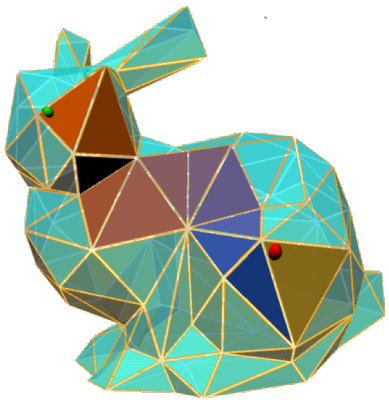


Triangle Meshes:
Intro

- Points, edges, triangles
- Simple triangle mesh
- Valid imbedding
- Dual graph and spanning trees
- Euler formula
- Corner operators
- Vertex normals
- Containment
- Traversals
- Geodesic paths and fields
- Smoothing
- Subdivisions

Start

Please close your laptops and silence & do not use your phones

Take lots of hand-written notes

Ask each time you need clarification/confirmation

Learning Objectives

Learn how to define, represent, and process triangle meshes

These are essential in graphics, animation, and modeling:

- Polygonal meshes may be trivially converted to triangle meshes
- Triangle meshes are simpler to implement than polygonal meshes
- Triangle meshes guarantee that its faces are planar
- GPU is engineered for fast processing of triangle meshes
- Triangle meshes may be used as control for subdivision surfaces

Elements of a Triangle-Mesh, M, **in 3D**

What is a vertex of M?

What is an edge of M?

What is a triangle of M?

Why are edges and triangles defined as relatively open?

Elements of a Triangle-Mesh, M

What is a vertex of M?

A point (location) associated with an ID between 0 and n_v-1

What is an edge of M?

The relative interior of the convex hull of two vertices of M

The line-segment between them, but excluding the end-points

What is a triangle of M?

The relative interior of the convex hull of three vertices of M

The face interpolating between them, but excluding the borders

It does not contain its edges and its vertices

Why are edges and triangles defined as relatively open?

So that, in a proper imbedding, all elements are pairwise-disjoint

A point of M belongs to only one element (unambiguous loyalty)

Topology of a **Simple Triangle-Mesh** (**STM**)

An **STM** (Simple Triangle-Mesh), M, is defined by a set of **vertices**, a set of **edges**, and a set of **triangles**, such that:

Each **edge** of M interpolates 2 different vertices of M

Each **triangle** of M interpolates 3 different vertices of M

Each **vertex** has at least three incident triangles

Each **edge** has exactly two incident triangles

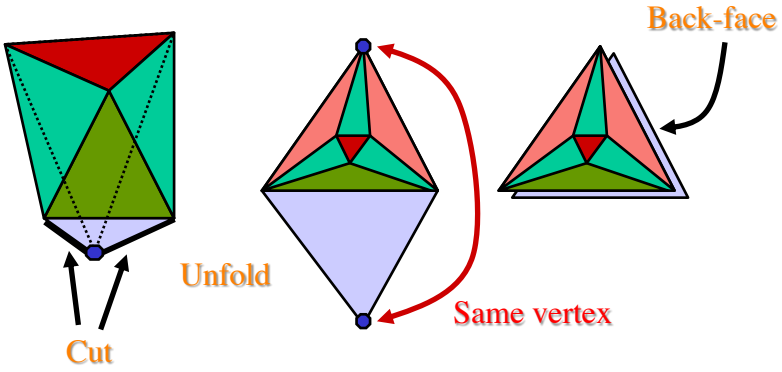
It is formed by a single **shell** (a single edge-connected component)

It's **genus** is 0 (no handles, planar graph)

Examples of topologies of surface that cannot be approximated by a simple T-mesh

The edges of an STM form a planar graph

The edges of an STM may be drawn in the plane without crossing



Combinatorial properties of a simple T-mesh

Assume that STM M has V vertices, E edges, and T triangles:

- T = 2V – 4

  About twice more triangles than vertices
- E = 3V – 6

How are these properties (topological invariants) derived?

- Euler formula F-E+V=2 with 2E = 3T (counting edge-uses)
- Cauchy's proof

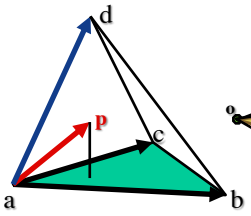
Point-in-solid test

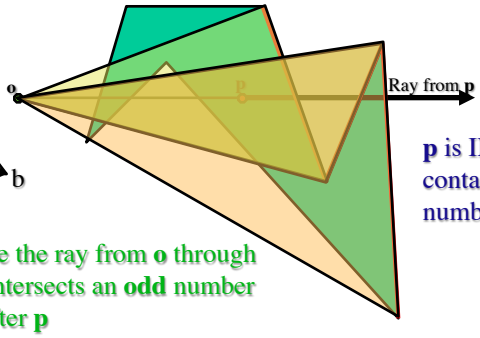
A point p lies inside a solid S bounded by triangles T_i of an STM when p lies inside an **odd number of tetrahedra**, each defined by an arbitrary point o and the 3 vertices of a different triangle T_i.

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 $(\underline{ab}\times\underline{ac})\bullet\underline{ad} > 0$

The test does not assume proper orientation of the triangles!





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

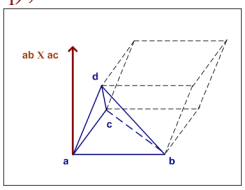
p is in because the ray from **o** through **p** to infinity intersects an **odd** number of triangles after **p**

Volume of a solid bounded by an STM

Given solid S, bounded by **consistently oriented STM**
where a triangles T_i has vertices (b_i,c_i,d_i)
and an arbitrary origin o,
let H_i denote the **tetrahedron** with vertices (o,b_i,c_i,d_i) of T_i.

The volume of S is one sixth of the sum of $v(o,b_i,c_i,d_i)$, for all i.

$v(o, b_i, c_i, d_i) = (\underline{ob_i}\times\underline{oc_i})\bullet\underline{od_i}$



Note that it is independent on the choice of o

Works for all oriented watertight triangle meshes (not only STM)

- Higher Genus
- Non-manifold

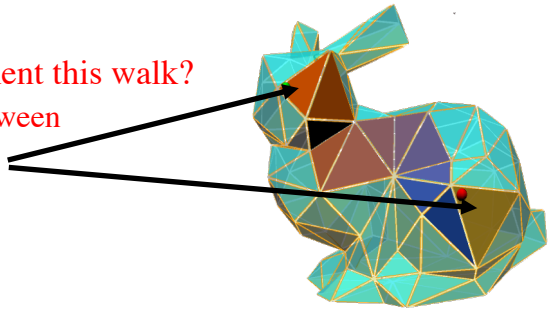
Adjacency and **incidence**

To traverse & process a STM, from each triangle we access:

- The three vertices that it is **incident** upon
- Three edge-**adjacent** (neighbor) triangles

How would you implement this walk?

Shortest graph-path between two given triangles



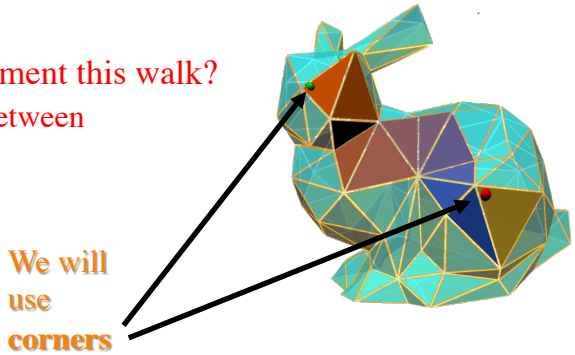
Adjacency and **incidence**

To traverse & process a STM, from each triangle we access:

- The three vertices that it is **incident** upon
- Three edge-**adjacent** (neighbor) triangles

How would you implement this walk?

Shortest graph-path between two given triangles



Oriented STM

We assume that the STM is **oriented**

Vertices of each triangles appear clockwise from outside

We want a compact data structure for the **connectivity**

t-v **incidence**, t-t & v-v **adjacency**,

Previous algorithms manipulate ordered tuples:

next(t,v): next vertex after v clockwise around triangle t,

swing(t,v): next triangle after t clockwise around vertex v

We use the term **corners** for tuple (f,v

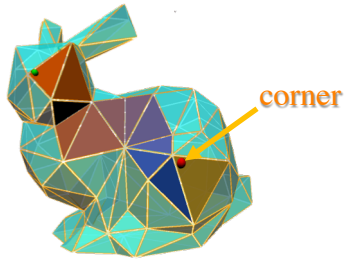
We assign consecutive **integer IDs** to:

Vertices

Faces

Corners (3 per triangle)

But not to edges: 2 corners per edge



Each corner identifies a different **half-edge (dart, edge-use, oriented edge)**

Corner operators for **STM**s

We organize corner-operators in 3 groups: primary, inverse, other.

It is convenient to use 4 **primary corner-operators**:

c.v: **vertex** of corner c

c.t: **triangle** of corner c

c.n: **next corner** around c.t

c.s: **swing corner** around c.v

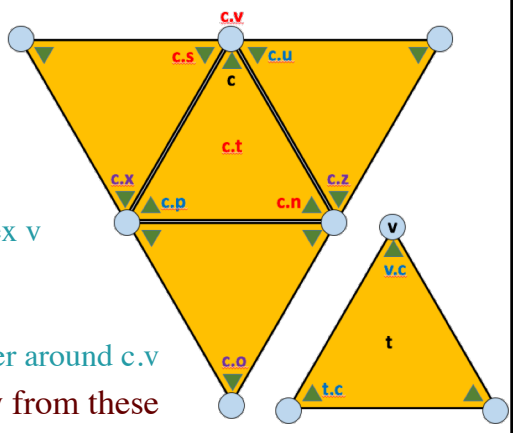
… and their **inverses**:

v.c: **a** corner incident on vertex v

t.c: **a** corner of triangle t

c.p: **previous** corner in c.t

c.u: (previous) **unswing** corner around c.v



Others may be derived easily from these

c.z = ?

c.o = ?

Quiz (individual, paper, name)

Express the left and right operators in terms of the primary ones

c.l =

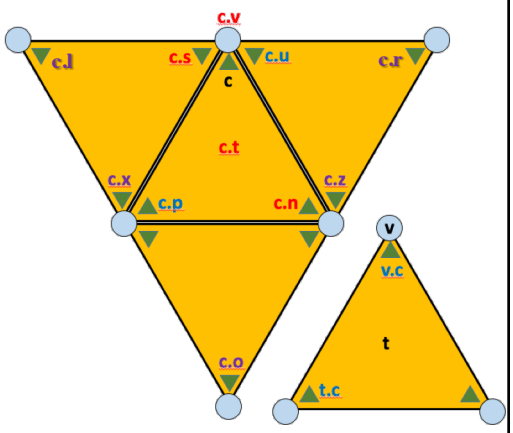
c.r =

Express these

c.p =

c.u =

in terms of the primary ones



Cost of storing all 8 tables: primary & inverse

In T-mesh:

C = 3T : each triangle has 3 corners

T ≈ 2V: follows from Euler eq, assuming relatively small genus & bdry

Naïve: Store int V[c],T[c],N[c],S[c],C_v[v],C_t[t],P[c],U[c]

32-bit INT references to corners, vertices, or triangles

Connectivity storage ≈ 6C+V+T refs = **39**V refs = 1248 V bits

How does this compare to the cost of storing **geometry**?

Connectivity takes **26** times more space than **vertex coordinates**!

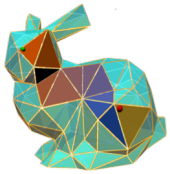
Assuming only 16 bits per coordinate, geometry cost = 48 V bits

Compact Representations of Triangle Meshes

Objective: Reduce storage cost of STM **connectivity** and still support **constant** (amortized) **time cost operations** for:

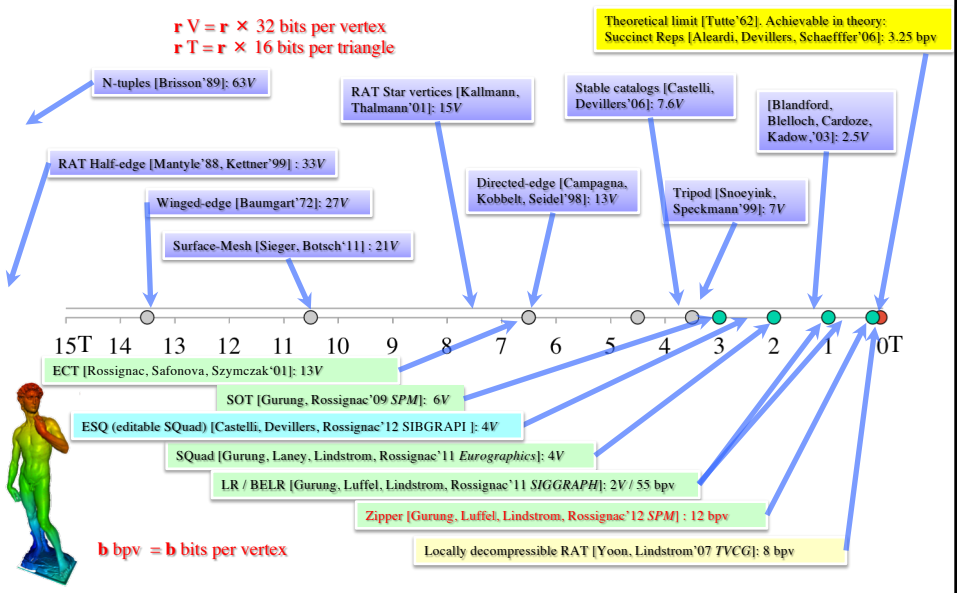
RAT (Random Access and Traversal) in

CAT (Constant Amortized Time)



with former GaTech students: Peter Lindstrom (LLNL), Topraj Gurung (Google), Mark Luffel (Apple)… and other collaborators

Some prior art: Mesh **connectivity** storage cost

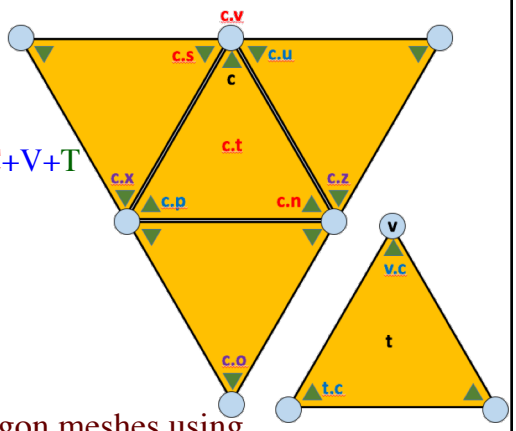


Rule 1: Do not store c.p and c.u

These can be computed by cascading other operators:

c.p = c.s.n.s (or simply c.n.n)

c.u = c.n.s.n



Cost = 4C + F + V = **27** V ref

30% savings over 39 V of 6C+V+T

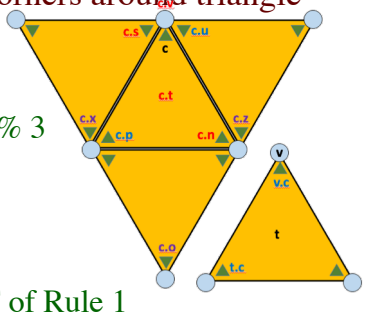
This trick works also for polygon meshes using

c.p = c.s.n.s

Rule 2: **Order** corners around faces (***ECT***)

Assign consecutive IDs to consecutive corners around triangle

c.p , c, c.n



For triangle meshes, no need to store:

c.n: it can be computed as 3 × c.t + (c+1) % 3

c.t: it can be computed as c/3

t.c: it can be computed as 3 × c

Cost = 2 C + V = **13** V ref

77% savings, **52%** savings over 4C+V+T of Rule 1

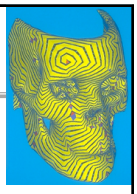
We will use two variants of this:

*Extended **Swing** table* stores V[c], **S**[c], C[v]

*Extended **Corner** table* stores V[c], **O**[c], C[v]

c.s = c.n.o.n

Rule 3: vertex/triangle match (*SOT*)

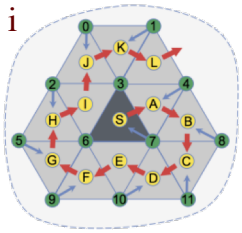


Match each vertex with a different incident triangle

Simple **linear cost** algorithm (TS & EB traversal)

Match each vertex with the C-triangle that discovered it

Reorder triangles: vertex i matched with triangle i



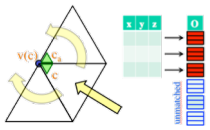
Store only the O[] table

No need to store C[] and V[]:

v.c: it is 3 × v

c.v: compute it by **iterating** c=c.s until c < 3 × V and c%3==0

Cost = 1C = **6**V ref

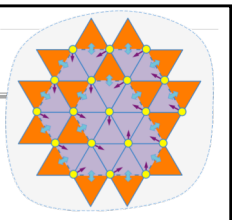


85% savings, **54%** savings over 2C+V of Rule 2

SOT: Compact representation for tetrahedral meshes

Gurung & Rossignac, **Solid and Physical Modeling (SPM)** 2009

Rule 4: Pair triangles (*SQuad*)

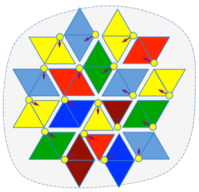


Pair adjacent matched & unmatched triangles

with an adjacent unmatched triangle

incident on the matched vertex : **Form quad**

Linear cost match&pair algorithm



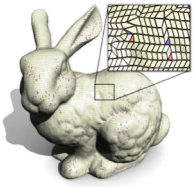
Nearly perfect pairing (~97%)

Spiraling (depth-first) traversal (as in Edgebreaker)

Pair C-triangles with non-C (right or left) neighbors

Store additional information for rare **exceptions**

Store only part of S[] table



No need to store swings between paired faces

Cost = 2 T = **4** V ref

90% savings, **33%** savings over 6V of Rule 3

SQuad: Compact Representation for Triangle Meshes

Gurung, Laney, Lindstrom, Rossignac. **Eurographics** 2011

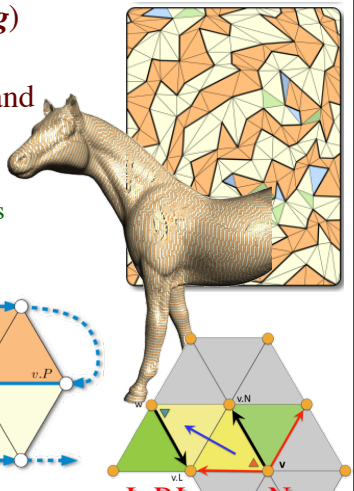
Rule 5: Order quad diagonals along loop (*LR*)

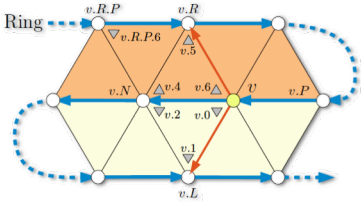
Select match&pair quads so that **quad diagonals** form a ***nearly* Hamiltonian cycle** (the ***ring***)

- Simple, linear cost **ring-expander** algorithm

For each ring vertex v, store references v.L and v.R, to 2 adjacent ring vertices

- next v.N and previous v.P ring vertex are implicit
- Store refs for connectivity around non-ring vertices
- Can compute most c.o without more info





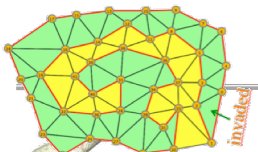
v.L.P.L == v.N

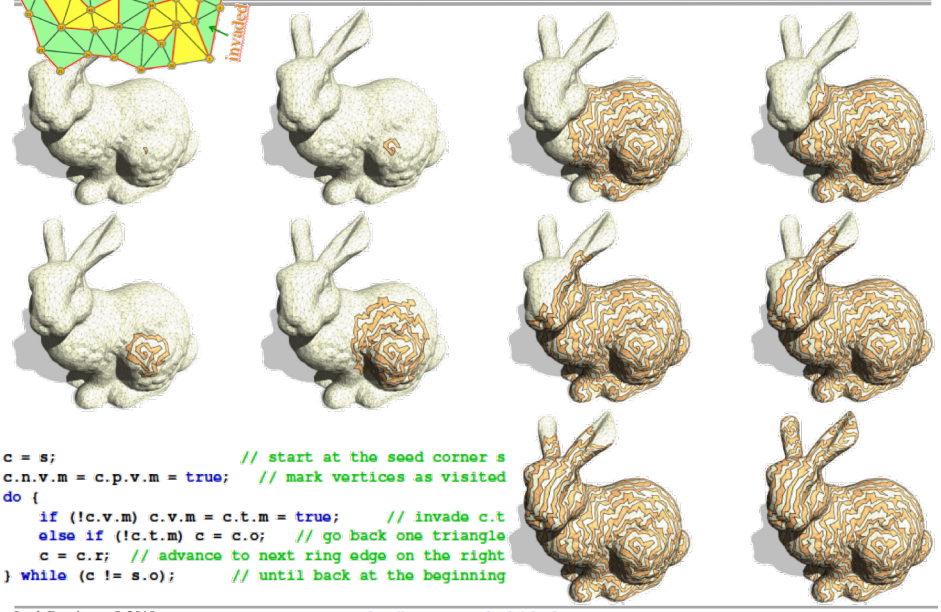
Cost ≈ 2.16 V ref

- 94% savings
- 46% savings over Rule 4
- 1/3 of incidence cost

LR: compact connectivity representation for triangle meshes. Gurung, Luffel, Lindstrom, Rossignac. **SIGGRAPH**, 2011.

Ring-expander *algorithm* (*LR*)





```
c = s;                          // start at the seed corner s
c.n.v.m = c.p.v.m = true;   // mark vertices as visited
do {
    if (!c.v.m) c.v.m = c.t.m = true;      // invade c.t
    else if (!c.t.m) c = c.o;    // go back one triangle
    c = c.r;  // advance to next ring edge on the right
} while (c != s.o);       // until back at the beginning
```

Rule 6: Store LR as deltas (***BELR***)

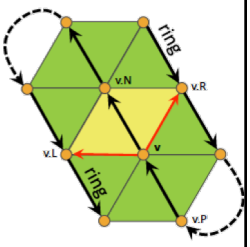
BELR: ("BEtter" or "Bit Efficient" LR)

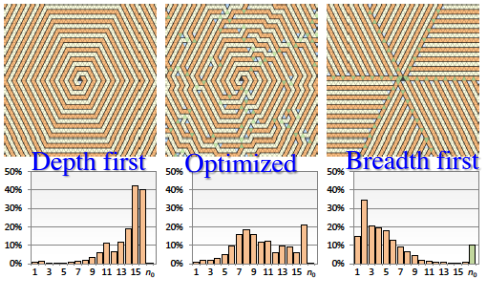
Instead of storing v.L and v.R, store (v.L-v)%V and (v.R-v)%V

Optimize ring formation so that most fit into 16-bits

Cost ≈ 52 V bits

96% savings over 39 V, **25%** savings over Rule 5





LR: compact connectivity representation for triangle meshes.
Gurung, Luffel, Lindstrom, Rossignac. **SIGGRAPH**, 2011.

Rule 7: Predict the deltas (***Zipper***)

Instead of storing v.L-v & v.R-v, **store v.L-v.P.L & v.R-v.P.R**

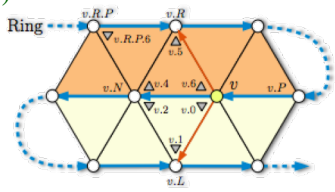
when delta is in [0,3] store using 2 bits, otherwise store the full ref (**exception**)

Force exceptions every 32 vertices (=>constant time)

Use ***popcount*** to get v.L and v.R without loop

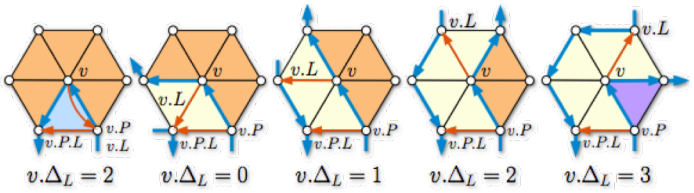
Bend the ring to reduce the number of exceptions

Use hashing to reduce storage cost for exceptions



Cost ≈ **12** V bits (*can be further reduced to 10 V bits*)

99.2% savings over 39 V, **77%** savings over Rule 6



Performance

Representation	Time
Surface Mesh	1.00
CT	0.85
LR	0.83
Zipper	1.18

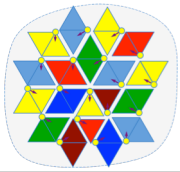
Zipper: A compact connectivity data structure for triangle meshes.
Gurung, Luffel, Lindstrom, Rossignac. **Computer-Aided Design, SPM** 2012

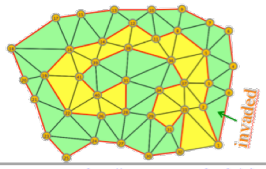
Summary of rules for compact reps of STM

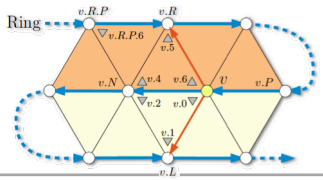
1. **Do not store operator** inverses when they can be computed
2. **Order corners** around triangles
3. **Order triangles** to synch with matching vertices
4. **Pair** matched and adjacent unmatched triangles into quads
5. Order quad diagonals around nearly **Hamiltonian cycle**
6. **Store deltas** rather than references
7. **Store corrections** to delta predictions

1248-to-12 reduction of storage over storing 8 look-up tables

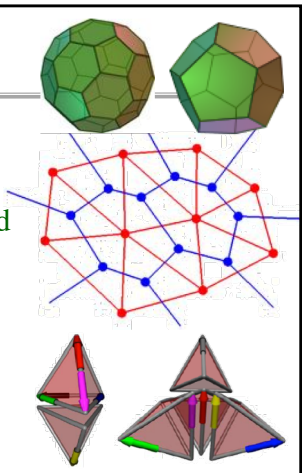
Linear cost & fast access and traversal operators (no decompression)







Extensions beyond STM



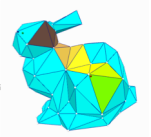
Domain

- T-meshes with genus and holes
- Polyhedron surface: Y=mesh, general manifold
- Tet mesh (improve on SOT)
- Mixed Hex/Tet mesh for FE analysis
- Non-manifold triangle complex
- Simplicial complex

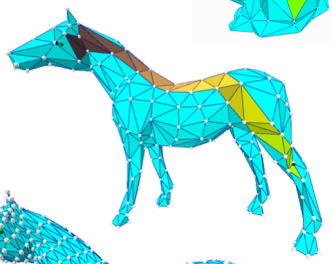
Application

- Compressed Transmission, Progressive refinement
- In memory processing
- Streaming (as constructed or user-guided)
- Construction & editing

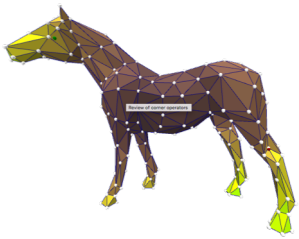
Mesh processing



Shortest path between corners (triangles)

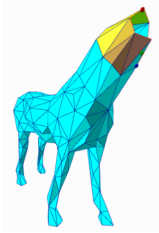


Isolation



Refinement (subdivision)

Holes & borders



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);
```

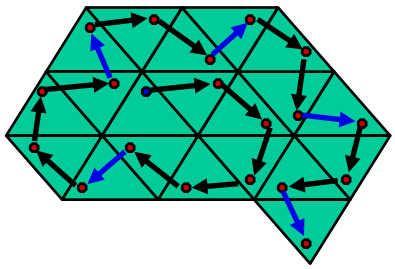
Label vertices **(for example as 1, 2, 3 …)**

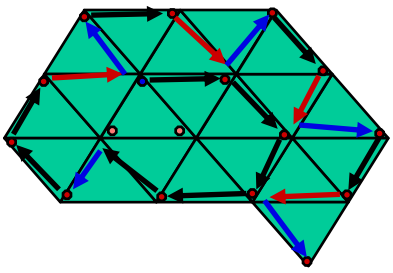
Label vertices with consecutive integers

Label(c.n.v); Label(c.n.n.v); Visit(c);

Visit(c)

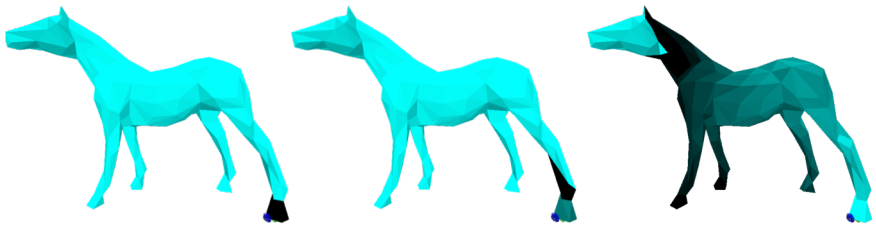
```
IF NOT labeled(c.v) THEN Label(c.v);
mark c.t;
IF NOT marked(c.r.t) THEN visit(c.r);
IF NOT marked(c.l.t) THEN visit(c.l);
```



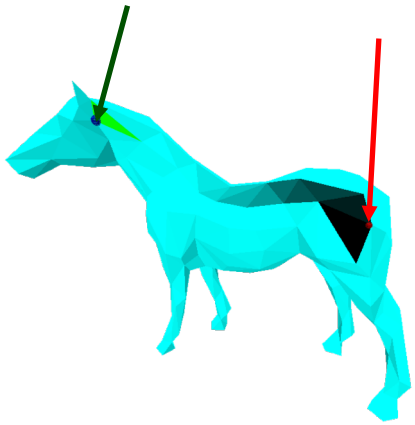


Compute geodesic distance

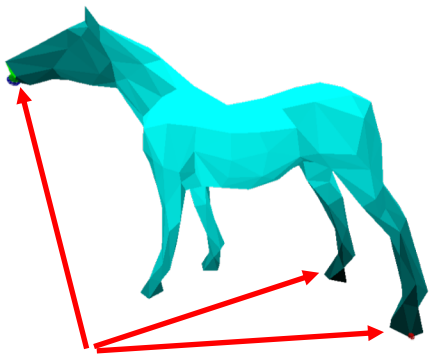
```
void computeDistance(int maxr) {
  int tc=0;
  int r=1;
  for(int i=0; i<nt; i++) {Mt[i]=0;};  Mt[t(c)]=1; tc++;
  for(int i=0; i<nv; i++) {Mv[i]=0;};
  while ((tc<nt)&&(r<=maxr)) {
    for(int i=0; i<nc; i++) {if ((Mv[v(i)]==0)&&(Mt[t(i)]==r)) Mv[v(i)]=r;};
    for(int i=0; i<nc; i++) {if ((Mt[t(i)]==0)&&(Mv[v(i)]==r)) {Mt[t(i)]=r+1; tc++;};};
    r++;  };
  rings=r;   }
```



Path and isolation

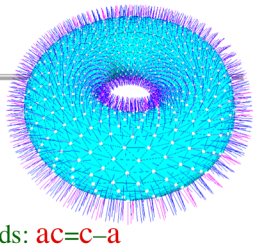


How would you represent and compute the path?



How would you represent and compute isolation and extremities?

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

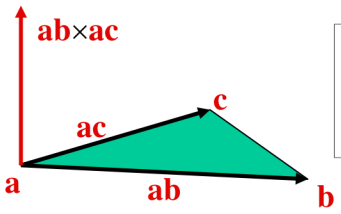
Note that N is independent of the position of vertex a

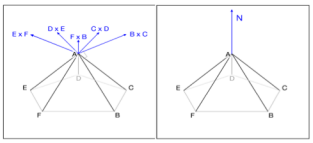
(b–a)×(c–a)+(b–a)×(c–a)+… = b×c+a×a–b×a–a×c + c×d+a×a–c×a–a×d +… –a×b +…

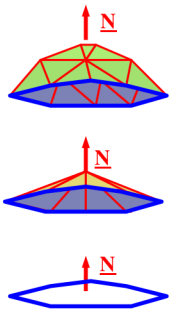
a×a = 0 , –a×c and –c×a cancel out, same for all other cross-products containing a

We are left with = b×c + c×d + … which does not depend on a

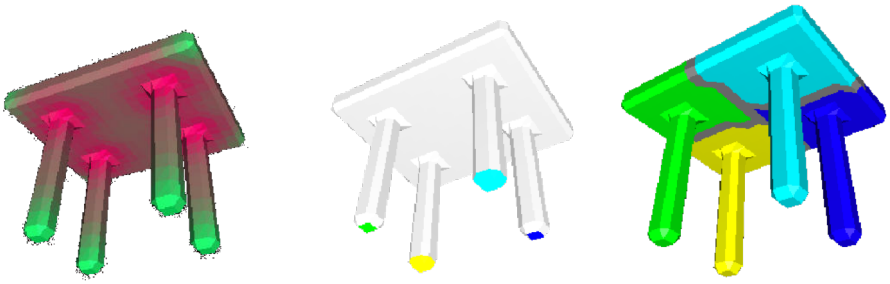
Then divide N by its norm to make it a unit vector



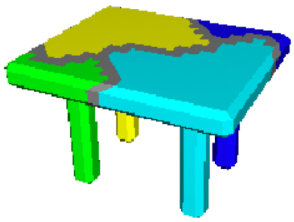




Segmentation

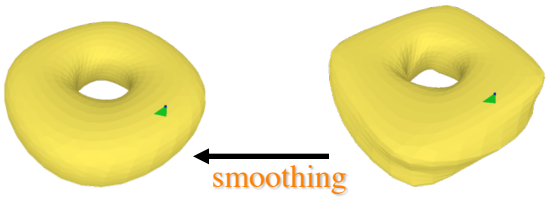


How would you represent and compute these?



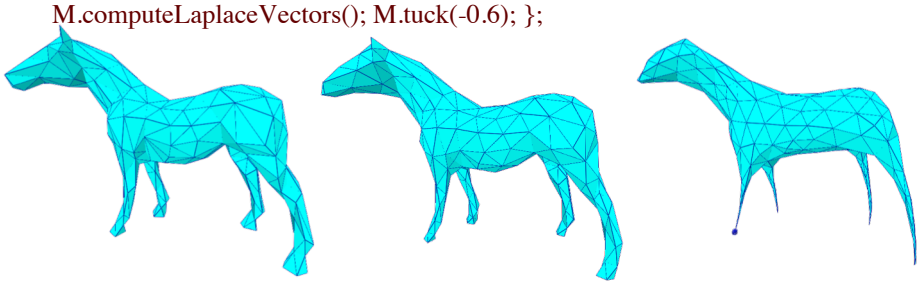
Smoothing

```
void keyPressed() {
  …
  // bi-laplace smoothing
  if (key=='S') {
   computeVertexNormals();
   tuck(0.6);
   computeVertexNormals();
   tuck(-0.6);
   };
```

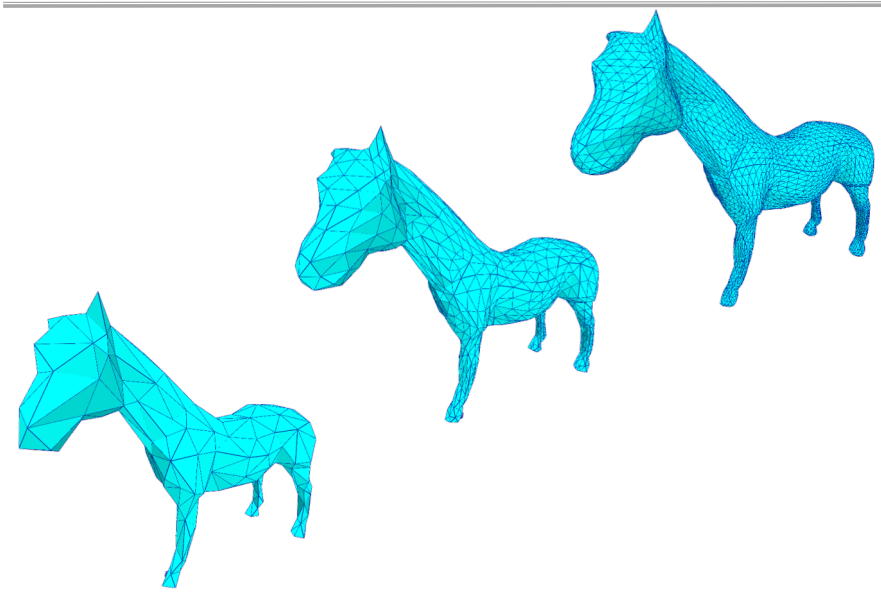


Smoothing

```
void computeLaplaceVectors() {
  for (int i=0; i<nv; i++) {Nv[i].setTo(0,0,0); Valence[i]=0;};
  for (int i=0; i<nc; i++) {Valence[v(i)]++; };   }
  for (int i=0; i<nc; i++) {Nv[v(p(i))].add(g(p(i)).vecTo(g(n(i))));};
  for (int i=0; i<nv; i++) {Nv[i].div(Valence[i]);};   };
void tuck(float s) {for (int i=0; i<nv; i++) {G[i].addScaledVec(s,Nv[i]);};};
 if (key=='S') {
    M.computeLaplaceVectors(); M.tuck(0.6);
    M.computeLaplaceVectors(); M.tuck(-0.6); };
```



Butterfly subdivision



An intuitive formulation for inserted points

First insert the new points in the middle of the edges (average of first neighbors)

m = **average**(f)

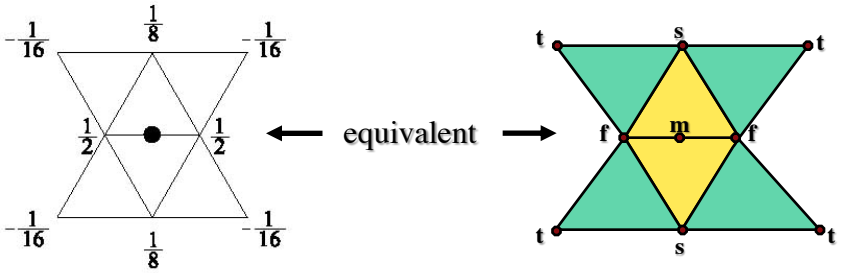
Then compute adjustment vectors (between averages of third and second neighbors)

d = (**average**(s) – **average**(t))/4

Then adjust all the m points by displacing them by the corresponding d

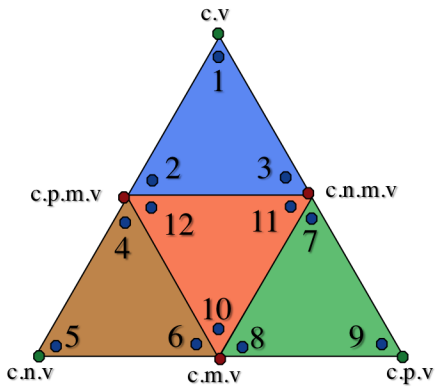
m = m + d

Then compute the new triangles



Generating the V entries for 4 new triangles

Write the 12 V-table entries in terms of c



c.v
c.p.m.v
c.n.m.v
c.p.m.v
c.n.v
c.m.v
c.n.m.v
c.m.v
c.p.v
c.m.v
c..n.m.v
c.p.m.v

Butterfly subdivision & smoothing