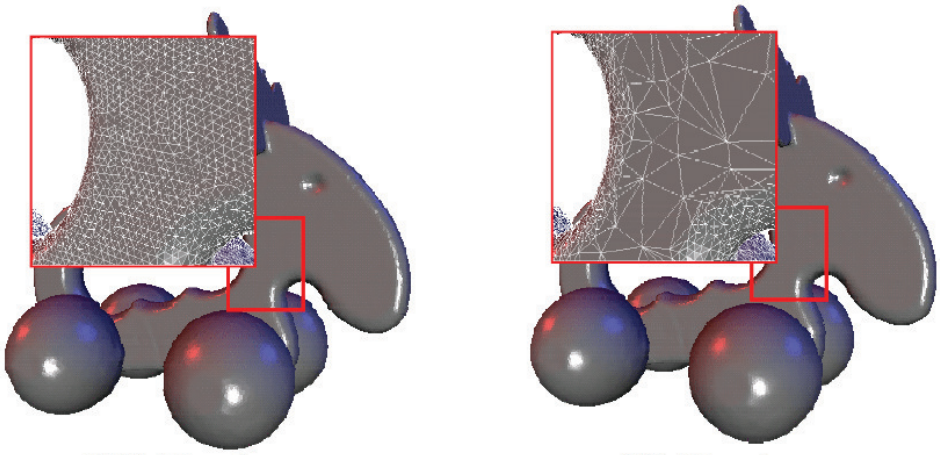
Motivation

- Oversampled 3D scan data

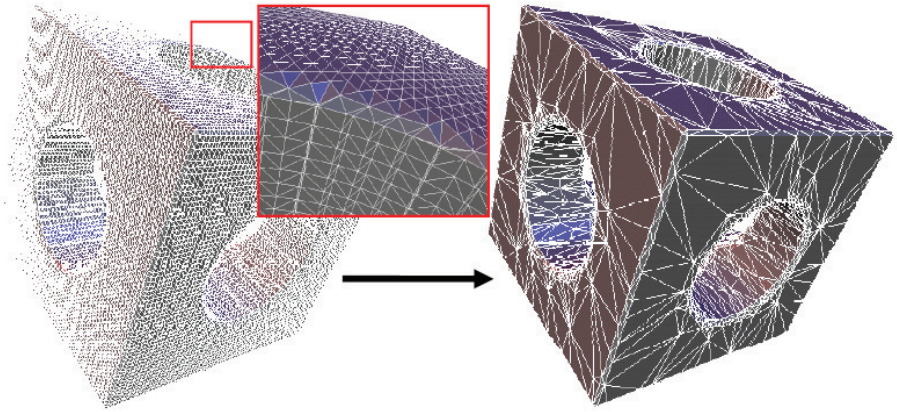


~150k triangles ~80k triangles

There are several reasons to want to simplify 3D models. In the case of scanning technologies, as we do not know the shape of the model to capture, we acquire as much data as we can. Still, some regions are going to be planar enough to be candidates for geometry simplification.

Motivation

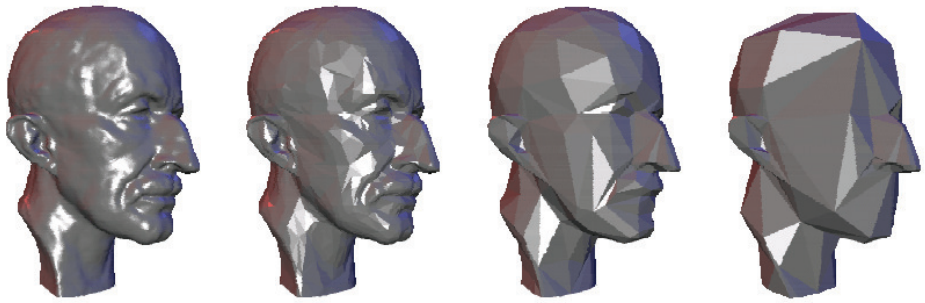
- Overtessellation: E.g. iso-surface extraction



Some modeling algorithms like Marching Cubes, which is useful to extract meshes from volumetric models, also produce overtessellation. In these cases we also need simplification.

Motivation

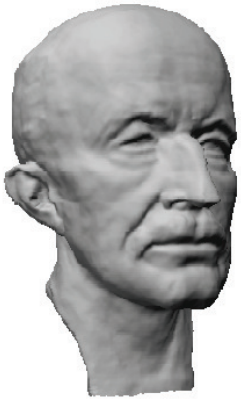
- Multi-resolution hierarchies for
  - efficient geometry processing
  - level-of-detail (LOD) rendering



But the reason we are interested in geometry simplification in this course is that it allows us to render more complex scenes. If we have a high detail model we can generate progressively simplified versions to compose a level-of-detail (LOD) hierarchy. Then we can use the versions of the scene's models that allow us to render the scene in real-time while maintaining the quality of the resulting render.

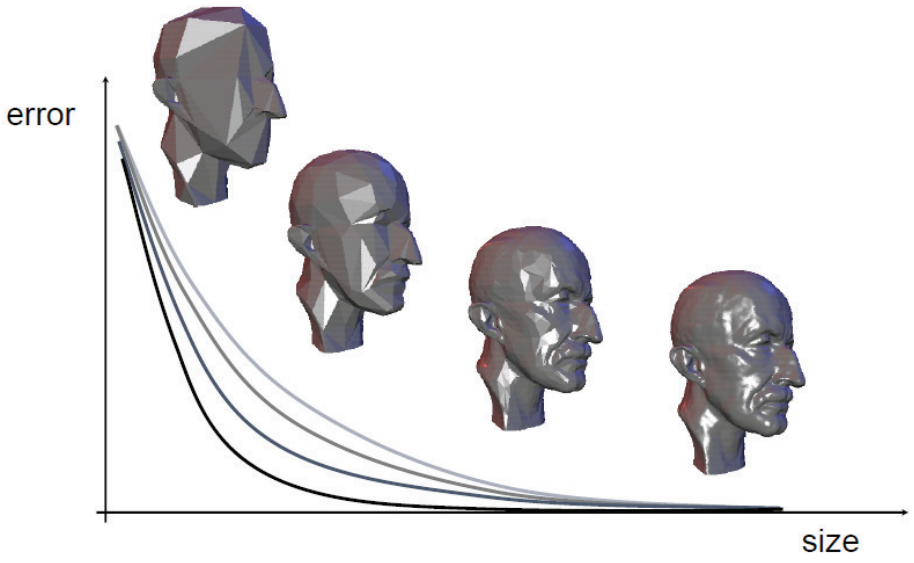
Motivation

- Adaptation to hardware capabilities



This is especially useful when the capabilities of the available hardware are limited. Even more, ideally, creators can model their scenes without any regard for performance, and then the application may use simplification and LOD strategies to adapt the geometry to the hardware's power.

Size-quality tradeoff



Still, these progressive simplification strategies have limits. There is an inverse relationship between error and model size (in number of primitives). The problem is that this inverse function is not linear. If I double the number of primitives I will not reduce the geometrical error by half. Also, as I reduce the number of primitives the rate at which the error increases does not remain stable. The result of this is that, for a model, there is a limit to how much I can reduce a model, and still get a similar enough shape.

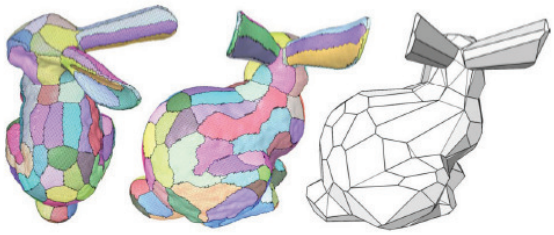
Problem Statement

- Given: $\mathcal{M} = (\mathcal{V}, \mathcal{F})$
- Find: $\mathcal{M}' = (\mathcal{V}', \mathcal{F}')$ such that
  1. $|\mathcal{V}'| = n < |\mathcal{V}|$ and $\|\mathcal{M} - \mathcal{M}'\|$ is minimal, or
  2. $\|\mathcal{M} - \mathcal{M}'\| < \epsilon$ and $|\mathcal{V}'|$ is minimal
- Respect additional fairness criteria
  - normal deviation, triangle shape, scalar attributes, etc.

So what we want is, given a mesh M composed of a set of vertices V and a set of faces F, find a simplified mesh M'. We are also going to want M' to approximate M. We will get that by requiring one of two things. Either M' has a certain number of primitives and we minimize the amount of error between M and M'. Or M' has as few primitives as possible, while guaranteeing that the error between M and M' is below a certain threshold. The first strategy is mainly useful for rendering, while the second is the one to choose for engineering applications. We can also design our simplification algorithms so that they respect other criteria.

Mesh Decimation Methods

- **Vertex clustering**
- **Incremental decimation**
- Resampling
- Mesh approximation



There are four main types of simplification algorithms. Vertex clustering and incremental decimation are the most useful and popular.

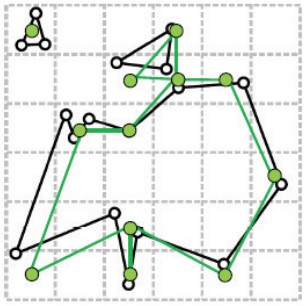
Vertex Clustering

- Cluster Generation
- Computing a representative
- Mesh generation
- Topology changes

Vertex clustering was the first published simplification algorithm. It is a very simple algorithm that can be very easily adapted to GPUs.

Vertex Clustering

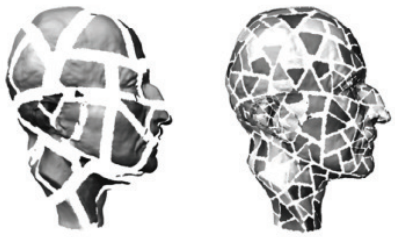
- Cluster Generation
  - Uniform 3D grid
  - Map vertices to cluster cells
- Computing a representative
- Mesh generation
- Topology changes



First we compute the bounding box of our model, which we divide into cells using a uniform grid. We compute which cells contain vertices from our model. As a result, we can cluster the vertices of the input mesh. Each cell determines one such cluster. For each cluster we will generate one vertex of the output simplified mesh.

Vertex Clustering

- Cluster Generation
  - Hierarchical approach
  - Top-down or bottom-up
- Computing a representative
- Mesh generation
- Topology changes



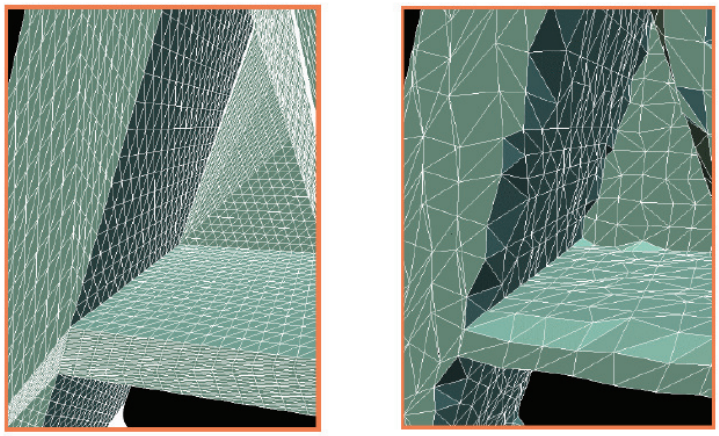
It is also possible to generate the clusters without using a uniform grid. We may use any other spatial data structure. Also, as these spatial data structures can be built top-down or bottom-up, so can the clusters.

Vertex Clustering

- Cluster Generation
- Computing a representative
  - Average/median vertex position
  - Error quadrics
- Mesh generation
- Topology changes

Now, we need to compute the position of the output vertex resulting from each cluster. There are several methods to do this.

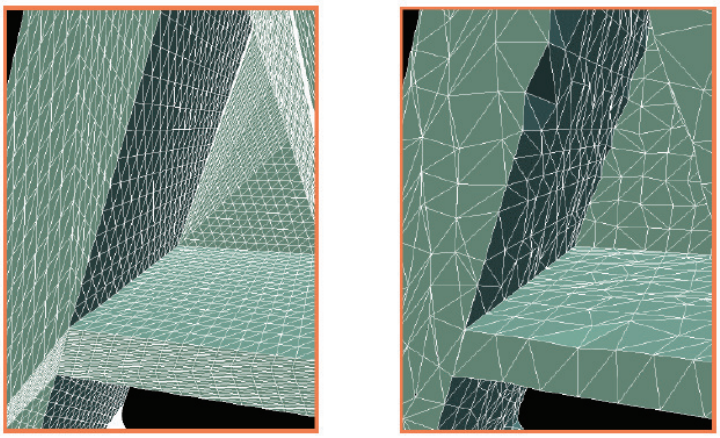
Vertex Clustering



Average vertex position

The simplest method is to compute the average of the vertices in each cluster. The problem with this approach is that the computed vertex does not represent the shape of the original mesh well enough.

Vertex Clustering

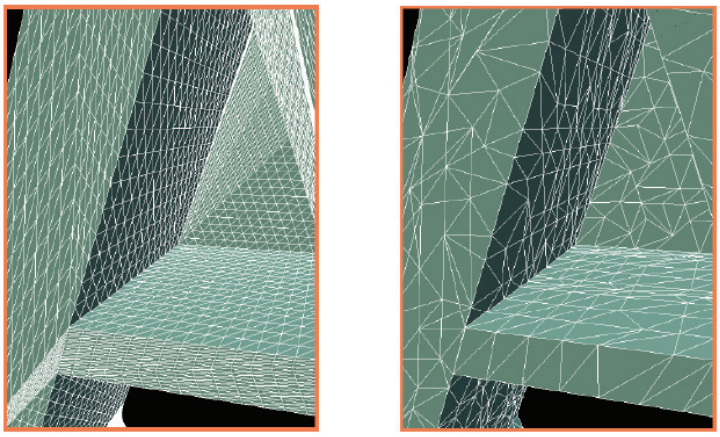


Median vertex position

Instead we can compute the 3D analog of the median. This is what is done in [1]. The result is better, but as this operator can only be approximated by using iterative filters the result is still lacking.

[1] “Mesh Smoothing via Mean and Median Filtering Applied to Face Normals”, Hirokazu Yagou, Yutaka Ohtake, Alexander G. Belyaev

Vertex Clustering

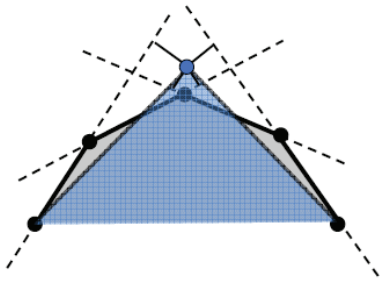


Error quadrics

The method that works best is using a quadric approximation to the shape local to the cluster. This is connected to the vertex-plane distance error we saw on the previous session.

Quadratic Error

- Minimize distance to neighboring triangles' planes



The idea is that each representative vertex has to represent the shape of its cluster. This shape comes from the triangles that are adjacent to the vertices in the cluster. So, once we have the triangles connected to the cluster, how do we compute this vertex? We compute it so that it minimizes the distance to the planes of the triangles.

Quadratic Error

- Squared distance of point p to plane q:

$$p = (x, y, z, 1)^T, \quad q = (a, b, c, d)^T$$

$$dist(q, p)^2 = (q^T p)^2 = p^T (q q^T) p =: p^T Q_q p$$

$$Q_q = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

As in any minimization problem, we will use least squares to get the position of the representative vertex p. The distance between p and one of the planes q will be $a \cdot p_x + b \cdot p_y + c \cdot p_z + d$. If we square this distance, we can represent it as the matrix product $p^T Q p$, where Q is called the quadric of plane q. The matrix Q can be directly computed from the parameters a, b, c, and d of the plane q.

Quadratic Error

- Sum distances to planes q_i of vertex' neighboring triangles:

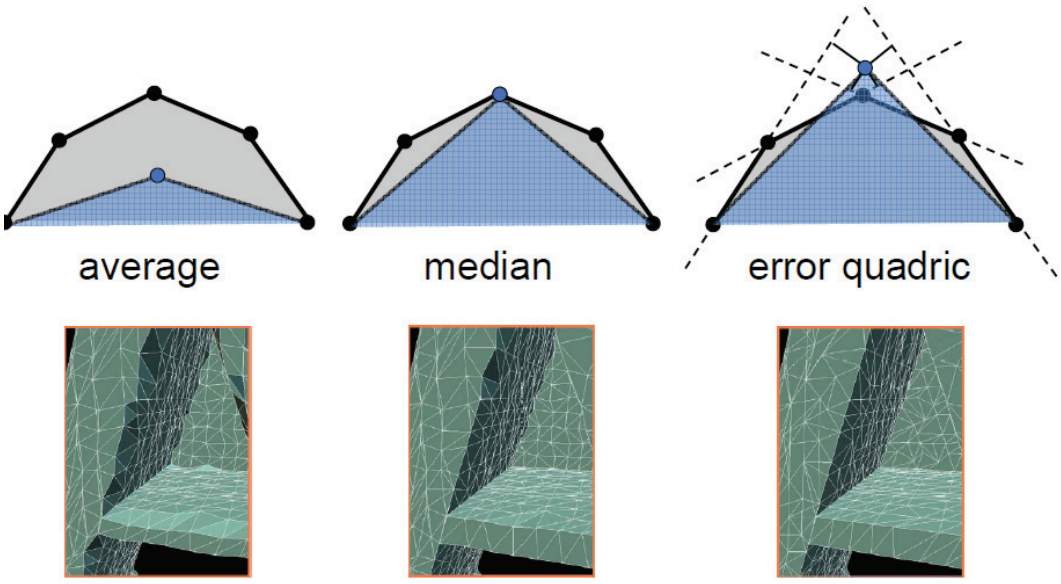
$$\sum_i dist(q_i, p)^2 = \sum_i p^T Q_{q_i} p = p^T \left(\sum_i Q_{q_i} \right) p =: p^T Q_p p$$

- Point p^* that minimizes the error satisfies:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} p^* = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

But for each vertex we have several triangles and so several planes. We need to compute the sum of the squared distances between p and all the planes. It just so happens that this sum can also be expressed as a matrix product $p^T Q_p p$. In fact Q_p is the sum of the quadric matrices of all the planes. Now we need to find the point p* that minimizes this sum. For that we may derive the expression and make it equal to zero. The result is an equation $Q^* p^* = 0$, where Q^* is just Q_p with its last row replaced by that of an identity matrix.

Comparison



Of course, depending on the quality we want we may choose any of this methods for the computation of the representative of each cluster. Still, quadric errors give the best result in all cases.

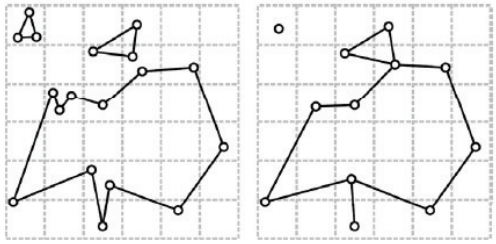
Vertex Clustering

- Cluster Generation
- Computing a representative
- Mesh generation
  - Clusters $p \leftrightarrow \{p_0,\ldots,p_n\}$, $q \leftrightarrow \{q_0,\ldots,q_m\}$
  - Connect (p,q) if there was an edge (p_i,q_j)
- Topology changes

Once we have the representative vertices, we already have all the vertices of the output mesh. We can collect them into a vector, so that the position of each output vertex is its index. We still need the triangles of the output mesh, but they may also be computed from the input triangles.

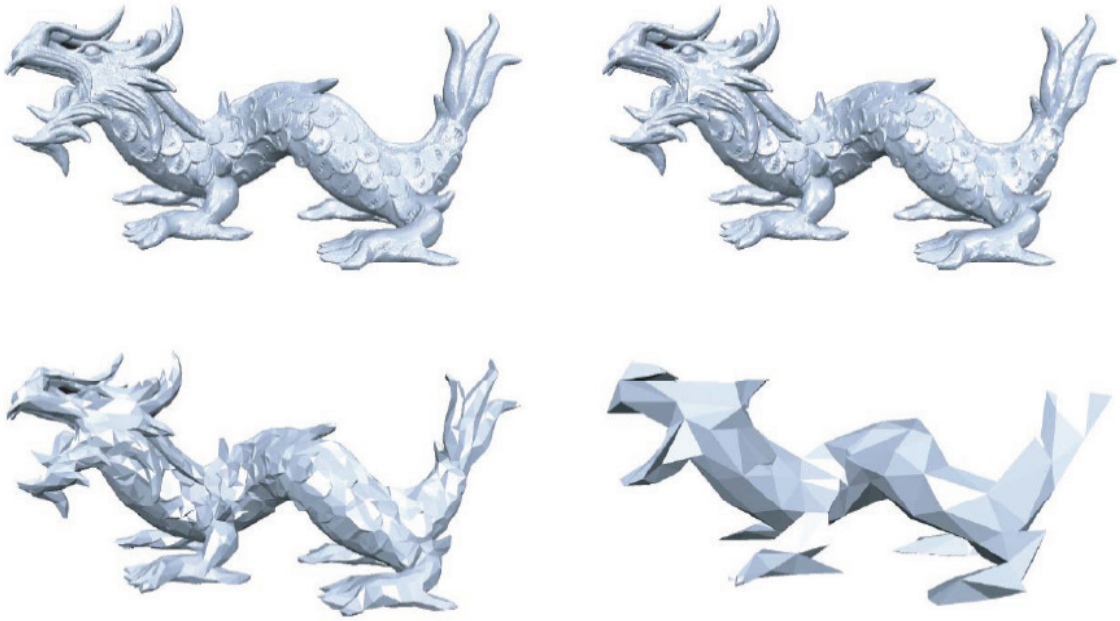
Vertex Clustering

- Cluster Generation
- Computing a representative
- Mesh generation
- Topology changes
  - If different sheets pass through one cell
  - Can be non-manifold



For each input triangle, we substitute the index of each of its input vertices to the corresponding output vertex index. Remember that each input vertex was in a cluster, so it has a corresponding output vertex. If all the three indices of the updated triangle are different, then we add it as a new triangle of the output mesh. If this is not the case, then the triangle has degenerated into a vertex or a segment and we do not add it to the output mesh.

Incremental Decimation



The other popular strategy is to apply incremental decimation.

Incremental Decimation

- General Setup
- Decimation Operators
- Error metrics
- Topology changes

Greedy Approach

- For each region
- evaluate quality after decimation
- enqeue(quality, region)

- Repeat:
- get best mesh region from queue
- apply decimation operator
- update queue
- Until no further reduction possible

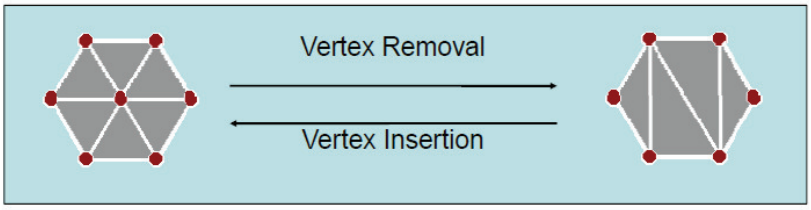
This method uses a greedy approach. First, we initialize a queue with all the decimation operations possible in the input mesh. This operations are ordered by the error they introduce. Then, iteratively, we extract one operation from the queue, apply it to the mesh, and update the queue correspondingly (as the error of some of the enqueued operations may have changed). We do this until we cannot apply any more reduction.

Decimation Operator

- Vertex Removal
- Edge Collapse
- Half Edge Collapse

As we already saw there are three main decimation operators. Vertex clustering we have already used. And as we will see edge collapse can be separated into two different operations depending on how we apply it.

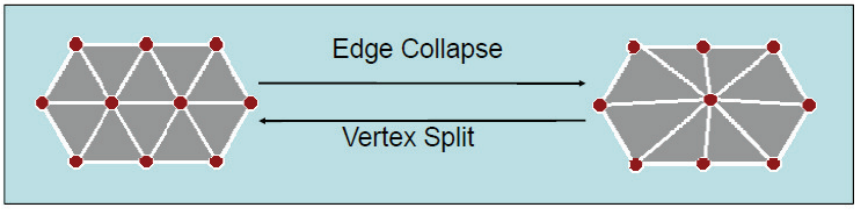
Vertex Removal



- Remove vertex
- Re-triangulate hole
  - Combinatorial degrees of freedom

Vertex removal deletes a vertex from the mesh, leaving a hole behind that needs to be re-triangulated. As such, it respects the topology of the mesh, but we have several options for the re-triangulation. Depending on the shape, some of these may cause self-intersections.

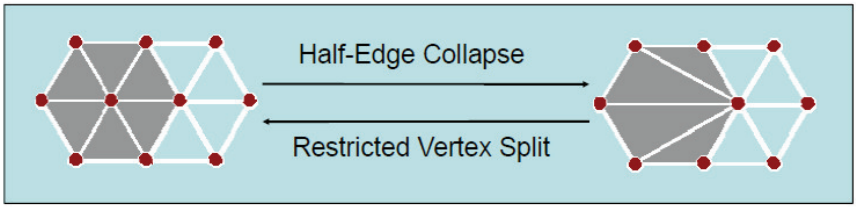
Edge Collapse



- Merge two adjacent vertices
- Define new vertex position
  - Continuous degrees of freedom

With edge collapse we move two vertices connected by an edge to a common position and, in the process, remove one of those vertices and the two faces adjacent to the selected edge. This can be done gradually, as the two vertices may move progressively to the selected final position. Also, we can define an inverse operation, the vertex split, that separates a single vertex into two connected by a new edge, while introducing two faces on two edges connected to the original vertex. We can have a vertex split that restores the topology we had before doing an edge collapse.

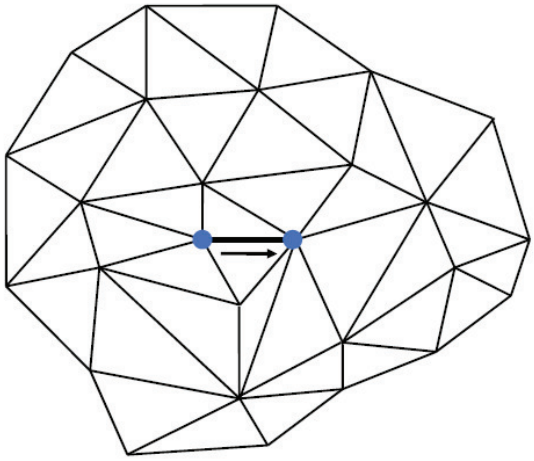
Half Edge Collapse



- Collapse edge into one end point
  - Special case of vertex removal
  - Special case of edge collapse
- No degrees of freedom

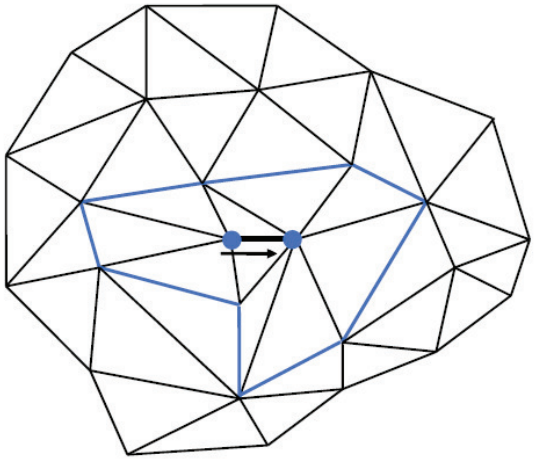
Instead of moving the two vertices to a new position that best represents the geometry we are removing, we can move one of the vertices to the position of the other. We call this a half-edge collapse, because the direction of the collapse can be defined by one of the half-edges of the selected edge.

Half-Edge Collapse

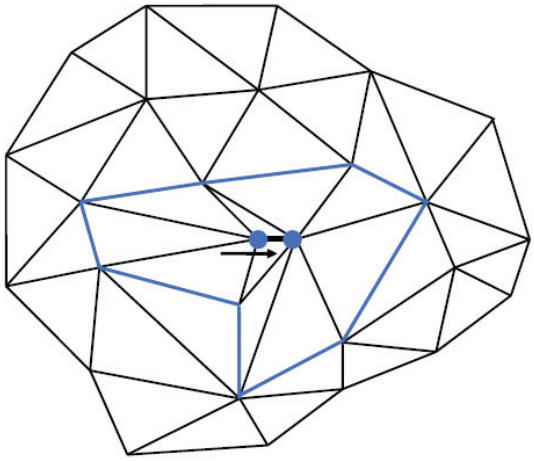


The half-edge collapse can also be applied gradually.

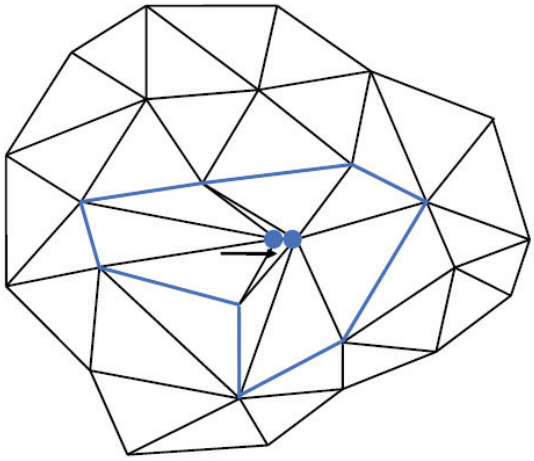
Half-Edge Collapse



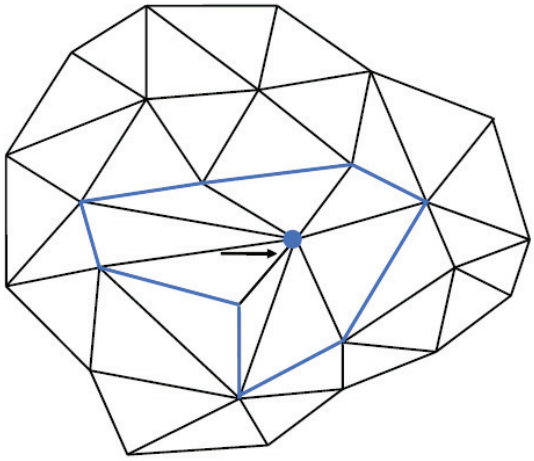
Half-Edge Collapse



Half-Edge Collapse

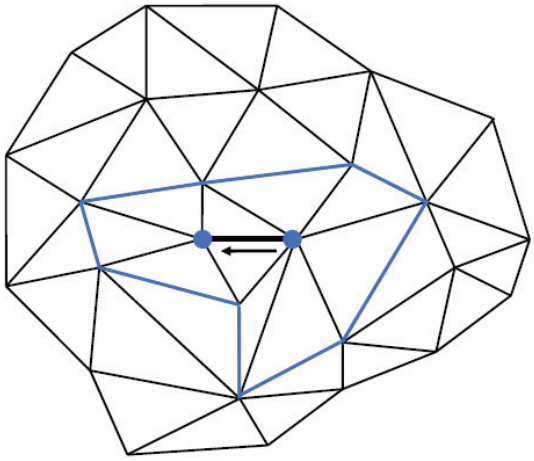


Half-Edge Collapse



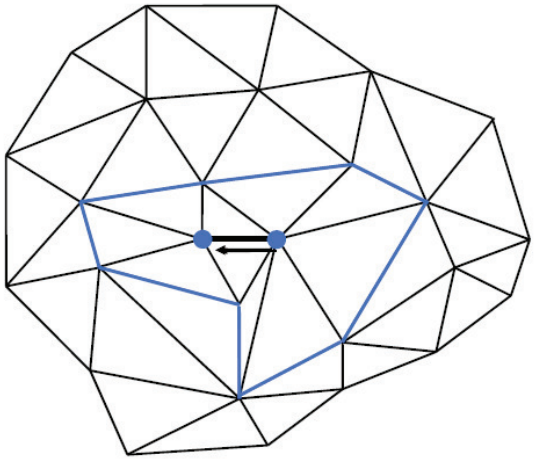
And in many cases it works as expected.

Half-Edge Collapse

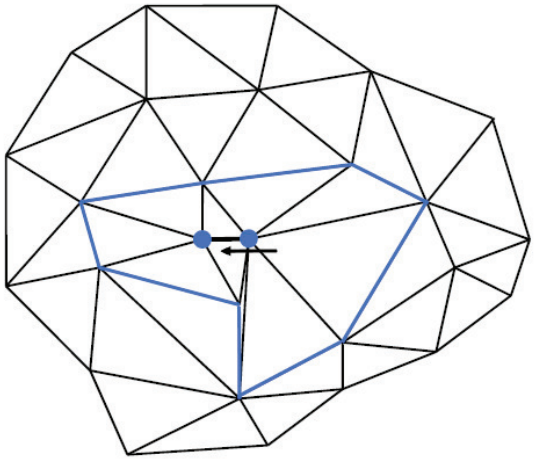


But in some cases it may cause problems, ...

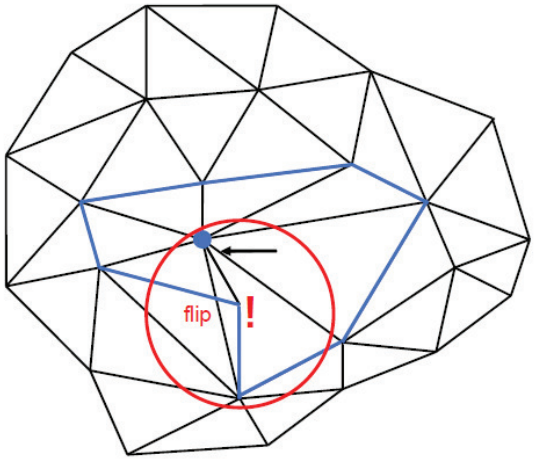
Half-Edge Collapse



Half-Edge Collapse



Half-Edge Collapse

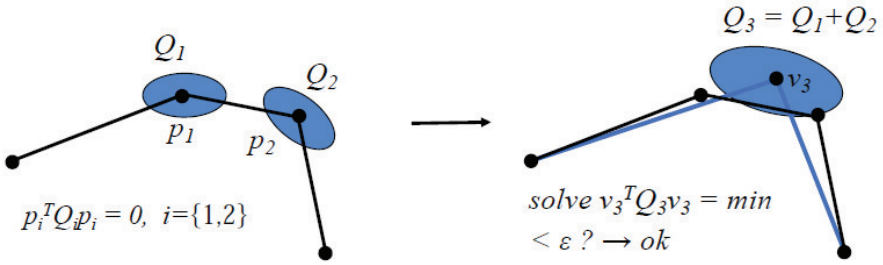


47

… as we may see in this case. One of the faces has changed the direction of its normal by 180 degrees and that part of the mesh has folded onto itself. We say that that face has flipped because of the half-edge collapse. That is something we need to detect and avoid. Detecting it is as simple as computing the dot product of the normals of the affected faces before and after the collapse.

Quadratic Error Metric for Edge Collapse

- Error quadrics
  - Squared distance to planes at vertex
  - No bound on true error



We said that the operations on the queue needed to be ordered by the error they introduced but, what error? Here we can use several approximations, but the most useful is the same we saw for vertex clustering, the error quadrics. One advantage of using this is that the quadric matrix of the vertex that results from an edge collapse can be computed as the summation of the quadric matrices of the vertices of the collapsed edge.

Project III: Surface Decimation Using Quadratic Error Metric

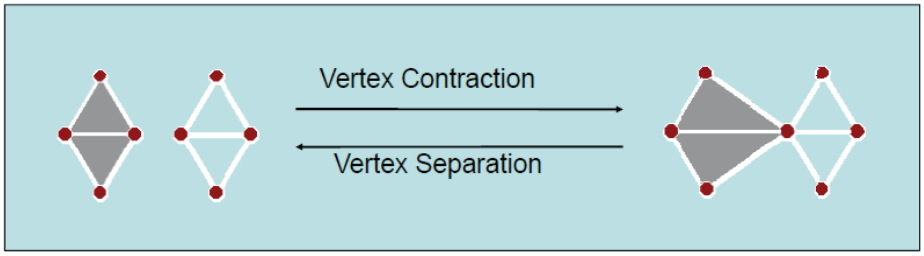
1. Compute the $\mathbf{Q}$ matrices for all the initial vertices.
2. Select all valid pairs.
3. Compute the optimal contraction target $\bar{\mathbf{v}}$ for each valid pair $(\mathbf{v}_1, \mathbf{v}_2)$. The error $\bar{\mathbf{v}}^T(\mathbf{Q}_1 + \mathbf{Q}_2)\bar{\mathbf{v}}$ of this target vertex becomes the *cost* of contracting that pair.
4. Place all the pairs in a heap keyed on cost with the minimum cost pair at the top.
5. Iteratively remove the pair $(\mathbf{v}_1, \mathbf{v}_2)$ of least cost from the heap, contract this pair, and update the costs of all valid pairs involving $\mathbf{v}_1$.

Ref: Michael Garland and Paul S. Heckbert, Surface simplification using quadric error metrics, SIGGRAPH97

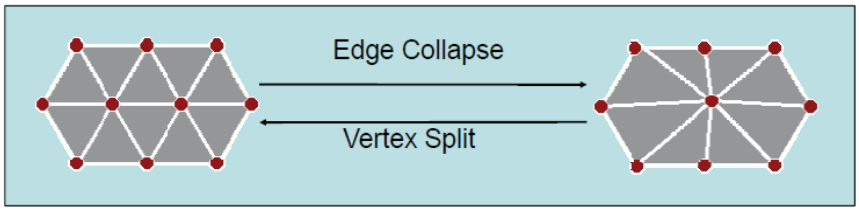
So, this is the final algorithm that Garland and Heckbert introduced for incremental decimation, and that is still in use today. It is known as Quadric Error Edge Collapse, because of the combination of the error metric and the decimation operator.

Topology Changes

- Merge vertices across non-edges
  - Change mesh topology



- What about edge collapse?

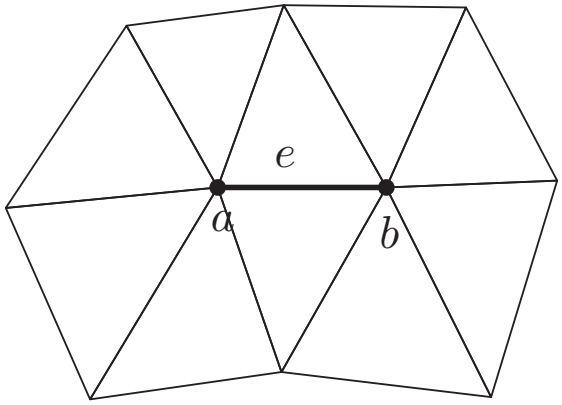


Now, vertex clustering accepts meshes, even if they are not 2-manifolds. It also does not respect the topology of the input mesh, and can produce results that are not 2-manifolds. But what about edge collapse?

Topology Preserving Edge Collapse

- A sufficient condition: assume M is a manifold without boundary, collapsing e does not change the topology if

$$Lk(a) \cap Lk(b) = Lk(e)$$



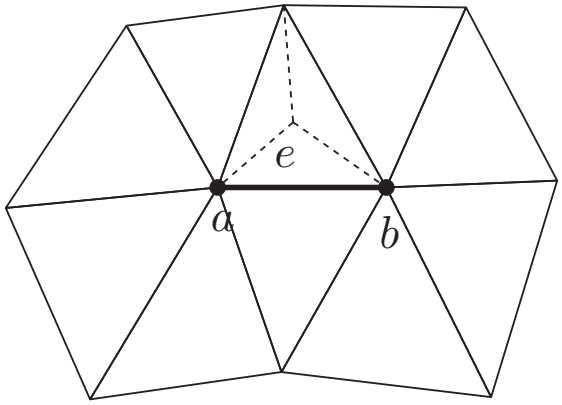
Ref: T. Dey, H Edelsbrunner, S. Guha and D. Nekhayes, Topology Preserving Edge Contraction, Publ. Inst. Math. (Beograd) (N.S)

Edge collapse does not modify the topology of the input mesh if we take a single problematic case into account. Let us say we want to check if an edge e can be collapsed. This edge has two vertices a and b. If the intersection of the vertices adjacent to a and those adjacent to b is the same as the two vertices adjacent to edge e, then and only then we may collapse e.

Topology Preserving Edge Collapse

- A sufficient condition: assume M is a manifold without boundary, collapsing e does not change the topology if

$$Lk(a) \cap Lk(b) = Lk(e)$$



Ref: T. Dey, H Edelsbrunner, S. Guha and D. Nekhayes, Topology Preserving Edge Contraction, Publ. Inst. Math. (Beograd) (N.S)

In the picture you can see one topology configuration where this is not the case. The intersection results in three vertices. If you try to collapse edge e, two of the faces disappear because of the operation, but two others may not. Remember that the vertices have 3D positions and may not be on the same plane, regardless of the picture being 2D.