Introduction to Geometric Processing through Optimization

Gabriel Taubin
Brown University

Computer representations of piecewise smooth surfaces have become vital technologies in areas ranging from interactive games and feature-film production to aircraft design and medical diagnosis. One of the dominant surface representations is polygon meshes. Most computer graphics applications require simple and efficient geometry-processing algorithms to operate on the very large polygon meshes used. In general, developing these algorithms involves fundamental concepts from pure mathematics, algorithms and data structures, numerical methods, and software engineering.

As an introduction to the field, this article shows how to formulate several geometry-processing operations to solve systems of equations in the "least-squares" sense. The equations are derived from local geometric relations using elementary concepts from analytic geometry, such as points, lines, planes, vectors, and polygons. Simple and useful tools for interactive polygon mesh editing result from the most basic descent strategies to solve these optimization problems. Throughout the article, I develop the mathematical formulations incrementally, keeping in mind that the objective is to implement simple software for interactive editing applications that works well in practice. You can implement higher-performance versions of these algorithms by replacing the simple solvers proposed here with more advanced ones.

Representing Polygon Meshes

We can represent polygon meshes in many ways; here, I use an array-based indexed face set representation, in which a polygon mesh P = (V, X, F) is composed of a finite set V of vertex indices, a table of 3D vertex coordinates X = $\{x_i\colon i \in V\}$ indexed by a vertex index, and a set F of polygon faces, in which a face $f = (i_1, \ldots, i_{n_f})$ is a sequence of nonrepeating vertex indices. The number n_f of vertex indices in a face can vary from face to face, and, of course, every face must have a minimum of $n_f \geq 3$ vertices. Two cyclical permutations of the same sequence of vertex indices are regarded as the same face, such as (0, 1, 2) and (1, 2, 0), but when a sequence of vertex indices results from another one by inverting the order, such as (0, 1, 2) and (2, 1, 0), we regard the two sequences as different faces and say that the two faces have opposite orientations. This representation does not support faces with holes, which is perfectly acceptable for most applications. For each face $f = (i_1, \ldots, i_{n_f})$, $V(f) = (i_1, \ldots, i_{n_f})$ is the set of vertex indices of the face considered as a set.

In terms of data structures, if N_V is the total number of vertices of the mesh, we can assume that the set of vertex indices is {0, …, N_V − 1}, composed of consecutive integers starting at 0. The vertex coordinates are represented as a linear array of 3 N_V floating-point numbers, and the set of faces F is represented as a linear array of integers resulting from concatenating the faces. To handle polygon meshes with different-sized faces (that is, a different number of vertex indices per face), we append a special marker at the end of each face, such as the integer –1 (which is never used as a vertex index) to indicate the end of the face. For example, [0, 1, 2, –1, 0, 2, 3, –1] would be the representation for the set of faces F of a polygon mesh composed of two triangular faces, f_0 = (0, 1, 2) and f_1 = (0, 2, 3), and V = {0, 1, 2, 3}. The particular order of the faces within the linear array is not important, but once a particular order is chosen, we will assume that it does not change. I will refer to a face's relative location within the array as its face index. If N_F is the total number of faces, then the set of face indices is {0, …, N_F – 1}.

Polygon Mesh Smoothing

Large polygon meshes are usually generated by measurement processes, such as laser scanning or structured lighting, that result in measurement errors or noise in the vertex coordinates. In some cases, systematic errors are generated by algorithms that generate polygon meshes, such as isosurface algorithms. In general, we must remove the noise to reveal the hidden signal but without distorting it. Algorithms that attempt to solve this problem are referred to as smoothing or denoising algorithms. It is probably fair to say that the whole field of digital geometry processing grew out of early solutions to this problem.

My goal here is to offer a simple and intuitive methodology for attacking the problem in various ways. You can expand on it with similar approaches to formulate other, more complex problems, such as large-scale deformations for interactive shape design. In smoothing algorithms, noise removal is constrained to changes to the values of the vertex coordinates X. Neither the set of vertex indices V nor the faces F of the polygon mesh are allowed to change.

Perhaps the simplest and oldest method to remove noise from a polygon mesh is Laplacian smoothing. In classical signal processing, noise is removed from signals sampled over regular grids by convolution—that is, by averaging neighboring values. Laplacian smoothing is based on the same idea: each vertex coordinate x_i is replaced by a weighted average of itself and its first-order neighbors. But to properly describe this method, we first need to formalize a few things.

The Primal Graph of a Polygon Mesh

The graph, or more precisely, the primal graph (we will see the dual graph later), G = (V, E) of a polygon mesh P = (V, X, F) is composed of the set of polygon mesh vertex indices V as the graph vertices and the set E of mesh edges as the graph edges. A mesh edge is an unordered pair of vertex indices e = (i, j) = (j, i) that appear consecutive to each other, irrespective of order, in one or more faces of the polygon mesh. In that case, we say that the face and the edge are incident to each other. The set of edges incident to a face f is E(f); n_f is the number of edges in this set (equal to the number of vertices in the face. For example, for the face f = (0, 1, 2), it is E(f) = {(0, 1), (1, 2), (2, 0)}. Note that one or more faces might be incident to a common edge. The set of faces incident to a given edge e = (i, j) is F(e), which has n_e number of incident faces. A boundary edge has exactly one incident face, a regular edge has exactly two incident faces, and a singular edge has three or more incident edges. We say that two vertices i and j are first-order neighbors if the pair (i, j) of vertex indices is an edge. For each vertex index, the set $V(i) = \{j: (i, j) \in E\}$ is the set of first-order neighbors of i, and n_i is the number of elements in this set.

In terms of data structures, we can represent the mesh edges (i, j) as a linear array of $2N_E$ vertex indices, where N_E is the total number of polygon mesh edges. To make each edge's representation unique, this array stores either the pair (i, j) if $i < j$ or (j, i) if $j < i$. Although this array is constructed strictly as a function of the faces, and as such, does not add any new information, constructing and storing it as an additional data structure is beneficial: several of the iterative algorithms discussed here can be efficiently implemented as a linear traversal of the edge array. However, implementing the same algorithms by traversing the face array usually increases complexity or creates special cases that complicate the algorithms.

To efficiently construct the array of edges from the array of faces, we use an additional data structure to represent a graph over the set of vertex indices. This graph data structure is initialized with the set of vertex indices and an empty set of edges. The graph data structure supports two efficient operations: get(i, j) and insert(i, j). The operation get(i, j) returns the edge index assigned to the edge (i, j), if such an edge exists, and a unique identifier such as –1, which is not used as an edge index if the edge (i, j) does not yet belong to the set of edges. If the edge (i, j) does not yet exist, the operation insert(i, j) appends the pair of indices to the array of edges and assigns its location in the array to the edge as the unique edge index. In this way, the index 0 is assigned to the first edge created, and consecutive indices are assigned to edges created later. An efficient implementation of this graph data structure can be based on a hash table.

For some algorithms, it is useful to have an efficient method to determine the number n_e of incident faces per edge, as well as to access those faces' indices. The graph data structure can also be extended to support this functionality. We can represent number n_e as an additional field in the record used to represent the edge e in the graph data structure or as an external variable-length integer array. Each value is initialized to 1 via the insert(i, j) operation during the graph data structure's construction and incremented during face array traversal every time the get(i, j) operation returns a valid face index. We can represent the sets of faces F(e) incident to the edges as an array of variable-length arrays indexed by the edge index and construct them as well during the graph data structure's construction.

Vertex Evolution Algorithms

A large family of polygon mesh-editing algorithms follow three steps. First, for each vertex index i of the polygon mesh, compute a vertex displacement vector Δx_i. Second, after all the vertex displacement vectors are computed, apply the vertex displacement vectors to the vertex coordinates, $x'_i = x_i + \lambda\Delta x_i$, where λ is a fixed-scale parameter (either user defined or computed from the polygon mesh data). Finally, replace the original vertex coordinates X with the new vertex coordinates X′. These three steps are repeated for a certain number of times specified in advance by the user or until a certain stopping criterion is met.

All the algorithms discussed here belong to this family. In terms of storage, these algorithms require an additional linear array of $3N_v$ floating-point numbers to represent the vertex displacement. The vertex coordinates are updated using this procedure in linear time as a function of the number of vertices. Of course, the time and storage complexity of evaluating the vertex displacements (to determine the scale parameter or whether the stopping criterion is met) must be added to the algorithm's overall complexity. In general, algorithms with linear time and storage complexity as a function of polygon mesh size are the only algorithms that scale properly for practical use with very large polygon meshes.

Laplacian Smoothing

As I mentioned earlier, Laplacian smoothing replaces each vertex coordinate x_i with a weighted average of itself and its first-order neighbors. More precisely, for each vertex index i, we compute a vertex displacement vector

$$\Delta x_i = \frac{1}{n_i}\sum_{j\in V(i)}(x_j - x_i)$$

as the average over the first-order neighbors j of vertex i (of vectors $x_j - x_i$). After we compute all these displacement vectors as functions of the original vertex coordinates X, we apply the vertex displacement to the vertex coordinates with a scale parameter in the range $0 < \lambda < 1$ ($\lambda = 1/2$ is usually a good choice).

To compute vertex displacement vectors, we need an efficient way of finding all the first-order neighbors of each vertex index, particularly the number of elements in the sets of first-order neighbors. Unfortunately, the data structures introduced so far do not provide such methods. However, because each edge (i, j) contributes a term to the sums defining both displacement vectors Δx_i and Δx_j, we can accumulate all the displacement vectors together while linearly traversing the array of edges. During the same traversal, we also accumulate the number of each vertex's first-order neighbors, so that we can normalize the vertex displacement vectors.

Descent Algorithms

Let us consider the sum of the squares of the edge lengths $\|x_i - x_j\|$ as a function of a polygon mesh's vertex coordinates:

$$E(x) = \sum_{(i,j)\in E}\|x_j - x_i\|^2,$$

where x is the table of vertex coordinates X regarded as a column vector of dimension $3N_V$, resulting from concatenating all the N_V 3D vertex coordinates x_i. Note that as a function of x, this function is quadratic, homogeneous, and non-negative definite. Consequently, it attains the global minimum 0 when all the edges have zero length or, equivalently, when all the vertex coordinates have the same value. As a result, starting from noisy vertex coordinates X, computing the global minimum of this function does not constitute a smoothing algorithm, but taking a small step along a descent direction toward a local minimum (which in this case, is the global minimum) is still a valid heuristic. Gradient descent is the simplest iterative algorithm to locally minimize a function such as this one. The negative of the gradient of the function E(x) is the direction of steepest descent. We can look at the gradient of E(x) as the concatenation of the N_V 3D derivatives of E(x) with respect to the vertex coordinates x_i:

$$\frac{\partial E}{\partial x_i} = \sum_{j\in V(i)}(x_i - x_j) = -\sum_{j\in V(i)}(x_j - x_i).$$

To construct a descent algorithm, we still need to choose a positive scale parameter λ so that the vertex-coordinates update rule

$$x'_i = x_i - \lambda\frac{\partial E}{\partial x_i}$$

actually results in the value of the function to decrease: $E(x') < E(x)$. Even though the negative of the gradient is the steepest descent direction, selecting too large a value of λ (called overshooting) usually results in the function's value actually increasing. In this case, an iterative algorithm based on an improper rule to choose λ might result in oscillatory behavior or even divergence.

After we choose a descent direction v, to find the optimal value for λ, we can look at the 1D problem of minimizing E(x + λv) with respect to λ. In our case, because the function E(x) is quadratic, this is a simple 1D quadratic optimization problem that we can solve in closed form. Explicitly, we can compute the optimal value for λ by solving the linear equation

$$\lambda\left(\sum_{(i,j)\in E}\| v_i - v_j \|^t\right)+\left(\sum_{(i,j)\in E}(v_i - v_j)^t(x_i - x_j)\right)=0.$$

Because a second traversal of the list of edges is necessary to accumulate the two coefficients of this linear equation, for long meshes, this step is usually avoided and replaced with user-specified values for λ.

Although we can already observe a close connection between this method and Laplacian smoothing, the descent directions chosen in both cases are not identical. Why is that so?

The Jacobi Iteration

The Jacobi iteration is the simplest iterative method to solve large square diagonally dominant systems of linear equations Ax = b, where the ith equation is solved independently for the ith variable, keeping the other variables fixed and resulting in a new value for the ith variable. After we use this method to determine the new values for all the variables, the old variables are replaced by—or displaced in the direction of—the new variables, and the process repeats for a fixed number of iterations or until convergence. Under certain conditions, the method converges to the solution of the system of equations.

In the context of optimizing a quadratic function E(x), the system of equations to be solved corresponds to making the function's gradient equal to zero. Even when the performance function is not a quadratic function of the variables, we can use this method to construct a properly scaled descent algorithm. This generalized Jacobi iteration is equivalent to minimizing the function E(x) with respect to the ith variable independently. If we write the new value for the variable x_i as the old value plus a displacement, $x_i + \Delta x_i$, in the case of the sum of square edge lengths function, determining the displacement Δx_i reduces to solving the equation

$$\begin{aligned}0 &= \frac{\partial E}{\partial x_i}\left(x_1,\ldots,x_i+\Delta x_i,\ldots,x_{N_V}\right)\\ &= \sum_{j\in V(i)}\left(\Delta x_i + x_i - x_j\right) = n_i\Delta x_i - \sum_{j\in V(i)}\left(x_j - x_i\right),\end{aligned}$$

which results in the Laplacian-smoothing displacement vector. We can determine the optimal value for the parameter λ by minimizing E(x + $\lambda\Delta$x) with respect to λ, as described earlier, although λ = 1/2 usually works well in practice.

How to Fix Laplacian Smoothing

Laplacian smoothing is a simple, easy-to-implement algorithm. It produces smoothing, but when too many iterations are applied, the shape of the polygon mesh undergoes significant and undesirable deformations. As I mentioned, this is because the function E(x) being minimized has a global minimum (actually, infinitely many, but unique modulo a 3D translation) that does not correspond to the result of removing noise from the original vertex coordinates. Any converging descent algorithm will approach that minimum, which is why we observe significant deformations in practice. In our case, in Laplacian smoothing, all the vertex coordinates of the polygon mesh to converge to their centroid:

$$\frac{1}{N_V}\sum_{i=1}^{N_V} x_i.$$

In the literature, this problem is referred to as shrinkage. Many algorithms, based on different mathematical formulations ranging from signal processing to partial differential equations, have been proposed over the past 15 or more years to deal with, and solve, the shrinkage problem, but I will not survey these algorithms here.

For the sake of simplicity, I take the viewpoint that the shrinkage problem is a direct result of the "wrong" performance function being minimized. Consequently, I address the shrinkage problem by modifying the performance function being minimized. However, after constructing each new performance function, I follow the same simple steps described earlier, of minimizing the function with respect to each variable independently to obtain a properly scaled descent vector and then updating the variables as in Laplacian smoothing by displacing the vertex coordinates in the direction of this descent vector. Finally, I repeat the process for a predetermined number of steps or until convergence based on an error-tolerance stop test.

Vertex Position Constraints

The most obvious way to prevent shrinkage is to update all the vertex coordinates. More formally, we partition the set of vertex indices V into two disjoint sets: a set V_C of constrained vertex indices and a set V_U of unconstrained vertex indices.

We also partition the vector of vertex coordinates x into a vector of constrained vertex coordinates x_C and a vector of unconstrained vertex coordinates x_U. We keep the same sum of squares of edge lengths function $E(x) = E(x_U, x_V)$, but we regard it as a function of only the unconstrained vertex coordinates x_U, with the constrained vertex coordinates x_V regarded as constants. As such, this function is still quadratic and nonnegative definite, but it is no longer homogeneous.

Generally, this function still has a unique minimum (modulo a translation in this case) that has a closed-form expression, and the minimum does not correspond to placing all the vertices at a single point in space. If we apply the same approach I described earlier to compute a descent direction by minimizing $E(x_U)$ with respect to each unconstrained variable independently, we end up with the same descent vectors as in Laplacian smoothing and the same descent algorithm, but here, only the unconstrained vertex coordinates. So, this algorithm's computational cost is roughly the same as that for Laplacian smoothing.

Unfortunately, we still see shrinkage. In general, it is not clear which vertices should be constrained and which should be free to move, but within an interactive modeling system that allows for interactive vertex selection, this effectively smoothes out selected portions of a polygon mesh, which is useful in practice.

Rather than keeping the constrained vertices at their original positions, we can assign them new target positions, in which case the constrained vertices can be updated first and then kept fixed during the algorithm iterations. Unfortunately, if the constrained vertex displacements are large compared with the average edge length, this algorithm might produce noticeable shape artifacts during the iterations.

An alternative is to switch to a soft-constraints strategy, in which all the variables are free to move again and the constraints are satisfied in the least-squares sense by adding one or more terms to the function being minimized. For example, in our case, we consider this function

$$E(x) = \sum_{(i,j)\in E} \| x_j - x_i \|^2 + \mu \sum_{i\in V_C} \| x_i - x_i^0 \|^2,$$

where μ is a positive constant, the second sum is over the constrained vertices, and x_i^0 is a target constrained-vertex position provided as input data to the algorithm. By minimizing each variable independently, we obtain the same expression as in Laplacian smoothing for the displacements Δx_i corresponding to the unconstrained vertices, and

$$\Delta x_i = \frac{1}{n_i + \mu}\left(\sum_{j\in V(i)} \left(x_j - x_i\right) + \mu\left(x_i^0 - x_i\right)\right)$$

for the displacement Δx_i corresponding to the constrained vertices.

Face-Centroid Constraints

To produce acceptable results, we must constrain many of the vertices. Rather than imposing constraints on vertex positions, we impose similar constraints on some or all of the face centroids. The intuition here is that the face centroids are weighted averages of the face vertex coordinates because of the smoothing process, so the problem is how to transfer that smooth-shape information back from the face centroids to the vertex coordinates. Continuing with a soft constraints approach, we can consider the following performance function, which looks very similar to the one used to impose soft vertex constraints:

$$E(x) = \sum_{(i,j)\in E} \| x_j - x_i \|^2 + \mu \sum_{f\in F_C} \| x_f - x_f^0 \|^2,$$

where F_C is the subset of constrained faces (it could be all the faces). For each f, we express the centroid x_f as the average of the face vertex coordinates:

$$x_f = \frac{1}{n_f}\left(x_{i_1} + \ldots + x_{i_{n_f}}\right),$$

so that we can regard the overall function as a function of only the vertex coordinates, and x_f^0 as the target 3D point value for the face centroid. For example, x_f^0 could be the face centroid's initial value before we apply any smoothing. Even though we would start the algorithm with the term of the performance function corresponding to the face-centroid constraints identical to zero, it could become nonzero after one or more iterations while the overall function decreases.

By applying the generalized Jacobi strategy of minimizing with respect to each variable independently, we obtain the following expression for each displacement Δx_i:

$$\Delta x_i = \frac{1}{n_i + \mu \sum_{f\in F_C(i)} \frac{1}{n_f}} \left(\sum_{j\in V(i)} \left(x_j - x_i\right) + \mu \sum_{f\in F_C(i)} \frac{1}{n_f}\left(x_f^0 - x_f\right)\right),$$

where $F_C(i)$ is the subset of constrained faces f containing vertex index i. These displacements and normalization factors can be accumulated as in previous algorithms by initializing to zero, traversing the array of edges, traversing the array of constrained faces, and then normalizing. Once we compute the displacements, we can update the vertex coordinates as in Laplacian smoothing.

Face-Normal Constraints

None of the constraints discussed so far allow for direct control of local surface orientation. A smoothing algorithm that can selectively control this orientation is a useful tool in an interactive polygon-mesh-editing system, and yet another possible way to prevent the shrinkage problem of Laplacian smoothing. To control surface orientation, we must introduce surface-normal vectors into the performance function to be minimized. As we have done for the face centroids, one possibility is to derive an expression for a face-normal vector as a function of the face vertex coordinates and then add an error term to the performance function for all or some of the faces. Because doing so results in nonlinear equations to solve for the displacement vectors, we propose a simpler alternative. We consider the following performance function,

$$E(x) = \sum_{(i,j)\in E} \| x_j - x_i \|^2 + \mu \sum_{f\in F_N} \sum_{(i,j)\in E(f)} \left(u_f^t (x_j - x_i) \right)^2,$$

where the first term is the sum of square edge lengths as in all the previous performance functions, and the second term is a sum over a subset F_N of faces in which we want to impose the face-normal constraint. For each such face-edge pair, we impose as a soft constraint that the face-normal vector u_f be orthogonal to the face boundary vector $x_j - x_i$. The user provides face normal vectors u_f as additional input to the algorithm.

However, our polygon mesh representation does not force faces to be planar. For this to happen, the soft constraint must be satisfied for all the face boundary vectors of each face. Note that with the constrained face-normal vectors regarded as constants, this performance function is also quadratic and homogeneous in the vertex coordinates. In this case, the displacement vectors satisfy the following linear equations:

$$\left(n_i I + \mu \sum_{f\in F_N(i)} \sum_{j\in V(f,i)} n_f n_f^t \right) \Delta x_i =$$

$$\sum_{j\in V(i)} (x_j - x_i) + \mu \sum_{f\in F_N(i)} \sum_{j\in V(f,i)} n_f n_f^t (x_j - x_i),$$

where I is the 3 × 3 identity matrix and V(f, i) = V(f) ∩ V(i) is the set of vertices that belong to face f and are first-order neighbors of vertex i (there are exactly two such vertices when the face is known to contain vertex i). The 3 × 3 matrix on the left side multiplies Δx_i, which is symmetric and positive definite and can be accumulated during the mesh traversal along with the other sums. We can easily invert it using Cholesky decomposition.

Smoothing Face-Normal Vectors

Assume that we have a face-normal vector u_f for every face of the polygon mesh. We concatenate these face-normal vectors to form a vector u of dimension $3N_F$, which we consider not a user-provided constant but a new variable. Of course the variables u and x are not independent. But rather than imposing their relations as hard constraints, we regard them as independent variables and represent their relations as soft constraints, as in the case of face-normal constraints.

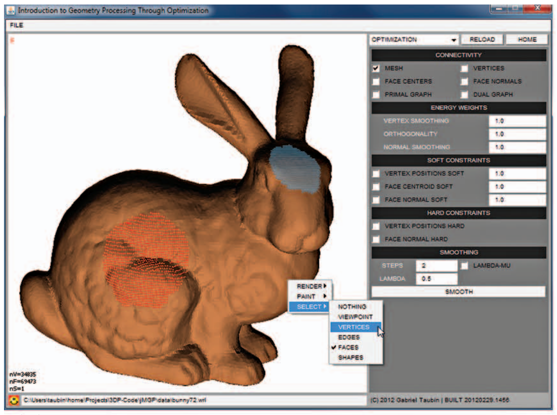
To remove noise from the face-normal vector, we consider the performance function

$$E(u) = \sum_{(f,g)\in E^*} \| u_f - u_g \|^2,$$

which is the sum over the dual-mesh edges of the square differences of face-normal vectors. The set of dual-mesh edges E* consists of pairs (f, g) of faces sharing a regular edge (one that has exactly two incident faces). Formally, a mesh's dual graph has the mesh's faces as dual-graph vertices and the dual-mesh edges as dual-graph edges. It is important to note the similarity between this performance function and the sum of square edge lengths.

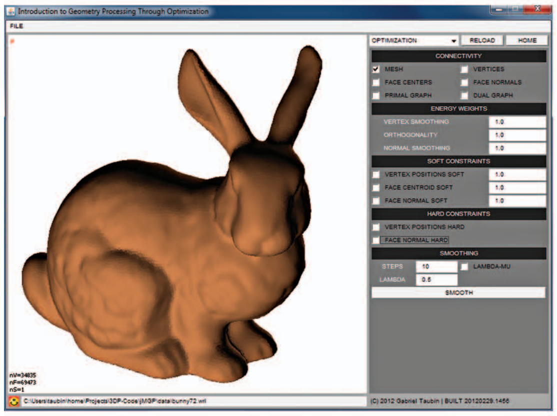
If we initialize the face-normal vectors from the vertex coordinates, we can use this performance function to first remove noise from the face-normal vectors and then use the smoothed face normal as face-normal constraints in a second smoothing process applied to the vertex coordinates. This second process of smoothing the vertex coordinates with face-normal constraints is an integration of smoothed face normals. Applying our strategy to this performance function, we can compute a displacement vector Δu_f for the face-normal vectors. Because after we apply these face-normal displacements the face-normal vectors are no longer unit length, we normalize the updated face-normal vectors to unit length and then perform the face-normal integration step.

An alternative is to consider the following performance function,



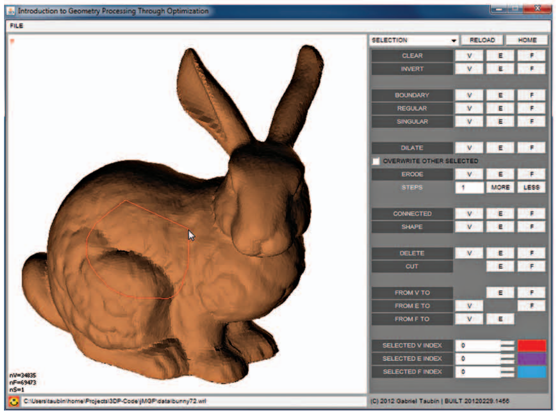
[✓ MESH]
VERTEX SMOOTHING 1.0
ORTHOGONALITY 1.0
NORMAL SMOOTHING 1.0
VERTEX POSITIONS SOFT 1.0
FACE CENTROID SOFT 1.0
FACE NORMAL SOFT 1.0
VERTEX POSITIONS HARD
FACE NORMAL HARD
STEPS 2
LAMBDA-MU
LAMBDA 0.5
✓ FACES

(a)


Introduction to Geometry Processing Through Optimization
FILE
OPTIMIZATION
RELOAD
HOME
CONNECTIVITY
ENERGY WEIGHTS
SOFT CONSTRAINTS
HARD CONSTRAINTS
SMOOTHING
SMOOTH
(C) 2012 Gabriel Taubin | BUILT 20120229.1456

[✓ MESH]
VERTEX SMOOTHING 1.0
ORTHOGONALITY 1.0
NORMAL SMOOTHING 1.0
VERTEX POSITIONS SOFT 1.0
FACE CENTROID SOFT 1.0
FACE NORMAL SOFT 1.0
STEPS 10
LAMBDA 0.5

(b)



STEPS 1
SELECTED V INDEX 0
SELECTED E INDEX 0
SELECTED F INDEX 0

(c)

Figure 1. Example. A sample application implements all concepts described in this article, including (a) smoothing, (b) applying an algorithm to the mesh in Figure 1a without constraints, and (c) using a selection panel to specify multiple subsets of vertices and faces.

$$E(x,u) = \sum_{(i,j)\in E} \| x_j - x_i \|^2 + \sum_{f\in F} \sum_{(i,j)\in E(f)} \left(u_f^t (x_j - x_i) \right)^2 + \gamma \sum_{(f,g)\in E^*} \| u_f - u_g \|^2,$$

and apply our minimization strategy to the variables x and u together. In this way, we can determine displacement vectors Δx_i and Δu_f as functions of x and u and then update both variables at once, followed by face-normal unit length normalization. I leave the details of these derivations to you.

Note that the two approaches discussed here allow for hard constraints to be applied to a subset of face-normal vectors. As in the case of hard vertex-coordinate constraints, the constrained values are just not updated. Also, we can impose soft face-normal constraints by adding yet another term to the last performance function (the sum over a subset of faces of square errors between face-normal vectors and target face-normal vectors).

We have considered a number of ways to add constraints to Laplacian smoothing. All these strategies can be combined into a single general polygon-mesh-smoothing that allows for hard and soft vertex position constraints on disjoint subsets, vertices and soft face-centroid constraints on a subset of faces, and hard and soft face-normal constraints on a disjointed subset of faces.

Overall, the performance function has six terms, and the contribution of each can be controlled through six corresponding weights. An implementation of what I described in this article, written in Java, appears in Figure 1. You can download the software at http://mesh.brown.edu/optimization, along with a more detailed description of its operation. By selectively setting the weights of some of the six terms of the performance function to zero, and by constraining all the vertex positions or all the face-normal vectors not to change, you can implement all the algorithms discussed in this article, including Laplacian smoothing.

Gabriel Taubin is a professor in Brown University's School of Engineering and the editor in chief of this magazine. Contact him at taubin@brown.edu.