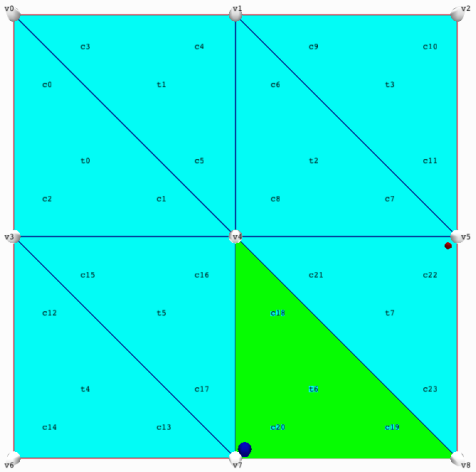
Mesh Processing

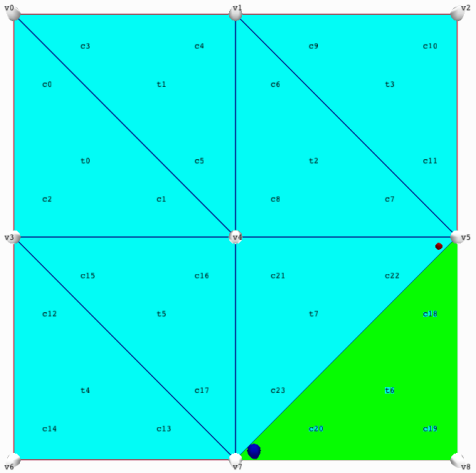
Georgia Tech College of Computing
Interactive & Intelligent Computing

- Corner operators
- Computing O
- Flipping edges
- Collapsing edges
- Geodesic distances, paths, and isolations
- Filling holes
- Smoothing
- Subdivision
- Compression

Flip edge

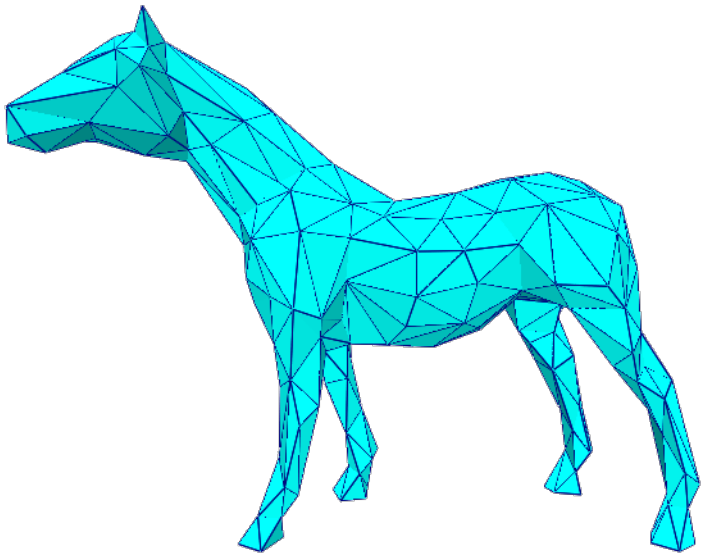
```
void flip(int c) {      // flip edge opposite to corner c
  if (b(c)) return;  // no opposite triangle
    V[n(o(c))]=v(c); V[n(c)]=v(o(c));
    int co=o(c);
    O[co]=r(c);
    if(!b(p(c))) O[r(c)]=co;
    if(!b(p(co))) O[c]=r(co);
    if(!b(p(co))) O[r(co)]=c;
    O[p(c)]=p(co); O[p(co)]=p(c);
   }
```

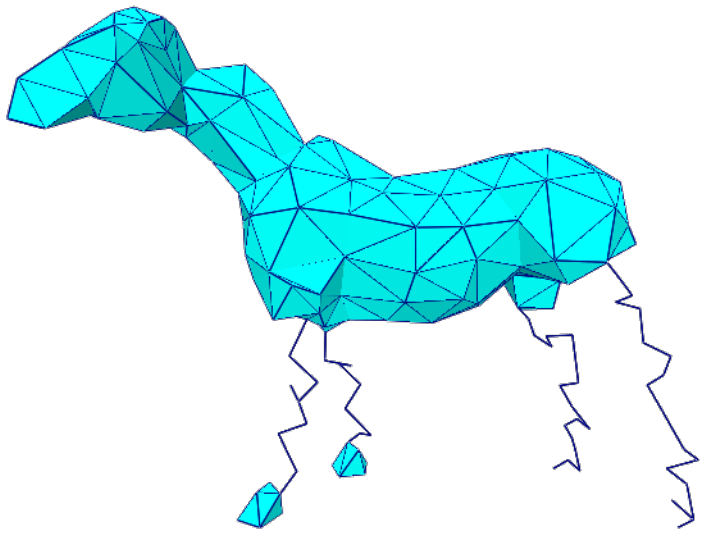




Shorten edges by flipping

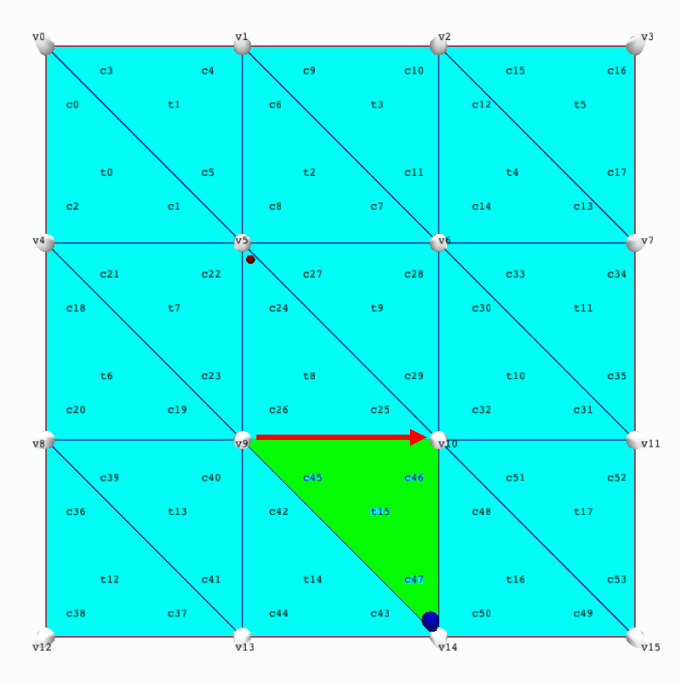
```
void flipWhenLonger() {
  for (int c=0; c<nc; c++)
    if (d(g(n(c)),g(p(c)))>d(g(c),g(o(c)))) flip(c);  }
```

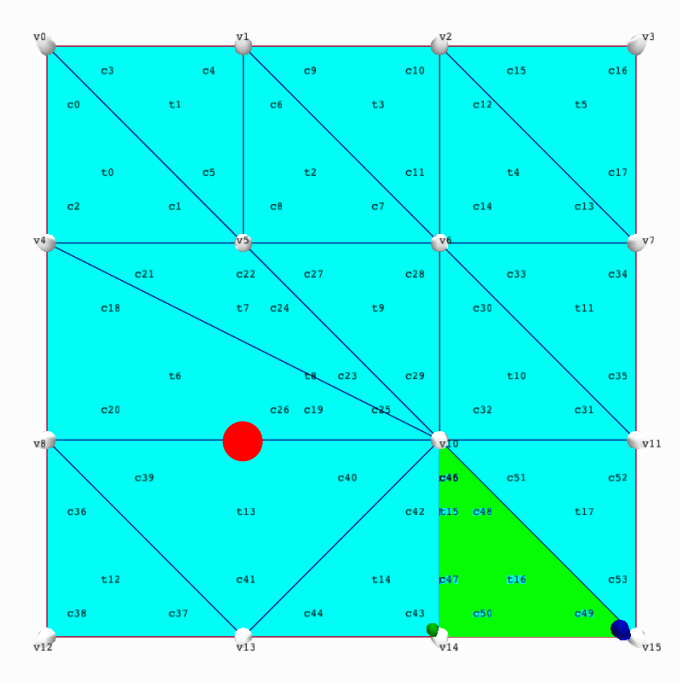




Collapse an edge

```
void collapse(int c) { if (b(c)) return;
  int b=n(c), oc=o(c), vpc=v(p(c));
  visible[t(c)]=false; visible[t(oc)]=false;
  for (int a=b; a!=p(oc); a=n(l(a))) V[a]=vpc;
  O[l(c)]=r(c); O[r(c)]=l(c);    O[l(oc)]=r(oc); O[r(oc)]=l(oc);  }
```



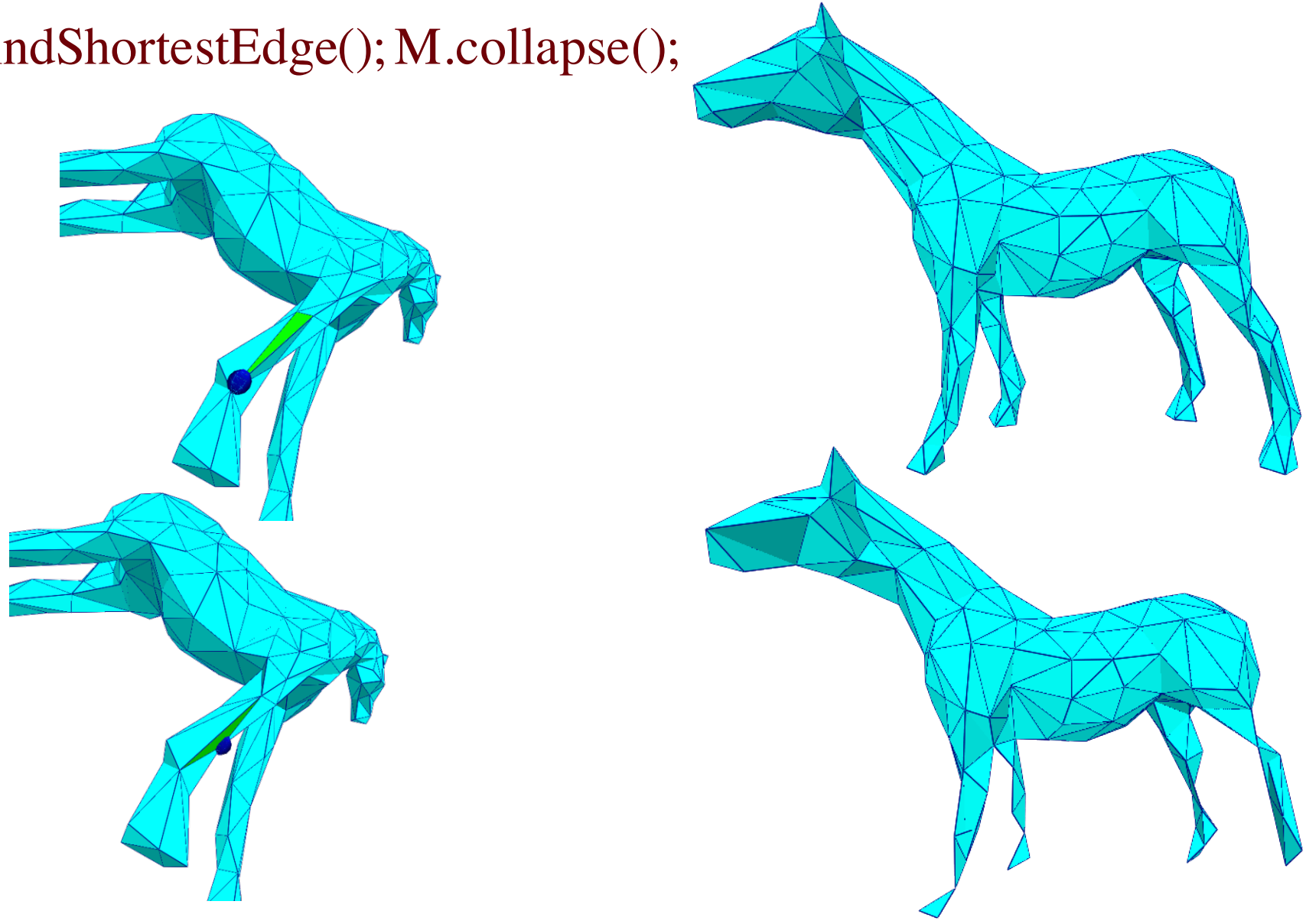


Find shortest edge

```
int cornerOfShortestEdge() {  // assumes manifold
  float md=d(g(p(0)),g(n(0)));   // init shortest length found so far
  int ma=0;                      // init facing corner
  for (int a=1; a<nc; a++)       // for all other corners
    if (d(g(p(a)),g(n(a)))<md)       {  // if found a shorter edge
      ma=a; md=d(g(p(a)),g(n(a))); }; // update
  return ma;
  }
```

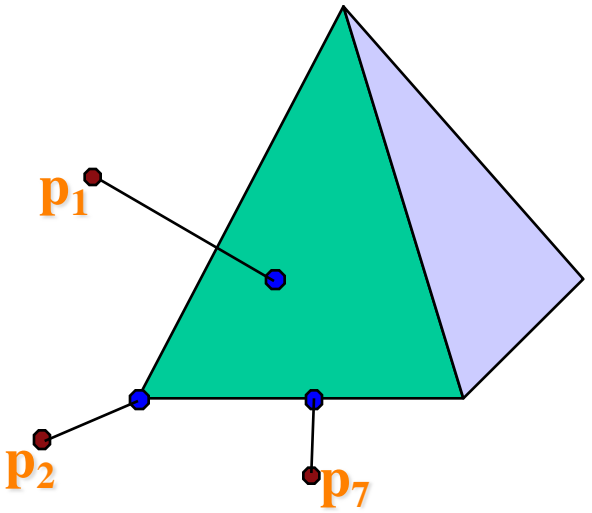
Collapse shortest edges

M.findShortestEdge(); M.collapse();



Distance between a point and a T-mesh

- The minimum distance may occur
  - In the middle of a face (p_1)
  - In the middle of an edge (p_7)
  - At a vertex (p_2)



- The distance to the solid bounded by a T-mesh S is
  - 0 it p is inside the solid
    - use parity of the number of intersections of ray with S
  - D(p,S) otherwise

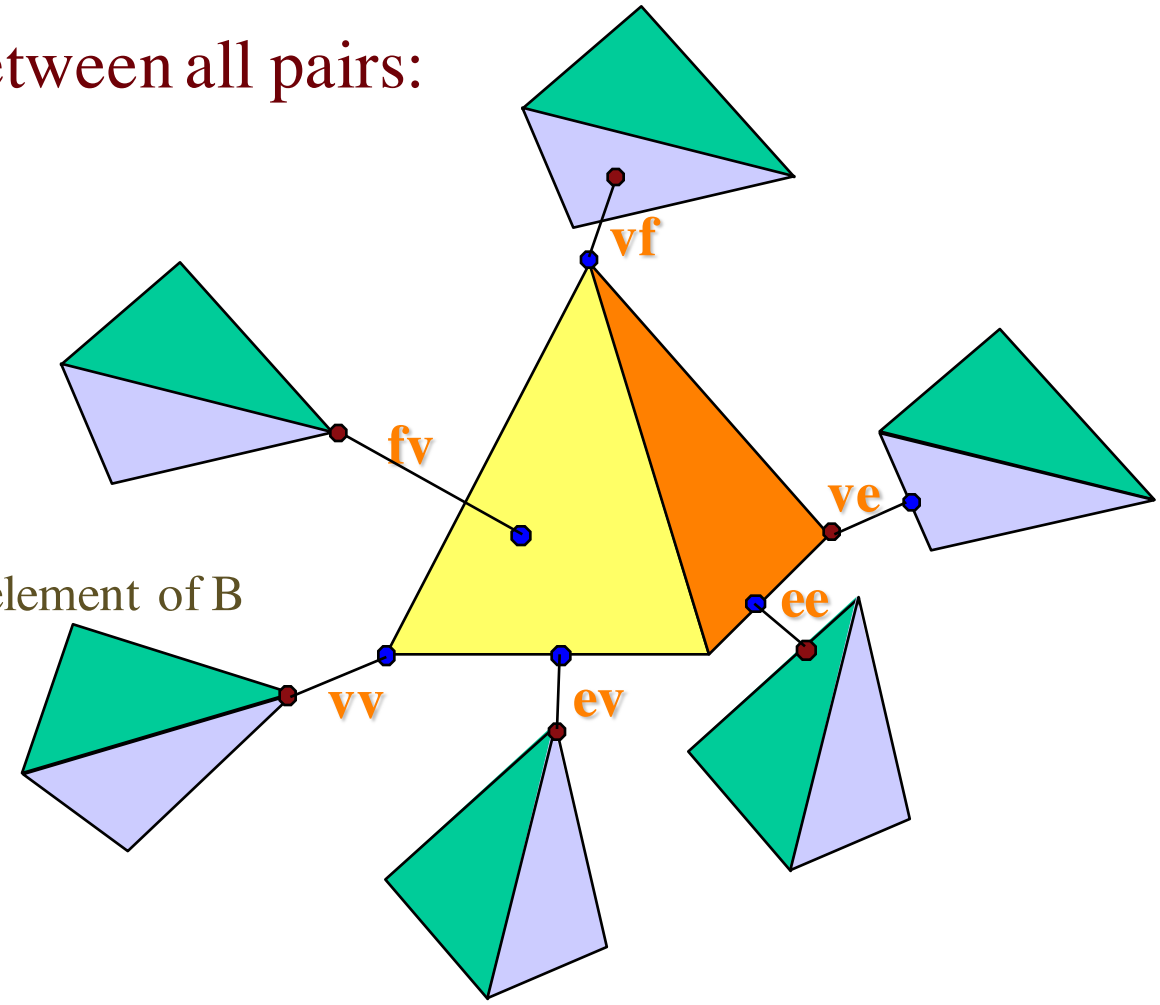
Exact point-surface distance for T-meshes

- D(p,S) ≡ Min(D(p,T) : T is a triangle of S),using previous slide
- Speed-up
  - d:= min(D(p,v) : v is a vertex of S)
  - For each triangle T of S do
    - D(p,plane(T))<d then if normal projection of P on T then d:= D(p,plane(T))
  - For each edge E do
    - D(p,line(E))<d then if normal projection of P on E then d:= D(p,line(E))
- Further speed-up
  - Preprocessing
    - Compute a minimal sphere around each triangle
    - Group neighboring triangles and compute spheres around groups
    - Build tree of spherical containers by recursively merging groups
  - Query for a point p
    - Go down the tree, picking at each node the sphere whose center is closest to p
    - d:= distance to leaf triangle
    - Go down the tree again in depth first order, but only visit branches whose spheres are closer to p than d
      - At each visited leaf, update d

Distance between two T-meshes

- Minimum of distances between all pairs:
  - face/vertex
  - edge/vertex
  - vertex/vertex
  - vertex/edge
  - vertex/face
  - edge/edge

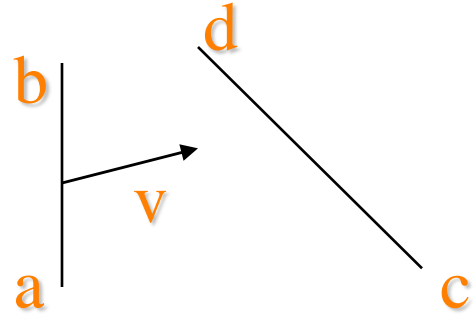
  of an element of A and an element of B



N:=(ab×cd) / ||ab×cd||
Dist:= N•ac
ee test ?

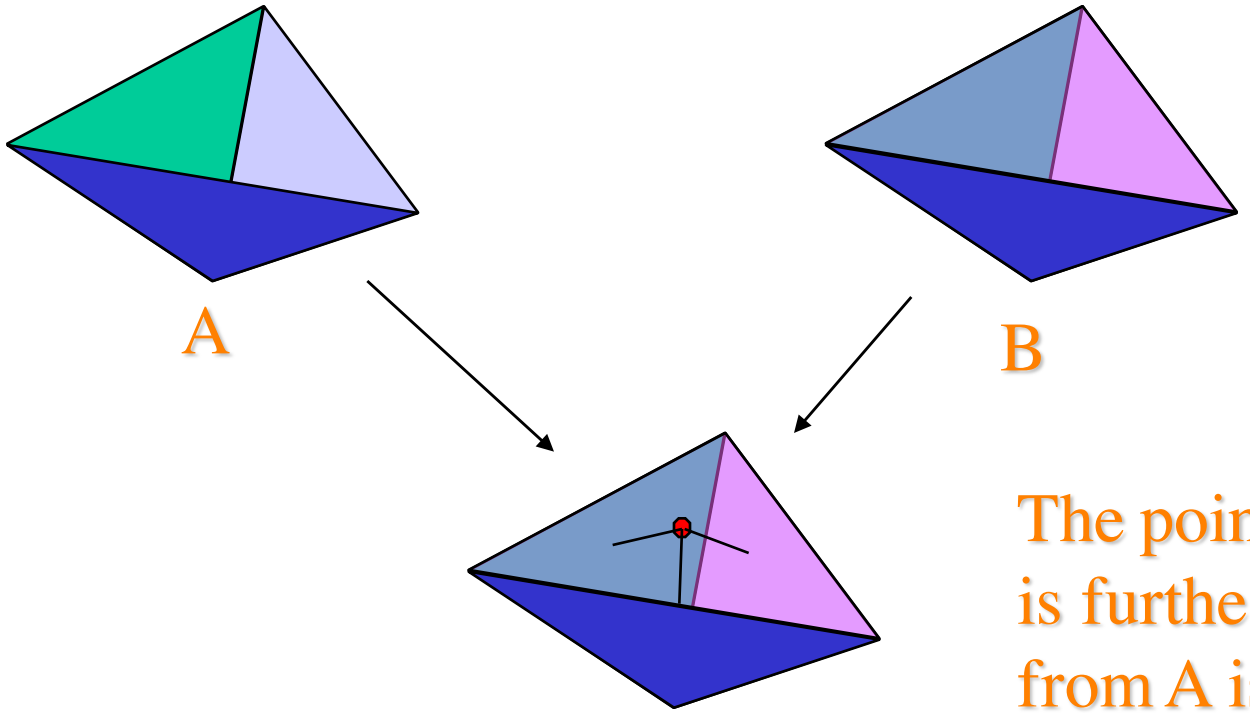
Collision prediction between T-meshes

- Assume A moves with constant velocity v and B is fixed
  - If both move, compute collision in local coordinate system of B
- Collision may occur between
  - A vertex p of A and (the closure of) a triangle T of B
    - **Intersect Ray(p,v) with T**
  - The closure of a triangle T of A with a vertex p of B
    - **Intersect Ray(p,-v) with T**
  - An edge (a,b) of A with an edge (c,d) of B
    - **Check when the volume of tetrahedron (a+tv,b+tv,c,d) becomes zero**
      - solve $(cd\times(ca+tv))\bullet(cb+tv)=0$ for t
      - $(cd\times(ca+tv))\bullet cb + (cd\times(ca+tv))\bullet tv)=0$
      - $(cd\times ca+t(cd\times v))\bullet cb + (cd\times ca+t(cd\times v))\bullet tv)=0$
      - $(cd\times ca)\bullet cb + t(cd\times v)\bullet cb + (cd\times ca)\bullet tv + t^2(cd\times v)\bullet v = 0$
      - $(cd\times ca)\bullet cb + t(cd\times v)\bullet cb + (cd\times ca)\bullet tv = 0$, because $(cd\times v)\bullet v = 0$
      - $t = (ca\times cd)\bullet cb \,/\, ((cd\times v)\bullet cb - (cd\times v)\bullet ca)$
      - **$t = (ca\times cd)\bullet cb \,/\, (ab\times cd)\bullet v$**
    - **Make sure that, at that time, the two edges intersect**
      - **Not just the lines**



Hausdorff distance between T-meshes

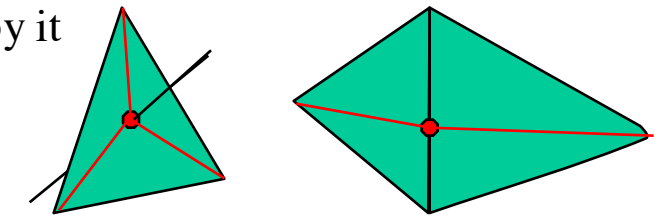
- Expensive to compute,
  - because it can occur away from vertices and edges!



The point of B that is furthest away from A is closest to 3 points on A that are inside faces

Intersection between T-meshes

- The triangle meshes A and B intersect if and only if the closure of an edge of one intersects the closure of a triangle of the other
  - The closure of an edge is the union of the edge with its vertices
  - The closure of a triangle is the union of the triangle with its boundary
  - These include the cases where an edge intersects or touches an edge or when a vertex touches a triangle
- How do we compute the intersection?
  - Assume A and B are represented by corner tables
  - Do all pairs of edge-triangle intersection tests
    - Each time you find an intersection, insert it in both meshes and subdivide the triangles to restore a valid triangulation and a correct corner table
    - With each vertex that you insert, associate the IDs of the 3 triangles involved
      - Two bounded by the edge and one intersected by it
  - Identify the intersection edges
    - They are new edges whose vertices share 2 IDs



Compacting the V & O tables

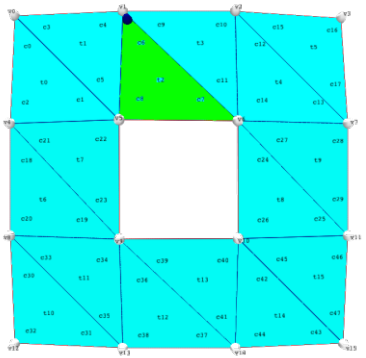
```
void excludeInvisibleTriangles () {for (int b=0; b<nc; b++)
  if (!visible[t(o(b))]) O[b]=-1;}

void compactVO() {
 int[] U = new int [nc];
 int lc=-1; for (int c=0; c<nc; c++) {if (visible[t(c)]) {U[c]=++lc; }; };
  for (int c=0; c<nc; c++) {if (!b(c)) {O[c]=U[o(c)];} else {O[c]=-1;}; };
  int lt=0;
  for (int t=0; t<nt; t++) {
    if (visible[t]) {
      V[3*lt]=V[3*t]; V[3*lt+1]=V[3*t+1]; V[3*lt+2]=V[3*t+2];
      O[3*lt]=O[3*t]; O[3*lt+1]=O[3*t+1]; O[3*lt+2]=O[3*t+2];
      visible[lt]=true;
      lt++; }; };
  nt=lt; nc=3*nt;    }
```

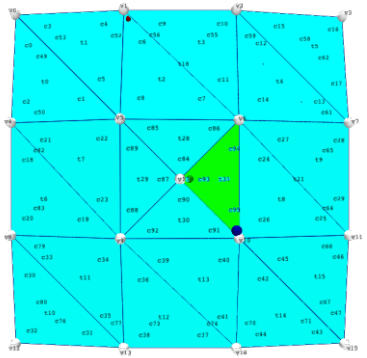
Compacting the G and V tables

```
void compactV() {
  int[] U = new int [nv];
  boolean[] deleted = new boolean [nv];
  for (int v=0; v<nv; v++) {deleted[v]=true;};
  for (int c=0; c<nc; c++) {deleted[v(c)]=false;};
  int lv=-1; for (int v=0; v<nv; v++) {if (!deleted[v]) {U[v]=++lv; }; };
  for (int c=0; c<nc; c++) {V[c]=U[v(c)]; };
  lv=0;
  for (int v=0; v<nv; v++) {
    if (!deleted[v]) {
      G[lv].setToPoint(G[v]);  deleted[lv]=false; lv++; }; };
  nv=lv;
 }
```

Fill a hole

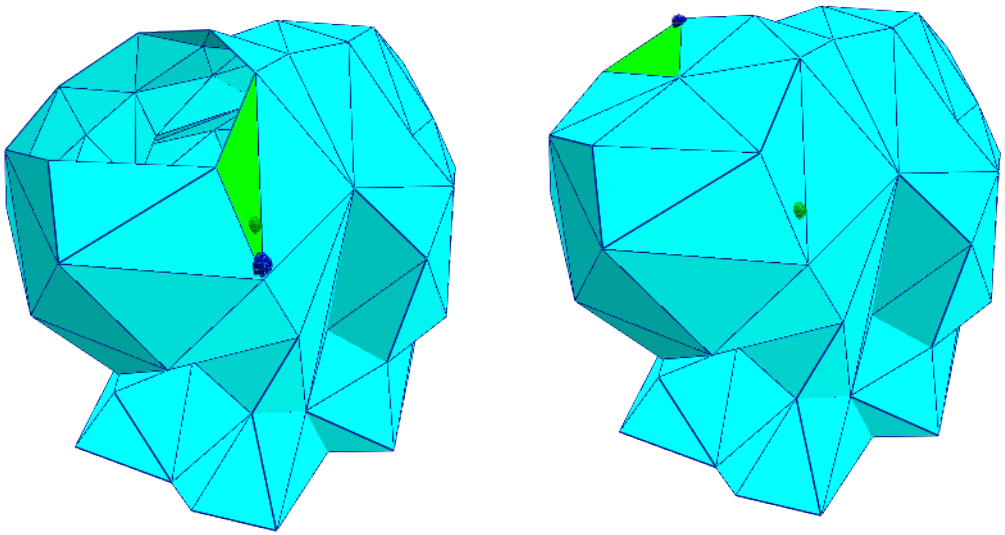
```
void fanThisHole(int cc) {if(!b(cc)) return ;
G[nv].setTo(0,0,0);   // tip vertex of fan
 int o=0;             // tip corner of new fan triangle
 int n=0;             // triangle count in fan
 int a=n(cc);         // corner running along the border
 while (n(a)!=cc) {if(b(p(a))) { // walk around, if a is left-end of border edge
     G[nv].addPt( M(M(g(a),g(n(a))),S(g(a),V(g(p(a)),g(n(a))))) );
     o=3*nt; V[o]=nv; V[n(o)]=v(n(a)); V[p(o)]=v(a); visible[nt]=true; nt++;
    O[o]=p(a); O[p(a)]=o; O[n(o)]=-1; O[p(o)]=-1; // link opposites for o
     n++;}; // increase triangle-count in fan
  a=s(a);} // next corner along border
 G[nv].mul(1./n); // divide fan tip to make average
 a=o(cc);       // reset a to walk around again
 int l=n(a);   // keep track of previous
 int i=0; while(i<n) {a=s(a); if(v(a)==nv) {i++; O[p(a)]=l; O[l]=p(a); l=n(a);};}
 nv++;  nc=3*nt; }; // update vertex count and corner count
```





Find and fill all holes

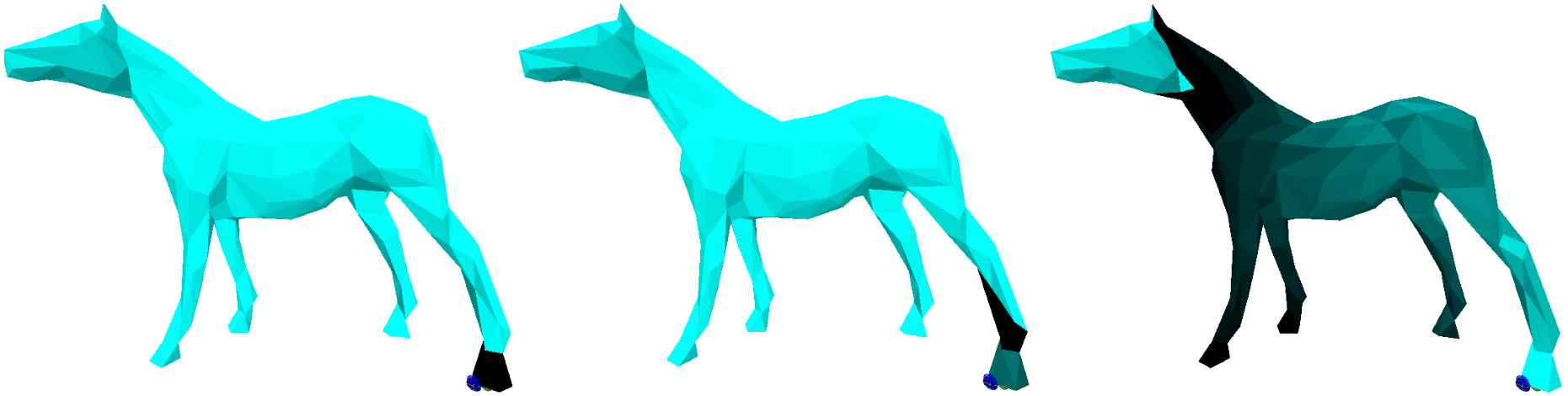
```
void fanHoles() {for (int cc=0; cc<nc; cc++)
    if (visible[t(cc)]&&b(cc)) fanThisHole(cc);   }
```



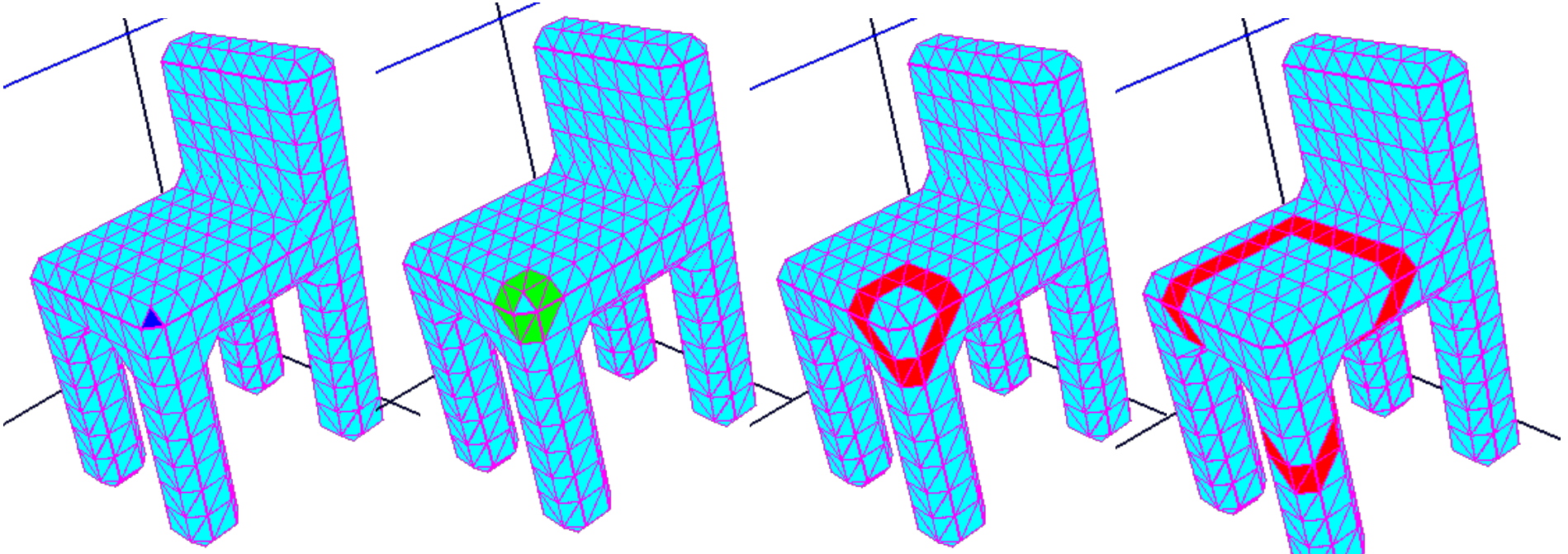
Or may fill 2 holes to make a through-hole

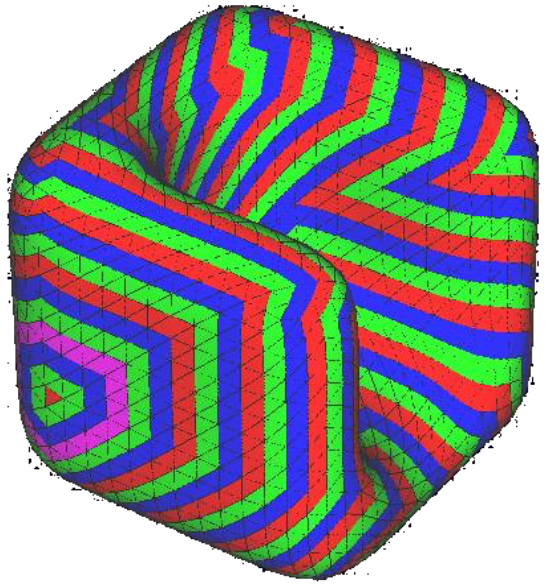
Compute geodesic distance

```
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;   }
```

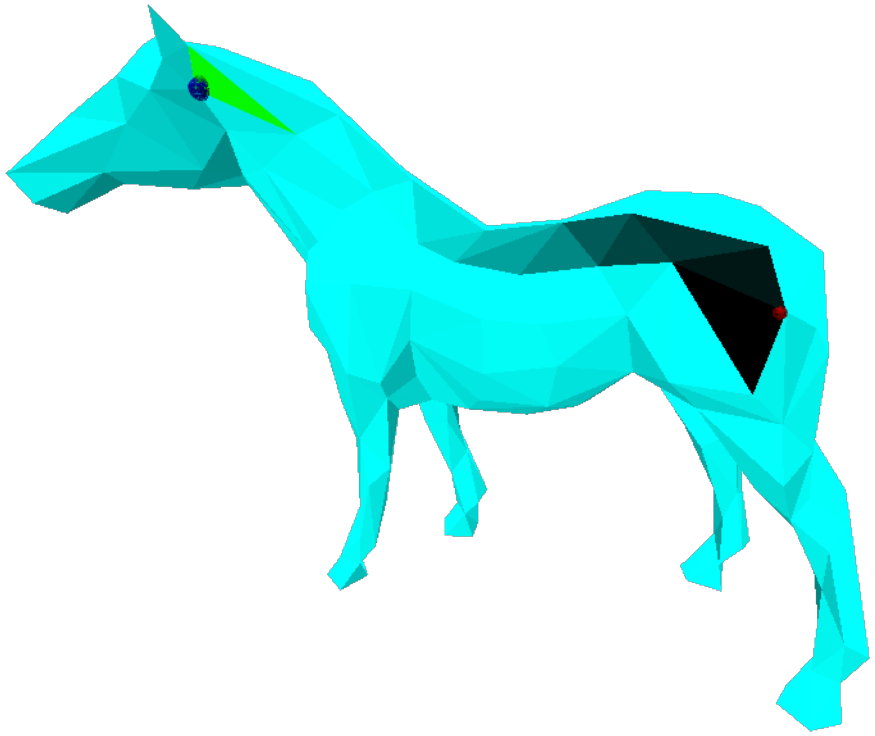


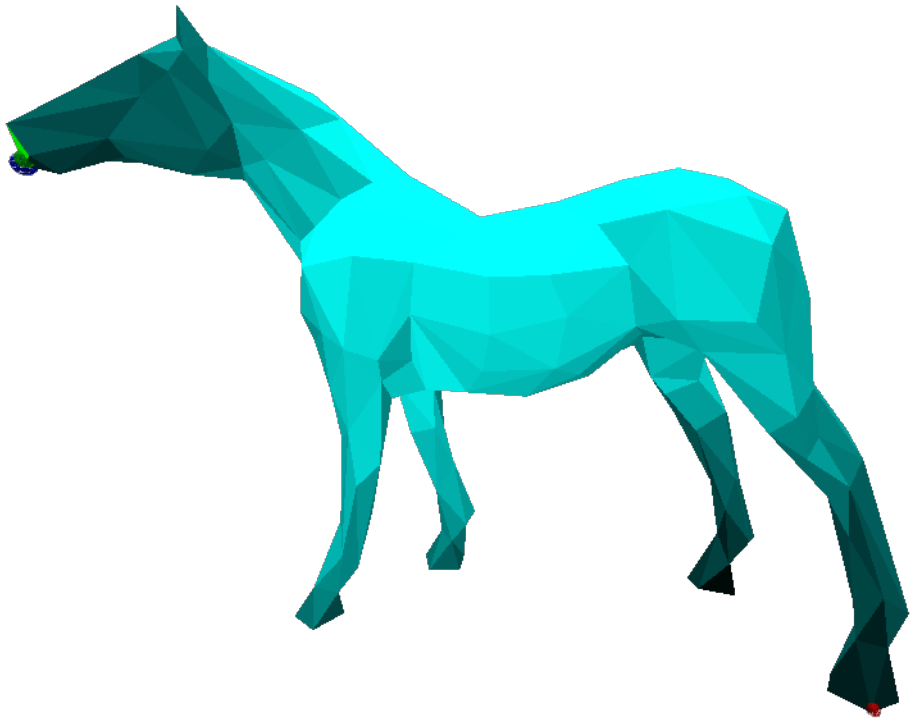
Geodesic (graph) distance



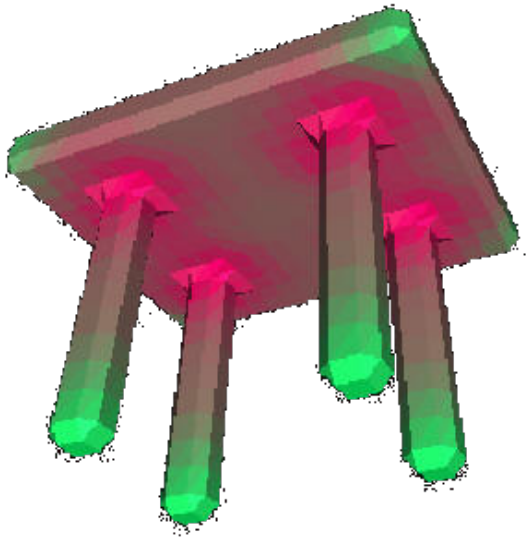


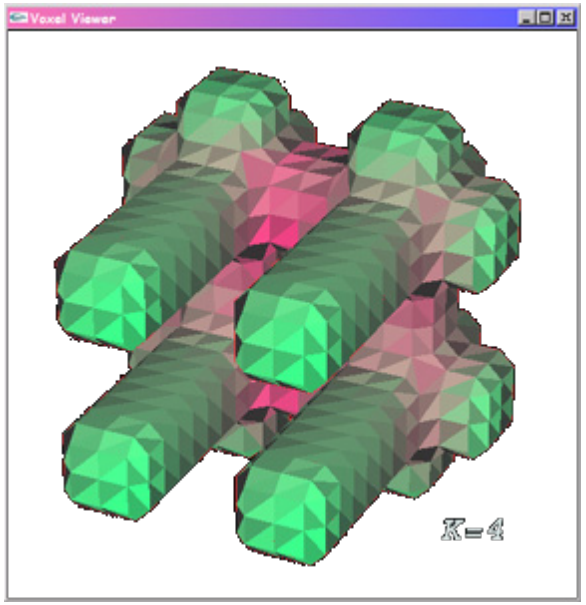
Path and isolation

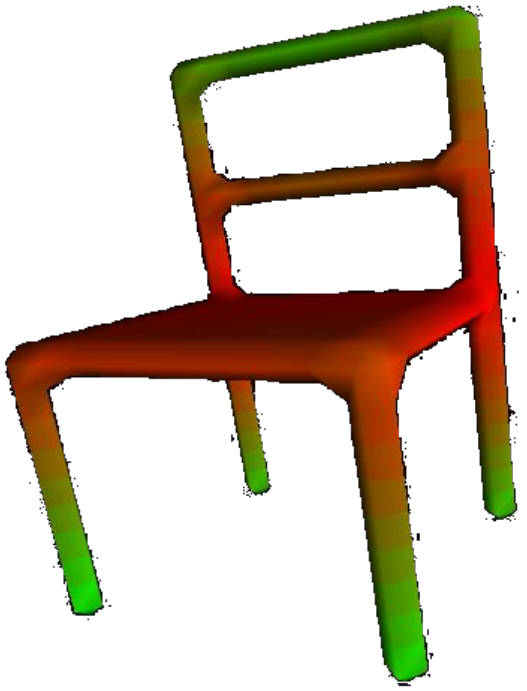




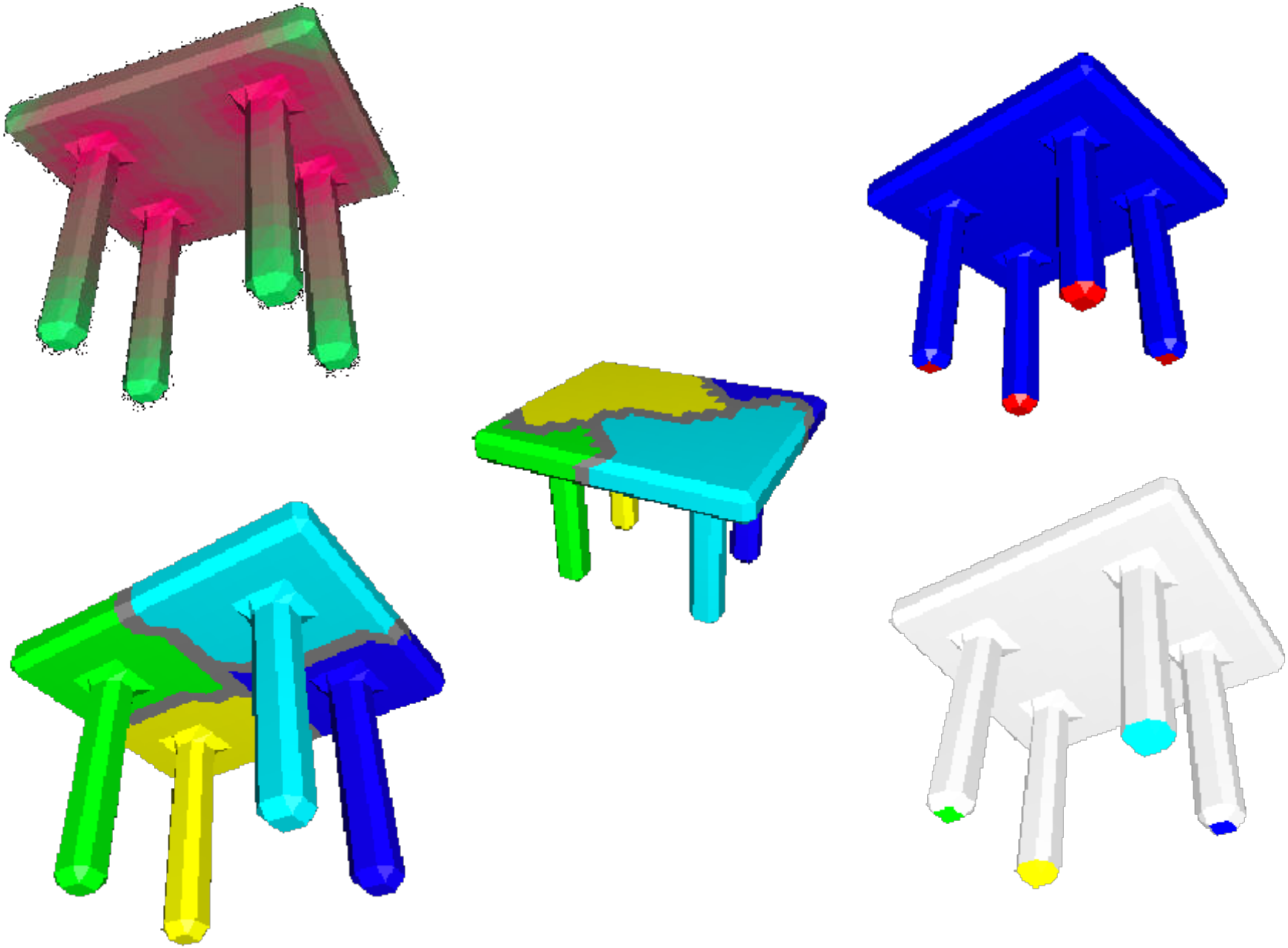
Isolation



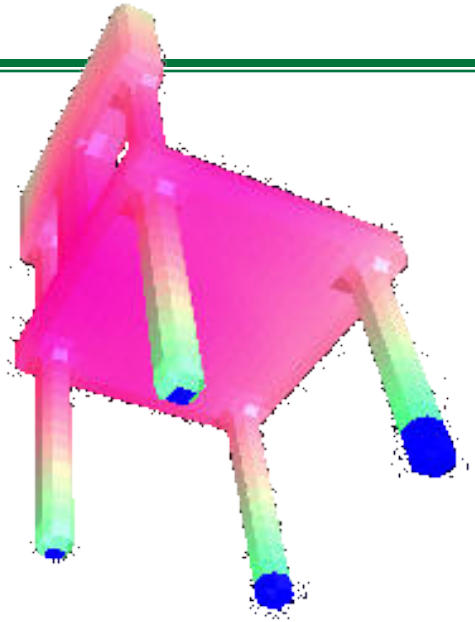


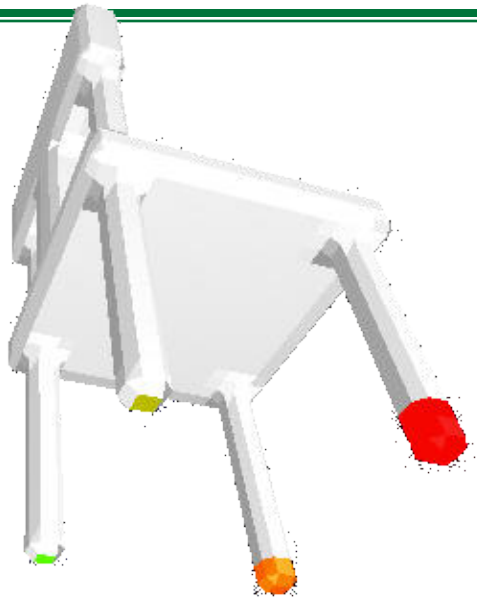


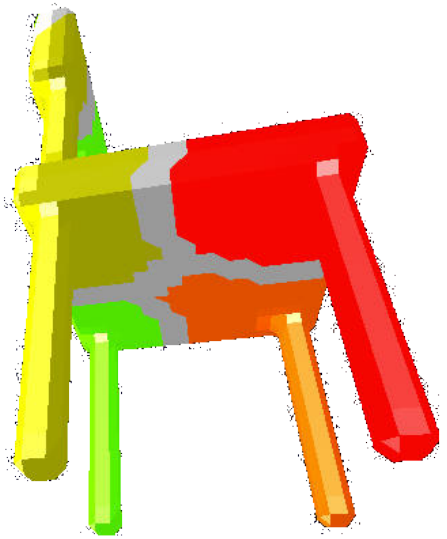
Segmentation

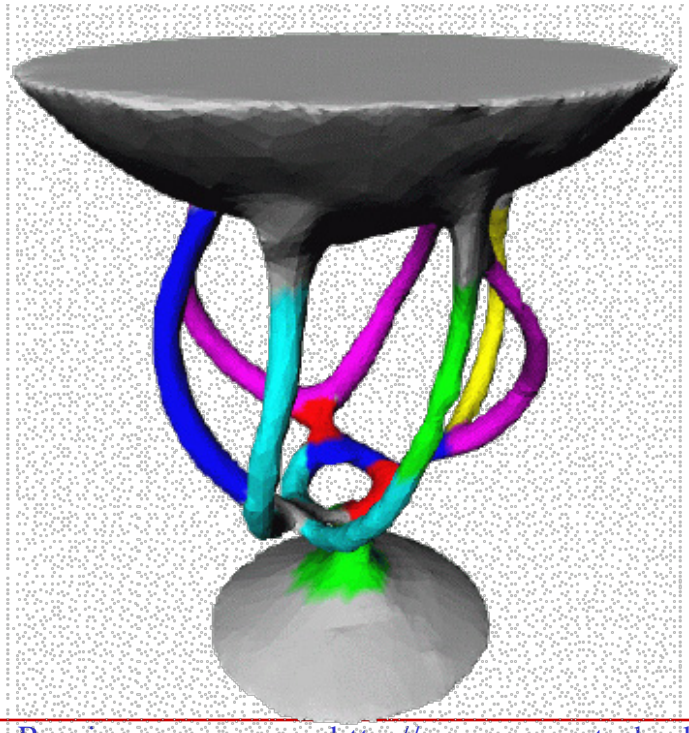


More segmentation

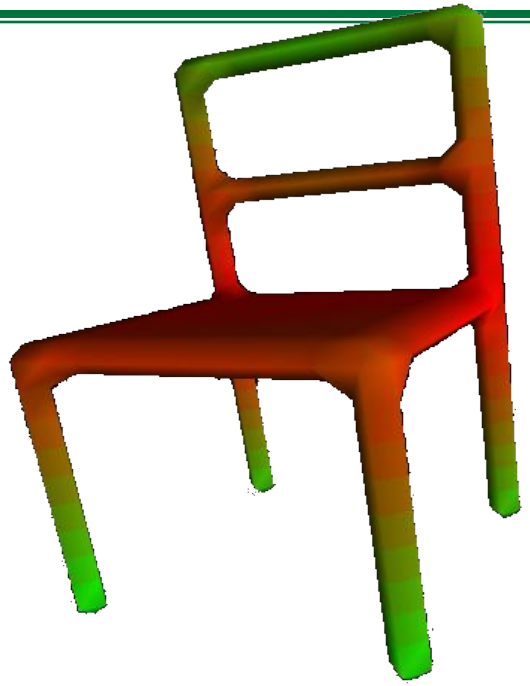


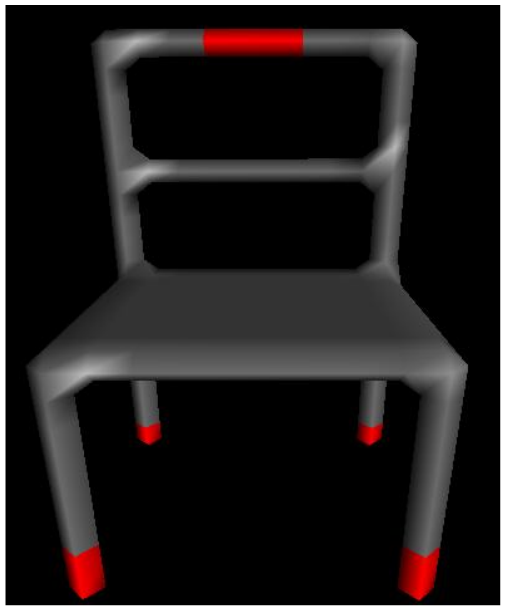


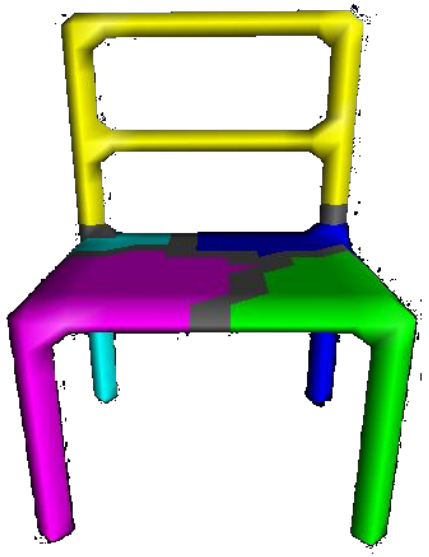


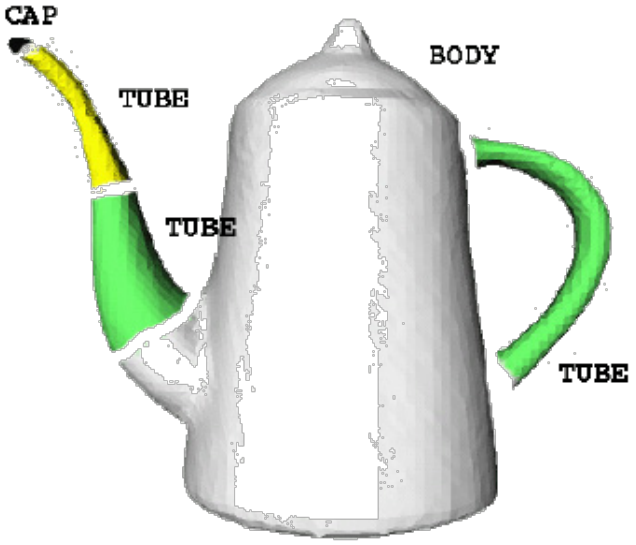


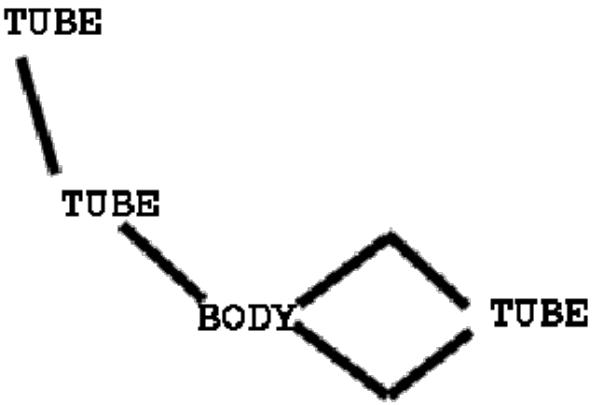
Skeleton



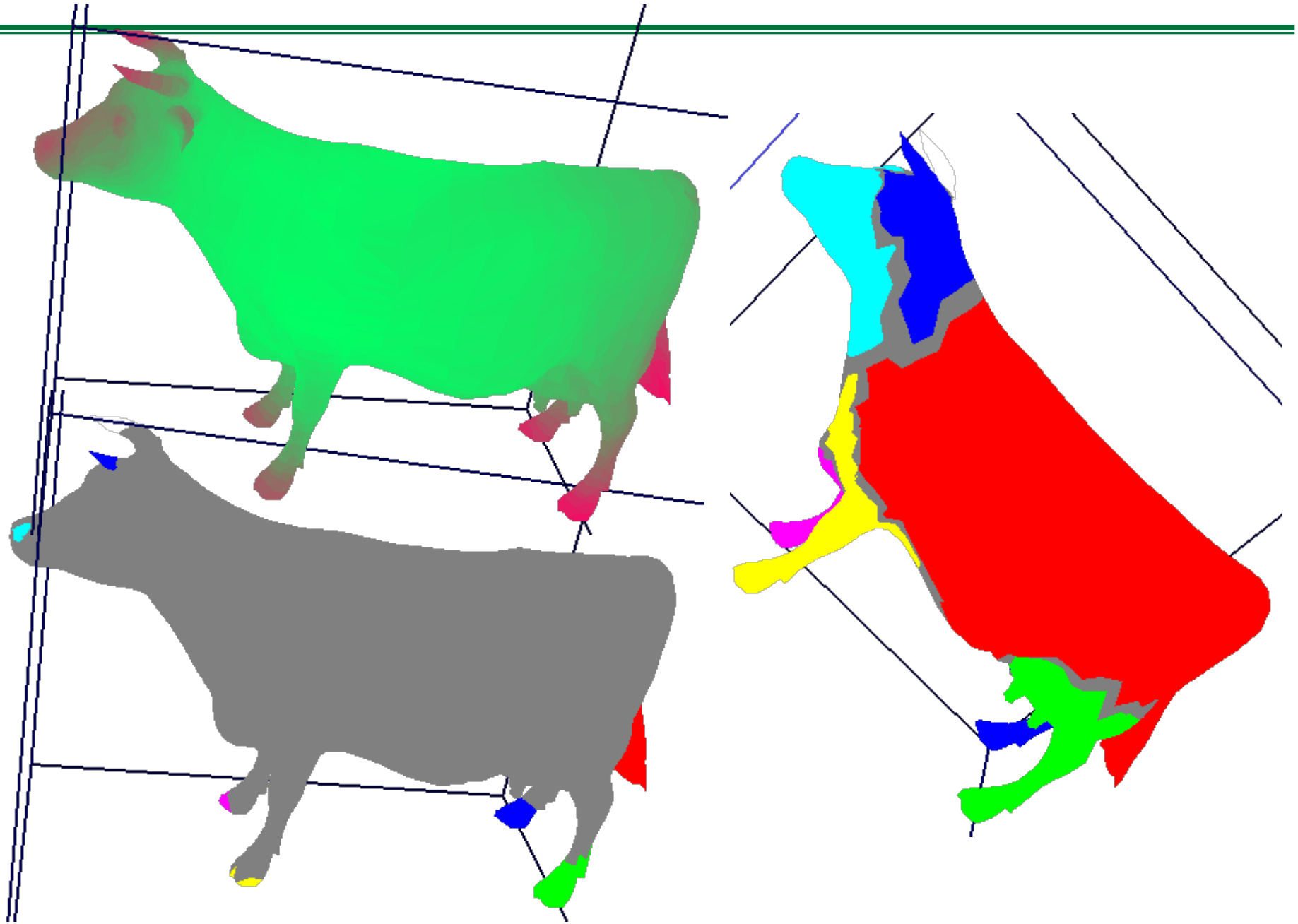






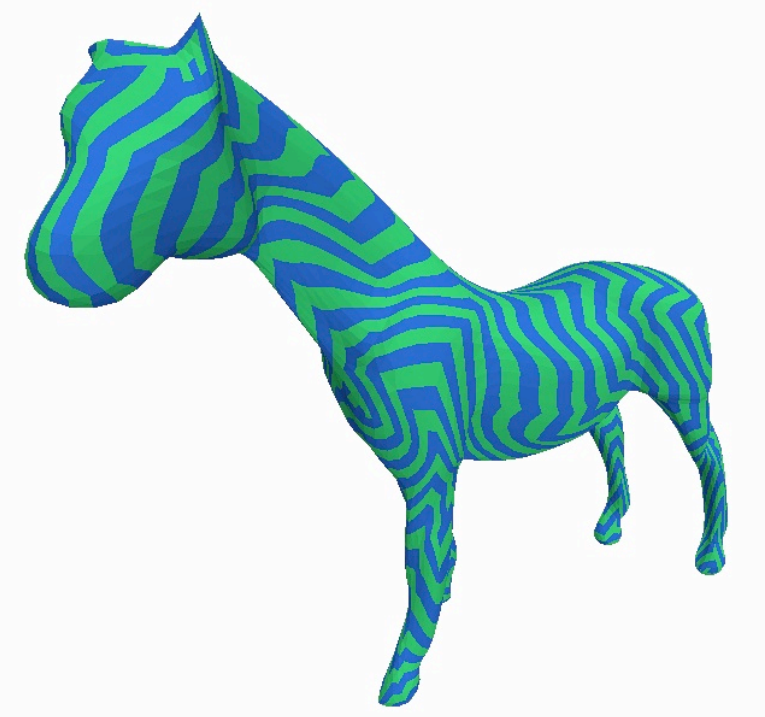


More chicken?



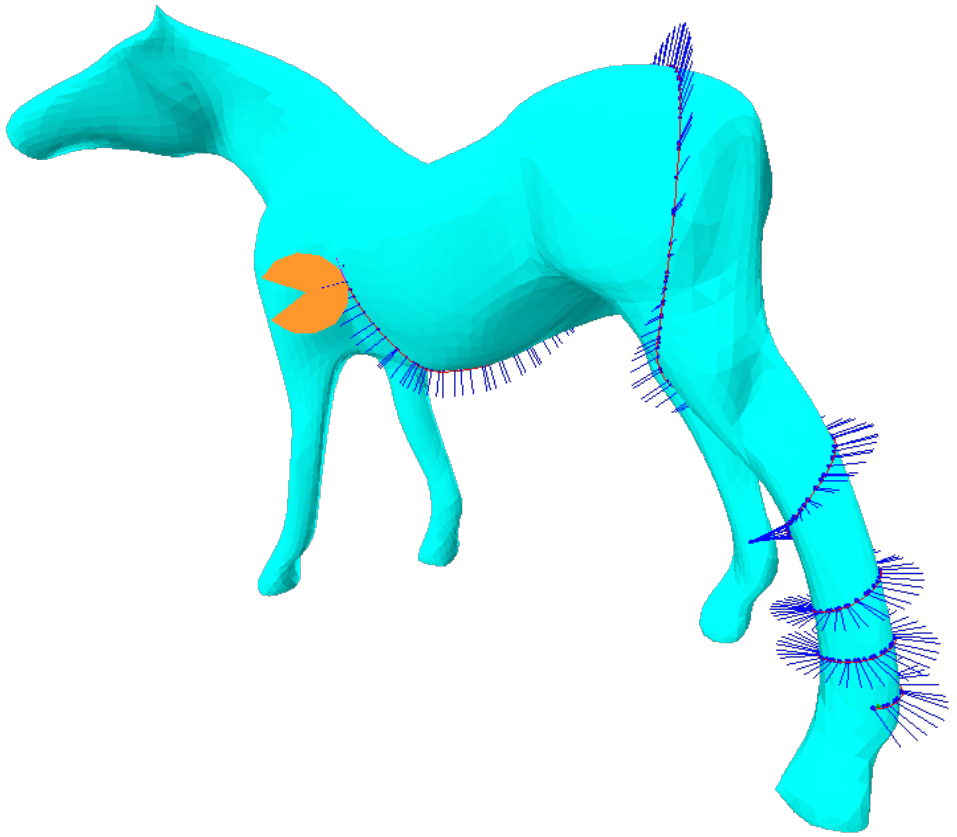
Hamiltonian cycle

- Cycle of edges that visits each vertex only once
- Splits mesh into 2 corridors
- Useful for compression
- May not exist or may be hard to find
- May often be computed, especially for subdivided meshes



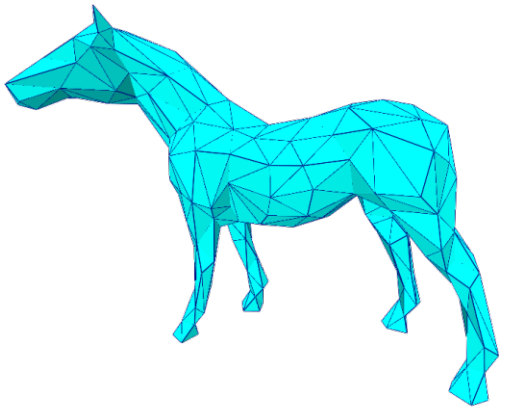
Ride on a mesh

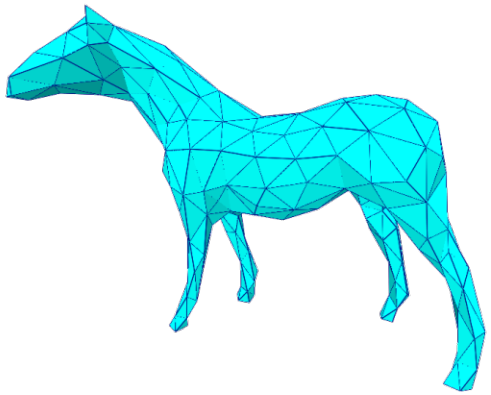
- Propagate velocity from one triangle to the next
- Compute new velocity using ray-reflection formula

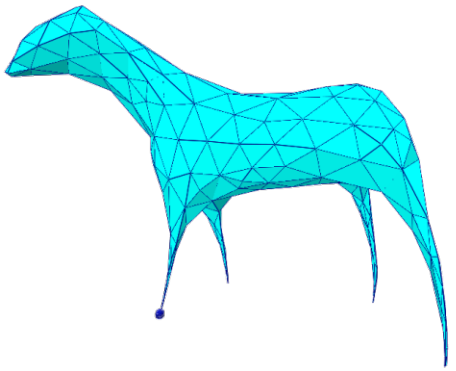


Smoothing

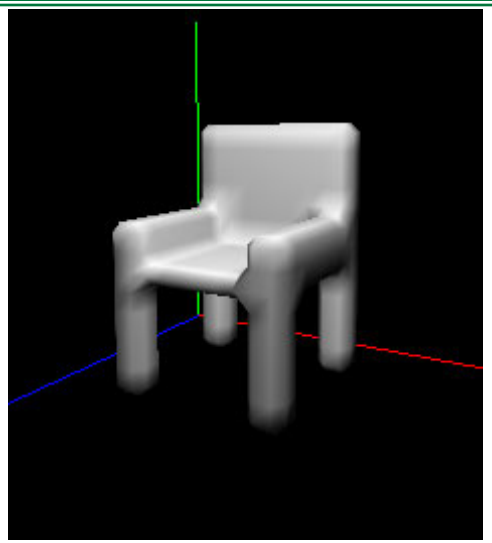
```
void computeLaplaceVectors()  {
  for (int i=0; i<nv; i++) {Nv[i].setTo(0,0,0); Valence[i]=0;};
  for (int i=0; i<nc; i++) {Valence[v(i)]++;  };   }
  for (int i=0; i<nc; i++) {Nv[v(p(i))].add(g(p(i)).vecTo(g(n(i))));};
  for (int i=0; i<nv; i++) {Nv[i].div(Valence[i]);};    };
void tuck(float s) {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); };
```

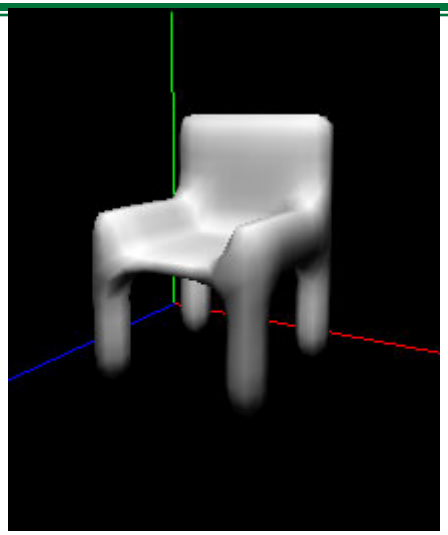


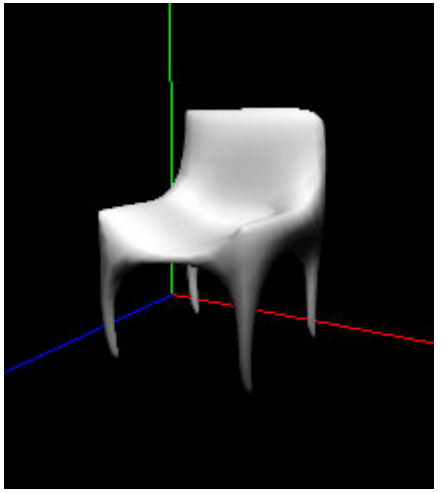


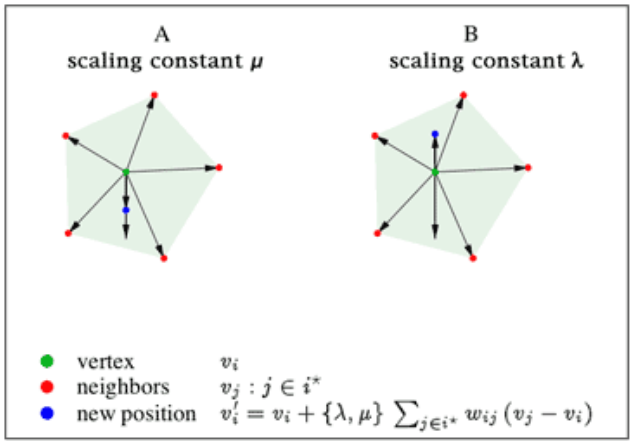


Smoothing (Taubin)

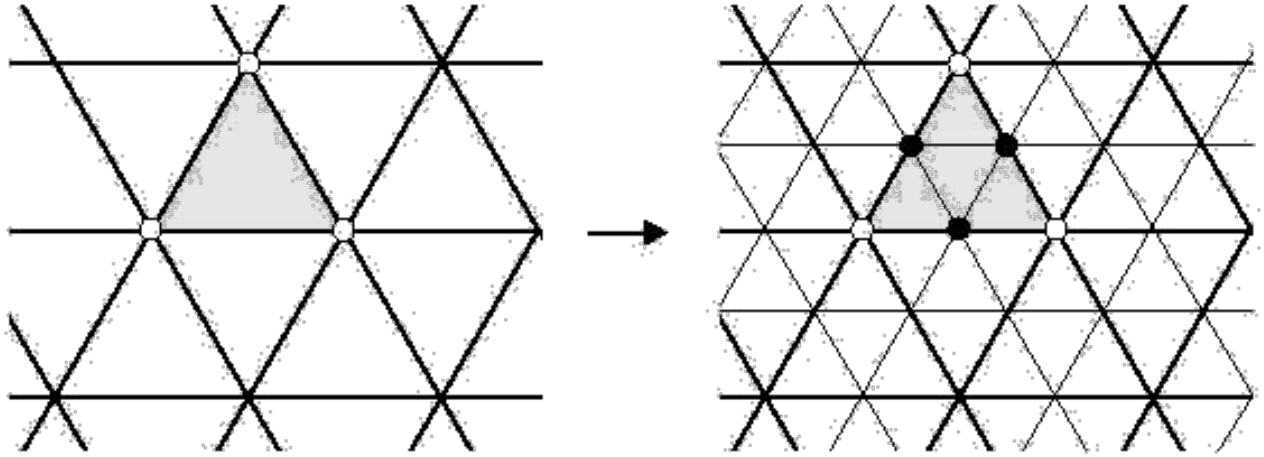


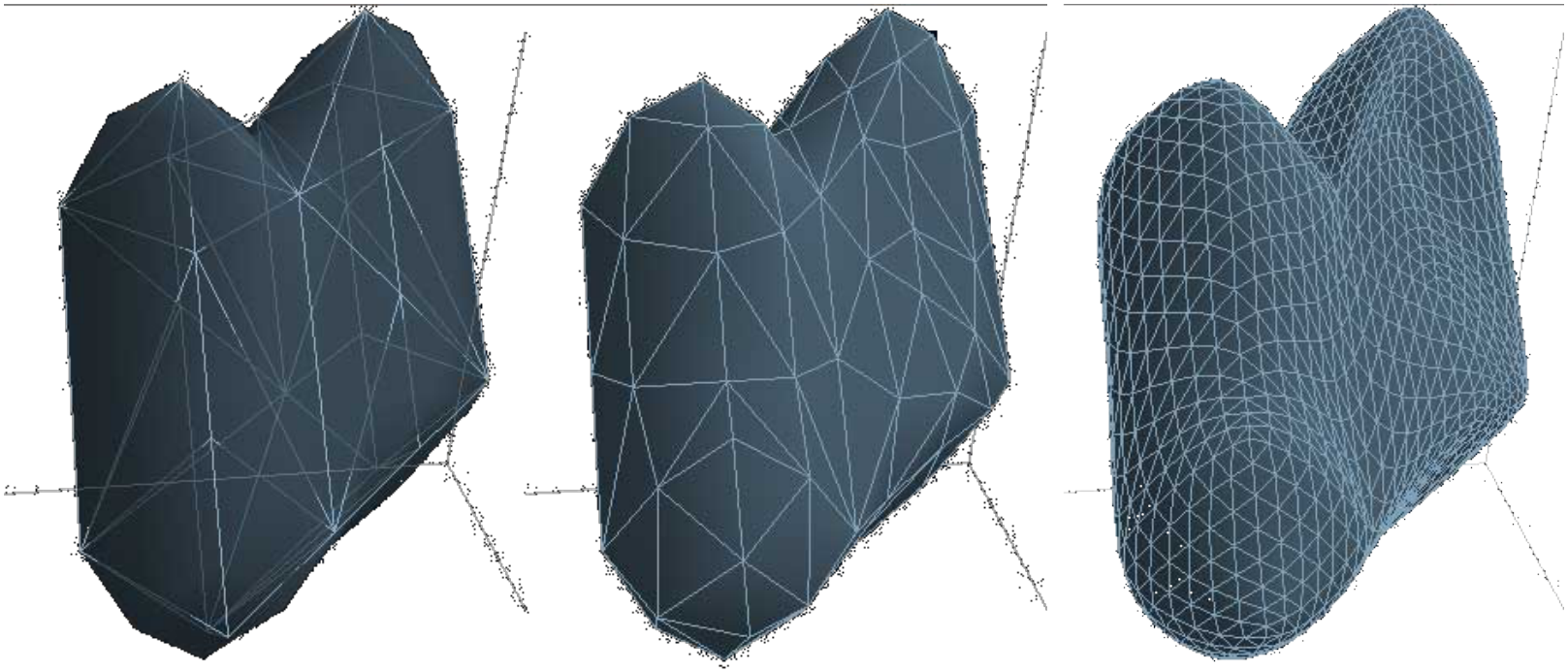






Subdivision

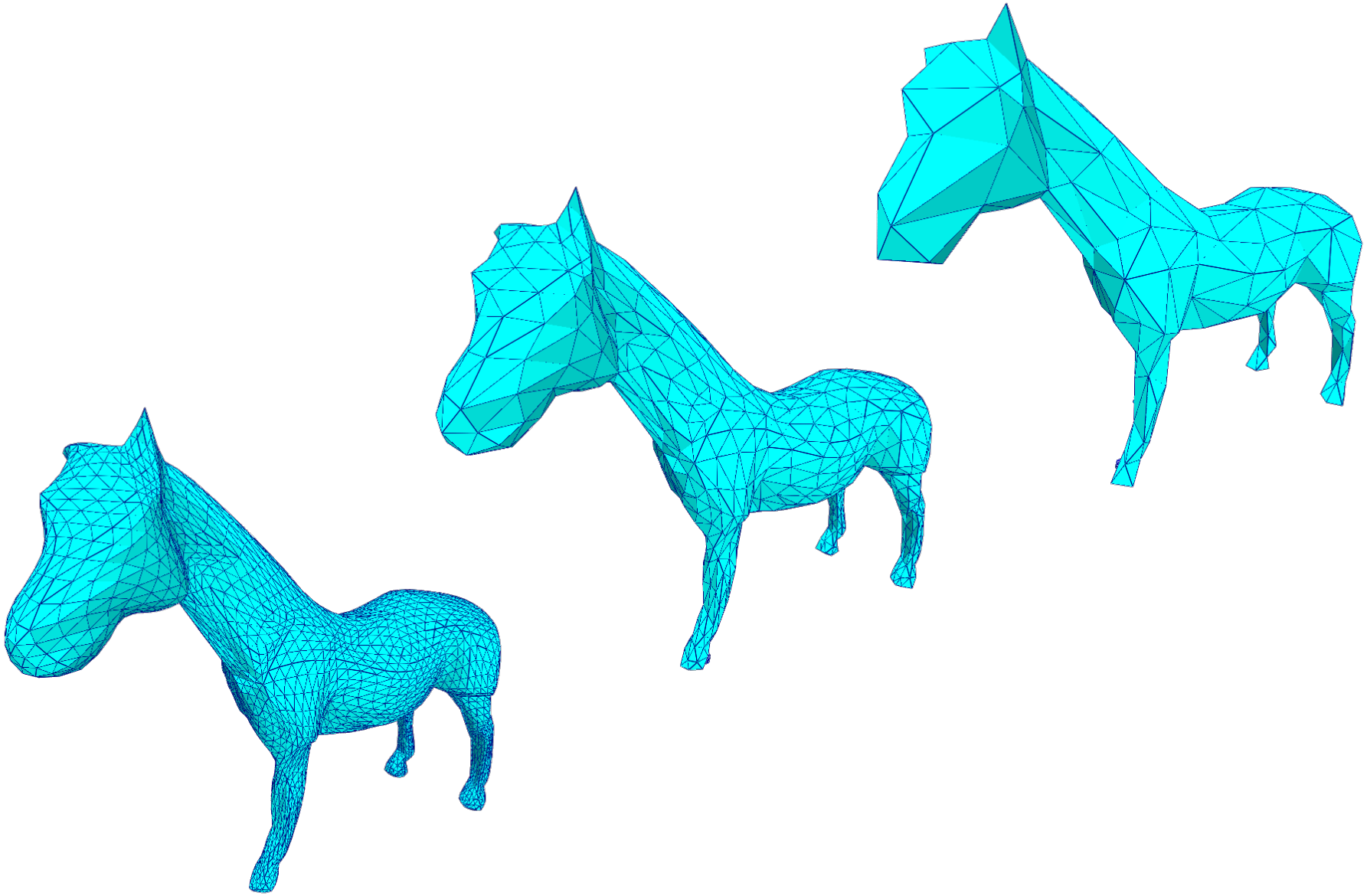




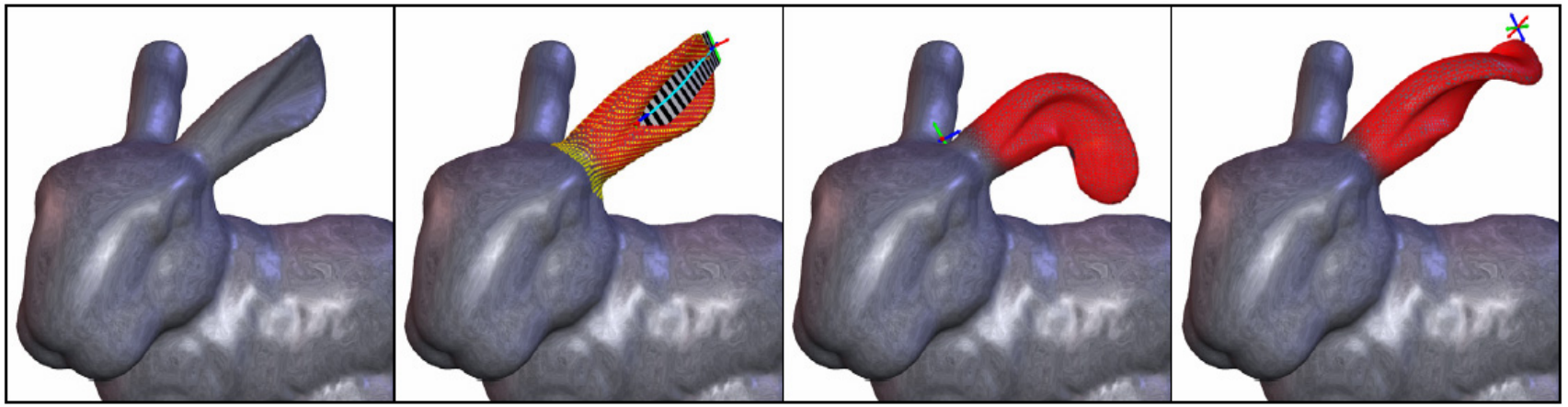
Subdivision

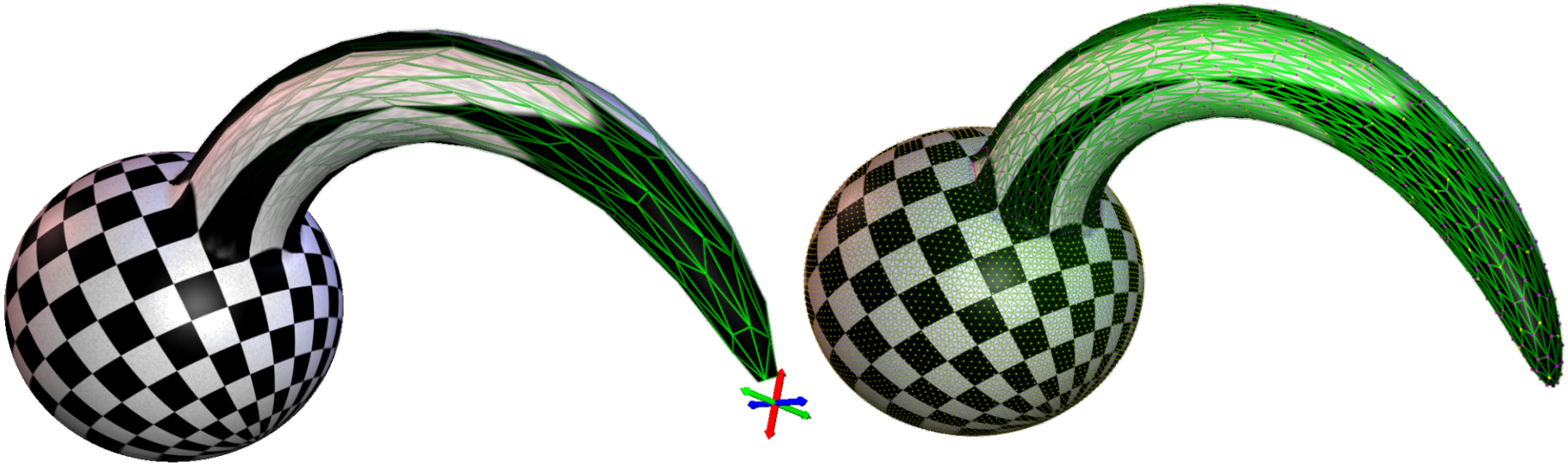
```
void splitEdges() {for (int i=0; i<nc; i++) {
  if(b(i)) {G[nv]=midPt(g(n(i)),g(p(i))); W[i]=nv++;}
  else {if(i<o(i)) {G[nv]=midPt(g(n(i)),g(p(i))); W[o(i)]=nv; W[i]=nv++; }}}}
void bulge() {   // tweaks the mid-edge vertices
  for (int i=0; i<nc; i++) {
   if((!b(i))&&(i<o(i))) {    // no tweak for mid-vertices of border edges
    if (!b(p(i))&&!b(n(i))&&!b(p(o(i)))&&!b(n(o(i))))
     {G[W[i]].addScaledVec(0.25,midPt(midPt(g(l(i)),g(r(i))),
          midPt(g(l(o(i))),g(r(o(i))))).vecTo(midPt(g(i),g(o(i))))); } } } }
void splitTriangles() {    // splits each tirangle into 4
  for (int i=0; i<3*nt; i=i+3) {
    V[3*nt+i]=v(i); V[n(3*nt+i)]=w(p(i)); V[p(3*nt+i)]=w(n(i));
    V[6*nt+i]=v(n(i)); V[n(6*nt+i)]=w(i); V[p(6*nt+i)]=w(p(i));
    V[9*nt+i]=v(p(i)); V[n(9*nt+i)]=w(n(i)); V[p(9*nt+i)]=w(i);
    V[i]=w(i); V[n(i)]=w(n(i)); V[p(i)]=w(p(i));    };
  nt=4*nt; nc=3*nt;  };
```

Subdivision examples



Adaptive refinement





Self-crossing