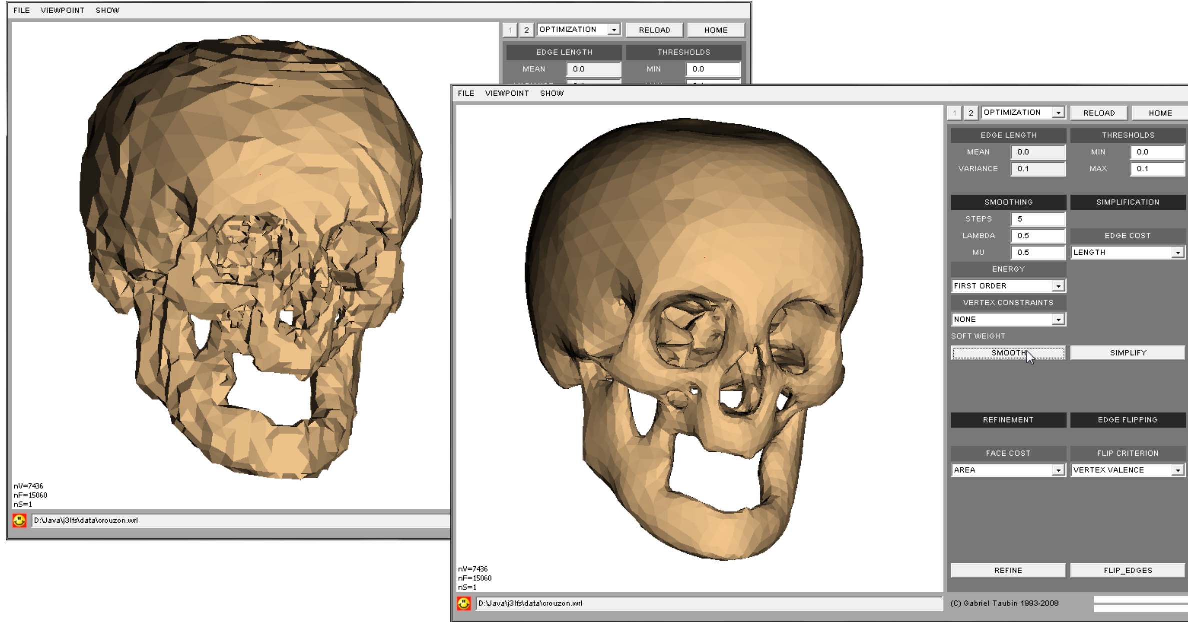
1 J3DPanelOptimization.java



As in the previous two assignments, you need to implement your code in this class. I am giving you a template with the user interface shown above. Your implementation may require more parameters and/or different ways of interacting with those that I defined. Feel free to modify the user interface and class variables to satisfy your needs. If you find bugs or missing functionality that you cannot fix yourself, please let me know.

The goal of this assignment is to implement four operations: Smoothing with support for soft and hard constraints, Edge collapse simplification, Adaptive red-blue triangle subdivision, connectivity optimization by edge flipping. If you structure your code well, you will be able to reuse large portions of the code you wrote in previous assignments.

2 Smoothing

We want to implement Laplacian smoothing and a number of related mesh smothing algorithms. We also want to be able to impose constraints on some vertex positions. To do so in a consistent manner, with well structured and modular algorithms, we first show that the Laplacian smoothing step can be explained as the Jacobi iteration for the following quadratic energy function

$$\mathcal{E}_3(x^0, x^1, x^2) = \sum_{e=(i,j)} \phi_{ij} \, \|x_i - x_j\|^2 \tag{1}$$

where $\phi_{ij} = \phi_{ji}$ are non-negative scalars, $x_i = (x_i^0, x_i^1, x_i^2)^t$ is the location of the i-th vector in 3D, and the sum is over all the edges of a mesh with V vertices, E edges, and F faces. However, since the sum of square norms can be split into three sums, one for each coordinate

$$\mathcal{E}_3(x^0, x^1, x^2) = \mathcal{E}(x^0) + \mathcal{E}(x^1) + \mathcal{E}(x^2) = \sum_{e=(i,j)} \phi_{ij} \, (x_i^0 - x_j^0)^2 + \sum_{e=(i,j)} \phi_{ij} \, (x_i^1 - x_j^1)^2 + \sum_{e=(i,j)} \phi_{ij} \, (x_i^2 - x_j^2)^2$$

and these three sums here are function of independent variables (first, second, and third coordinates), it is sufficient to consider the one dimensional energy

$$\mathcal{E}(x) = \sum_{e=(i,j)} \phi_{ij} \, (x_i - x_j)^2 \tag{2}$$

minimizing the original energy is equivalent to minimizing each one of the three independently. That is, we look at $x = (x_1, \ldots, x_V)^t$ as a one dimensional signal defined on the mesh vertices. All the energy functions which we will consider in this assignment can be split into three independent energy functions of one dimensional vertex signals, but this is not true in general. In some cases the energy containing the three coordinates of the vertices have to be used to derive new algorithms without splitting.

2.1 Matrix Formulation

To analyze the problem and to derive new algorithms it is convenient to derive equivalent matrix formulations. We will actually use both, because our implementations will be based on the formulas with explicit coordinates.

The energy $\mathcal{E}(x)$ of equation 2 can be written in matrix form as follows

$$\mathcal{E}(x) = (Lx)^t \phi(Lx) = x^t [L^t \phi L] x = x^t A x \tag{3}$$

where $A = [L^t \phi L]$ is a symmetric non-negative definite$V \times V$ matrix (positive definite in general), L is a $E \times V$ matrix, with one row per edge, and one column per vertex; and ϕ is a $E \times E$ diagonal matrix. For each edge $e = (i, j)$, the e-th row of L is defined as

$$L_{eh} = \begin{cases} 1 & \text{if } h = i \\ -1 & \text{if } h = j \\ 0 & \text{otherwise} \end{cases}$$

so that $(Lx)_e = (x_i - x_j)$. Note that for each edge $e = (i, j) = (j, i)$ we need to choose one of the two vertices as the one with the 1 in the other with the -1. For example, if $i < j$ then assign 1 to i and -1 to j. The energy function is independent of these choices. The e-th diagonal element of the matrix ϕ is $\phi_{ij} = \phi_{ji}$.

2.2 Derivatives with respect to vectors and matrices

If $f(x)$ is a scalar function of a vector variable $x = (x_1, \ldots, x_V)^t$, we denote the vector of first order derivatives with respect to the V variables $\frac{\partial f}{\partial x}$. That is

$$\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_V} \end{pmatrix}$$

Although we don't need derivatives with respect to matrix variables, let me mention that if $f(M)$ is a scalar function of a $R \times C$ matrix variable

$$M = \begin{pmatrix} m_{11} & \cdots & m_{1C} \\ \vdots & \ddots & \vdots \\ m_{R1} & \cdots & m_{RC} \end{pmatrix}$$

the $R \times C$ matrix of first order derivatives with respect to the V variables is denoted

$$\frac{\partial f}{\partial M} = \begin{pmatrix} \frac{\partial f}{\partial m_{11}} & \cdots & \frac{\partial f}{\partial m_{1C}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial m_{R1}} & \cdots & \frac{\partial f}{\partial m_{RC}} \end{pmatrix}$$

If the functions $f(x)$ or $f(M)$ are vector or matrix functions, then $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial M}$ can be defined in a consistent manner as block matrices. For example, if $f(x) = x$, what is $\frac{\partial f}{\partial x}$? And what if $f(x) = Ax$, where A is any matrix with V columns? More generally, show that the derivative of an inner product of two vector valued functions $f(x)$ and $g(x)$

$$\frac{\partial (f^t g)}{\partial x} = \left(\frac{\partial f}{\partial x}\right)^t g + f^t \left(\frac{\partial g}{\partial x}\right) \ .$$

2.3 The Jacobi Iteration

We consider here a quadratic energy function

$$\mathcal{E}(x) = x^t A x + 2b^t x + c \tag{4}$$

where $x = (x_1, \ldots, x_V)^t$ is our V-dimensional vector of variables, A is a symmetric positive definite $V \times V$ matrix, b is a V-dimensional vector, and c is a scalar. Minimizing this energy is the well know Least-Squares (LS) problem. When the matrix A is positive definite this problem has a global minimum which can be computed in various ways. On one hand, a necessary condition for an extremum is that all the first derivatives vanish

$$\frac{\partial \mathcal{E}}{\partial x}(\hat{x}) = 0 \ ,$$

where $\hat{x}$ is the global minimizer. Using the relation derived above for the derivatives of an inner product of matrices, and the fact that computing derivatives is a linear operation, we have

$$\frac{\partial \mathcal{E}}{\partial x}(x) = 2(Ax + b) \ .$$

It follows that the minimizer is $\hat{x} = -A^{-1}b$, however computing it when the number V of variables is large is a fundamental problem in numerical computations. If the matrix A is only non-negative definite, the minimizer is not unique. If x is a minimizer, and $Ay = 0$, then $y + x$ is another minimizer. We care about this because this is the case with Laplacian smoothing. Most algorithms to solve large and sparse linear systems are iterative. In the case of LS problems, these descent algorithms are normally descent algorithms, where if x^N is the estimate of the minimizer at time N, the estimate at at time $N + 1$ is computed as the result of applying a displacement $x^{N+1} = x^N + \delta x^N$ to the time N estimate in such a way that the energy decreases $\mathcal{E}(x^{N+1}) \leq \mathcal{E}(x^N)$, and it can be proven that the sequence of estimates converges to the minimizer. Jacobi is one of the simplest descent algorithms of this kind. To compute the i-th coordinate δx_i of the displacement vector δx, as a function of a current estimate x, we solve the linear equation

$$\frac{\partial \mathcal{E}}{\partial x_i}(x) = (x_1, \ldots, x_i + \delta x_i, \ldots, x_V) = 0$$

which we regard as a linear equation in the single variable δx_i. If we write all these equations together in matrix form, we get

$$Ax + D\delta x + b = 0$$

where D is a $V \times V$ diagonal matrix formed with the diagonal elements of the matrix A. Finally, the displacement vector can be written in matrix form as

$$\delta x = -D^{-1}(Ax + b)$$

and in coordinates as

$$\delta x_i = -\frac{1}{a_{ii}} \left(b_j + \sum_{j=1}^{V} (a_{ij} x_j) \right)$$

Note that in order for these eqautions to be well defined we need $a_{ii} \neq 0$ for all i. Otherwise we would be dividing by zero. In fact, for the method to converge we actually need more. We need the matrix to be diagonally dominant. But we will not go into these details here. This condition is satisfied for Laplacian smoothing. In the case of the energy of equation 3, where $A = J^t \phi J$ and $b = 0$, show that

$$a_{ij} = \begin{cases} \sum_{h \in i^\star} \phi_{ih} & \text{if } j = i \\ -\phi_{ij} & \text{for each edge } e = (i,j) \\ 0 & \text{otherwise} \end{cases}$$

where $i^\star$ is the set of vertices j conncted to i by an edge $e = (i, j)$. Finally, the Jacobi displacement turns out to be the Laplacian smoothing step

$$\Delta x_i = \sum_{j \in i^\star} w_{ij}(x_j - x_i)$$

where $w_{ij} = \phi_{ij}/\phi_i$ and $\phi_i = \sum_{h \in i^\star} \phi_{ih}$, as expected. Note that this analysis provides another explanation for the shrinking problem of Laplacian smoothing: for a positive definite matrix A, $\hat{x} = 0$ if $b = 0$. If A is only non-negative definite, any x so that $Ax = 0$ is a minimizer. In the case of Laplacian smoothin, where the diagonal matrix ϕ is non-singular, we have $[J^t \phi J]x = 0$ if and only if $Jx = 0$. In general this only happens for vectors x with all equal coordinates.

2.4 Implementing FIR Filters With Jacobi Iterations

As in the case of Laplacian smoothing, for any energy function we can implement the $\lambda - \mu$ algorithm, or any other FIR filter defined by a polynomial transfer function, using the displacement vectors computed in the Jacobi iteration:

$$x^{N+1} = x^N + \lambda_N \delta x^N$$

where $\lambda_1, \lambda_2, \ldots$ is a properly sequence of displacement factors. If we set $\lambda_k = \lambda$ with $0 < \lambda < 1$ we have the classical Laplacian smoothing algorithm. If we take $\lambda_{2k} = \lambda$ and $\lambda_{2k+1} = \mu$ with $0 < \lambda < -\mu$, we have the $\lambda - \mu$ algorithm.

2.5 Imposing Hard Constraints

Imposing hard constraints on a subset of the vertices is very easy. If $I \subseteq \{1, \ldots, V\}$ is the subset of vertices to be constrained, and $\bar{x}_i$ is the target location of each contrained vertex, we first make the constrained vertex values equal to their target values. Then we iterate until the termination criterion is satisfied: 1) compute the displacement vectors as in the unconstrained case for the unconstrained vertices; 2) displace the unconstrained vertices in the direction of the displacement vectors.

2.6 Imposing Soft Constraints

Another way to impose constraints is by adding an additional term to the energy function which penalizes the deviation of constrained vertex values from their target values, such as

$$\mathcal{E}(x) = \sum_{e=(i,j)} \phi_{ij}\,(x_i - x_j)^2 + \sum_{i \in I} \mu_i (\bar{x}_i - x_i)^2 \tag{5}$$

where $\mu_i > 0$ for $i \in I$. If we also define $\mu_i = 0$ for $i \notin I$, and μ as the diagonal matrix with μ_i in the i-th diagonal position, we can rewrite this energy function in matrix form as

$$\mathcal{E}(x) = x^t [L^t \phi L] x + (x - \bar{x})^t \mu (x - \bar{x}) = x^t A x + 2b^t x + c \tag{6}$$

where $A = L^t \phi L + \mu$, $b = -\mu\bar{x}$, and $c = \bar{x}^t \mu \bar{x}$. The Jacobi displacement is defined as before

$$\delta x = -D^{-1}(Ax + b) = -D^{-1}([L^t \phi L]x + \mu(x - \bar{x})$$

Show that in this case we have

$$a_{ij} = \begin{cases} \mu_i + \sum_{h \in i^\star} \phi_{ih} & \text{if } j = i \\ -\phi_{ij} & \text{for each edge } e = (i, j) \\ 0 & \text{otherwise} \end{cases}$$

and the Jacobi displacement can be written in coordinates as follows

$$\delta x_i = \frac{\mu_i(\bar{x}_i - x_i) + \sum_{j\in i^\star} \phi_{ij}(x_j - x_i)}{\mu_i + \phi_i}$$

or as an affine combination of the displacement toward the target position and the Laplacian vector

$$\delta x_i = (1 - \epsilon_i)(\bar{x}_i - x_i) + \epsilon_i \Delta x_i$$

where

$$\epsilon_i = \frac{\phi_i}{\mu_i + \phi_i} \; .$$

In conclusion, this can be implemented as a minor modification of the unconstrained Laplacian smoothing algorithm. Also, both hard and soft constraints can be applied at the same time. For this we need a partition of the set of vertices into three subsets: hard constraints, soft constraints, and unconstrained. Target values must be provided for the hard and soft constraints.

2.7 Other Energy Functions

The main problem with Laplacian smoothing with hard or soft constraints is that mesh smoothness in the neighborhood of constrained vertices cannot be controled. One alternative quadratic energy function which can be used to address this problem is

$$\mathcal{E}(x) = \sum_{i=1}^{V} \nu_i (\Delta x_i)^2 \tag{7}$$

where $\nu_i > 0$ for $i = 1, \ldots, V$. Again, it is sufficient in this case to consider the one dimensional vertex signal case. However, using this energy function may create artifacts because there is no control on the edge lengths. A better approach is to introduce an additional term in the energy function

$$\mathcal{E}(x) = \sum_{e=(i,j)} \phi_{ij}\,(x_i - x_j)^2 + \sum_{i=1}^{V} \nu_i (\Delta x_i)^2 + \sum_{i\in I_S} \mu_i (\bar{x}_i - x_i)^2 \tag{8}$$

and additional hard constraints: $x_i = \bar{x}_i$ for $i \in I_H$.

To derive an expresion for the Jacobi displacement, we can write this energy function as the sum of three quadratic energy functions

$$\mathcal{E}(x) = (x^t A_1 x + 2b_1^t x + c_1) + (x^t A_2 x + 2b_2^t x + c_2) + (x^t A_3 x + 2b_3^t x + c_3)$$

The Jacobi displacement vector is still $\delta x = -D^{-1}(Ax + b)$, where $A = A_1 + A_2 + A_3$, $b = b_1 + b_2 + b_3$, D_j is the diagonal matrix formed with the diagonal elements of A_j, and $D = D_1 + D_2 + D_3$. This displacement vector can be written as an affine combination of the three Jacobi vectors $\delta_j x = -D_j^{-1}(A_j x + b_j)$ as follows

$$\delta x = [D^- 1 D_1]\delta_1 x + [D^- 1 D_2]\delta_2 x + [D^- 1 D_3]\delta_3 x \; .$$

Note that $[D^- 1 D_j]$ is a positive diagonal matrix and $[D^- 1 D_1] + [D^- 1 D_2] + [D^- 1 D_3]$ is the identity matrix, meaning that the sum of diagonal elements is equal to one. If a_{jii} is the i-th diagonal element of the matrix A_j and D_j, and $\beta_{ji} = a_{jii}/(a_{1ii} + a_{2ii} + a_{3ii})$, we have $\beta_{1i} + \beta_{2i} + \beta_{3i} = 1$ and we can rewrite this equation in coordinates as an affine combination

$$\delta x_i = \beta_{1i}\, \delta_1 x_i + \beta_{2i}\, \delta_2 x_i + \beta_{3i}\, \delta_3 x_i$$

The only thing that remains to be done to implement the smoothing algorithm based on the complete energy function of equation 8, is to find an expression in coordinates for the Jacobi displacement corresponding to the quadratic

energy function of equation 7. This time we will do the analysis in coordinates directly. We start by computing first order derivatives of the energy function of equation 7

$$\frac{1}{2}\frac{\partial}{\partial x_i}\left(\sum_{h=1}^{V}\nu_h(\Delta x_h)^2\right) = \sum_{h=1}^{V}\nu_h\,\Delta x_h\,\frac{\partial \Delta x_h}{\partial x_i} = \left(\sum_{h:i\in h^\star} w_{hi}\,\nu_h\Delta x_h\right) - \nu_i\Delta x_i$$

because,

$$\Delta x_h = \sum_{j\in h^\star} w_{hj}(x_j - x_h)\;, \qquad \sum_{j\in h^\star} w_{hj} = 1 \qquad \text{and} \qquad \frac{\partial \Delta x_h}{\partial x_i} = \begin{cases} -1 & \text{if } h = i \\ w_{hi} & \text{if } i \in h^\star \\ 0 & \text{otherwise} \end{cases}$$

(remember that $w_{hi} \neq w_{ih}$ in general, and $h \in i^\star$ is not necessarily equivalent to $i \in h^\star$). Again, the analysis is easire in matrix form. Writing $\Delta x = (W - I)x$, where W is the matrix of weigths w_{ij} with $w_{ij} = 0$ if vertex j is not a neighbor of vertex i, and I the identity matrix, we have

$$\sum_i \nu_i\,(\Delta x_i)^2 = x^t(W-I)^t\nu(W-I)x\;,$$

the matrix A turns out to be $A = (W-I)^t\nu(W-I)$ and the i-th diagonal element of this matrix is

$$a_{ii} = \nu_i + \sum_{h:i\in h^\star}\nu_h\,w_{hi}^2$$

2.8 Algorithmic Details

We have all the equations we need to implement a smoothing algorithm based on minimizing the energy function of equation 8, with hard and/or soft constraints. What remains to figure out is how to implement this efficiently. The idea is as in the basic Laplacian smoothing algorithm to accumulate various quantities while traversing the edges. The overall Jacobi displacement vector is an affine compbination of the old Laplacian vector, the displacement of soft constrained vectors toward their targets, and the Jacobi vector of the third term analyzed above. Since the the last Jacobi vector can be written as a function of the Laplacian vectors, we will have to do two passes through the edges to accumulate all the quantities needed. Finally the affine combination vector can be evaluated and normalized. The details are up to you.

3 Simplification

We want to implement edge collapse simplification in such a way that various different edge costs could be used. Here we have a high level pseudocode description of the algorithm

```
// 0) initialize empty heap
// 1) for each edge
//      if edge if collapsible
//        compute edge score
//        insert in heap
// 2) initialize empty independent set of edges
// 3) while heap is not empty
//      delete edge with min score
//      if score larger than max score threshold
//         break
//      if deleted edge is independent of all edges in the independent set
```

```
//          include edge in independent set
// 4) if independent set of edges is not empty
//      initialize vMap as identity map
//      assign vertex index to pair of collapsed vertices
//      determine coordinates of collapsed vertices (save on a separate buffer)
//        determine coordinates of collapse per vertex properties as well
//      create newCoordIndex from coordIndex and vMap
//        delete per face and per corner properties of deleted triangles as well
// 5) rebuild edges, faces, and selection buffer
//
// 6) We need an option to select the vertices not affected by this operation,
//    so that constrained smoothing could be applied to optimize the position of
//    the new vertices
```

This code fragment shows how to use the new `Heap` class, which is part of the `mesh` package

```
GraphFaces g        = ifs.getEdges();

Heap       h        = new Heap();
float      edgeCost = 0.0f;
boolean    insertInHeap = false;
// we need to save the ends of edges inserted in heap
VecInt     hV = new VecInt();

GraphEdge  e        = null;
for(iV=0;iV<nV;iV++) {
  for(e=g.getFirstEdge(iV);e!=null;e=g.getNextEdge(e)) {
    iE  = e.getIndex();
    iV0 = e.getVertex(0);
    iV1 = e.getVertex(1);

    // determine if edge should be inserted in heap
    // and compute edgeCost here

    if(insertInHeap) {
      hV.pushBack(iV0);
      hV.pushBack(iV1);
      h.add(edgeCost);
    }
  }
}

// select edges in increasing cost order
while((iH=h.delMin())>=0) {
  // Heap.delMin() returns the 'time' of insertion in the heap

  edgeCost = h.getLastKey();
  iV0      = hV.get(2*iH  );
  iV1      = hV.get(2*iH+1);
```

```
    e        = g.getEdge(iV0,iV1);
    iE       = e.getIndex();

    // ...

  }
```

4 Subdivision

We want to implement red-blue adaptive triangle subdivision in such a way that different face refinement criteria could be used. Here we have a high level pseudocode description of the algorithm

```
// 0) initialize empty set of marked vertices
// 1) for each triangle
//      if triangle must be subdivided
//         mark the three vertices of the triangle
// 2) for each edge
//      if the two ends are marked
//        create mid-edge vertex (as in the iso-curve algorithm)
//        create associated per vertex properties
// 3) for each triangle
//      if three vertices are marked
//        split into four triangles
//        create associated per face and per corner properties
//      else if two vertices are marked
//        split into two faces
//        create associated per face and per corner properties
//      else
//        preserve the triangle as it is
//        preserve associated per face and per corner properties
// 4) make new edges, faces, and selection
// 5) we need an option to select all the original vertices
//    so that constrained or unconstrained smoothing could be applied
```

5 Optimization

We want to implement edge flipping triangle optimization in such a way that different edge flipping criteria could be used. Note that since neither the number of vertices nor the number of triangles change, we can atempt to implement the connectivity changes in place, i.e. by modifying the `coordIndex` array directly, rather than using a separate output buffer. Here we have a high level pseudocode description of the algorithm

```
// 0) initialize empty heap
// 1) for each edge
//      if edge flip is allowed
//        compute edge score
//        insert in heap
// 2) initialize empty independent set [of edges]
// 3) while heap is not empty
//      delete edge with min score
```

```
//      if score larger than max score threshold
//         break
//      if deleted edge is independent of all edges in the independent set
//         include edge in independent set
// 4) for each edge in the independent set
//      flip edge [can this be done in place?]
//      update per face and per corner properties
// 5) rebuild edges, faces, and selection buffer
//
// 6) Support an option to select the vertices not affected by this operation,
//    so that constrained or unconstrained smoothing could be applied to optimize
//    the position of afected vertices
```