Volume 0 (1981), Number 0 pp. 1–11 COMPUTER GRAPHICS *forum*

A Linear Variational System for Modeling From Curves

James Andrews[†1,2] Pushkar Joshi[‡2] and Nathan Carr[§2]

[1]U. C. Berkeley
[2]Adobe Systems Inc.

Abstract

We present a linear system for modeling 3D surfaces from curves. Our system offers better performance, stability, and precision in control than previous non-linear systems. By exploring the direct relationship between a standard higher-order Laplacian editing framework and Hermite spline curves, we introduce a new form of Cauchy constraint that makes our system easy to both implement and control. We introduce novel workflows that simplify the construction of 3D models from sketches. We show how to convert existing 3D meshes into our curve-based representation for subsequent editing and modeling, allowing our technique to be applied to a wide range of existing 3D content.

Categories and Subject Descriptors (according to ACM CCS): COMPUTER GRAPHICS [I.3.5]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations;

1. Introduction

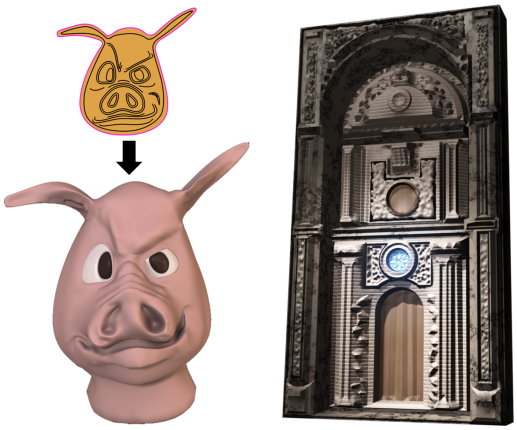


Figure 1: *Example shapes modeled using our framework.*

Sketch-based modeling of smooth shapes has allowed novice users to quickly create complex, organic 3D shapes from sketches. Users unfamiliar with 3D modeling can intuitively model and edit 3D shapes by drawing curves that denote significant shape features. These curves act as constraints for the 3D model: the surface must pass through these constraints and is usually kept smooth in all unconstrained regions. The user controls the 3D shape by modifying the constraint curves.

In this paper we show how to extend a standard linearized Laplacian framework to create a fast, robust curve-based freeform modeling system. Offering control directly analogous to that of Hermite splines, we allow direct control of tangents and curvatures along the constraint curves. Our technique can represent a vast majority of shapes users would expect to create with a sketch-based interface. We demonstrate how users can convert existing 2D artwork into 3D shapes easily and quickly.

Any new modeling system cannot afford to ignore the vast libraries of existing shapes that are stored as polygonal meshes. Therefore, we show how to reverse engineer existing meshes into our curve-based representation for subsequent editing.

2. Related Work

The literature for methods that simplify the construction of geometric models is vast, and contains approaches that

† e-mail: jima@eecs.berkeley.edu
‡ e-mail: pushkarj@adobe.com
§ e-mail: ncarr@adobe.com

use almost every surface representation studied in geometric modeling (polygon meshes, parametric patches, implicit functions, etc.). In this section, we limit our discussion to the area in which we claim our contributions: surface modeling via feature (or sketched) curves.

Subdivision surfaces are a popular choice of linear surface primitive, and have been extended to support editing with feature curves. For example, [Lev99] uses combined subdivision surface schemes to interpolate networks of curves. [BLZ00] provide subdivision surfaces with normal control at the boundary (without adding inconvenient control net vertices). A parametric or subdivision-based N-sided patch needs the control net that bounds the boundary patch, creating unnecessary internal control vertices that are difficult to manage. A potentially cumbersome modification of the internal control structure is needed to create sharp creases or holes. Independent of subdivision surfaces, N-sided patches with normal control at the boundary have a rich history in geometric modeling (e.g. [Gre83]). In his thesis, Loop described a method for constructing an N-sided Bezier patch from a convex polygon with G^1 boundary conditions [Loo92]. [KS08] describe a sketch-based interface (their "SketchCad" tool) for producing surface patches that interpolate user-drawn sketches with G^1 continuity. Such N-sided Bezier-like patches have also been implemented in commercial products like FreeDesign [GPR09].

Our desire for a simple, intuitive, yet powerful representation has led us instead to variational patches, where the surface is computed as the solution of a linearized optimization problem (the so-called "PDE method" of [BW90]). Our method is based in particular on the work of [BK04] and [SK01]. We add the ability to conveniently specify constraints as curves and a new method to control the surface behavior near these constraints.

Recent modeling interfaces targeted at novice users produce surfaces from curves. A common goal in these systems is to better capture the modeler's intent without burdening them by constraints imposed by the underlying mathematical/topological representation. The Teddy system showed that with the right surface representations and user interfaces, 3D modeling can be made accessible to novice users [IMT99]. The developers of "ShapeShop" [SWSJ05] use implicit surfaces to produce shapes with complex topologies. Nealen et al. [NSACO05] implemented a sketch-based interface for modifying existing meshes. Karpenko et al. [KH06] examined the issue of creating surfaces by inferring 3D shape from feature curves drawn in 2D by artists. FiberMesh built upon the ideas of Teddy, providing a richer surface representation and enhanced user interface [NISA07]. Joshi et al. [JC08] used a similar variational system to create 3D shapes from existing 2D content, and Olsen and Samavati [OS10] took a similar approach to modeling from sketches on images. Gingold et al. [GZ09] developed a surface modeler that can infer position and normal features from the shading applied to the model. Both the works of [NSACO05] and [GZ09] describe a method for specifying the surface normal at a constraint curve by rotating the frame of the Laplacian to match the prescribed surface normal. In contrast, our method simplifies the ability to set the surface normal independently on *either* side of the constraint. There has also been significant work in the area of modeling 3D space curves: the "ILoveSketch" system [BBS08] provides an elegant, artist-centric interface for drawing curves in 3D. The user-interfaces developed in many of these works are complimentary to our patch representation; in fact, our surface representation could be used as an improved, underlying system in many of the above-mentioned sketch-based interfaces.

3. Linear *vs*. Non-Linear Solvers

Nealen et al. [NISA07] recently explained two disadvantages of a linear solver very similar to ours: (1) planar position constraints generate planar solutions, and (2) the shapes generated by a linear bi-Laplacian system do not always distribute curvature in a desirable way. We introduce the direct manipulation of Cauchy constraints that solve both these problems, allowing us to, for example, produce approximations of spheres from circles used as constraint curves. Like other linear mesh-based solvers, our system depends on the initial domain used to define edge weights — so far, we have not observed this dependence to be a significant issue. A true non-linear solution should be independent of any geometry used to initialize the solver, barring local minima in energy space. Even though non-linear systems can capture more complex energies (e.g., [EP09]), a number of advantages reside in linear-system-based mesh solutions:

- **Performance**: FiberMesh [NISA07] uses a very fast non-linear smooth surface solver, but it still comes at a real performance cost compared to the linear system. Our basic (bi-Laplacian) solve is roughly equivalent to one iteration of the FiberMesh non-linear solver, which typically takes 5-10 iterations to converge. The method of Wardetzky et al. [WBH*07] is also impressively fast, but does not quite achieve interactive rates yet. Other methods, such as Xu et al. [XZ07], are not designed for interactive use and take on the order of minutes to converge for relatively small examples.

- **Convergence**: Any iterative non-linear solve requires some consideration of its convergence. In fact, in most cases, one cannot guarantee that the solver will converge, nor can one guarantee that a non-converging solver is sufficiently close to the solution [WBH*07]. We do not have a formal analysis of the FiberMesh solver, but have observed that in many simple cases (especially with concave constraint curves) it does not appear to converge in practice, at least as implemented in the publicly released demonstration application. The mesh surface instead oscillates between several states (Fig. 2(bottom)). In some cases the mesh surface

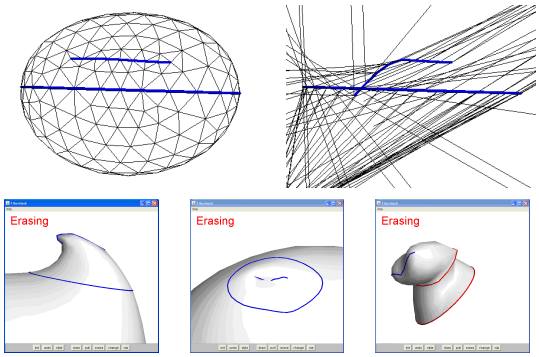


Figure 2: *Top row shows a simple case where horizontally dragging a constraint curve drawn on an ellipsoid causes the FiberMesh system to become unstable. Bottom row shows configurations in which the FiberMesh system produces an oscillating surface that never converges. This test was performed by invoking the eraser tool with no constraints selected to make the solver perform additional iterations.*

appears to diverge, rapidly inflating so extremely that the triangles become invalid, which is disastrous for its iterative solver. This occurs in very simple and easily-encountered situations (Fig. 2(top)). These issues are not surprising, as it is common for the stability of this kind of global non-linear optimization to depend on the initial configuration [HKS92].

- **Predictability**: Our linear system behaves like a 2-manifold, discrete analog to the familiar Hermite splines. Therefore, its behavior tends to be predictable: small changes to a constraint result in small changes to the local surface. In contrast, non-linear systems can have critical points where sudden, large changes in the solution can occur. For example, in the FiberMesh system a small change to a constraint can often result in surprisingly large changes to the surface. Perhaps the most noticeable and common issue along these lines is that a portion of the surface can suddenly become enormous, covering up other constraints lines and making further editing difficult.

Our system requires the artist to manage sparse tangent constraints in addition to position constraints. The introduction of tangent constraints, while possibly viewed as a burden, allows the artist an intuitive fine-grained level of control over surface detail. This coupled with the performance and robustness of the linear solve make it an appealing choice upon which to build a modeling system.

Finally, the linearization will also allow us to derive linear expressions for smooth vertices in terms of the constraints. In addition to potentially providing further performance gains, this turns out to be invaluable for the reverse engineering process (Section 6.2).

4. The Laplacian Modeling Framework

Our approach is to (1) define constraint curves and vertices on an input mesh surface, and (2) edit the surface by modifying those constraints, while keeping the surface smooth near unconstrained regions. In this section, we focus on the mesh-based Laplacian framework that underlies our smooth surface solver and how we implement our constraint curves within that framework. Approaches for using this Laplacian framework to obtain surface meshes are described in Sections 5 and 6.

4.1. Understanding the Laplacian Modeling Framework

Our basic smooth surface primitive closely follows the work of [BK04]: we assume we're given either a closed mesh with internal constraints, or an open mesh with constraints on the boundary. Given this triangulated domain, we solve $\Delta^k(\mathbf{p}) = 0$ for k typically equal to 2 or 3, where Δ is a discrete Laplacian operator. Specifically, the higher-order Laplacian is defined recursively at a mesh vertex as:

$$\Delta^k(\mathbf{u}) = \sum_i w_i(\Delta^{k-1}(\mathbf{u}) - \Delta^{k-1}(\mathbf{v_i})) \qquad (1)$$

$\mathbf{v_i}$ are the one-ring neighbors of vertex $\mathbf{u}$, $\Delta^0(\mathbf{u}) = \mathbf{u}$ and w_i are the cotangent weights [PP93] scaled by inverse vertex area. Cotangent weights are computed in a fixed initial domain and held constant to linearize the system. By formulating this equation for all unconstrained vertices in a mesh, we obtain a sparse symmetric positive-definite linear system of the form $\mathbf{Ax} = \mathbf{b}$, which we factor and solve with the SuperLU solver [DEG*99]. This approach is commonly used and is not novel, but we develop an intuitive understanding of its behavior by analogy to the 1-manifold case, and use this analogy to guide our development of a useful Cauchy constraint.

Consider the behavior of this modeling primitive in the simpler 1-manifold case of a curve. Solving for a curve with $\Delta^2(\mathbf{u}) = 0$ corresponds to constraining the fourth derivative (with respect to the user-defined parameterization) of a parametric curve to be zero, which is a property of a cubic Hermite curve. This curve is completely defined by constraining the positions and first derivatives at its end points — a Cauchy boundary condition (i.e., a combination of Dirichlet and Neumann boundary conditions). From these boundaries, we control the tangent direction and also the tangent strength. Likewise, the $k = 3$ case corresponds to a quintic Hermite spline (where we control the position, first and *second* derivatives), and the $k = 1$ case is a straight line (where we control only the position). The stable behavior and flexible artistic control of these ubiquitous curve primitives is what the method of [BK04] generalizes to 2-manifolds. Unlike the traditional extension of Hermite splines to surfaces via regular grids, this method allows us to generalize to "patches" of completely arbitrary topology, arbitrarily

complex boundaries, and with surface derivatives specified everywhere on the boundary. In our system, the constraints that specify the Cauchy boundary conditions (and optionally also higher-order derivative conditions) are called Cauchy constraints.

4.2. Cauchy Constraints as 1-D Laplacians

As discussed in Section 3, Cauchy constraints are essential to our 2-manifold solver – not just for additional artistic control, but also for avoiding the "planarity problem" discussed by [NISA07] and for mitigating the domain-dependence of our solution. For best results, we found that these Cauchy constraints should be implemented using "external" Cauchy constraints (we provide a more detailed argument in Appendix A). External constraints can be added by introducing non-visible *ghost geometry* along constraints to allow for higher-order surface control. Such ghost geometry has also been used previously for fluid simulations [FAMO99]. From the perspective of the solver, the ghost geometry completes missing neighborhood information around the mesh constraints, thus allowing basic operators (e.g., Laplace-Beltrami) to be directly evaluated at those locations. Although Cauchy constraints are simple to understand for 1-manifolds, extending these constraints to the 2-manifold case raises some interesting problems. First, generating consistent connected strips of ghost geometry around open boundaries can be a cumbersome task in the presence of sharp concavities [SK01]. Second, it is unclear how to introduce additional consistent strips of ghost geometry for internal constraints (i.e. constraints that are not on an open boundary of the mesh).

We address the above issues by maintaining *local* consistency of the mesh connectivity. We observe that local consistency is sufficient for providing artistic control over the surface near the constraint. That is, the ghost geometry does not need to be globally consistent with the rest of the mesh. Instead, *locally* completing a constraint vertex neighborhood by adding ghost vertices is sufficient.

In practice, the process of inserting ghost geometry to maintain local consistency is extremely simple: we complete the Laplacian formulation by adding ghost vertices that are mirrored versions of the constraint vertex neighbors (as shown in Fig. 3). Ghost vertices allow the computation of 1-D Laplacians at constraint vertices. Appendix A provides the pseudocode for incorporating ghost vertices in an arbitrary-order mesh-based Laplacian system. Moreover, this method of completing the Laplacian in a mesh-based system is topologically convenient (no explicit geometry is added to the mesh) yet very powerful (we can control position and tangent behavior along every point of the constraint curve).

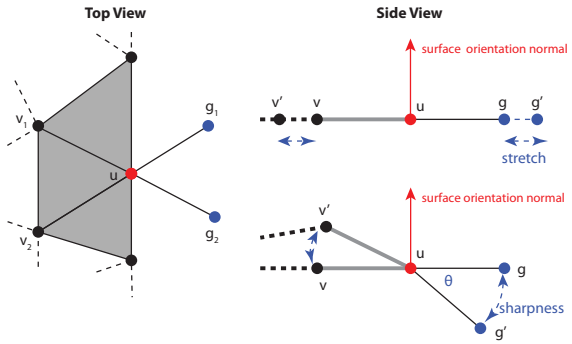


Figure 3: *The ghost vertex interpretation of the micro-Laplacian of* $\mathbf{u}$ *from the perspective of neighbors* $\mathbf{v_1}$ *or* $\mathbf{v_2}$ *is shown (left): vertices* $\mathbf{v_1}$ *and* $\mathbf{v_2}$ *are mirrored across* $\mathbf{u}$ *to form* $\mathbf{g_1}$ *and* $\mathbf{g_2}$ *respectively. The parameters used to influence the position of internal vertex* $\mathbf{v}$ *by moving the ghost vertex* $\mathbf{g}$ *connected to a constraint vertex* $\mathbf{u}$ *is shown (right). The three ghost vertex parameters are the orientation (or average surface normal at* $\mathbf{u}$*), the sharpness, and the stretch. Moving the ghost vertex* $\mathbf{g}$ *to a new position* $\mathbf{g}'$ *moves vertex* $\mathbf{v}$ *to position* $\mathbf{v}'$ *after the linear solve.*

4.3. The Micro-Laplacian and its Conceptual Basis

We can insert ghost geometry to specify surface behavior at boundary vertices as well as at interior constraint vertices. For the sake of understanding the theoretical underpinnings of the ghost geometry, we introduce the concept of a "micro-Laplacian." The micro-Laplacian demonstrates that the notion of local consistency actually corresponds to adding more detail to the constraints than the mesh resolution would normally permit. We remind the reader that the exposition in this sub-section explains why the ghost vertices work, and understanding this explanation is not necessary for implementing this paper.

The micro-Laplacian is best understood by considering the case of an isolated constraint vertex $\mathbf{u}$, shown in Fig. 4. We interpret such a constraint as representing a tiny hole in the mesh — a hole much smaller than the mesh resolution could represent — with each side of the hole seeing a different piece of geometry. To represent this, for each vertex $\mathbf{v_i}$ adjacent to $\mathbf{u}$, we replace $\mathbf{u}$ with virtual constraint vertex $\mathbf{u_i}$ which is only visible to $\mathbf{v_i}$. Vertex $\mathbf{v_i}$ then creates a ghost vertex $\mathbf{g_i}$ reflected across $\mathbf{u_i}$. Each $\mathbf{u_i}$ has the same position as $\mathbf{u}$, but has its own Laplacian. Note that $\mathbf{u_i}$ is not visible to $\mathbf{u_{i-1}}$ or $\mathbf{u_{i+1}}$ or vice-versa; similarly, $\mathbf{g_i}$ is connected only to $\mathbf{u_i}$, but not to $\mathbf{u_{i+1}}$ or $\mathbf{u_{i-1}}$. This means that the mesh connectivity is locally consistent, but is not globally consistent. This mental model allows us the freedom to control the surface orientation separately at each side of the constraint vertex, allowing us, for example, to place a sharp point at the constraint vertex. Note that the Laplacian at constraint vertex $\mathbf{u}$

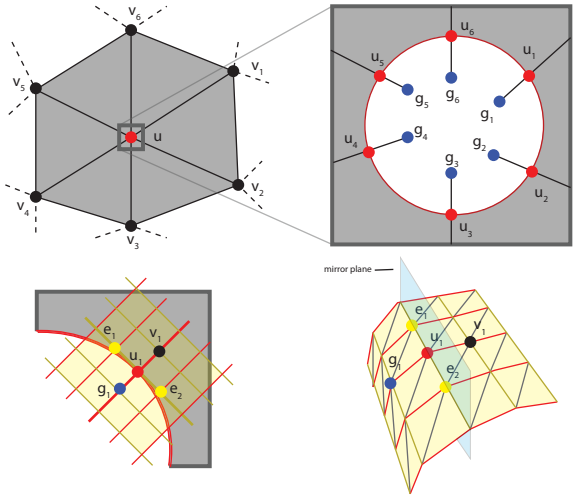


Figure 4: *An isolated single-vertex constraint interpreted as representing a tiny hole in the mesh (top). We define a "micro-Laplacian" which allows us to simulate these small-scale details by defining a separate Laplacian at constraint* $\mathbf{u}$ *for each incoming vertex* $\mathbf{v_i}$*. We can visualize the micro-scale geometry (bottom) as a hinge bending across the local tangent direction of the constraint edge tessellated as shown, and extending as far as needed to define our free vertex order-k Laplacians.*

is different based on which neighbor $\mathbf{v_i}$ is being considered. (The stability of our system with these inconsistent Laplacians has withstood thorough testing without any issues.)

To define the Laplacian of this virtual geometry, we invent a simple, explicit model for the shape of the virtual geometry. We define the edge of the "hole" to be a straight line in our desired local tangent direction, and the surrounding geometry to form a cylindrical patch or hinge. As shown in the bottom row of Fig. 4, the micro-Laplacian operates on this cylindrical shape, i.e., on the positions of the internal vertex $\mathbf{v_1}$, corresponding constraint vertex $\mathbf{u_1}$, ghost vertex $\mathbf{g_1}$ and orthogonal edge endpoints $\mathbf{e_1}$ and $\mathbf{e_2}$. The cylindrical shape bends along the local tangent direction of the virtual constraint edge, and is flat in the orthogonal direction. Because the Laplacian is a measure of curvature, and there is by our definition no curvature except along the bending direction, we need only to consider the edges in the direction of bending to compute the Laplacian. In Appendix B we show how this holds in practice even when we explicitly define a tessellation for the micro-scale geometry: all terms of the Laplacian of $\mathbf{u_i}$ may be dropped except for the positions of $\mathbf{u_i}$, $\mathbf{v_i}$, and a ghost vertex $\mathbf{g_i}$ (which is defined initially by reflection of $\mathbf{v_i}$ across $\mathbf{u_i}$, shown in Fig. 3). By symmetry the two remaining terms must have equal weight. Therefore, for the Laplacian of constraint vertices in Eqn. 1, we use the ghost vertices to get:

$$\Delta^k(\mathbf{u_i}) = 2w\Delta^{k-1}(\mathbf{u_i}) - w\Delta^{k-1}(\mathbf{v_i}) - w\Delta^{k-1}(\mathbf{g_i}) \qquad (2)$$

Because the scale of our virtual geometry is arbitrary, we let $w = 1$ in Eqn. 2. Because $\mathbf{g_i}$ is positioned such that the Laplacian is initially zero, this will only affect how much $\mathbf{g_i}$ must move as stretch is introduced in Section 4.4 (we formulate the linear system weights with the assumption that the initial stretch is zero).

Although we have explained the micro-Laplacian in the context of an isolated constraint vertex, the concept applies wherever a sudden change or discontinuity in the Laplacian is desired. In practice, it also works well when no discontinuity is required. In fact, for simplicity we use the micro-Laplacian for *all* Cauchy constraints in our examples. Note that some care must be taken to treat micro-Laplacians consistently when discontinuities are not desired: for example, they should be rotated about a consistent axis (as described in Section 4.4).

4.4. Parameterizing Cauchy Constraints for the Artist

Controlling the surface near constraints by directly modifying the ghost vertex positions is not intuitive to artists and designers. Instead, we encapsulate first-order Cauchy constraints with the parameters illustrated in Fig. 3: orientation, sharpness, and stretch, which are the parameters that the artist sees and manipulates. Orientation defines the average surface normal at a constraint. Sharpness defines the angle between that average surface normal and the local surface normal on each side of the constraint – for example, a 0 degree sharpness parameter creates a flat surface, while a 45 degree sharpness parameter creates a right-angled crease at an internal constraint curve, or a right-angled cone at an isolated constraint vertex. Stretch is constrained to always stretch the surface away from the constraint, and has an effect equivalent to increasing the magnitude of the derivative for a Hermite spline – the surface maintains its local tangent direction farther away from the constraint. Orientation can be controlled by any standard rotation UI (such as an arc ball), while sharpness and stretch are simple 1-dimensional parameters which can be specified by sliders.

To define a ghost vertex $\mathbf{g_i}$ (at constraint vertex $\mathbf{u}$ from vertex $\mathbf{v_i}$) using these parameters, we must additionally define some axis of rotation $\mathbf{r_i}$ about which the sharpness parameter will crease the surface. For a point constraint, we choose $\mathbf{n} \times (\mathbf{v_i} - \mathbf{u})$ as the axis, where $\mathbf{n}$ is the initial average surface normal at $\mathbf{u}$. For a curve constraint, we choose the tangent direction $\mathbf{d}$ of the curve at that vertex $\mathbf{u}$, or $-\mathbf{d}$ if $(\mathbf{n} \times (\mathbf{v_i} - \mathbf{u})) \cdot \mathbf{d} < 0$ (to rotate in opposite directions on opposite sides of the crease). Our parameters then define a rotation matrix $\mathbf{O}$ for orientation, a crease angle θ for sharpness, and a stretch parameter s. If we define a rotation matrix

$\mathbf{R_i}$ for sharpness as a rotation by θ about axis $\mathbf{r_i}$, then we arrive at the formula:

$$\mathbf{g_i} = \mathbf{u} + \mathbf{OR_i}\left(1 + \frac{s}{||\mathbf{u} - \mathbf{v_i}||}\right)(\mathbf{u} - \mathbf{v_i}) \qquad (3)$$

Note that we scale the stretch parameter by the inverse of the length of the edge, so the amount of stretch introduced is independent of mesh resolution.

Typically, these parameters are specified for a whole curve at once, or smoothly defined along regions of curves. This higher-level view also allows our constraints to be independent of mesh resolution. When new vertices are introduced along a constraint curve, for example by refining the mesh, we simply interpolate the orientation, sharpness and stretch parameters.

5. Generating New 3D Models from Curves

We now describe our first application of the Laplacian framework: a system for modeling new geometry from curves. We demonstrate the capability of our system to generate high-quality results in practice.

Drawing a 2D sketch of a smooth object can often be easier than building that object with a 3D modeling tool. With that in mind, we designed our workflows for easily converting an existing sketch into a 3D model. We focused on sketches of objects that were mostly smooth, but had some sharp features. The input to the system is a vector art file that specifies paths (usually Bézier curves) that make up the sketch. We require that all non-closed paths in a given 2D sketch be bounded by a closed path. Our system samples the input curves and converts them into piece-wise linear connected segments, i.e. polylines; we then convert these line segments into a high-quality triangulation using the software package "Triangle" [She96].

This triangulated, flat piece of geometry becomes a live patch that may be inflated into 3D, edited, and also joined with other such patches to create a complete, optionally-watertight model. By using the same approximations for each boundary curve, we ensure a consistent triangulation across boundaries of patches that may share a polyline and avoid troublesome T-junctions. G^1 continuity may be enforced across patch boundaries by automatically keeping the orientation parameter consistent for boundary vertices that are shared by two or more patches. (In general for order-k Laplacian systems, G^{k-1} continuity may be enforced; we used the bi-Laplacian ($k = 2$) system for all examples in this section.) Within each patch, input curves may be marked as Cauchy constraints, smooth position constraints (with fixed position and Laplacian computed using the local geometry), or inactive constraints, explained below.

5.1. Inactive Constraints

One challenge in surface modeling from a 2D sketch is to convert existing 2D curves into 3D space curves. Our solution is to simply deactivate some interior constraint curves. An "inactive" constraint is essentially a passive curve that stays on the surface, gets modified along with the surface, but does not affect the surface. By changing parameters stored at the active constraints, we can modify the surface and turn the inactive constraints from flat 2D curves into 3D space curves. The user can activate the inactive constraints at any time when their 3D shape meets the user's expectations. See Fig. 5 for an example. Because drawing smooth curves in 3D space is more difficult than drawing smooth curves in 2D, we believe that harnessing the patch inflation mechanism is an easy way to produce 3D constraints.

5.2. Making Patches from Constraints

The user may add additional constraints to the existing, inflated patch. Eventually, a single patch may not be sufficient to construct the desired shape. At that stage, the user may choose to decompose the single patch into multiple, disjoint, and possibly abutting patches. Our preferred approach for doing so is to incrementally break apart the patch into separate patches. The user selects a constraint curve and asks the system to make a separate patch with the constraint curve as the boundary. If the constraint curve is closed, it is turned into a hole in the original patch. If the constraint curve is not closed, the original patch is unchanged and the system closes the open constraint curve by connecting its endpoints with a line segment. The 3D position and orientation required for every point along the boundary of the new patch are taken from the original patch surface. The new patches can be edited independently from the original patch and further decomposed into more patches. See Fig. 6 for an example of decomposing a patch into smaller ones for the purpose of adding more detail, and Fig. 7 for an example of patch decomposition to create several free floating patches.

Decomposing a single patch into multiple patches has several design advantages. Most importantly, the user gets local control over the shape being modeled: adding or manipulating the constraints on one patch will not affect other patches. There are also computational advantages: the locality of patch edits means that only the affected patch needs to be updated. If multiple patches are affected, separate independent threads can be trivially dispatched to carry out the computation.

5.3. Preliminary User Feedback

A professional designer used our prototype implementation to build the examples shown in Figs. 5, 6 and 7. The shape in Fig. 7 took about 30 minutes to create. The pig in Fig. 5(C) took about 10 minutes to create, though the artist spent an additional three hours refining that result to arrive at Fig. 5(E). Overall, the designer was very pleased by the ease with which he was able to model organic shapes as well as shapes with sharp edges. However, our user interface

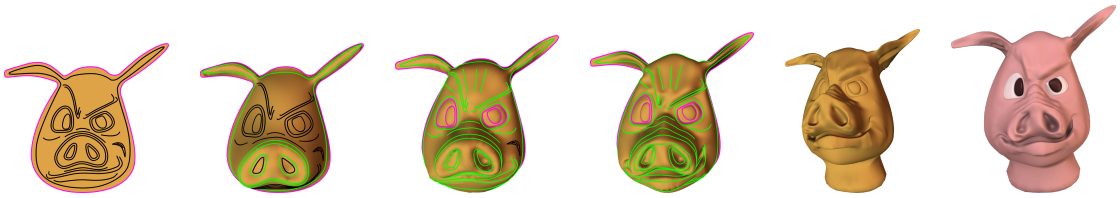


Figure 5: *The designer starts with a sketch and deactivates all internal constraints (shown in black) (A). By activating and moving some constraints, the designer obtains the desired 3D positions of the inactive constraints (B). Cauchy constraints (pink) are used to define the eyes, and additional smooth position constraint lines (green) are added on the 3D surface for more control (C). The final states of the constraints (D) gives the surface (E) that is then shaded and rendered (F).*

Figure 6: *The designer makes a patch out of the 2D sketch (left), and by editing parameters (position and sharpness) at internal constraints makes a 3D model (middle). In order to add more detail, selected internal faces (along the side panels) are made into separate patches and edited by drawing more Cauchy constraint curves (right).*

was designed primarily for testing our system — we believe that a better user interface for deforming the patch constraint curves could substantially reduce the design time. A subset of our system has also been implemented as a part of the Repoussé feature in Adobe® Photoshop® CS5 Extended.

6. Reverse-Engineering Existing Meshes

In this section, we explore the potential of our system to reverse engineer existing shapes into our shape representation. Instead of purely focusing on editing the existing geometry, we envision reverse engineering as the basis for a "mesh import" module that would allow the model to be edited seamlessly, as if it had been created with our tool set originally. Therefore, instead of following the traditional mesh editing approach of "baking" the existing mesh Laplacians in our representation, we attempt to encode all geometric detail at constraint curves, using our Cauchy constraints. We show that a surface representation comprised of our Cauchy constraints is capable of expressing a large variety of shapes. We noticed that the first- and second-order Laplacian solves do not always produce cylindrical surfaces properly, instead

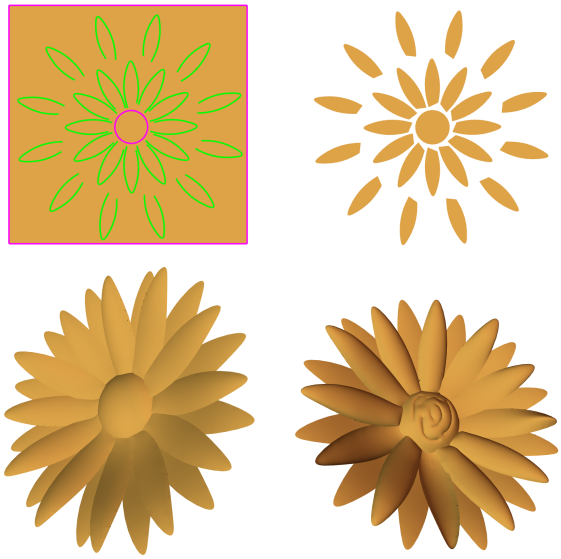


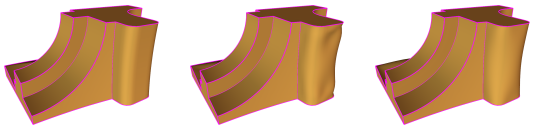
Figure 7: *The designer produces the different sketch components (top left), each of which was made into a new patch (top right). By deforming the patch boundaries, the different patches come together to form a flower (bottom left). Further edits can be directly added on the 3D surface of each patch (bottom right).*

tending to collapse into an hour-glass shape; fortunately, using a tri-Laplacian (or higher) solve (with the corresponding Cauchy constraints) tends to mitigate this issue (e.g., Fig. 9).

We re-use the existing mesh structure while converting it into our patch representation. Given a set of important feature lines or points to mark as constraints, we set the Cauchy constraint vertices appropriately to match the surface. We use two heuristics to identify candidate feature lines: a dihedral angle threshold (from [GSMCO09]) for mechanical or CAD-style parts, and ridge/valley lines (from [OBS04]) for organic meshes. For the Cauchy constraints, we find it often

100 sec.

Figure 8: *Left is the input smooth shape with feature curves extracted by ridges and valleys. Middle is the result obtained by reducing feature curves (with computation time listed). Right is an example of a simple edit to the bunny shape.*



300 sec. 8 sec.

Figure 9: *Left is the input shape with feature curves extracted by analyzing dihedral angles. Middle shows a tri-Laplacian fit with a complete setup, giving a "wobbly" shape. Right shows a tri-Laplacian fit with a reduced setup, giving the best result and performance. The times needed for fitting are listed below each result.*

suffices to use a naive guess – simply set the constraint orientation to match the surface normal at the constraint vertex. This works well when the constraints are relatively dense, and control of shape behavior away from constraint lines is not as important as matching the shape near constraint lines. This core process is extremely fast, requiring only a simple pass over the mesh to identify features. We can optionally refine our core reverse engineering process in two ways: reducing the number of feature curves, and improving the fit of the patches to the input mesh by optimizing the ghost vertex positions.

6.1. Reducing Internal Feature Curves

When fewer feature curves are desired, we use a greedy algorithm to select the important curves. We add one curve at a time wherever the point-to-point error (defined below) is highest. We stop when the maximum point-to-point error at any vertex is smaller than some threshold (0.1 in our examples, assuming the input mesh is scaled to just fit in a 5x5x5 box). To bootstrap this greedy approach, we require an initial surface to evaluate the point-to-point error. For patches with a boundary, we use the boundary curves to construct an initial surface. In the case of closed models, we select the longest internal feature curve as a starting point to get an initial surface. This process, inspired by [SCO04], is demonstrated for the bunny shape in Fig 8. As with the greedy process described in [SCO04], this process is slow because it requires we solve the system once per every feature curve we select. (If speed of reverse engineering is important, the "combined local maxima" optimization suggested in [SCO04] should apply here as well.)

6.2. Optimizing the Cauchy Constraints

As in all our examples, we use the micro-Laplacian formulation for our Cauchy constraints. Optimization of the Cauchy constraints is equivalent to optimization of the ghost vertex positions. To accelerate the optimization process, we precompute a linear expression for each vertex $\mathbf{v_i}$ in terms of each ghost vertex $\mathbf{g_j}$ and a constant vector $\mathbf{k_i}$. This linear expression takes the form:

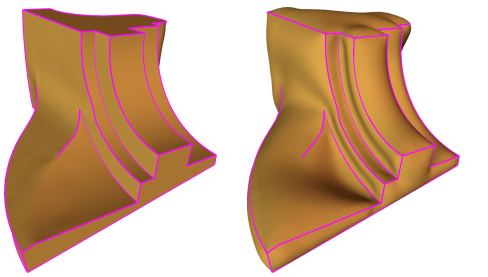


Figure 10: *Left is the editable fandisk shape obtained after fitting – see Fig. 9(middle). We change parameters at constraint curves to create an inflated version of the fandisk(right). Note that such control using a sparse set of feature curves would be very difficult to obtain with existing methods.*

$$\mathbf{v_i} = \mathbf{k_i} + \Sigma_j w_{ij} \mathbf{g_j} \tag{4}$$

In this equation, our ghost vertex positions can be thought of as new dimensions in a *ghost vertex basis*, where our vertex positions are now expressed in that basis as $(3+n)$-dimensional vectors $(k_{i1}, k_{i2}, k_{i3}, w_{i1}, ..., w_{in})$, rather than 3-vectors in our previous standard 3D basis. To compute the $\mathbf{k_i}$

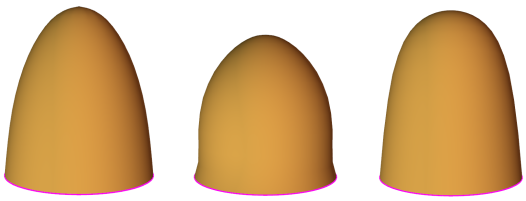


Figure 11: *Left is the target shape we wish to reconstruct. Middle shows the shape preferred by the point-to-point error metric. Left shows the shape preferred by the point-to-plane error metric. In both cases, we used a tri-Laplacian solve for our patches.*

and w_{ij} values, we solve the same Laplacian $\mathbf{Ax} = \mathbf{b}$ system from Eqn. 1, transformed into the new ghost vertex basis. First, we express the right hand side $\mathbf{b}$ in that basis:

$$\mathbf{b_i} = \mathbf{l_i} + \Sigma_j m_{ij} \mathbf{g_j} \quad (5)$$

The weights m_{ij} are the contribution of each ghost vertex to a given right hand side vector $\mathbf{b_i}$. The constant vector $\mathbf{l_i}$ is the total weighted contribution of fixed vertex locations to the same $\mathbf{b_i}$. We can solve for basis weights w_{ij} and the constant terms $\mathbf{k_i}$ by solving $\mathbf{Aw} = \mathbf{m}$ and $\mathbf{Ak} = \mathbf{l}$ (where $\mathbf{w}$, $\mathbf{m}$, $\mathbf{k}$ and $\mathbf{l}$ are the matrices of all w_{ij}, y_{ij}, $\mathbf{k_i}$ and $\mathbf{l_i}$ weights respectively). This requires one linear solve per ghost vertex ($\mathbf{Aw} = \mathbf{m}$) and three additional solves (one for X, Y, and Z) for the constant vectors $\mathbf{k_i}$ ($\mathbf{Ak} = \mathbf{l}$). With all $\mathbf{k_i}$ and w_{ij} computed, we are now free to modify our ghost vertex locations and update the mesh using Eqn. 4, without needing to re-solve the linear system. We note that this general approach has been used by others to accelerate mesh processing [BK04, JMD*07].

To optimize our mesh, we define an error metric for each free vertex. Using this metric and Eqn. 4, we determine the ghost vertex positions by solving for the linear least squares solution that minimizes this error. We have experimented with two error metrics: first, a "point to point" metric that measures the distance between the position returned by our patch solver and the given vertex position. Second, a "point to plane" metric that measures the distance between the position returned by our patch solver and the plane defined by the given vertex with its normal. Neither choice is perfect: the point-to-point metric will not distort texture, but tends to "wobble" around the correct surface (Fig. 9(middle)), while the point-to-plane metric fits some surfaces better (Fig. 11) but is slower and often underconstrained.

In the method described so far, each 3D coordinate of each ghost vertex is treated as a separate variable in the fitting process. We call this the "complete" setup. Instead, for each closed constraint curve, we could assign the *same* sharpness and stretch parameters to all the ghost vertices. If we set sharpness and stretch to move each ghost vertex in a local 2D frame, this permits a linear expression for each vertex in terms of just two free parameters. For higher-order ghost vertices, we use different parameters for each order of ghost vertex. We call this setup the "reduced" setup. We notice that reducing the degrees of freedom in this fashion can dramatically improve both the speed of the system and the quality of the results, as it can forbid "wobbly" solutions that our error metric alone does not sufficiently penalize. See Fig. 9 for an example of fitting using the complete and reduced setups. Fig. 10 and Fig. 8 show examples of fitting with sharp and smooth edges, respectively.

7. Drawbacks and Future Work

Our future work primarily consists of exploring better user interfaces for specifying Cauchy constraints; in particular, we need a method for intuitively specifying second- and higher-order Cauchy constraints. Along with a better user interface, our future work will address some of the drawbacks of our current implementation. We need to improve the feature curve extraction for smooth shapes — currently our implementation requires that feature curve edges be aligned with the mesh edges, which can produce crooked edges for coarse meshes. We also need to invest more time evaluating the best error metric for the fitting process — our point-to-point and point-to-plane metrics were simple to implement and worked well enough for our examples, but we expect that more sophisticated mesh error metrics will produce better fitting results.

A related limitation arises from the quality of tessellation. To improve the accuracy of the solution, we need adaptive tessellation, which is a well-studied problem in the finite element community. Mesh tessellation also impacts rendering quality and the interpolation of normals for badly shaped triangles can lead to shading artifacts. Such artifacts can be reduced by performing anisotropic remeshing (again, a well-studied problem in the meshing community).

The addition or removal of any constraint (or changing the type of a constraint) requires a re-triangulation of the domain and re-initialization and factorization of the linear system. Although we cannot avoid this setup cost, we have noticed that the setup time is small enough to not interfere with an interactive user experience. A large majority of our patches contain about 6000 free vertices for which the time for setup (triangulation, matrix building, and matrix factorization) is about 0.5 sec. and is fully interactive for the back-solve (on a laptop with an Intel 2.4GHz Core2 Duo CPU with 2GB RAM). Although our system is fast enough for the purposes of the examples in this paper, we have room for performance improvement (with optimized code) to match the numbers reported by [BBK05].

8. Acknowledgements

Thanks to Daichi Ito for being our first test user, and creating the artwork shown in Figs. 5, 6 and 7. Thanks to Pete Falco for suggesting work in Section 6. This work was supported in part by the National Science Foundation (NSF award #CMMI-1029662 (EDI)).

References

[BBK05] BOTSCH M., BOMMES D., KOBBELT L.: Efficient linear system solvers for mesh processing. In *Mathematics of Surfaces XI*. Springer Berlin / Heidelberg, 2005, pp. 62–83. 9

[BBS08] BAE S.-H., BALAKRISHNAN R., SINGH K.: Ilovesketch: as-natural-as-possible sketching system for creating 3d curve models. In *ACM UIST* (2008). 2

[BK04] BOTSCH M., KOBBELT L.: An intuitive framework for real-time freeform modeling. *ACM Siggraph* (2004). 2, 3, 9, 10

[BLZ00] BIERMANN H., LEVIN A., ZORIN D.: Piecewise smooth subdivision surfaces with normal control. In *ACM Siggraph* (2000). 2

[BW90] BLOOR M. I. G., WILSON M. J.: Using partial differential equations to generate free-form surfaces. *Comput. Aided Des. 22*, 4 (1990), 202–212. 2

[DEG*99] DEMMEL J. W., EISENSTAT S. C., GILBERT J. R., LI X. S., LIU J. W. H.: A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications 20*, 3 (1999). 3

[EP09] EIGENSATZ M., PAULY M.: Positional, metric, and curvature control for constraint-based surface deformation. *Comput. Graph. Forum 28*, 2 (2009). 2

[FAMO99] FEDKIW R. P., ASLAM T., MERRIMAN B., OSHER S.: A non-oscillatory eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *J. Comput. Phys. 152* (July 1999), 457–492. 4

[GPR09] GAO K., PARK H., ROCKWOOD A.: Feature based styling. whitepaper, 2009. 2

[Gre83] GREGORY J.: n-sided surface patches. *Mathematics of Surfaces* (1983), 217–232. 2

[GSMCO09] GAL R., SORKINE O., MITRA N. J., COHEN-OR D.: iwires: an analyze-and-edit approach to shape manipulation. In *ACM SIGGRAPH* (2009). 7

[GZ09] GINGOLD Y., ZORIN D.: Shading-based surface editing. *ACM Siggraph* (2009). 2, 10

[HKS92] HSU L., KUSNER R., SULLIVAN J.: Minimizing the squared mean curvature integral for surfaces in space forms. *Experimental Mathematics 1* (1992), 191–207. 3

[IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: A sketching interface for 3d freeform design. In *ACM SIGGRAPH* (1999). 2

[JC08] JOSHI P., CARR N.: Repoussé: Automatic inflation of 2d artwork. In *Eurographics Symposium on Sketch-Based Interfaces and Modeling* (2008). 2

[JMD*07] JOSHI P., MEYER M., DEROSE T., GREEN B., SANOCKI T.: Harmonic coordinates for character articulation. *ACM Siggraph* (2007). 9

[KH06] KARPENKO O. A., HUGHES J. F.: Smoothsketch: 3d free-form shapes from complex sketches. *ACM Siggraph* (2006). 2

[KS08] KARA L., SHIMADA K.: Supporting early styling design of automobiles using sketch-based 3d shape construction. In *Computer-Aided Design and Applications* (2008), CAD Solutions LLC. 2

[Lev99] LEVIN A.: Interpolating nets of curves by smooth subdivision surfaces. In *ACM SIGGRAPH* (New York, NY, USA, 1999). 2

[Loo92] LOOP C.: *Generalized B-spline Surfaces of Arbitrary Topological Type*. PhD thesis, University of Washington, 1992. 2

[NISA07] NEALEN A., IGARASHI T., SORKINE O., ALEXA M.: Fibermesh: designing freeform surfaces with 3d curves. *ACM Siggraph* (2007). 2, 4

[NSACO05] NEALEN A., SORKINE O., ALEXA M., COHEN-OR D.: A sketch-based interface for detail-preserving mesh editing. In *ACM SIGGRAPH* (2005). 2

[OBS04] OHTAKE Y., BELYAEV A., SEIDEL H.-P.: Ridge-valley lines on meshes via implicit surface fitting. In *ACM SIGGRAPH* (2004), pp. 609–612. 7

[OS10] OLSEN L., SAMAVATI F.: Image-assisted modeling from sketches. In *Proceedings of the Graphics Interface* (2010). 2

[PP93] PINKALL U., POLTHIER K.: Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics 2*, 1 (1993), 15–36. 3

[SCO04] SORKINE O., COHEN-OR D.: Least-squares meshes. In *Proceedings of the Shape Modeling International* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 191–199. 8

[She96] SHEWCHUK J. R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996, pp. 203–222. 6

[SK01] SCHNEIDER R., KOBBELT L.: Geometric fairing of irregular meshes for free-form surface design. *Computer Aided Geometric Design 18*, 4 (2001), 359–379. 2, 4, 10

[SWSJ05] SCHMIDT R., WYVILL B., SOUSA M., JORGE J.: Shapeshop: Sketch-based solid modeling with blobtrees, 2005. 2

[WBH*07] WARDETZKY M., BERGOU M., HARMON D., ZORIN D., GRINSPUN E.: Discrete Quadratic Curvature Energies. *Computer Aided Geometric Design 24*, 8-9 (Nov 2007), 499–518. 2

[XZ07] XU G., ZHANG Q.: G2 surface modeling using minimal mean-curvature-variation flow. *Comput. Aided Des. 39*, 5 (2007), 342–351. 2

Appendix A: Notes on the Laplacian Framework

Internal vs External Cauchy Constraints

There are several possible ways to constrain surface derivatives at mesh vertices. We follow the method (à la [SK01]) of fixing positions of fictional, "external" vertices beyond the mesh boundary on which the constraint vertex lies. A common alternative is to fix the positions of "internal" vertices adjacent to a constraint vertex, variants of which appear in [BK04] and [GZ09]. At relatively coarse mesh resolutions (which are common in practice) the internal constraint solutions satisfy the gradient constraint exactly at the expense of surface smoothness. The surface tends to have an undesirable, jagged crease near internal fixed vertices — this problem is exacerbated for higher-order Laplacians, where more internal vertices are constrained. These properties are demonstrated in Fig. 12, showing a simple 1-manifold example which allows us to compare the results of both discretizations to an exact solution.

Pseudocode

To construct the Laplacian matrix, we use a recursive function to build each row of the matrix individually as done in algorithm 1. The recursive function "buildEqn" constructs an equation that expresses the position of a free vertex as a function of its neighbors — see bottom half of algorithm 2 for a standard Laplacian and top half of algorithm 2 for our modified version with Cauchy constraints.

The first few arguments to both functions are: matrix "M" and the index of row "r" corresponding to the free vertex, vertex "v" whose Laplacian is being computed, the order

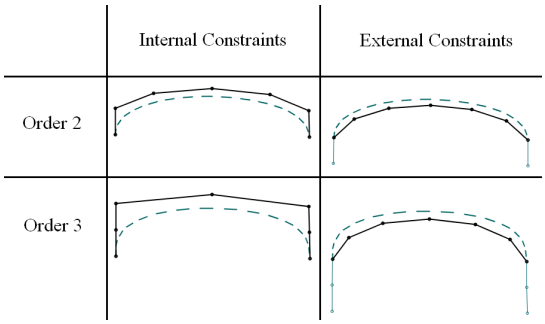


Figure 12: *A 1-manifold example comparing internal and external constraint vertices for a coarse mesh. The dotted line shows the true Hermite curve solution. For fine meshes both methods converge to the true solution.*

"o" of that Laplacian, and the weight "wt" with which that Laplacian will be considered. We have two additional arguments to the buildEqn function of algorithm 2: the "sV" variable to remember the source vertex we came from, and the "gNum" variable to track which ghost vertex we are considering: gNum 0 is the constraint vertex and gNum 1 through (laplacianOrder-1) indicates the ghost vertices in order, extending out from the constraint. The function "getEdgeWeight(v,n)" returns the cotangent weight of the one-ring neighbor n of vertex v.

The right-hand-side of the equation is built by the "setRHS" function in algorithm 2. If gNum is zero, setRHS(r, v, gNum, sV, -wt) adds the position of vertex v with weight -wt, else the position of the indicated ghost vertex with weight -wt.

Algorithm 1 Build Laplacian Matrix

```
M = zeros(numUnconstrainedVertices)
for r = 0; r < numUnconstrainedVertices; r++ do
    buildEqn(M, r, getVertex(r), laplacianOrder, 1)
end for
```

Algorithm 2 Build Laplacian Equation with Constraints

```
function buildEqn(Matrix M, int r, vertex v, int o, float wt, vertex sV=null, int gNum=0)
  if v.isConstraint() then
    if o == 0 then
      setRHS(r, v, gNum, sV, -wt)
    else
      buildEqn(M, r, v, o-1, 2*wt, sV, gNum)
      buildEqn(M, row, v, o-1, -wt, sV, gNum+1)
      if gNum == 0 then
        buildEqn(M, row, sV, o-1, -wt, v)
      else
        buildEqn(M, row, v, o-1, -wt, sV, gNum-1)
      end if
    end if
  else
    if o == 0 then
      M[row, v.column()] += wt
    else
      wt *= 1.0/v.area()
      for all neighbor n of vertex v do
        buildEqn(M, v, o-1, wt*getEdgeWeight(v, n))
        buildEqn(M, n, o-1, -wt*getEdgeWeight(v, n), v)
      end for
    end if
  end if
end function
```

Appendix B: Micro-Laplacian on an Explicit Tesselation

Though it is intuitively clear that the non-bending terms should drop from the micro-scale Laplacian, as discussed in Section 4.3, it is not immediately obvious that this occurs with the discrete Laplacian operator we have chosen for our optimization. An arbitrary discretization may introduce errors that prevent the other edges from cleanly cancelling. Fortunately, we are free to define the exact tessellation of our micro-scale geometry, and we can show that for a simple uniform tessellation (Fig. 4(bottom row)), the non-bending terms do indeed drop from the equation. Our assumed shape bends along the local tangent direction of the virtual constraint edge and is flat in the orthogonal direction. We choose a regular and consistent tessellation for this hinge geometry, with the additional constraint that vertices $\mathbf{e_1}$, $\mathbf{u_1}$ and $\mathbf{e_2}$ are collinear.

To compute the Laplacian at $\mathbf{u_1}$, we separately consider the contribution of three different sets of edges to the Laplacian Eqn. 1. Refer to Fig. 4. We call the edges that extend in parallel lines (i.e., the edges connecting $\mathbf{e_1}$ to $\mathbf{u_1}$, and $\mathbf{u_1}$ to $\mathbf{e_2}$ as shown in red in Fig. 4) the "transverse" edges. We call the edges connecting $\mathbf{v_1}$ to $\mathbf{u_1}$ and $\mathbf{u_1}$ to $\mathbf{g_1}$ the "constraint" edges. Finally, we refer to remaining edges as the "diagonal" edges. We will show that for computing the Laplacian at $\mathbf{u_1}$, we only need to consider the contributions of the constraint edges.

The transverse edges contribute nothing to the first-order ($k = 0$) Laplacian, as the two transverse edges at a vertex are in the opposite direction and of equal length. They contribute nothing to the higher-order Laplacians either, because the order $k > 0$ Laplacian values are identical along the edges (the bending and tessellation being the same), so the difference of any order Laplacians across the edges is zero. Therefore, the contributions of the transverse edges may be ignored completely for the Laplacian at $\mathbf{u_1}$.

By construction, the constraint edges are perpendicular to the transverse edges in the base domain, as shown in Fig. 4. Contributions of the diagonal edges to the Laplacian at $\mathbf{u_1}$ are therefore zero due to the cotangent weights being zero for right angles. Therefore, the only edges we need to consider while computing the Laplacian at $\mathbf{u_1}$ are the two constraint edges.