Chanhaeng Lee, 40228578

Project Report

Description and Result

Data Structure For Mesh Optimization.

This project uses the halfedge-based data structure for the mesh optimization algorithm, because the data structure offers numerous advantages, including efficient traversal of the neighborhood of a vertex, face, or edge through an oriented search. Figure 1 illustrates the data structure used in the project.

When a triangle is given, the `struct Vertex` is created for each vertex of the triangle, and then `struct Face` is created with `struct HalfEdge`s. The halfedges are oriented so that a face is traversed in the counterclockwise/clockwise order using the next/prev pointer. Each halfedge stores a pointer of a vertex it points to, while a vertex saves pointers to the halfedges that can be traversed from the vertex. The `HalfEdge* sym` of `struct HalfEdge` points to the opposite halfedge that belongs to another face. The `struct Edge` represents an undirected edge that is used for edge operations. An edge stores a pointer to a halfedge. In case there are two halfedges for one edge, the halfedge with a higher ID is saved for the edge data. The last created halfedge is selected as the halfedge pointer for a face.

Data Preparation For Mesh Optimization.

In addition to the halfedge-based data structure, three additional processes are required to implement the mesh optimization algorithm for a given mesh: 1) sampling random points from the mesh surface, 2) normalizing vertex positions of the mesh, 3) finding the closest face to each sampled point.

For the first process, a face is randomly selected, and two barycentric coordinates are generated in the uniform distribution $\mathcal{U}_{[0,1]}$. Because the last barycentric coordinate is determined automatically, a vertex in the face can be sampled randomly. In this project, 10,000 points are sampled for a mesh surface.

Normalizing the vertex positions of the mesh is transforming the vertices to fit in the unit cube, where the center of the cube is located at $(0,0,0)$ and the minimum/maximum coordinates of the cube is $(-1,-1,-1)/(1,1,1)$. The randomly sampled points are also transformed using the same transform matrix. The mesh optimization algorithm is then applied to the vertices in this normalized space.

A k-d tree data structure is used to efficiently find the closest face to each sampled point. The splitting plane for the k-d tree is determined by selecting the longest axis of a node's Axis-Aligned Bounding Box (AABB) and finding the median of the coordinate of the points along the axis.

Energy Function of Mesh Optimization

The energy function of mesh optimization is defined as follows:

$$\mathrm{E}(\mathrm{K},\mathrm{V}) = \mathrm{E}_{\mathrm{dist}}(\mathrm{K},\mathrm{V}) + \mathrm{E}_{\mathrm{rep}}(\mathrm{K}) + \mathrm{E}_{\mathrm{spring}}(\mathrm{K},\mathrm{V}),$$

where K is a simplicial complex and V is a set of vertex positions defining a mesh $M = (K, V)$. In order to minimize the energy function, the paper divides the algorithm into two stages. The first stage is solving an inner minimization problem by optimizing V over fixed simplicial complex K. The second stage is solving an outer minimization over K. Figure 2 illustrates the idealized pseudo-code version of the algorithm as presented in the paper.

First Stage of Mesh Optimization

The first stage of mesh optimization is the procedure `OptimizeVertexPositions` for fixed simplicial complex. The distance energy $E_{dist}(K, V) = \sum_{i=1}^{n} d^2(x_i, \phi_V(|K|))$ and the spring energy $E_{spring}(K, V) = \sum_{\{j,k\}\in K} \kappa \left\| v_j - v_k \right\|^2$ will be minimized in this stage, where x_i is a randomly sampled point, $|K|$ is the topological realization for a simplicial complex, $\phi_V(|K|)$ is the mapping that is fully specified by the vertices V, and d^2 is the squared distance between the sampled point x_i and $M = \phi_V(|K|)$, κ is the spring constant, and v is the vertices of the mesh. The squared distance d^2 turns into the solution to the minimization problem:

$$d^2(x_i, \phi_V(|K|)) = \min_{b_i \in |K|} \|\mathrm{x_i} - \phi_{\mathrm{V}}(\mathrm{b_i})\|^2,$$

where $b_i \in |K|$ is the barycentric coordinate vector corresponding to the point $p \in \phi_V(|K|)$ closest to x_i.

Minimizing those equations is a linear least squares problem. However, solving it over all of vertices is not efficient because one edge operation such as edge collapse, edge split, and edge swap affect only the local area around one edge. Therefore, if a vertex V to be optimized is given with its adjacent vertices P and barycentric coordinates (u, v, w) evaluated by projecting a close sampled point x to the triangle VP_iP_j, the distance equation to be minimized is

$$\min_V \left\| (uV + vP_i + wP_j) - x \right\|^2.$$

The equation inside the norm operation can be rewritten into this to be solved in the linear least-squares form $Ax = b \rightarrow Ax - b = 0$:

$$(uV + vP_i + wP_j) - x = \begin{bmatrix} \mathrm{u} & 0 & 0 \\ 0 & \mathrm{u} & 0 \\ 0 & 0 & \mathrm{u} \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} - (x - vP_i - wP_j).$$

The same logic applies to the spring energy in the local context. The linear least-squares form of it is

$$\kappa\|V - P\|^2 = \left\| \sqrt{\kappa}(V - P) \right\|^2,$$

$$\sqrt{\kappa}(V - P) = \begin{bmatrix} \sqrt{\kappa} & 0 & 0 \\ 0 & \sqrt{\kappa} & 0 \\ 0 & 0 & \sqrt{\kappa} \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix} - \sqrt{\kappa} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}.$$

Because we know its linear least-squares form $Ax - b$, the x is evaluated by $(A^T A)^{-1} A^T b$, where x is actually the vertex V to be optimized in the linear least-squares form.

Legal Edge Collapse and Legal Edge Swap

As an edge collapse operation can cause a change of topological type, testing its legality to prevent the topological change is needed. According to the definitions in the paper, it defines an edge $\{i,j\} \in K$ to be a boundary edge if it is a subset of only one face $\{i,j,k\} \in K$, and a vertex $\{i\}$ to be a boundary vertex if it is from a boundary edge $\{i,j\} \in K$. Three conditions should be satisfied for the edge collapse to be legal:

1. For all vertices $\{k\}$ adjacent to both $\{i\}$ and $\{j\}$, $\{i,j,k\}$ is a face of K.
2. If $\{i\}$ and $\{j\}$ are both boundary vertices, $\{i,j\}$ is a boundary edge.
3. K has more than 4 vertices if neither $\{i\}$ nor $\{j\}$ are boundary vertices, or K has more than 3 vertices if either $\{i\}$ or $\{j\}$ are boundary vertices.

The condition for an edge swap operation that transforms the edge $\{i,j\} \in K$ into $\{k,l\} \in K$ is $\{k,l\} \notin K$.

The last edge operation, edge split, is always legal as it does not change the topological type of K. The proof for the legality of an edge swap is not discussed here since the edge swap condition is self-explanatory. The proof for the legality of an edge collapse will be discussed in the next.

Illustration for the Legality of Edge Collapse

The legality of an edge collapse can be illustrated based on three conditions. First, if we have only 3 vertices in K (a 2-simplex, triangle), the edge collapse on the complex causes the change of the topological type of K. Therefore, any complex K must have at least 4 vertices for an edge collapse. Starting with 4 vertices of the complex K, the first condition is automatically satisfied and the third condition is also satisfied with all 4 boundary vertices. There are two cases that collapse an edge in this case: one is collapsing the edge shared with two faces (Figure 3), and the other is collapsing a non-shared edge (Figure 4). The red box and the blue box represent the edge to be collapsed in the figures. The first case does not satisfy the condition 2, because $\{i,j\}$ is not a boundary edge with the boundary vertices $\{i\}$ and $\{j\}$. On the other hand, the second case satisfies the second condition. If we do the edge collapse on the first case, the result will be a 1-simplex (line segment), which results in a change of the topological type. Figure 5 shows the result of edge collapse on the second case. The legality of an edge collapse with at least 5 vertices is illustrated with homeomorphism in the paper.

Second Stage of Mesh Optimization

The second stage of mesh optimization consists of two procedures $GenerateLegalMove$ and $OptimizeVertexPositions$. The same operation at the first stage of the mesh optimization is used for $OptimizeVertexPositions$ of the second stage. $GenerateLegalMove$ handles the energy term $E_{rep}(K) = c_{rep}m$, where c_{rep} is a user-selectable parameter to control the tradeoff between geometric fit and compact representation, and m is the number of vertices in K. A larger c_{rep} value results in a mesh with fewer vertices, so the optimization algorithm minimizes $E_{rep}(K)$ with fewer vertices. The implementation of the second stage includes the procedure $\text{OptimizeVertexPositions}$ before an edge operation, to check if the adjusted vertex by the edge operation can lower the energy function $E(K,V)$.

The authors of the paper added more conditions to check edge operation eligibility in their implementation 3 years after publication. These conditions involve checking for sharp edges and the number of sharp edges connected to the vertices of the edge. An edge is considered sharp if it is a boundary edge or flagged as sharp based on the cosine value of dihedral angle with the edge being less than a designated value ($-1e^{31}$) or a user-specified value. This test was added to prevent boundary/crease migration inwards and corner migration.

Result of Mesh Optimization

For the mesh optimization, a cylindrical mesh M (Figure 6) with 8,320 vertices and 16,384 faces was used. This mesh was generated using cloth simulation in the 3d modeling tool Blender. The same parameter setting and the same optimization scheme were used as the authors did, where each loop in the process performs the first stage method, the second stage method, and finally the first stage method again. The entire process comprises 4 loops with different 4 spring constant ($1e^{-2}, 1e^{-3}, 1e^{-4}, 1e^{-8}$) and $c_{rep} = 1e^{-3}$. The resulting optimized mesh (Figure 7) has 206 vertices and 392 faces. The optimized result with $c_{rep} = 1e^{-4}$ (Figure 8) has 565 vertices and 1102 faces, indicating that a lower c_{rep} value results in more vertices in the optimized mesh.

However, the optimized mesh (Figure 9) with $c_{rep} = 1e^{-5}$ and 1,490 vertices and 2,931 faces has unexpected protruding vertices. It appears that larger c_{rep} values remove edges containing protruding vertices. The newly added conditions were applied to the optimization with $c_{rep} = 1e^{-5}$, but it didn't resolve this issue.

Conclusion

This project successfully achieved four objectives:

1. Implement the first stage.
2. Explain the legal moves for two edge operations.
3. Implement the second stage.
4. Present the algorithm results.

Although the result with $c_{rep} = 1e^{-5}$ is not satisfactory, the optimized meshes with larger c_{rep} values demonstrate the paper's algorithm ability to significantly reduce the number of vertices and faces while preserving the geometric features of the original mesh. As mentioned in the paper, the optimized mesh has similar protruding vertices without the spring energy term. Therefore, further experiments are needed to investigate the effect of the spring energy term on the protruding vertices.

Reference

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. and Stuetzle, W., 1993, September. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (pp. 19-26).

Botsch M. *Polygon Mesh Processing*. Natick, Mass.: A K Peters; 2010.

```
struct Vertex
{
    Vertex() = delete;
    Vertex(PoolAllocator* allocator)
        : arhe(allocator)
    {}


    TVector<HalfEdge*> arhe;
    uint32_t id;
    Vector3 point;
};

struct Face
{
    HalfEdge* herep;
    uint32_t id;
};

struct Edge
{
    HalfEdge* herep;
    bool is_sharp;
};

struct HalfEdge
{
    HalfEdge* prev;  // previous half edge in ring around face
    HalfEdge* next;  // next half edge in ring around face
    HalfEdge* sym;   // pointer to symmetric half edge (or 0)
    Vertex* vert;    // vertex to which this half edge is pointing
    Face* face;
    Edge* edge;
};

struct Mesh
{
    Mesh(PoolAllocator* allocator)
        : num_vertex(0)
        , num_face(0)
        , num_edges(0)
        , vertex_map(allocator)
        , face_map(allocator)
        , allocator(allocator)
    {}


    uint32_t num_vertex;     // id to assign to next new vertex
    uint32_t num_face;       // id to assign to next new face
    uint32_t num_edges;
    TUnorderedMap<uint32_t, Vertex*> vertex_map;
    TUnorderedMap<uint32_t, Face*> face_map;
    PoolAllocator* allocator;
};
```

Figure 1 - Halfedge-based data structure used in the project.

```
$OptimizeMesh(K_0, V_0)$ {
$K := K_0$
$V := OptimizeVertexPositions(K_0, V_0)$
"- $Solve\ the\ outer\ minimization\ problem.$"
$repeat$ {
  $(K', V') := GenerateLegalMove(K, V)$
  $V' = OptimizeVertexPositions(K', V')$
  $If\ E(K', V') < E(K, V)\ then$
    $(K, V) := (K', V')$
  $endif$
} $until\ convergence$
$return\ (K, V)$
}
```

Figure 2 - Mesh Optimization Algorithm

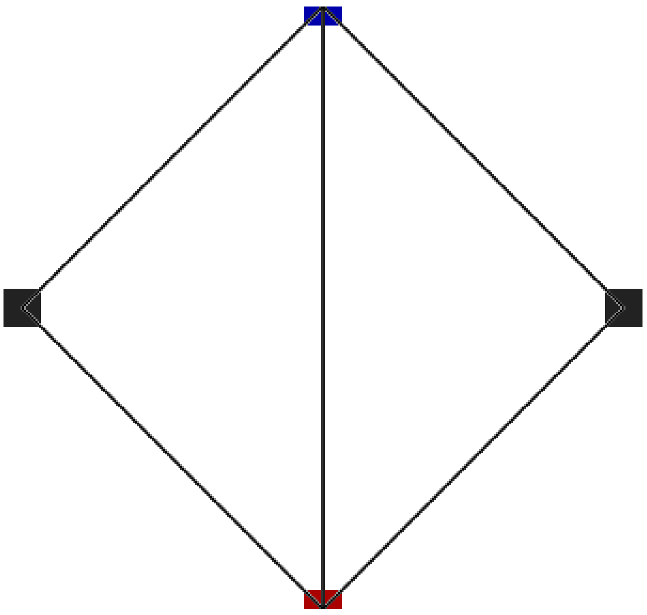

Figure 3 - Edge Collapse Case (1)

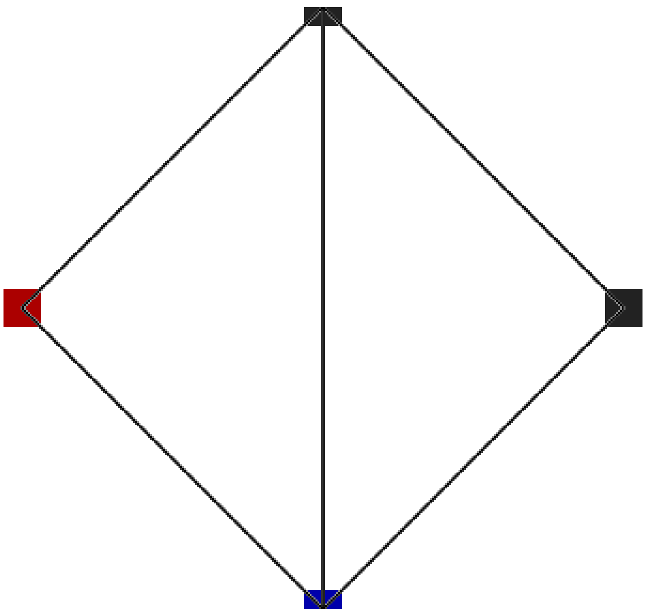

Figure 4 - Edge Collapse Case (2)

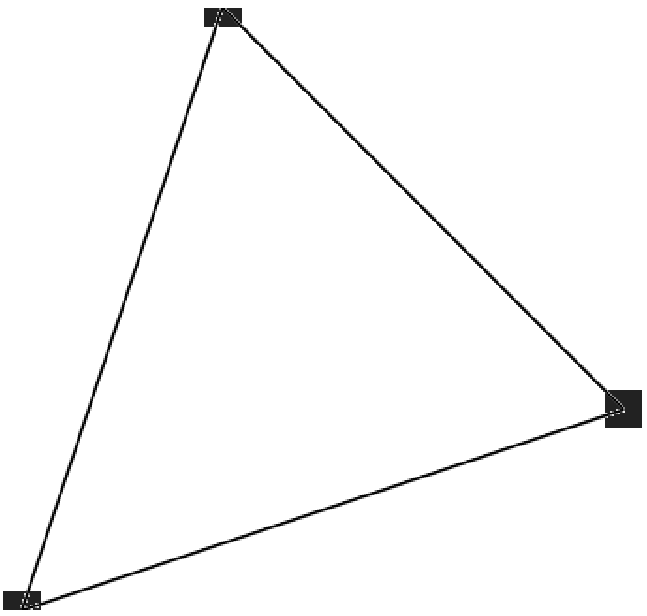

Figure 5 - Edge Collpase Result for Case (2)

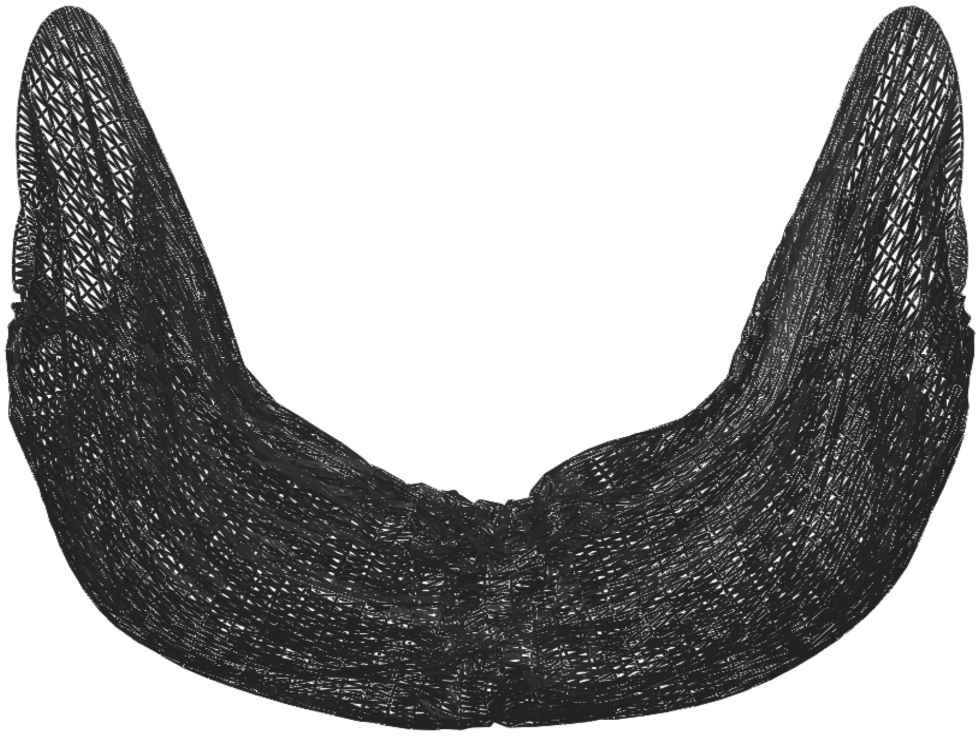

Figure 6 - Original Mesh M. 8,320 Vertices and 16,384 Faces.

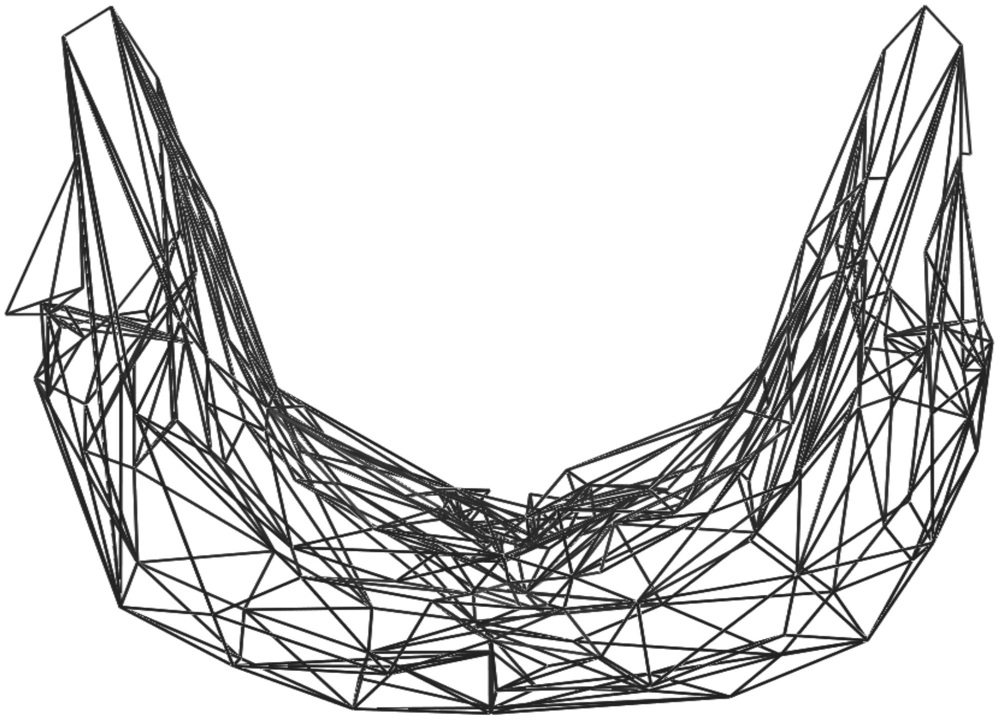

Figure 7 - Optimized Mesh M' from M with $c_{rep} = 1e^{-3}$. 206 Vertices and 392 Faces.

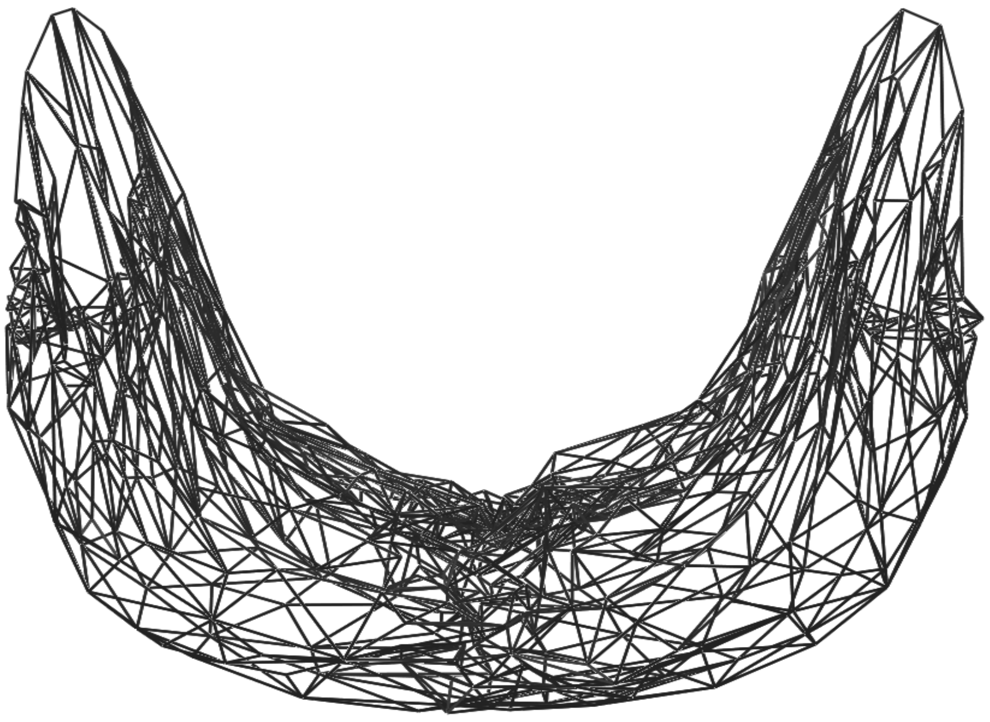

Figure 8 - Optimized Mesh M' from M with $c_{rep} = 1e^{-4}$. 565 Vertices and 1102 Faces.

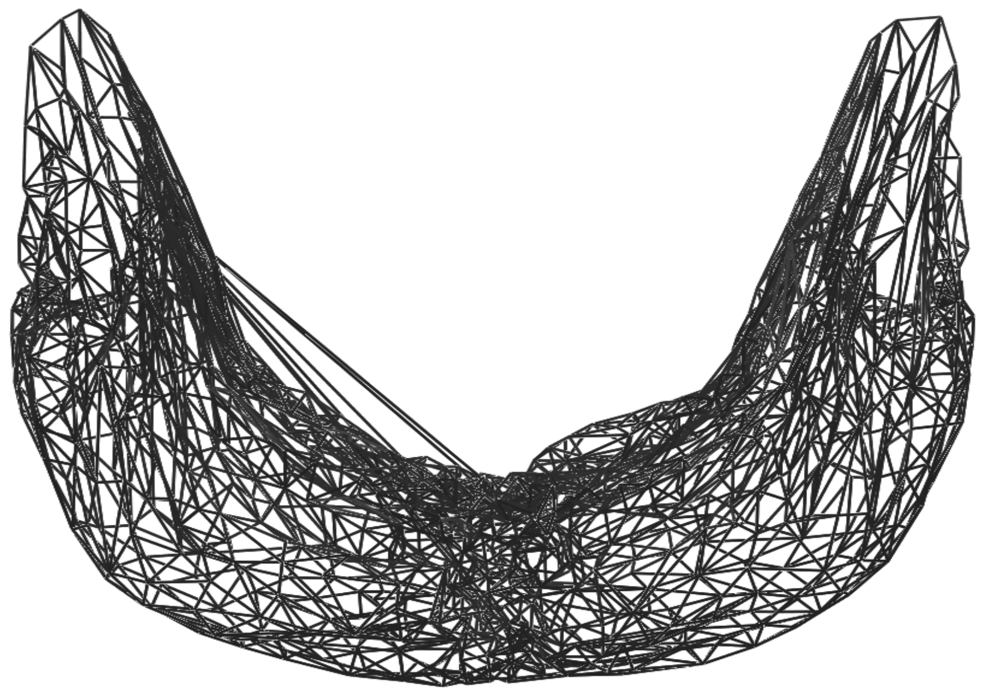

Figure 9 - Optimized Mesh M' from M with $c_{rep} = 1e^{-5}$. 1490 Vertices and 2931 Faces.