3D Geometry Processing
Exercise 5

Fall 2013

Hand in: 21.11.2013, 16:00

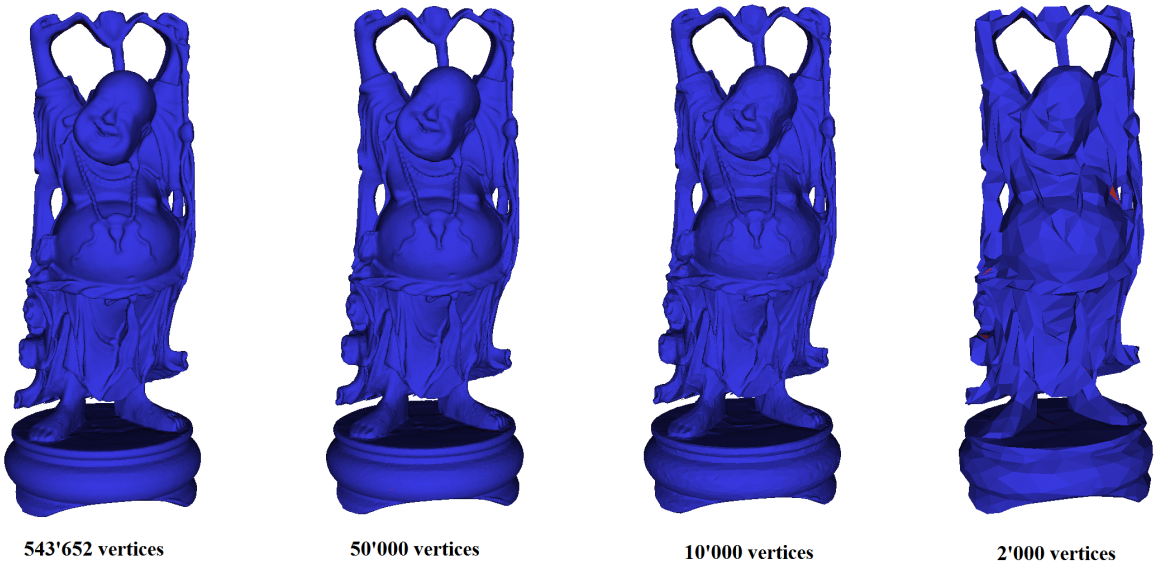

Figure 1: Demonstration of the quadric error decimation algorithm on the Buddha mesh. The mesh resolution can be greatly reduced without much visual impact. Note that fold-overs can occur, such that the 'inner' side of the triangles is visible (rendered in red).

In this assignment you will implement the quadric error decimation algorithm, also called QSlim, for half-edge structures. QSlim is based on greedily selecting and collapsing edges, based on a heuristic error estimation. It produces high quality simplifications while being fast, having a linear complexity in the number of edges and collapses.

1 Preparation: Half-Edge Collapses

The elementary editing operation used in the QSlim algorithm is the edge collapse. So your first task is to mplement half-edge collapses. In the class `HalfEdgeCollapse.java`, the following provided methods are provided, which take care of some gory details:

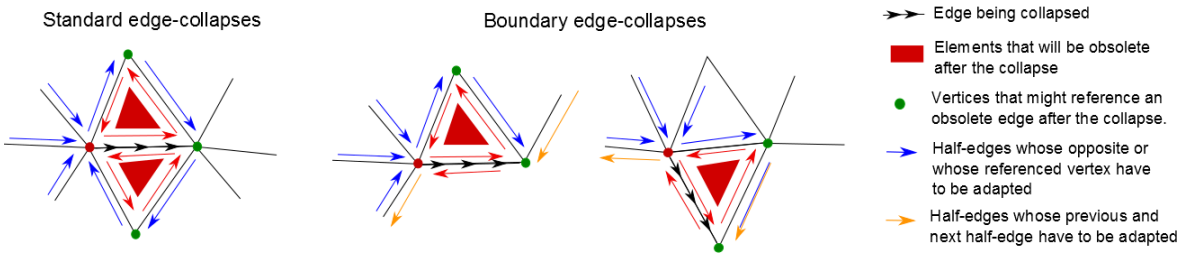

Figure 2: Depiction of the half-edge structure elements that need to be modified in an half-edge collapse. Finding safe edge references for the marked vertices is already implemented in `makeV2ERefSafe(HalfEdge e)`.

- `isHalfEdgeCollapsable(HalfEdge e)`: This method tests if it is possible to collapse the edge e without getting to an invalid half-edge structure and without changing the topology of the mesh.
- `makeV2ERefSafe(HalfEdge e)`: This method makes sure, that all the concerned vertices reference edges that will be valid after the collapse. In some cases, after calling this method, e.end() might reference an edge from e.start() and will only become a valid reference after the edge is completely collapsed. This should not matter, as you don't need to iterate around vertex e.end() while collapsing the edge. The rightmost picture in Figure 2 depicts the case where no valid reference for e.end() can be found at e.end().
- `isCollapseMeshInv(HalfEdge e, Point3f position)`: This method tests if collapsing the half-edge e into the provided position leads to a fold over. This method does not detec all mesh inversions, but many. Feel free to improve it and to experiment around with alternative implementations.

1. Implement the method `collapseEdge(HalfEdge e, Point3f newPos)`, which collapses the start vertex of the half-edge onto the end vertex of the half-edge while relinking half-edges and vertices, such that the half-edge structure remains valid. When the collapse is finished, the position newPos should be assigned to the remaining vertex. *Details*:
   - Do not remove the obsolete half-edges, faces and vertices from the array lists kept in the half-edge structure, but collect them in HashSets. The collected dead elements can then efficiently be removed all at once using the ArrayLists `removeAll(...)` method, as is done in the `finish()` method. This makes sure that m edge collapses in a half-edge structure with n edges have a complexity of $\mathcal{O}(m) + \mathcal{O}(n)$ (m collapses, and an iteration over all edges to remove the dead ones) and not $\mathcal{O}(n\ m)$ (iterating each of the m times over the n vertices, to remove the right one) .
   - The provided `assert*` methods are very useful to debug code and to test that the half-edge structure stays valid...
2. Test your code by randomly collapsing a large number of edges on any mesh, but make sure to also test this on a mesh which has some boundaries (e.g. the bunny5k mesh).

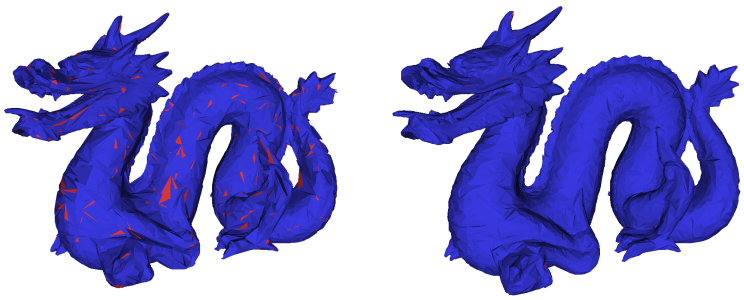

Figure 3: Random collapse without face flip tests and with face-flip tests on the dragon.obj mesh. Note that even with the test some faces get inverted.

You can also stress test your code on the very large buddha.obj or the dragon3.obj mesh collapsing hundred thousands of edges. Note that the dragon3.obj mesh is very ugly with many holes and degenerated faces. See Figure 3

3. Implement an algorithm that collapses all edges smaller than epsilon. Iterate multiple times over all half-edges, until no edges are collapsed anymore. Using very small epsilons, this effectively gets rid of degenerated edges. You can test your method on the buddha.obj mesh, using it with $\epsilon = 0.0001f$ already should reduce its size by about 10'000 vertices.

5 Points

2 QSlim

The quadric error decimation algorithm discussed in the lecture allows to reduce the number of vertices in a mesh dramatically, while staying close to the original mesh, as demonstrated in Figure 1. The approach is to greedily iteratively collapse the edge whose collapse has the smallest impact. To estimate the impact of a collapse, the following heuristic is used: an error matrix Q_v is associated to each vertex v, such that the quadratic form $v^T Q_v v$ describes the sum of squared distances to a set of planes. When two vertices v and w are merged, $Q_v + Q_w$ is used as a new error matrix; the sums of squared distances to plane sets are simply accumulated.

Construction of the Error Quadrics

The distance of a point $p = (x, y, z)$ to a plane with normal $n = (x_n, y_n, z_n)$ going through a point $p_0 = (x_0, y_0, z_0)$ can be computed by $\langle n, p \rangle - \langle n, p_0 \rangle$, which can be rewritten as:

$$(x_n, y_n, z_n, -\langle n, p_0 \rangle) \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

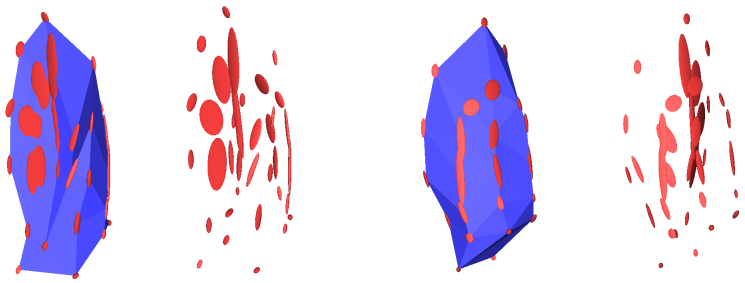

Figure 4: Visualization of the 0.04-isosurface of the error quadrics on the bunny_ear.obj mesh. Note how they spread out in planar regions and follow straight edges. If the isosurface does not extend in some direction, this means a collapse in that direction would be heavily penalized.

The squared distance of the point p to the described plane is given by

$$(x, y, z, 1)\mathbf{Q}_{n,p_0}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad \text{where } \mathbf{Q}_{n,p_0} = \begin{pmatrix} x_n \\ y_n \\ z_n \\ -\langle n, p_0\rangle \end{pmatrix}(x_n, y_n, z_n, -\langle n, p_0\rangle).$$

The quadric error matrix assigned to a vertex v is then built from the distance matrices of all incident planes, by summing over all incident faces f. In the following formula n_f is the face normal, p_f is some position on the face and Q_v is the sought error quadric:

$$Q_v = \sum_{f \in \mathcal{N}_1} Q_{n_f, p_f}.$$

When contracting an edge (v,w), the new error quadric is $\tilde{Q} = Q_v + Q_w$ and the cost of collapsing the edge (v,w) into the position $p = (x, y, z, 1)$ is defined to be $p^T \tilde{Q} p$.

1. Compute all per vertex error matrices and store them in a HashMap⟨Vertex, Transformation⟩ in the class QSlim.java.

   (a) To get a better intuition of the error matrices (and at the same time to validate your implementation), visualize the ϵ-isosets of the quadratic forms described by the matrices, i.e. render the sets $\{p : p^T Q_v p = \epsilon\}$ at every vertex, see Figure 4. Upon initialization, these ellipsoids can be determined by finding the eigenvalues and eigenvectors of the upper right 3x3 submatrix of Q (on initialization, all involved planes go through the same point,just considering the upper 3x3 matrix cooresponds to setting that point to zero). The found eigenvectors then are the main axes of the ellipsoid, and $\frac{\epsilon}{\sqrt{eigenvalue}}$ describe the axes lengths (figure out why (-:). You will find some useful methods in the class `Assignment5_vis.java`.

   (b) Be careful that the matrices do not have NAN or infinity values in them. These can for example occur if a face is degenerated, such that when computing and

normalizing its face normal, a division by zero occurs. For such faces, set the matrix Q_{n_f,p_f} to zero. Write a test, which tests all matrices on the buddha.obj or the dragon3.obj mesh for NANs and infinities.

2. Implement the quadric error decimation algorithm: add all potential collapses to a PriorityQueue (one for each half-edge) and prioritize the collapses by the cost of the collapse, using $(u + v)/2$ as target position for the collapse of the edge (u,v). Poll and collapse half-edges until a target number of vertices is reached.

   Before every collapse check for collapsability and mesh inversions. If a mesh inversion or an uncollapsable edge is detected, update the collapse costs to $(cost + 0.1) * 10$ and reinsert the collapse back into the queue. After a collapse, update the cost of the remaining vertex and the adjacent edges and mark collapses whose edge does not exist anymore as deleted (cf Details).

   Details:

   - Represent each potential collapse by a PotentialCollapse class containing a halfEdge, a target position and a cost. The PotentialCollapse class should implement Comparable, such that the Java PriorityQueue can be used to prioritize the potential collapses.
   - The Java PriorityQueue<> implements a heap but has some issues: it is very inefficient to delete elements in the PriorityQueue, and it does not support that the priority of elements change, once they have been inserted. To solve these these shortcomings, add a boolean field `isDeleted` to the PotentialCollapse class, and mark collapses as deleted instead of really removing them from the queue. In the same spirit, if the cost of a collapse changes, mark it as deleted and add a new PotentailCollapse with and updated cost to the queue. When polling, ignore pairs marked as deleted.
   - Finally, to mark a PotentialCollapse as deleted, you first need to efficiently find the PotentialCollapse corresponding to some edge. To do this, keep an additional hashmap, which stores for every half-edge its most current PotentialCollapse. The map can then be used as look-up table.

3. Use optimized positions: If q_{ij} are the entries of $\tilde{Q} = Q_v + Q_w$, and if the following linear equation is invertible, solving

$$\begin{pmatrix} q_{00} & q_{01} & q_{02} & q_{03} \\ q_{10} & q_{11} & q_{12} & q_{13} \\ q_{20} & q_{21} & q_{22} & q_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} u = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

   will result in the optimal contraction position u with the smallest error. This equation is obtained by listing the constraints $\frac{\partial}{\partial x} v^T Q v = 0$, $\frac{\partial}{\partial y} v^T Q v = 0$, $\frac{\partial}{\partial z} v^T Q v = 0$ and $w = 1$. If the linear equation is not invertible, keep using the average position.

5 Points

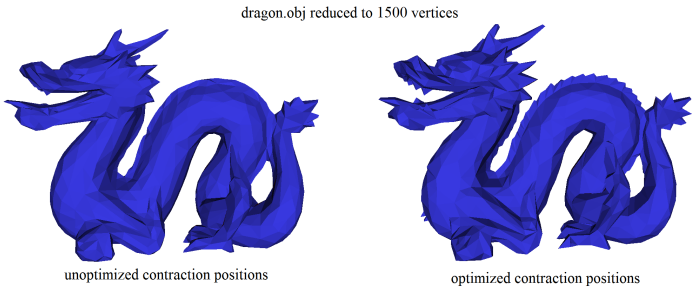

Figure 5: Comparision of using optimized contraction positions and the positions $(u + v)/2$ during quadric edge decimation. The algorithm was run on the dragon.obj mesh with a target of 1500 vertices.

3 Hand-In

Don't forget to:

1. Commit your code via the Ilias exercise page before the deadline.
2. Prepare your code demonstration and reserve a time slot for your demo via the Google Doc shared in the forum.