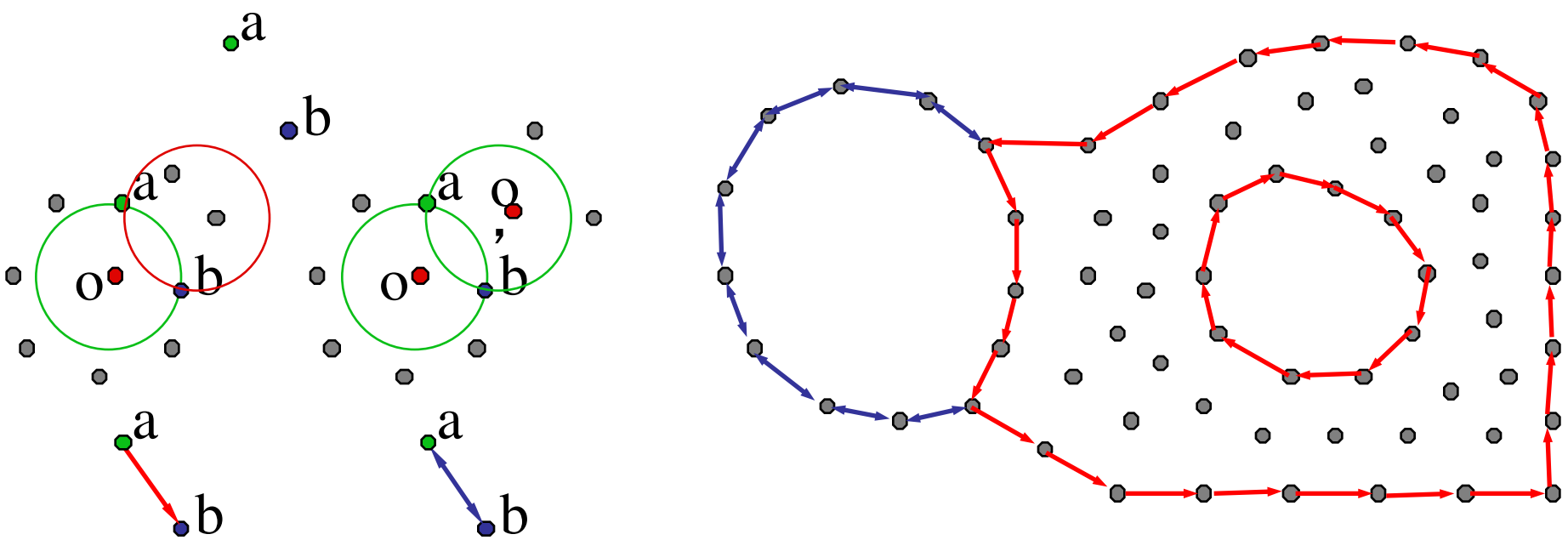
Prof. Jarek Rossignac
College of Computing
Georgia Institute of Technology

Lecture Objectives

- Learn how to **triangulate** an unstructured set of points in 3D
- Learn the **terminology**: Incidence, orientation, corner…
- Learn how to represent a simple triangle mesh using a **Corner Table** data structure
- Learn how to **build** a **Corner Table** from a Face/Vertex index file
- Learn how to implement and use the primary **operators** for traversing the mesh
- Learn how to **traverse** the mesh to estimate **surface normals** at vertices and to identify the shells
- Learn the formula for computing the **genus** of each shell
- Understand the **topological limitations** of the Corner Table and how to use it for representing meshes with holes

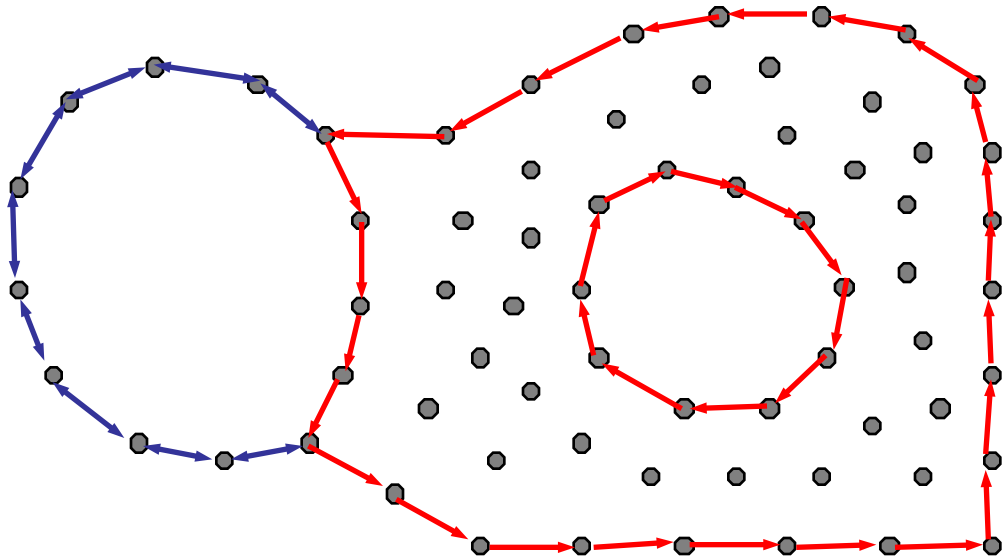
Connecting points in 2D

- Pick a radius r (from statistics of average distance to nearest point)
- For each ordered pair of points **a** and **b** such that ||**ab**||<2r, find the positions **o** of the center of a circle of radius r through **a** and **b** such that **b** is to the right of **a** as seen from **o**.
- If no other point lies in the circle circ(**o**,r), then create the oriented edge (a,b)



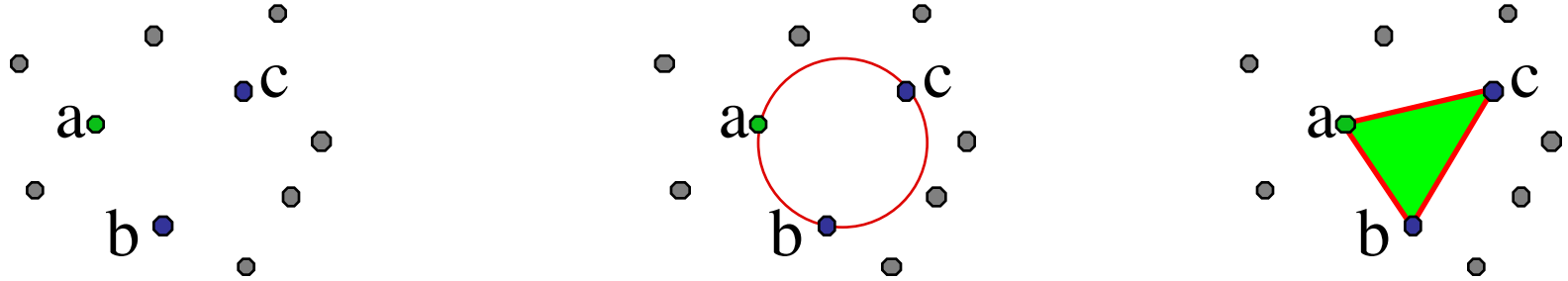
Interpreting the loops in 2D

- Chains of pairs of edges with opposite orientations (blue) form **dangling** polygonal curves that are adjacent to the **exterior** (empty space) on both sides.
- Chains of (red) edges that do not have an opposite edge form **loops**. They are adjacent to the exterior on their right and to interior on their left.
- Each **components** of the interior is bounded by one or more loops.
  - It may have **holes**



Connecting points in 3D

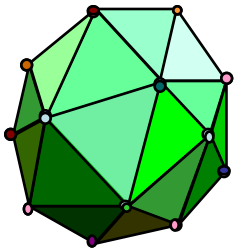
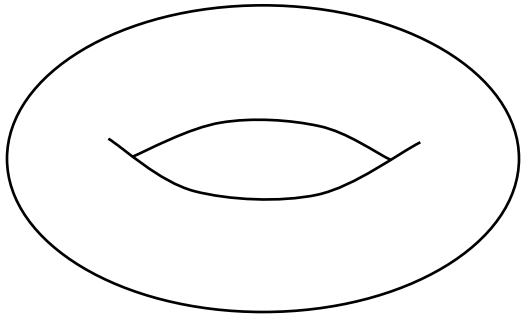
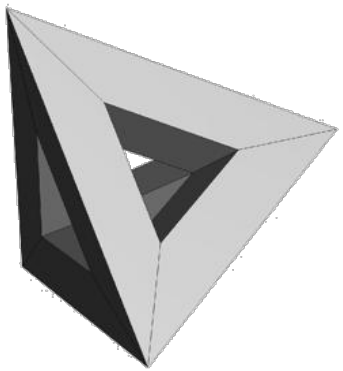
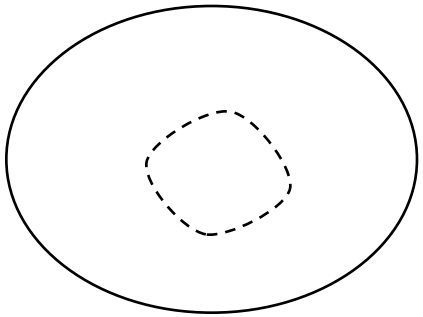
- Pick a radius r (from statistics of average distance to nearest point)
- For each ordered triplet of points **a**, **b**, and **c** if there is a point **o** such that sphere(**o**,**r**) passes through **a**, **b**, and **c** and contains no other point, then create the oriented triangle (**a**,**b**,**c**), so that they appear counterclockwise as seen from **o**.



- Each triangle has a neighbor across each edge (roll the ball)
  - Two triangles (with opposite orientation may have the same vertices)

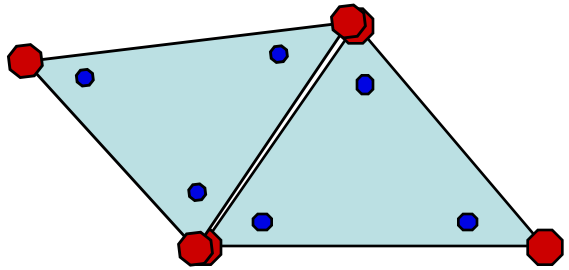
Interpreting the shells in 3D

- Pairs of triangles with opposite orientations form **dangling** surface sheets that are adjacent to the **exterior** (empty space) on both sides.
- Triangles that do not have an opposite triangle form **shells** where each edge has two adjacent triangles. They are adjacent to the exterior on their side from which the triangles appear counterclockwise and to interior on the other side.
  - Each shell may have zero (sphere), one (torus), or more handles (through-holes)
- Each **components** of the interior (**solid**) is bounded by one or more shells.
  - It may have internal **cavities** (holes bounded by their own shell) that are not accessible from the outside

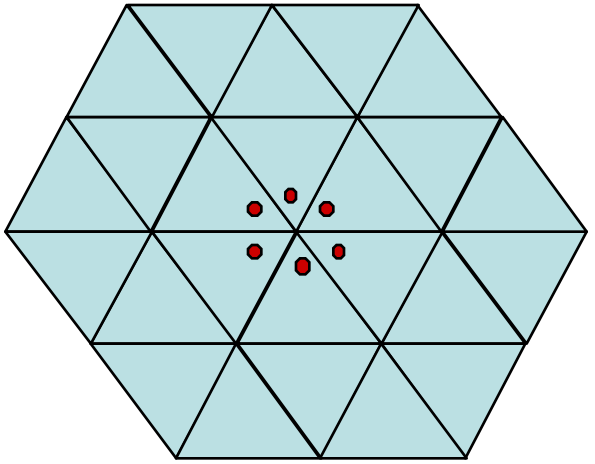





Representation as **independent** triangles

- For each triangle:
  - Store the location of its 3 vertices



	vertex 1			vertex 2			vertex 3		
Triangle 1	x	y	z	x	y	z	x	y	z
Triangle 2	x	y	z	x	y	z	x	y	z
Triangle 3	x	y	z	x	y	z	x	y	z



- Each vertex is repeated 6 times (on average)
- Expensive to identify an adjacent triangle
  - Not suited for traversing a mesh

Representing **vertices + incidence**

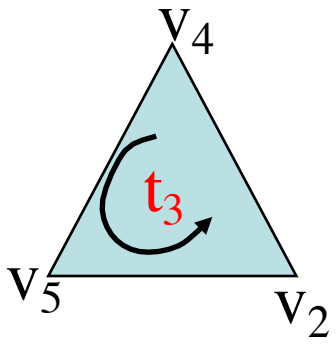
- **Samples:** Location of vertices + attributes (color, mass)
- Triangle/vertex **incidence:** specifies the indices of the 3 vertices of each triangle
  - Eliminates vertex repetition
  - But still does **not** support a direct access to neighboring triangles (**adjacency**)

Samples (vertices):

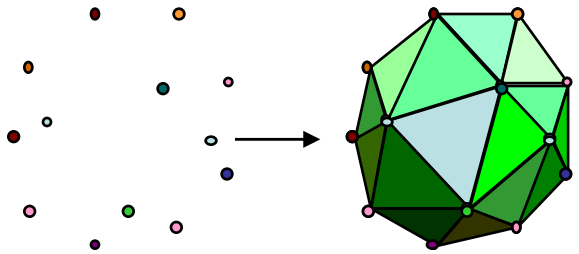
vertex 1	x	y	z	c
vertex 2	x	y	z	c
vertex 3	x	y	z	c
. . .	. . .			

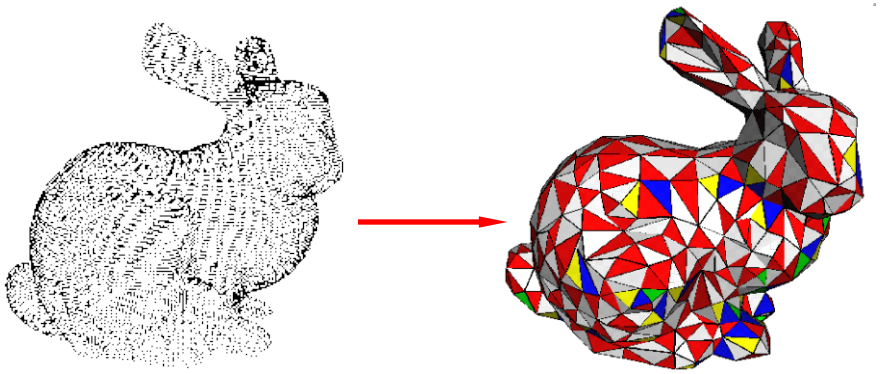
Triangle/vertex incidence:

Triangle 1	1	2	3
Triangle 2	3	2	4
Triangle 3	4	5	2
Triangle 4	7	5	6
Triangle 5	6	5	8
Triangle 6	8	5	1
. . .	. . .		



Order of vertex references defines outward direction (triangle orientation)

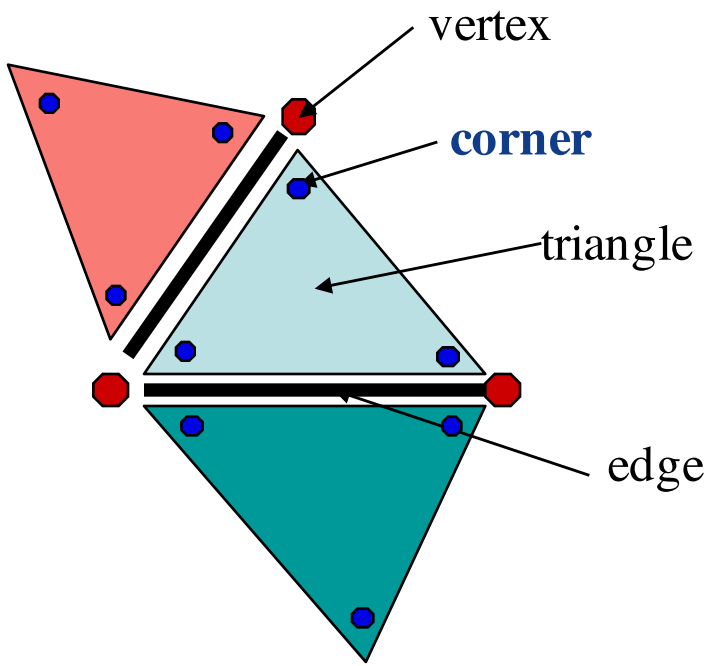




Corners, incidence and **adjacency**

Triangle/vertex incidence: identifies **corners**

- **Corner:** Abstract **association** of a **triangle** with a **vertex** (vertex-use)
  - A triangle has 3 corners
  - On average, 6 corners share a triangle



Triangle/triangle adjacency: Identifies neighboring triangles

- ✓ Neighboring triangles share a common edge
- ✓ Adjacency may be computed from the incidence
- ✓ Adjacency is convenient to **accelerate traversal** of triangulated surface
  - Walk from one triangle to an adjacent one
  - Estimate surface normals at vertices

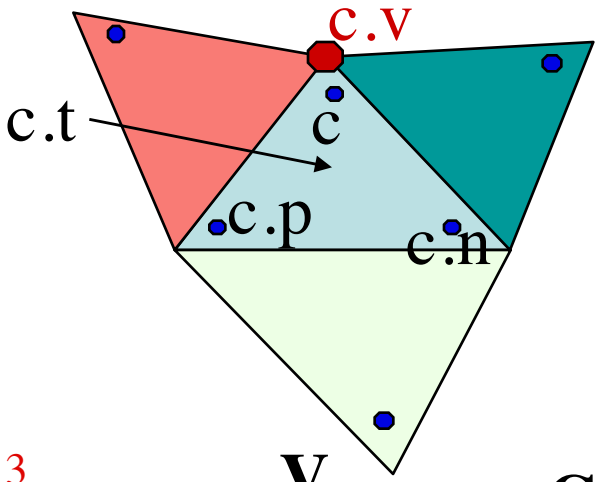
We will use the **Corner Table** to represent incidence and adjacency

Representing the incidence as the **V table**

- Integer IDs for **vertices** (0, 1, 2… V-1) & **triangles** (0, 1, 2…T-1)

- **V-table:**
  - Identifies the **vertex ID** c.v for each **corner** c
  - The 3 corners of a triangle are **consecutive**
    - Triangle **number**: c.t = c DIV 3
  - Corners order for a triangle respects **orientation**
    - Cyclic order in which corners are listed
    - Next corner **around triangle**: c.n = 3 c.t + (c+1)MOD 3
    - Previous corner: c.p=c.n.n

- **Samples stored in geometry table (G):**
  - Location of vertex v is denoted v.g
    - Location of vertex of corner c is denoted c.v.g
      - Implementation as arrays: G[V[c]]
  - G tables list coordinates for vertex v
    - v.g = (v.g.x, v.g.y, v.g.z)
    - or use short cut: (v.x, v.y, v.z)



V

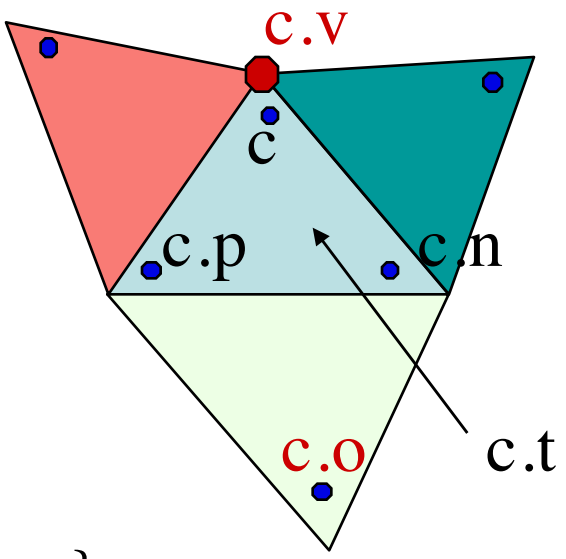
Triangle 0	1
Triangle 0	2
Triangle 0	3
Triangle 1	2
Triangle 1	1
Triangle 1	4
Triangle 2	1

G

vertex 1	x	y	z
vertex 2	x	y	z
vertex 3	x	y	z
vertex 4	x	y	z

Representing **adjacency** with the O table

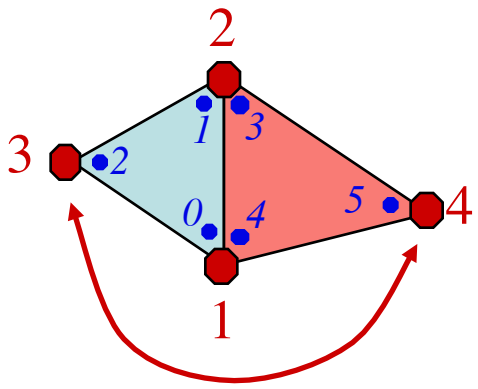
- For each corner **c** store:
  - **c.v** : integer reference to an entry in the G table
    - Content of V[c] in the V table
  - **c.o** : integer id of the opposite corner
    - Content of O[c] in the O table
- Computing the O table from V



For each corner a do For each corner b do

if (a.n.v==b.p.v && a.p.v==a.n.v) { O[a]:=b; O[b]:=a } ;

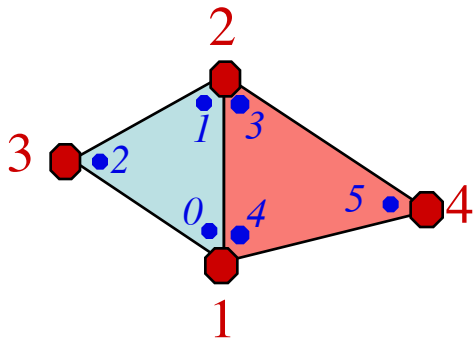
	V	O
Triangle 0 corner 0	1	7
Triangle 0 corner 1	2	8
Triangle 0 corner 2	3	5
Triangle 1 corner 3	2	9
Triangle 1 corner 4	1	6
Triangle 1 corner 5	4	2



	x	y	z
vertex 1	x	y	z
vertex 2	x	y	z
vertex 3	x	y	z
vertex 4	x	y	z

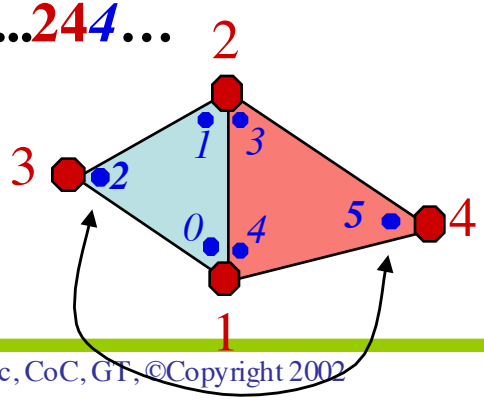
A faster computation of the O table

1. **List all of triplets {min(c.n.v, c.p.v), max(c.n.v, c.p.v), *c*}**
   - **230, 131, 122, 143, 244, 125, …**



	v	o	a
Triangle 1 corner 0	1		a
Triangle 1 corner 1	2		b
Triangle 1 corner 2	3		c
Triangle 2 corner 3	2		c
Triangle 2 corner 4	1		d
Triangle 2 corner 5	4		e

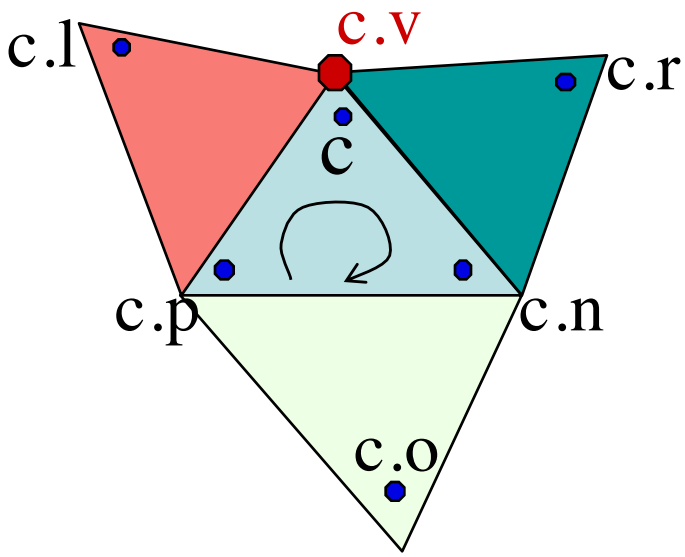
2. **Bucket-sort the triplets:**
   - **122, 125 …131… 143 …230…244 …**
3. **Pair-up consecutive entries 2k and 2k+1**
   - **(122, 125)…131… 143…230…244…**
4. **Their corners are opposite**
   - **(122,125)…131…143…230…244…**



	v	o	a
Triangle 1 corner 0	1		a
Triangle 1 corner 1	2		b
Triangle 1 corner 2	3	5	c
Triangle 2 corner 3	2		c
Triangle 2 corner 4	1		d
Triangle 2 corner 5	4	2	e

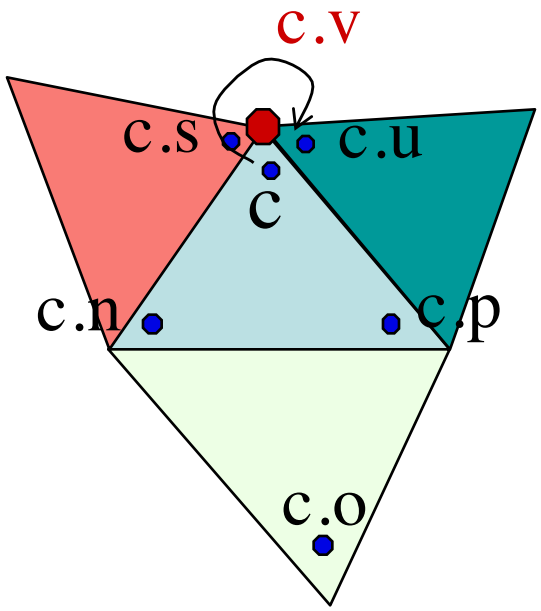
Accessing left and right neighbors

- Direct access to opposite corners of right and left neighbors
  - **c.l = c.n.o**
  - **c.r = c.p.o**

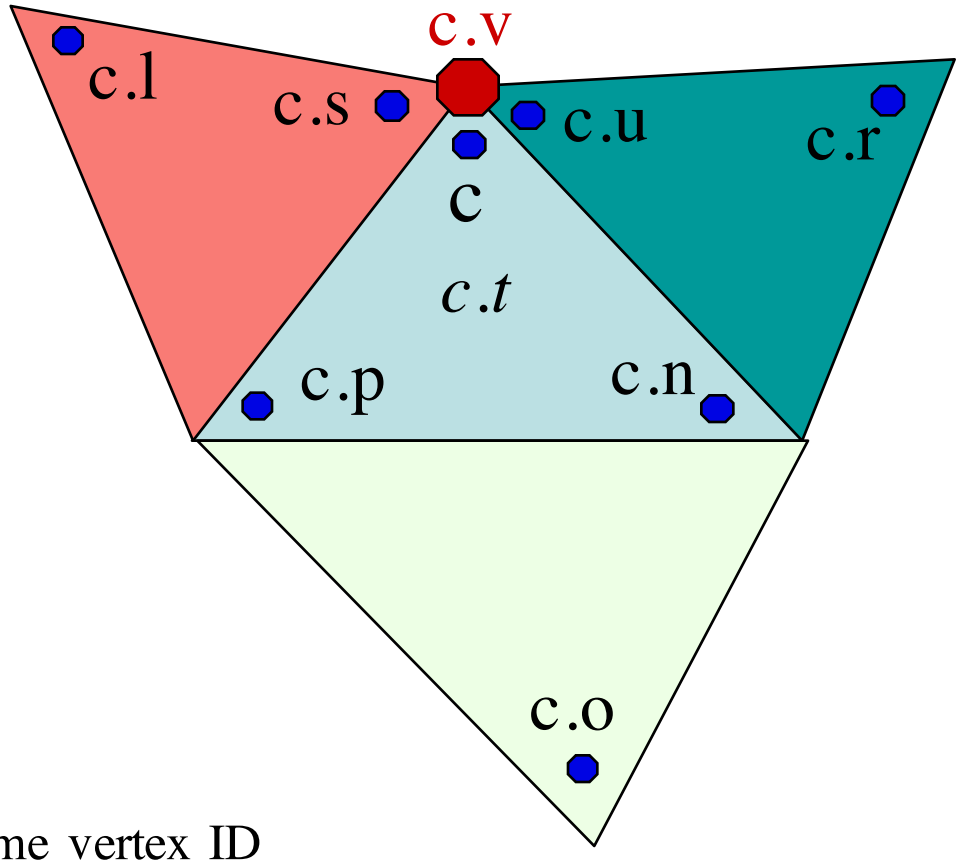


Accessing swing and unswing neighbors

- Useful for walking around a vertex
  - **c.s = c.n.o.n "swing"**
  - **c.u = c.n.s.n "unswing"**



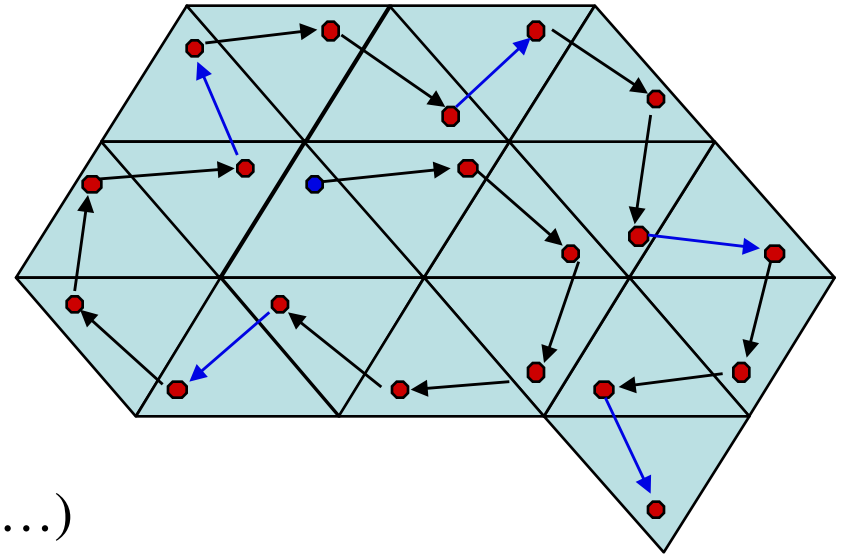
Summary notation

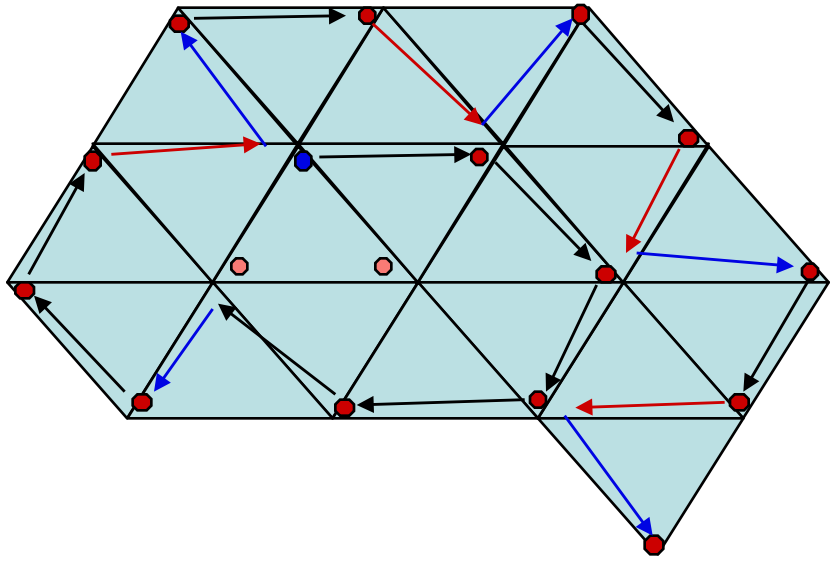


- Given corner c
  - c.v is the integer ID of its **vertex**
    - On average, 6 corners have the same vertex ID
  - c.v.g is the 3D point where c.v is located (**geometry**)
    - Must use .g for vector operations. Ex: c.n.v.g – c.v.g is vector along edge

Using adjacency table for T-mesh traversal

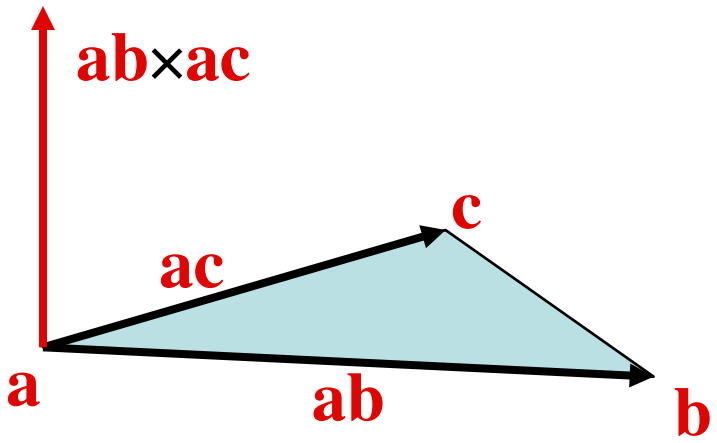
- **Visit T-mesh**
  - Mark triangles as you visit them
  - Start with any corner c and call Visit(c)
  - Visit(c)
    - mark c.t;
    - IF NOT marked(c.r.t) THEN visit(c.r);
    - IF NOT marked(c.l.t) THEN visit(c.l);
- **Label vertices** (for example as 1, 2, 3 …)
  - Label vertices with consecutive integers
  - Label(c.n.v); Label(c.n.n.v); Visit(c);
  - Visit(c)
    - IF NOT labeled(c.v) THEN Label(c.v);
    - mark c.t;
    - IF NOT marked(c.r.t) THEN visit(c.r);
    - IF NOT marked(c.l.t) THEN visit(c.l);

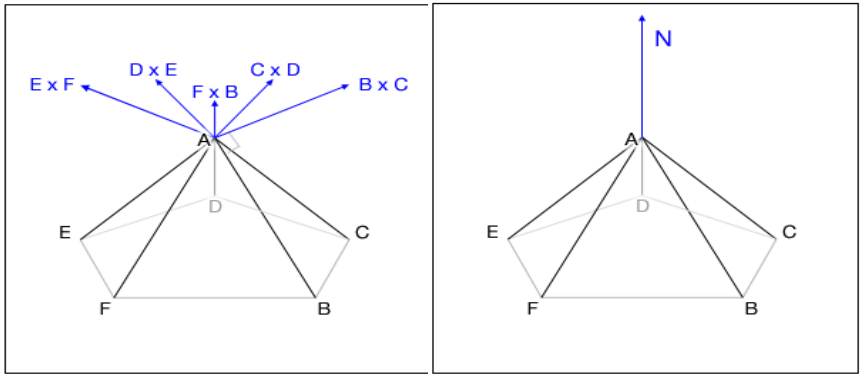




Estimating a vertex normal

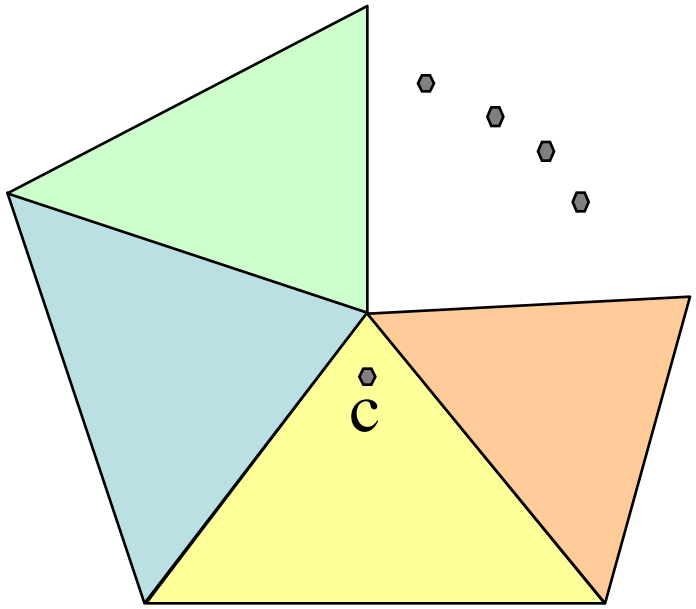
- At vertex **a** having **b**, **c**, **d**, **e**, **f** as **neighbors**
- **N=ab×ac+ac×ad+ad×ae+ae×af+af×ab**
  - The notation **U×V** is the **cross product** of the two vectors
  - The notation **ac** is the vector between **a** and **c**. In other words: **ac=c–a**
  - Note that **N** is independent of the position of vertex **a**
    - **(b–a)×(c–a)+(b–a)×(c–a)+… = b×c+a×a–b×a–a×c + c×d+a×a–c×a–a×d +… –a×b +…**
    - **a×a = 0** , **–a×c** and **–c×a** cancel out, same for all other cross-products containing **a**
    - We are left with = **b×c + c×d** + … which does not depend on **a**
- Then **divide N** by its norm to make it a unit vector





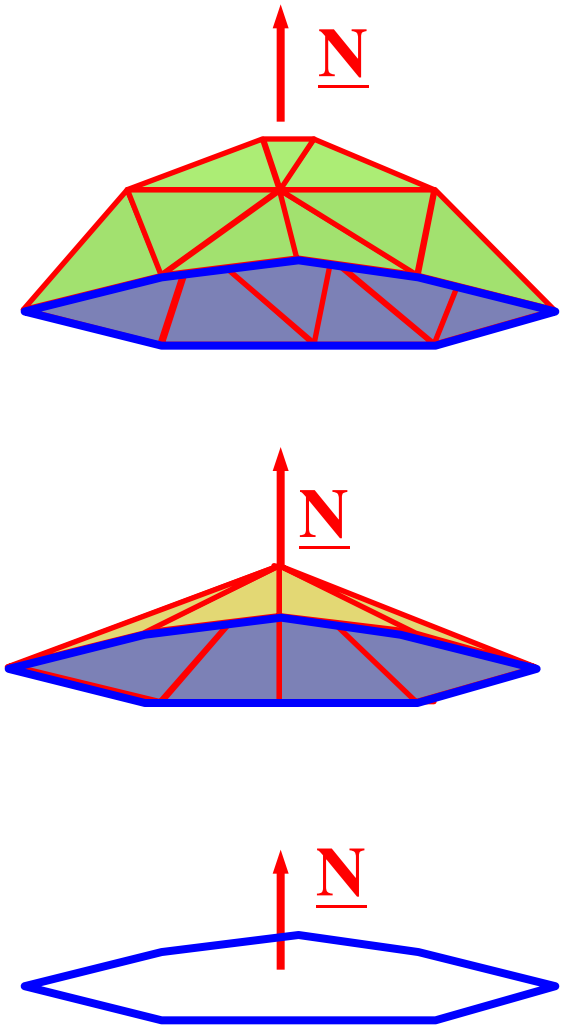
Exercise

- Given corner c, estimate the surface normal **n** at vertex c.v
  - Use the sum of cross-products approach proposed above to compute **N**
  - Then normalize it to get **n**
- Remember that **n** is independent of the position c.v.g of c.v



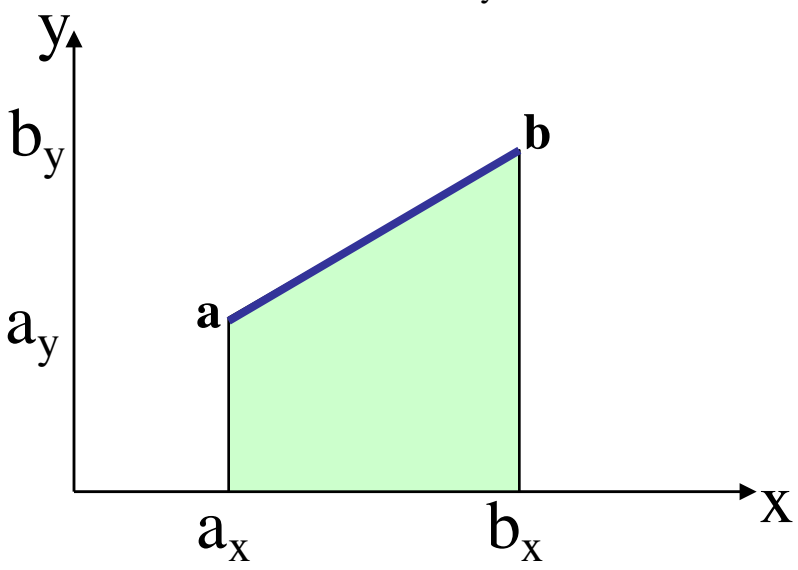
Computing the normal to a patch

- Consider a non-planar **patch** of triangles in 3D
- Consider its border (blue curve)
  - Border edges have a single incident triangle
- Estimate the normal **N** to the patch as the sum of the normals to its triangles weighted by the triangle areas (**ab**×**ac**).
- The result is independent on the position of the internal vertices (proof?)
- It only depends on the border.
- We could make a triangle-fan with some point **o** and compute **N**
  - Sum **oa**×**ob** for all border edges (**a**,**b**)
- Or we can use the projections of the border edges on in the x, y, and z directions…

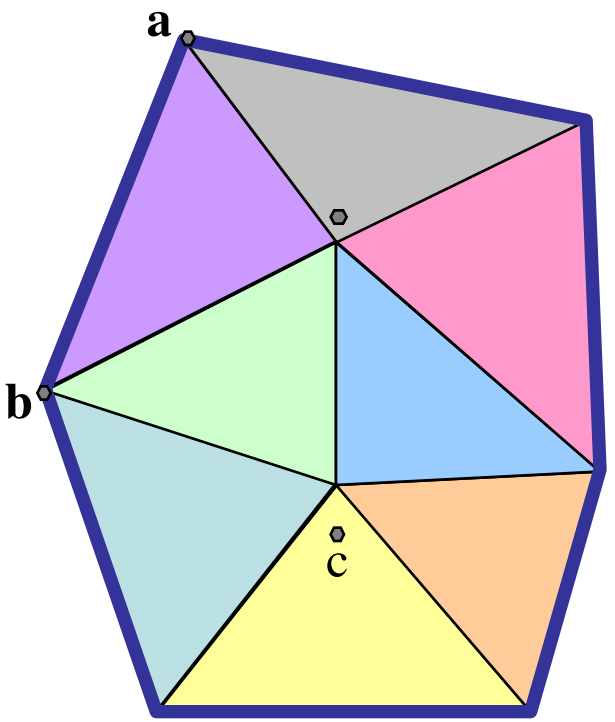


Faster computation of the normal N to a patch

- **N** may be computed from the projections of the border edges (**a**,**b**) onto the 3 principal planes:
- Compute signed areas of "shadows" of the border loop on the YZ, ZX, and XY planes
  - $\mathbf{N}_z$:= the sum of signed areas of 2D trapozoids under the projection of (**a**,**b**), for each border edge (**a**,**b**).
  - Same for $\mathbf{N}_x$ and $\mathbf{N}_y$



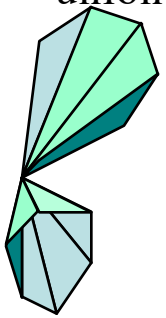
$$\text{area}(\mathbf{a},\mathbf{b}) := (a_y + b_y)(b_x - a_x)$$



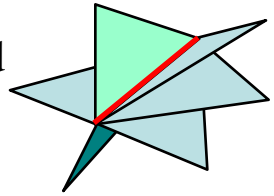
Assume T-mesh is an **orientable manifold**

- A set of triangles forms a **manifold** mesh when:
  - The 3 **corners** of a triangle refer to **different vertices** (not zero area)
  - Each **edge bounds** exactly **2 triangles**
  - **The** star **of each vertex v** forms a single cone (connected if we remove **v**)
    - **Star** = union of edges and triangles incident upon the vertex

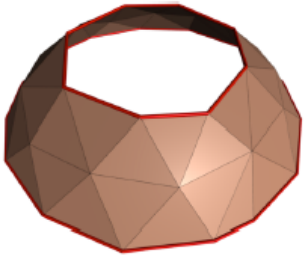
Non-manifold vertex



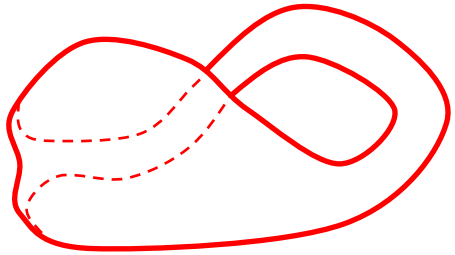
Non-manifold edge



border edges



- A manifold triangles mesh is **orientable** when:
  - Triangle can be oriented consistently

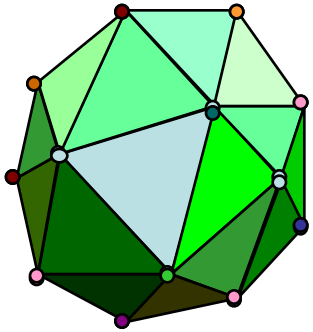


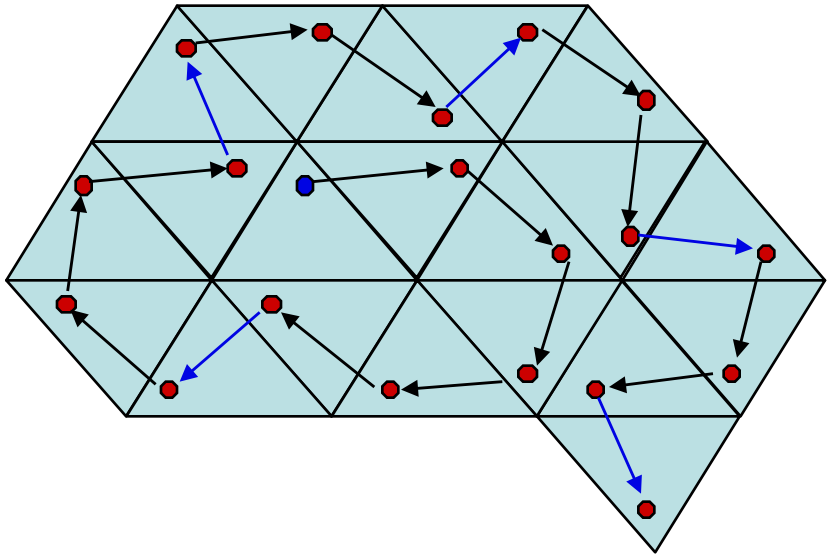
Klein bottle

- **Pseudo-manifold**
  - Start with a Manifold T-mesh
  - Displacing the **geometry** (vertices)
  - Different vertices may have the same location
  - Mesh may self-intersect

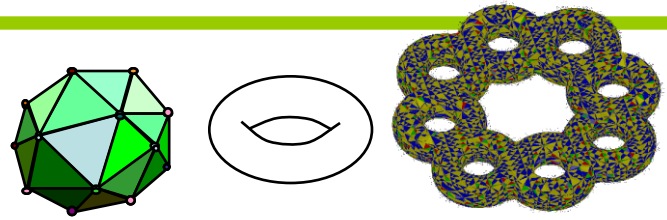
Shells: connected portions of T-meshes

- All triangles of a **shell** form a **connected** set
  - Two adjacent triangles are connected (through their common edge)
  - Connectivity is a transitive relation (can identify a mesh by invading it)
- **To identify a new shell**
  - Pick a new "color" (ID) and a virgin triangle
  - Use the Visit(c) procedure to reach all of the triangle of the shell and paint them

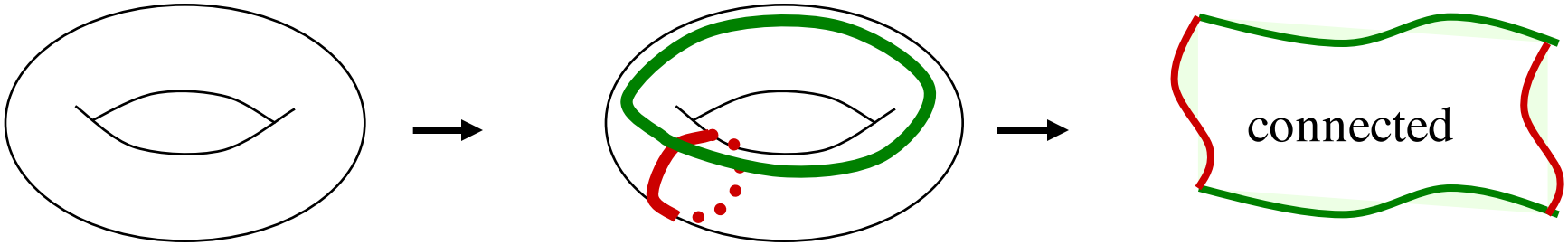




Genus (number of handles) in a shell



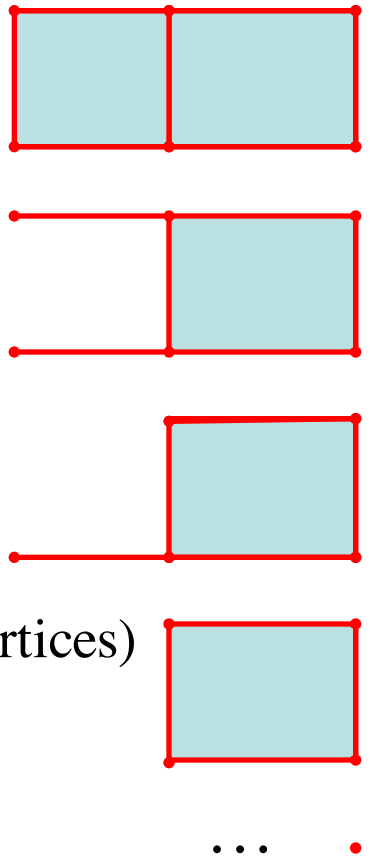
- **Handles** correspond to **through-holes**
  - A sphere has zero handles, a torus has one
- The **number G of handles** is called the **genus** of the shell
  - A handle cannot be identified as a particular set of triangles
- A T-mesh has **G handles** if and only if you can **remove at most 2G edge-loops without disconnecting** the mesh



- Genus of a shell may be computed using: **G = T/4 – V/2 + 1**
  - Remember as **T = 2V – 4 + 4G (note sign!)**
- In a zero-genus mesh, **T = 2V – 4** (from Euler-Poincare formula)

Proof of Euler's formula: **F–E+V =2**

- Formula attributed to Descartes (1639)
- Euler published first proof (1751)
- Used in Combinatorial Topology founded by Poincaré (c.1900)
- Cauchy's proof (1811)
  - Take a **zero-genus** polyhedron
    - Faces may have arbitrary number of edges
  - Assume it has F faces, E edges, V vertices
  - Remove one initial face
  - Repeatedly apply one or the other destructor
    - Remove one edge and one face
    - Remove one edge and one vertex
  - All preserve connectivity and F–E+V
  - All keep closed cells (faces with borders, edges with end-vertices)
  - You end up with a single vertex and an unbounded face
  - So: F–E+V=2

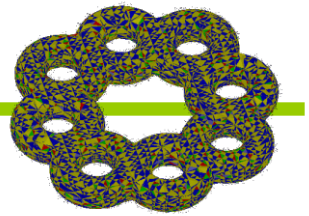


Applying V–E+F=2 to triangle meshes

- Euler formula for zero genus manifold polyhedron: F–E+V=2
- Triangle mesh: each face has 3 edges
  - We will use T to represent the number of faces: **T–E+V=2**
  - Let's count edge-uses U
  - Each triangle uses 3 edges: U = 3T
  - Each edge is used twice: U = 2E
  - Therefore: E = 3T / 2
    - Substituting 3T / 2 for E: **T–3T/2+V=2**
- Multiply by 2: **2T–3T+2V=4**
- Hence: **T = 2V – 4**
  - **There are roughly twice as many triangles as vertices!**
  - Check on a tetrahedron: T=4 and V=4

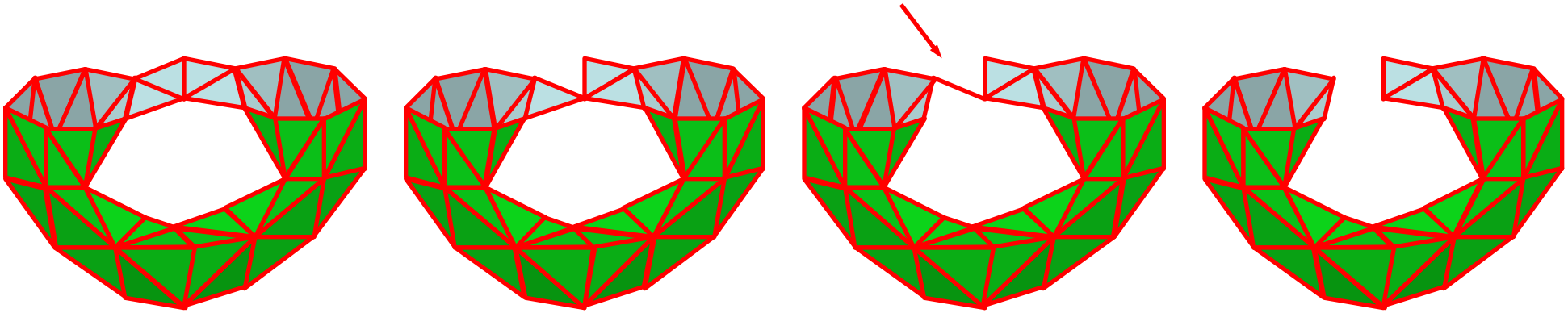
Note that 3T/2=E=T+V–2

Justifying T = 2V – 4 + 4H

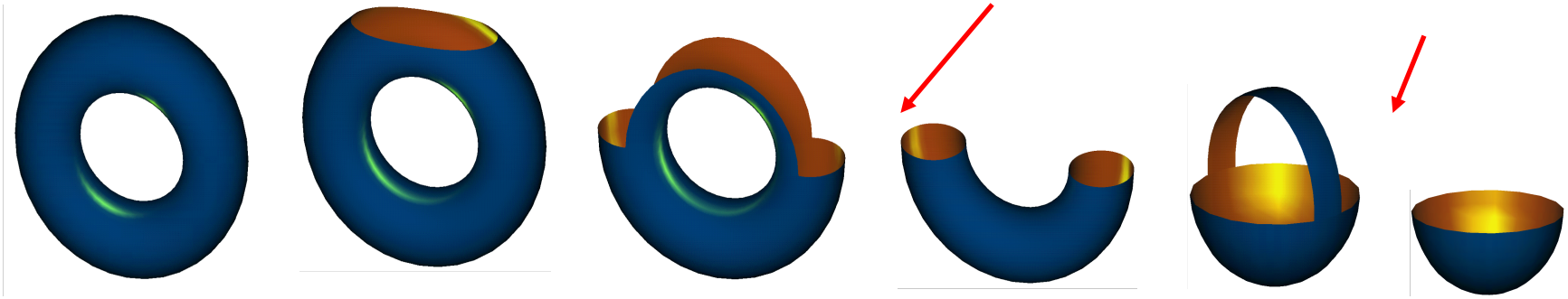


Apply Cauchy's proof to a triangle mesh with 1 handle

We must delete a **bridge** edge without deleting a vertex or a triangle



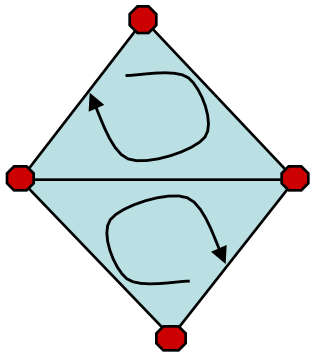
We have **2 bridge edges per handle**

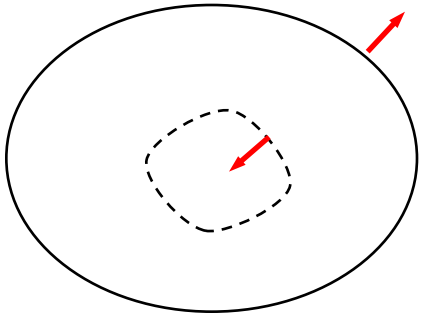


Now 3T/2=E=T+V–2+2H. Hence: **T = 2V – 4 + 4H**

Solids and cavities

- A **solid** (here restricted to be a connected manifold polyhedron) may be represented by its **boundary**, which may be composed of one or more manifold shells
  - One shell defines the external boundary
  - The other shells define the boundaries of internal cavities (**holes**)
- All the shells of a solid can be consistently **oriented**
  - It you were a bug sitting on the **outward side** of an oriented triangle you would have to turn clockwise (with respect to your up vector normal to the triangle) to look at the vertices in the order in which their IDs are stored in the V table
  - The outward side of each triangle must be adjacent to the exterior of the solid.



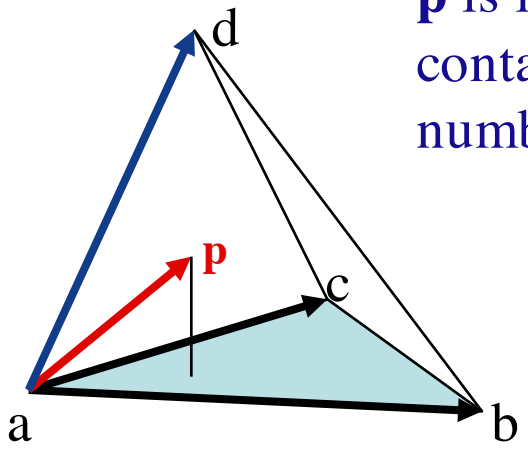


Point-in-solid test

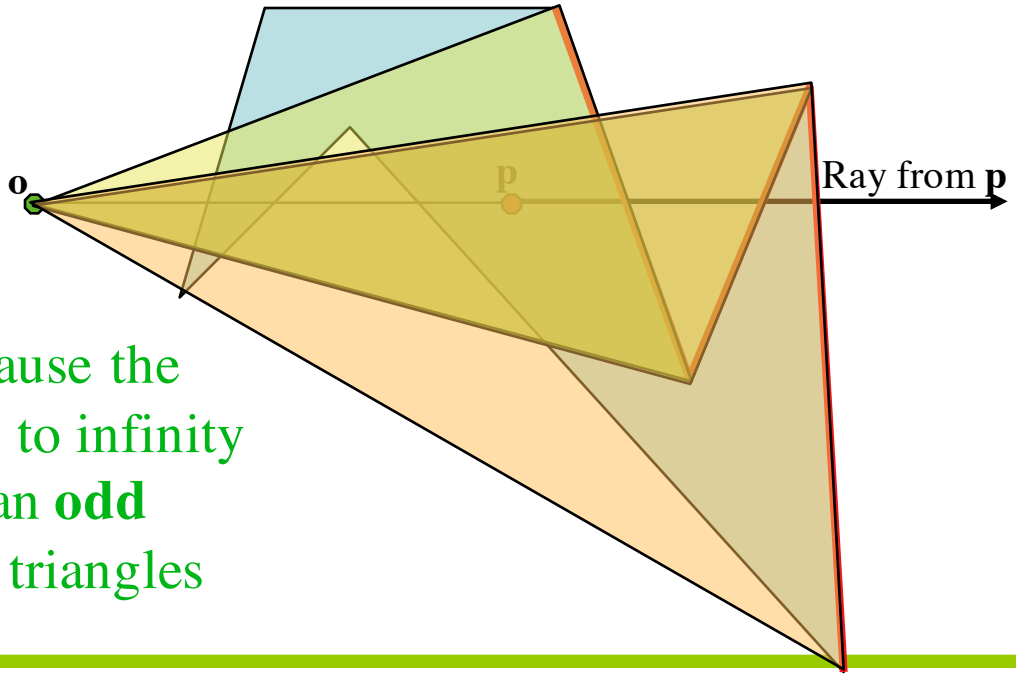
- A point **p** lies inside a solid S bounded by triangles T_i when p lies inside an **odd** number of tetrahedra, each defined by an arbitrary point **o** and the 3 vertices of a different triangle T_i.
  - Remember that point-in-tetrahedron test may be implemented as:

    PinT(**a**,**b**,**c**,**d**,**p**) := *same*(*s*(**a**,**b**,**c**,**d**),*s*(**p**,**b**,**c**,**d**),*s*(**a**,**p**,**c**,**d**),*s*(**a**,**b**,**p**,**d**),*s*(**a**,**b**,**c**,**p**))

    where s(**a**,**b**,**c**,**d**) returns (**ab**×**ac**)•**ad** > 0
  - The test does not assume proper orientation of the triangles!



p is in because it is contained in an **odd** number of tetrahedra



p is in because the ray from **p** to infinity intersects an **odd** number of triangles

Rationale

Consider an oriented ray R from o through p.

Assume for simplicity that it does not hit any vertex or edge of the mesh.

A point at infinity along the ray is OUT because we assume that the solid is finite.

Assume that the portion of the ray after p intersects the triangle mesh k times.

If we walk from infinity towards p, each time we cross a triangle, we toggle classification.

So, p is IN if and only if k is odd.

Let T denote the set of triangles hit be the portion of the ray that is beyond p.

Let H denote the set of tetrahedra that contain p and have as vertices the point o and the 3 vertices of a triangle in the mesh.

To each triangle of T corresponds a unique tetrahedron of H.

So, the cardinality of T equals the cardinality of H.

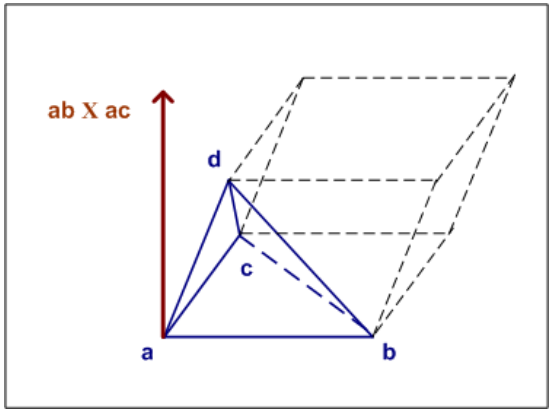
Hence we can use the parity of the cardinality of H.

Volume of a solid

- Given a solid S, bounded by **consistently oriented** triangles T_1, T_2, … T_n, let H_i denote the tetrahedron having as vertices an arbitrary origin **o** and the three vertices ($\mathbf{b_i}$,$\mathbf{c_i}$,$\mathbf{d_i}$) of T_i.

- The **volume** of S is one sixth of the sum of v(**o**,$\mathbf{b_i}$,$\mathbf{c_i}$,$\mathbf{d_i}$), for all i.
  - v(**o**, $\mathbf{b_i}$, $\mathbf{c_i}$, $\mathbf{d_i}$) returns $(\underline{\mathbf{ob}_i} \times \underline{\mathbf{oc}_i}) \bullet \underline{\mathbf{od}_i}$
  - Note that the volume of S is independent on the choice of **o**
  - Note that this evaluation requires that the triangles be consistently **oriented**
  - The formula also works for triangulated boundaries of non-manifold solids, provided that the orientation is consistent with the outward orientation of the faces.

- Applications:
  - Physically plausible simulation
  - Product design and optimization
  - Volume preserving 3D morphs and simplification



Examples of questions for tests

- Define incidence, adjacency, corner, shell, solid, genus
- Difference between handle (through-hole) and hole (cavity)
- Explain the content of a corner table
- Provide the implementation of the corner operators: c.v, c.o, c.t, c.n, c.p, c.r, c.l
- How can we identify the corner opposite to c?
- Explain how to build a Corner Table from a list of triangles?
- How to identify the shells of a mesh represented by a corner table?
- How to compute the genus (number fo handles) of each shell?
- Can we represent solid by its bounding triangles (not-oriented)
- How to test whether a vertex lies inside a solid
- How to compute the volume of a solid

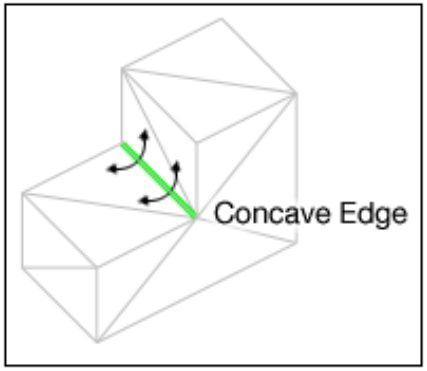
Questions to think about

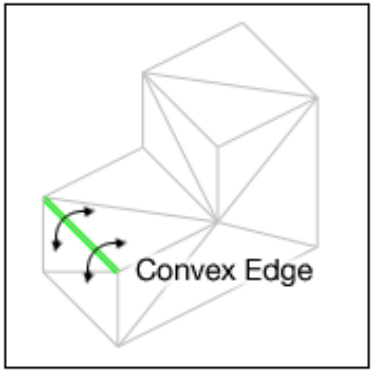
- How to pick the proper outward orientation for a triangle
- How to consistently orient the triangles of a shell
- How to test whether a point P is inside a shell S
- How to identify the shells that bound a solid
- How to identify the solids (and their bounding shells) from a corner table that represents all the triangles
- How to orient the shells bounding a solid
- How to identify the non-manifold vertices of a shell
- How to test whether a shell is free from self-intersections
- How to test whether two shells intersect one another
- What if the triangles do not form a water-tight shell

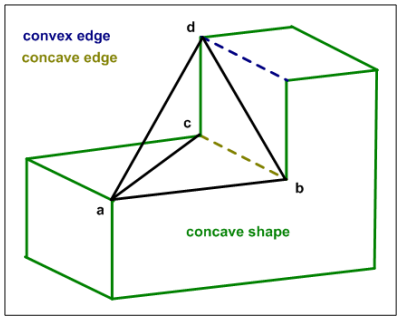
Practice test 1

Concavity test for an edge of a triangle mesh

- Assume that the triangle mesh is properly oriented and represented by a corner table.
- Each edge (**a**,**b**) of a triangle mesh may be identified by the opposite corner c of one of its incident triangles, so that c.n.v==a && c.p.v==b or vice versa. Let edge(c) denote this edge.
- Assuming that triangles are oriented counterclockwise, we say that edge(c) is **concave** when the vertices of c.t appear counter-clockwise from c.o.v.
- An edge that is not concave is either **convex** or **flat**.
- Write a very simple test for checking whether edge(c) is concave, convex, or flat. Use only vector notations and cross and dot products. Use the corner table operators. Present it as a function convex(c) that returns 1 if the edge is convex, 0 if it is flat, and -1 if it is concave.
- Make sure that your function can identify flat edges even in the presence of numerical round-off errors.
- Provide a drawing with the vertices and corners marked properly.







Practice test 1: Solution

- $\underline{\mathbf{N}}$:=(c.p.v.g–c.v.g)×(c.n.v.g–c.v.g) ;
  - Outward normal to c.t
- d:=(c.o.v.g–c.v.g)•$\underline{\mathbf{N}}$;
- If |d|<e then the edge is flat

  (epsilon trick to compensate for round-off errors)
- if d>e then the edge is concave.
- If d<e then the edge is convex.
  - Pick e carefully. Depends on model size, desired threshold for identifying flat edges.

Practice test 2

Smooth normals

- Assume that the triangle mesh is properly oriented and represented by a corner table.
- Given a corner c, write the code for estimating the surface normal to the vertex c.v from two layers of neighbors.
- Use only corner table operators and operations on vectors and points.
- Let (c.v.x, c.v.y, c.v.z) denote the 3 coordinates of vertex c.v
- The first layer of neighbors are those vertices connected to c.v by an edge.
- The second layer of neighbors are those connected by an edge to the first layer.
- Use the 3 shadow area method for computing the normal to a patch

Practice test 2: Solution

We visit all neighbors of a.v and turn around each, collecting the triangle normals.
We mark visited triangles to avoid double counting.
This solution does not use the 3 shadow areas.

```
Input: corner a
N:=0;                                   # Normal is initially null
b:=a;                                   # b will travel ccw around a.v
do {c:=b.n;                             # c will travel around ccw around b.n.v
      do {
          IF (c.t.m==0) {               # triangle already processed
             N+= (c.p.v.g–c.v.g)×(c.n.v,g–c.v.g);      # add cross-product
             c.t.b:=1 };                # mark triangle
          c:=c.r.n;                     # next corner around b.n.v
          } while (c != b.n);           # finished turning around b.n.v
      b:= b.r.n                         # next b
      } while (b != a);                 # stop turning around a.v
n:=N.u;                                 # make unit vector
```

Practice test 3

Shell containment

- Assume that you have two manifold shells A and B that do not intersect.
- Assume that each shell is properly oriented (so that its triangles appear counterclockwise when seen from the outside) and is represented by a corner table.
- Provide the pseudo-code for detecting whether A lies inside B, B lies inside A, or neither.
- Now, assume that A lies inside B. Provide the pseudo-code for testing whether a point **p** lies inside the solid S bounded by these two shells.

Practice test 3: Solution

- A is inside B if and only if the first vertex, v_o, of A is inside B.
  - A shell is connected. The shells do not intersect. So, A is either entirely inside B or entirely outside of it. Therefore we can use any vertex of A for the test.
- To test whether $\mathbf{v_o}$ is inside B, we pick a point **x** and make tetrahedra that each join x to a triangle of B. $\mathbf{v_o}$ is inside B if it lies in an odd number of these tetrahedra. The point-in-tetrahedron test is performed by comparing the signs of five mixed-products (see slides).
- To test whether B is inside A, just swap A and B above.

- Assuming that A lies inside B, the solid S is B–A, which is a simple CSG expression. Point **p** is in S if it is in B and out of A.
  - We already discussed how to test whether a point is inside a shell.
- An alternative solution is to treat A and B as a single set of triangles and count the parity of the tetrahedra they form with an arbitrary point **x** and that contain **p**. It has to be odd for **p** to be in S.