Interactive and Anisotropic Geometry Processing Using the Screened Poisson Equation

Ming Chuang*
Johns Hopkins University

Michael Kazhdan†
Johns Hopkins University

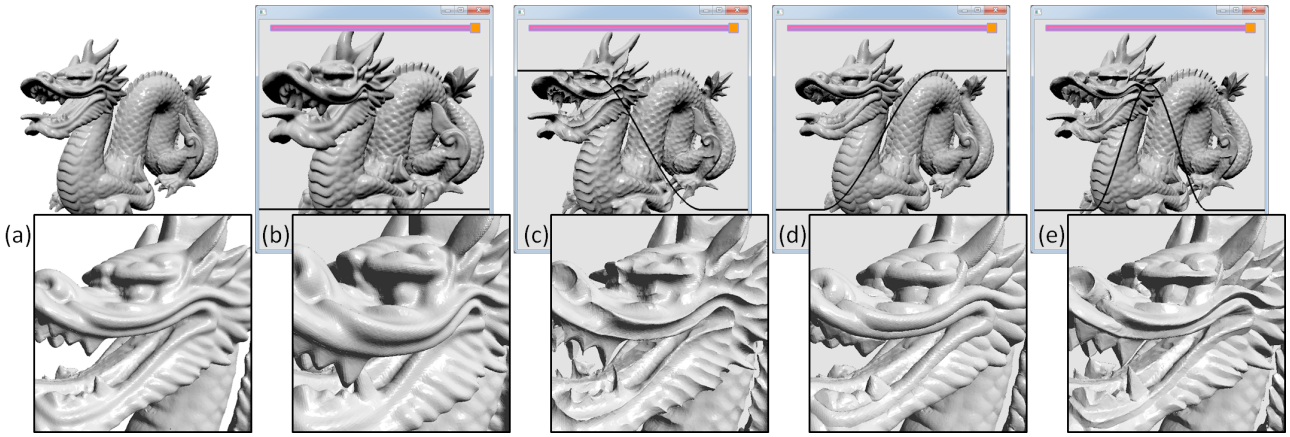


Figure 1: *Anisotropic detail sharpening: Starting with an initial model (a), global sharpening is applied to the geometry to enhance the detail (b). By adapting the direction of sharpening to the curvature in different ways, a rich space of geometry-aware sharpening filters are realized (c-e). Though the model consists of almost one million vertices and a new system is constructed and solved each time the filter is changed, our method still supports geometry processing at interactive rates.*

Abstract

We present a general framework for performing geometry filtering through the solution of a screened Poisson equation. We show that this framework can be efficiently adapted to a changing Riemannian metric to support curvature-aware filtering and describe a parallel and streaming multigrid implementation for solving the system. We demonstrate the practicality of our approach by developing an interactive system for mesh editing that allows for exploration of a large family of curvature-guided, anisotropic filters.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations

Keywords: Laplace-Beltrami, multigrid, real-time, surface editing

Links: DL PDF

*e-mail: ming@cs.jhu.edu
†e-mail: misha@cs.jhu.edu

1 Introduction

With the increased proliferation of 3D scanners, the ability to perform geometry-aware filtering has become an important aspect of geometry processing. This has included operations such as edge-aware smoothing, for removing the unwanted effects of scanner noise, and sharpening, for exaggerating geometric detail.

This type of processing is made hard by the fact that the specific filter is often not known in advance and an essential step in editing the geometry is determining the type of filter that should be used. Figure 1 shows an example in which the detail in the dragon (a) is enhanced using different sharpening filters (b-e). Although all the edits accentuate the detail, the specific effects vary with the filter profile and the desired editing effects are only realized through interactive exploration of the filter space.

Previous work has shown that the filtering of mesh geometry can be expressed in terms of the solution to a Poisson equation [Pinkall and Polthier 1993; Taubin 1995; Desbrun et al. 1999] and that geometry-awareness can be incorporated by anisotropically weighting the Laplace operator [Clarenz et al. 2000; Tasdizen et al. 2002].

However, using these methods in practice has proven challenging because applying a filter requires defining and solving a large sparse linear system – limiting these approaches either to small meshes or to non-interactive settings.

Contribution We address this challenge by proposing a real-time system for anisotropic filtering of geometric detail. The specific contributions of our approach are three-fold:

- We extend the screened Poisson formulation of gradient-domain image processing described by Bhat *et al.* [2008; 2010] to meshes, providing a general framework that supports localized editing using anisotropic filters.

- We describe the implementation of the first multigrid solver capable of relaxing mesh-based Poisson systems comprised of over $3\times400{,}000$ degrees of freedom, at a rate of 20 fps.

- We show that the integration required for defining the finite-elements system can be expressed independent of the choice of Riemannian metric, allowing us to not only solve the system, but also adapt it to a user-prescribed metric interactively.

Outline The remainder of the paper is organized as follow: We start with a brief survey of related work in Section 2. After that, we describe the continuous formulation of our system in Section 3 and its discretization in Section 4. We present the interactive construction and solution of the linear system in Section 5 and discuss our interface for exploring the space of filters in Section 6. We evaluate our approach in Section 7 and conclude with a summary and discussion of future directions in Section 8.

2 Related Work

The Laplace operator arises in numerous mesh processing applications. This section briefly reviews some of the more common applications in mesh editing and discusses methods for solving the associated linear system.

2.1 Mesh Editing

Using the analogy between the eigenvectors of the Laplacian and the Fourier basis, Taubin [1995] describes a frequency-space approach in which fairing is performed by low-pass filtering: First, the geometry of the mesh is represented as a function over the surface. Then, the function is decomposed in terms of the eigenvectors of the Laplacian. And finally, the coefficients are multiplied by a transfer function which decays with frequency (eigenvalue). Though a direct approach was infeasible, Taubin shows that an approximation to the smoothing can be implemented by leveraging results from scale-space theory [Witkin 1983] that relate the linearization of Gaussian smoothing to the Laplacian.

This approach is extended by Guskov *et al*. [1999] who use a progressive mesh structure in conjunction with an up-sampling operator to define the analog of a Laplacian pyramid [Burt and Adelson 1983] over a mesh. By modulating the levels of the pyramid in different ways, the authors simulate convolution with a broad class of transfer functions. Though this approach supports a broader class of shape edits, it is only approximates convolution in that the lower levels of the hierarchy do not explicitly correspond to subspaces spanned by the lower-frequency eigenvectors of the Laplacian. It is only with the recent work of Vallet and Lévy [2008], who use the shift-invert spectral transform in a pre-processing phase to localize the solution of the eigenvalue problem, that a direct frequency-space filtering approach has become possible.

In parallel with the signal processing interpretation, Kobbelt *et al*. [1998] formulate mesh fairing as the problem of energy minimization. In this context, the approach of Taubin can be interpreted as a Jacobi step of a solver aimed at minimizing the energy and larger-scale smoothing is performed by using a multigrid solver defined over a hierarchy of meshes. This approach is further developed in [Desbrun et al. 1999] which views Laplacian smoothing as a time-integration of the heat equation. In this context, the earlier work on mesh fairing can be interpreted as an explicit integration and Desbrun *et al*. propose a more efficient and more stable approach using semi-implicit integration.

From Homogeneity to Anisotropy While methods such as [Taubin 1995; Kobbelt et al. 1998; Desbrun et al. 1999; Guskov et al. 1999; Ohtake et al. 2000] perform mesh fairing by solving a homogenous system of equations, there has also been significant research in the use of anisotropic diffusion for performing feature-aware geometry processing [Clarenz et al. 2000; Meyer et al. 2002; Bajaj et al. 2002; Tasdizen et al. 2002]. The idea behind these approaches is to use the curvature information to define an inner-product on the tangent space that slows the diffusion along directions that cross feature lines.

2.2 Poisson Solvers

For many of the above methods, the editing of the geometry requires the solution of a Poisson-like equation. While general "black-box" algebraic multigrid solvers have been used, there have also been techniques focused on developing multiresolution hierarchies that support multigrid, including methods based on mesh simplification [Kobbelt et al. 1998; Aksoylu et al. 2003], graph coarsening [Shi et al. 2006], and octrees [Chuang et al. 2009].

3 Continuous Formulation

We begin by reviewing the screened Poisson equation and discussing its applications in geometry processing. Then, we describe how the method can be extended to account for a general Riemannian metric. (We refer the reader to the work of Bhat *et al*. [2008] for an elegant description of the screened Poisson equation in the context of image processing.)

3.1 The Screened Poisson Equation

Consider the linear system that arises when trying to interpolate the values of a function defined on a mesh M while simultaneously amplifying (resp. dampening) the gradients in order to achieve a sharpening (resp. smoothing) effect. Given an initial function F, the best-fit function G is obtained by finding the minimizer of the sum of square norms:

$$E(G) = \alpha^2 \left\| G - F \right\|^2 + \left\| \nabla_M G - \beta \nabla_M F \right\|^2$$

where α is a weight that trades off between the desire for value fidelity and gradient modulation, β is a gradient scaling term, and ∇_M is the gradient operator, taken with respect to the differential structure on M. Applying the Euler-Lagrange equation, the minimizer is the solution to the screened Poisson equation[1]:

$$\left(\alpha^2 \cdot \mathrm{id} - \Delta_M\right) G = \left(\alpha^2 \cdot \mathrm{id} - \nabla_M \cdot (\beta \nabla_M)\right) F \qquad (1)$$

where ∇_M and $\nabla_M \cdot$ are the gradient and divergence operators (with respect to M) and $\Delta_M = \nabla_M \cdot \nabla_M$ is the Laplace-Beltrami operator on M (the analog of the Laplacian on a manifold).

Applications to Geometry Processing Treating the original embedding of the surface as a function on M, we can edit the *geometry* using the screened Poisson equation. In particular, if we set $F : M \to \mathbb{R}^3$ to be the original coordinate function, $F(p) = \mathbf{x}^\circ(p) = p$, then the solution $G = \mathbf{x} : M \to \mathbb{R}^3$ becomes the function assigning

[1] Note that this expression of the linear system in terms of the screened Poisson equation assumes that the mesh does not have a boundary. In practice, we do not require such an assumption because we define the system by integrating the dot-products of gradients, rather than the product of functions and Laplacians, so our formulation defines the correct minimizer even for surfaces with boundary.

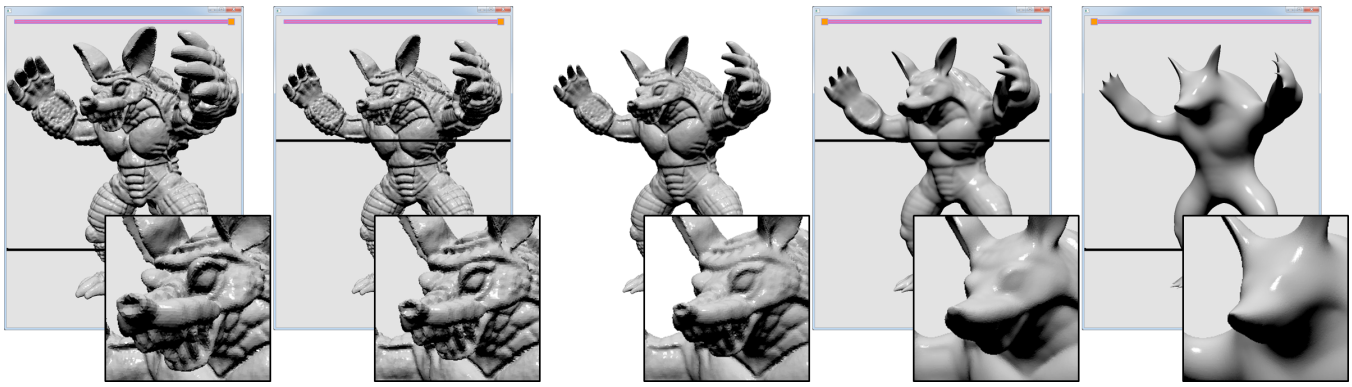


Figure 2: *Homogenous detail sharpening and smoothing: Examples of geometric effects obtained by solving a screened Poisson equation. The surfaces, from left to right, are obtained with: low fidelity and gradient amplification ($\alpha = 1$, $\beta = 2$), high fidelity and gradient amplification ($\alpha = 5$, $\beta = 2$), no gradient scaling (original model, $\beta = 1$), high fidelity and gradient dampening ($\alpha = 5$, $\beta = 0$), low fidelity and gradient dampening ($\alpha = 1$, $\beta = 0$).*

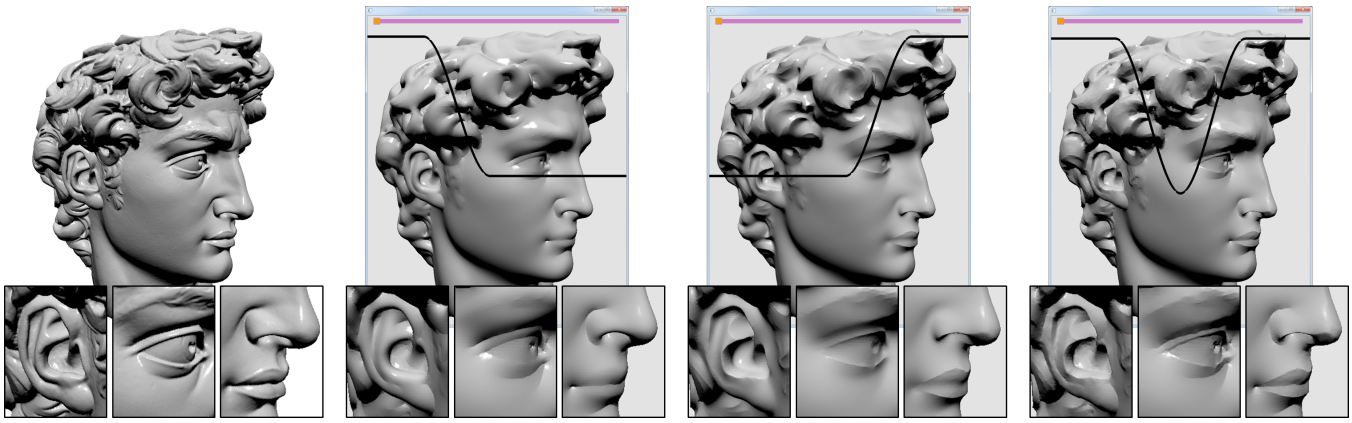


Figure 3: *Anisotropic detail smoothing: Examples of geometric effects obtained by adapting the Riemannian metric to the curvature. Starting with the original model (left), global smoothing constraints were applied. The surfaces, from left to right, are obtained by amplifying the fidelity term in directions of: large negative curvature, large positive curvature, large absolute curvature.*

new positions to the vertices of the mesh, with the parameters α and β dictating the fidelity and detail modulation of the new geometry.

Figure 2 shows examples achieved with this approach, when β is constant. The center image shows the original geometry, $\beta = 1$, the images to the left show the results with gradient amplification, $\beta > 1$, and the images to the right show gradient dampening, $\beta < 1$. For this visualization, models further from the center correspond to solutions with more weakly weighted interpolation constraints.

Relationship to Mesh Fairing An advantage of the screened Poisson formulation is that the use of a general scaling function supports a rich family of possible edits. For example, setting $\beta = 0$ in Equation 1, we get:

$$\left(\mathrm{id} - \frac{1}{\alpha^2} \cdot \Delta_M\right) \mathbf{x} = \mathbf{x}^\circ$$

which describes a semi-implicit step of mean-curvature flow with time-step $1/\alpha^2$. Thus, the mean-curvature fairing described by Desbrun *et al.* [1999] can be viewed as a specific instance of solving a screened Poisson equation for mesh filtering.

More generally, β can be defined to be any spatially varying function, providing a way to prescribe that the mesh should smoothed in some regions and sharpened in others. (See Figure 5.)

3.2 General Riemannian Metrics

Borrowing from Eckstein *et al.* [2007], we generalize the class of surface edits characterized by the screened Poisson equation by allowing the Riemannian metric to change. Specifically, if $\mathbf{A}$ is a spatially varying inner-product on the tangent space of M, we set:

$$E_{\mathbf{A}}(\mathbf{x}) = \left\|\mathbf{x} - \mathbf{x}^\circ\right\|_{\mathbf{A}}^2 + \left\|\nabla_M \mathbf{x} - \beta \nabla_M \mathbf{x}^\circ\right\|_{\mathbf{A}}^2.$$

In this case, the function minimizing the energy is the solution to the (anisotropic) screened Poisson equation:

$$\left(\sqrt{|\mathbf{A}|} - \nabla_M \cdot \sqrt{|\mathbf{A}|}\mathbf{A}^{-1}\nabla_M\right)\mathbf{x} = \left(\sqrt{|\mathbf{A}|} - \nabla_M \cdot \sqrt{|\mathbf{A}|}\mathbf{A}^{-1}\beta\nabla_M\right)\mathbf{x}^\circ. \tag{2}$$

Here, $|\mathbf{A}(p)|$ is the determinant of $\mathbf{A}$, corresponding to the (squared) area distortion at p and $\mathbf{A}^{-1}$ maps the (dual of the) gradient with respect to the metric defined by the embedding into the (dual of the) gradient defined with respect to the metric defined by $\mathbf{A}$.

Note that setting $\mathbf{A} = \alpha^2 \cdot \mathrm{id}$ gives the solution to the isotropic screened Poisson equation with weight α^2. In the context of mesh fairing ($\beta = 0$), this is a restatement of the fact that the speed of mean-curvature flow is inversely proportional to mesh size.

Figure 3 shows examples achieved with this approach, when β is set everywhere to zero (to smooth the model) and $\mathbf{A}$ is spatially vary-

ing. The image to the left shows the original geometry. To its right, we see results obtained by setting $\mathbf{A}$ to be large in directions of large negative curvature (center left), large positive curvature (center right), and large absolute curvature (right).

As the figures indicate, specifying that $\mathbf{A}$ should be large in a particular direction down-plays the gradient constraints fitting in favor of the interpolation constraint. So, for example, when $\mathbf{A}$ is large in directions of large negative curvature (center left and right), sharp concave creases are preserved. Similarly, setting $\mathbf{A}$ to be large in directions of large positive curvature (center right and right), sharp convex creases are preserved.

Spectral Analysis When α and β are constant, the spectral analysis of Bhat *et al.* carries over directly to the context of a mesh. If f_λ is an eigenvector of the Laplace-Beltrami operator with eigenvalue $-\lambda^2$, then solving the screened-Poisson equation is equivalent to multiplying the f_λ component of the input signal by:

$$H_{\alpha,\beta}(\lambda) = \frac{\alpha^2+\beta\lambda^2}{\alpha^2+\lambda^2} = \frac{1+\beta(\lambda/\alpha)^2}{1+(\lambda/\alpha)^2}.$$

This transfer function satisfies $H_{\alpha,\beta}(0)=1$ and $H_{\alpha,\beta}(\infty)=\beta$, implying that it preserves the lower frequencies and either dampens or amplifies higher frequencies, depending on whether β is less than or greater than one. The efficiency with which the transfer function converges to β in the frequency domain is determined by the fidelity value α, with smaller values resulting in more pronounced dampening or amplification at lower frequencies.

The transfer function can be also interpreted to be independent of the choice of fidelity weight. To this end, we recall that scaling the Riemannian metric by α defines a Laplace-Beltrami operator with the same eigenvectors but with modified eigenvalues, $\lambda \to \lambda/\alpha$. Thus, the property that $H_{\alpha,\beta}(\lambda) = H_{1,\beta}(\lambda/\alpha)$ can be interpreted to mean that in scaling the metric we still use the original ($\alpha=1$) transfer function, but now evaluated on the modified spectrum.

This generalizes to the metric defined by any (positive-definite) $\mathbf{A}$. That is, for constant β the transfer function remains:

$$H_\beta(\lambda) = \frac{1+\beta\lambda^2}{1+\lambda^2}$$

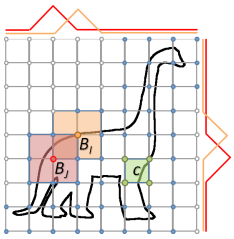
but now defined with respect to the spectrum of the Laplace-Beltrami operator defined by $\mathbf{A}$.

4 Discrete Formulation

We now describe the way in which a finite-elements approach can be used to discretize the continuous system. We briefly review the finite-elements of Chuang *et al.* [2009] on which we base our solution. Then, we describe how to construct the linear system and show that the computationally expensive integration required to define the finite-elements system can be expressed independent of the choice of metric and gradient scale.

4.1 Choosing a Discretization

To define a function space, Chuang *et al.* embed the mesh M in a regular 3D grid. Then, first-order B-spline are associated with the corners of the grid cells, and test functions $\{B_1(p),\ldots,B_n(p)\}$: $M \to \mathbb{R}$ are defined by restricting the B-splines to the surface of the mesh. Since only those B-splines whose support overlaps the geometry need to be considered in defining the linear system, the coefficients of the system can be indexed by the corners of an octree that is adapted to the surface.



The inset shows a visualization of the test-functions for a 2D curve. Since they are first-order, B-splines centered at a vertex are supported within the grid cells adjacent to that vertex (orange and red regions). Equivalently, for any grid cell that intersects the curve (green region), there are exactly four B-splines whose support intersect that cell (the B-splines centered on the cell's corners). As a result, many B-splines are supported off of the surface (white dots) and do not contribute to the definition of the system.

4.2 Discretizing the System

Representing the Embeddings The system is discretized by only considering solutions spanned by the test functions:

$$\mathbf{x}(p) = \sum_i \vec{\mathbf{c}}_i B_i(p), \qquad \vec{\mathbf{c}}_i \in \mathbb{R}^3.$$

Note that since the first-order B-splines satisfy the linear reproduction property, the initial embedding $\mathbf{x}^\circ(p)$ lies within the span of the test functions. In particular, setting I to be the index of the vertex centered at (i,j,k) and $\vec{\mathbf{c}}^\circ_I = (i,j,k)$, we have:

$$p = \mathbf{x}^\circ(p) = \sum_I \vec{\mathbf{c}}^\circ_I B_I(p).$$

For simplicity, we will continue to index the test-functions as B_I, with I denoting both the index of the test-function within the system and the 3D coordinates of the corner on which B_I is centered.

Representing the Homogeneous System We begin by considering the homogeneous system in Equation 1, assuming that both α and β are constant. This system is discretized by taking the dot-product of both sides with each of the test-functions, giving:

$$\left\langle (\alpha^2 \cdot \mathrm{id} - \nabla_M \cdot \nabla_M)\mathbf{x}, B_I \right\rangle = \left\langle (\alpha^2 \cdot \mathrm{id} - \nabla_M \cdot \beta\nabla_M)\mathbf{x}^\circ, B_I \right\rangle$$

for all test-function indices I. Since $\vec{\mathbf{c}}^\circ$ and $\vec{\mathbf{c}}$ are the coefficients of the original and new embeddings, this gives the linear system:

$$S^\alpha \vec{\mathbf{c}} = \vec{\mathbf{b}}^{\alpha,\beta}$$

where $\vec{\mathbf{b}}^{\alpha,\beta}$ is the constraint vector and S^α is the $n \times n$ system matrix, obtained by iterating over the grid cells c and integrating:

$$\vec{\mathbf{b}}^{\alpha,\beta}_I = \sum_J \left(\sum_c \int_{c\cap M} \alpha^2 B_I \cdot B_J + \beta \langle \nabla_M B_I, \nabla_M B_J \rangle dp \right) \vec{\mathbf{c}}^\circ_J$$

$$S^\alpha_{IJ} = \sum_c \int_{c\cap M} \alpha^2 B_I \cdot B_J + \langle \nabla_M B_I, \nabla_M B_J \rangle dp. \qquad (3)$$

Here, $\nabla_M B_I(p)$ is the gradient of the I-th B-spline with respect to the embedding of M, obtained by projecting the gradient of the 3D B-spline at p onto the tangent plane of M at p.

Representing the Anisotropic System The anisotropic system in Equation 2 corresponds to a choice of spatially varying, symmetric, positive-definite, bilinear form $\mathbf{A}$. Although we do not support all inner-products, we show how to support the curvature-driven forms commonly used for geometry-aware filtering.

Following the approach of Clarenz *et al.* [2000], we restrict ourselves to the subset of forms that diagonalize with respect to the

principal curvature directions. Setting $\{v_1(p), v_2(p)\}$ to the principal curvature directions at the point p and $\{\kappa_1(p), \kappa_2(p)\}$ to the associated principal curvature values, we have:

$$\mathbf{A}(p) = \alpha_1^2(p) \cdot (v_1(p) \otimes v_1(p)) + \alpha_2^2(p) \cdot (v_2(p) \otimes v_2(p))$$

where $v \otimes v$ is the matrix obtained by taking the outer-product of v with itself and $\alpha_i(p)$ is a positive function which depends only on the position p and the principal curvature κ_i:

$$\alpha_i(p) = \alpha(p, \kappa_i(p)) > 0.$$

(Though principal curvature directions are not uniquely defined at umbilics, the fact that α only depends on the position and the principal curvature values implies that the $\mathbf{A}(p)$ is still well-defined at these points.)

Incorporating the metric $\mathbf{A}$ and the scale term β into the linear system amounts to modulating the integral computed over each grid cell by the values of α_1, α_2, and β within the cell:

$$\begin{aligned}
\vec{\mathbf{b}}_I^{\mathbf{A},\beta} &= \sum_J \sum_c \Bigg(\int_{c \cap M} \alpha_1 \cdot \alpha_2 \cdot B_I \cdot B_J dp \\
&\quad + \int_{c \cap M} \beta \frac{\alpha_1 \cdot \alpha_2}{\alpha_1^2} \langle \nabla_M B_I, v_1 \rangle \langle \nabla_M B_J, v_1 \rangle dp \\
&\quad + \int_{c \cap M} \beta \frac{\alpha_1 \cdot \alpha_2}{\alpha_2^2} \langle \nabla_M B_I, v_2 \rangle \langle \nabla_M B_J, v_2 \rangle dp \Bigg) \vec{\mathbf{c}}_J^{\circ} \\
S_{IJ}^{\mathbf{A}} &= \sum_c \Bigg(\int_{c \cap M} \alpha_1 \cdot \alpha_2 \cdot B_I \cdot B_J dp \\
&\quad + \int_{c \cap M} \frac{\alpha_1 \cdot \alpha_2}{\alpha_1^2} \langle \nabla_M B_I, v_1 \rangle \langle \nabla_M B_J, v_1 \rangle dp \\
&\quad + \int_{c \cap M} \frac{\alpha_1 \cdot \alpha_2}{\alpha_2^2} \langle \nabla_M B_I, v_2 \rangle \langle \nabla_M B_J, v_2 \rangle dp \Bigg). \qquad (4)
\end{aligned}$$

Here, the term $\alpha_1 \cdot \alpha_2$ accounts for the area scale and the terms $1/\alpha_1^2$ and $1/\alpha_2^2$ account for the fact that scaling a function by α scales its second derivative by $1/\alpha^2$.

4.3 Metric and Gradient Scale Independent Integration

One of the difficulties of using the formulation in Equation 4 is that the system coefficients are expressed in terms of integrals of the test functions, weighted by the coefficients of the metric α_1 and α_2, and the scaling function β. Thus, it would seem that modifying either the metric or the gradient scales would require costly computation, prohibiting the interactive construction of the linear system.

We show that by making a simple assumption on the structure of these functions, the computationally expensive integration can be made independent of both the metric and gradient scaling.

Examining Equation 4, we observe that while the definition of the system requires that the B_I be weakly differentiable, the incorporation of the functions α_1, α_2, and β only require that they be integrable over a grid cell c. Thus, while we use first-order B-splines to represent the B_I, we can represent α_1, α_2, and β by functions that are piecewise constant on the grid cells.

This has the desirable property of neither raising the degree of the polynomial integrated over the triangles nor making it rational, so that simple quadrature rules can be used to compute the integrals.

Furthermore, the piecewise-constant representation of these functions means that we can pull their contribution outside the integrals, allowing us to compute the integrals off-line:

$$\rho_{IJ}(c) = \int_{c \cap M} B_I \cdot B_J dp \qquad \ell_{IJ}^i(c) = \int_{c \cap M} \langle v_i, \nabla_M B_I \rangle \langle v_i, \nabla_M B_J \rangle dp$$

and recombine them efficiently at run-time.

5 Interactive Surface Editing

We now describe our implementation of an interactive system for mesh editing. Starting with an input mesh and given prescribed editing constraints by the user, our system solves for the function $\mathbf{x}$ which, evaluated at the original vertices, gives the positions of the vertices on the edited surface. To do this, we must set up the linear system (Equation 3), solve the screened Poisson equation, and evaluate the positions of the new vertices.

We decompose the implementation of the system into two phases.

5.1 Preprocessing

Off-line, we read in the mesh, set the coefficients $\vec{\mathbf{c}}^{\circ}$ of the initial coordinate function $\vec{\mathbf{x}}^{\circ}(p)$, construct the vertex evaluation matrix E, and compute the integrals needed for setting the system coefficients.

Setting $\vec{\mathbf{c}}^{\circ}$

As described in the Section 4, we can set the coefficients of the original embedding function by using the linear-reproduction property of first-order B-splines, setting the value of coefficient $\vec{\mathbf{c}}_I^{\circ}$, centered at corner $I = (i, j, k)$, to $\vec{\mathbf{c}}_I^{\circ} = (i, j, k)$.

Constructing the Evaluation Matrix

To compute the positions of the vertices on the edited mesh, we will need to evaluate the new coordinate function:

$$\mathbf{x}(p) = \sum_I \vec{\mathbf{c}}_I B_I(p)$$

at each of the original vertices $\{\vec{\mathbf{v}}_1^{\circ}, \ldots, \vec{\mathbf{v}}_m^{\circ}\} \subset M$. Since evaluation is linear in the function coefficients, we can express this as a matrix multiplication. Specifically, letting $\vec{\mathbf{v}}$ denote the $m \times 3$-dimensional vector of new vertex positions, we have:

$$\vec{\mathbf{v}} = E\vec{\mathbf{c}}$$

where E is the $m \times n$ evaluation matrix, defined by $E_{jI} = B_I(\vec{\mathbf{v}}_j^{\circ})$.

Since the matrix E only depends on the test functions and the position of the original vertices, we only need to compute it once in the preprocessing phase.

Pre-Computing Integrals

As discussed in Section 4.3, the representation of the functions α_1, α_2 and β by piecewise constant functions makes it possible to compute the integrals independent of the choice of metric or gradient scale. In the preprocessing, we compute the integrals $\rho_{IJ}(c)$, $\ell_{IJ}^1(c)$, and $\ell_{IJ}^2(c)$, splitting the triangles of the mesh to the faces of the grid cells and using sixth-order quadrature rules [Cowper 1973] to integrate the polynomial functions over the (sub)-triangles.

5.2 Run-Time

In the on-line step, we construct the new system matrix and constraint vector, down-sample the matrices to obtain a multigrid hierarchy, solve for the coefficients of the new coordinate function, and evaluate the new vertex positions. Although we only update the system when the values of the functions α_1, α_2, and β are changed, we follow the real-time system for image editing proposed by McCann *et al*. [2008] and relax the linear system at every frame. Since we initialize the solution with the coefficients from the previous frame, when the metric and gradient scales are not being changed, the solution to the screened Poisson equation is continuously being refined and the residual norm decreases with each frame.

Constructing the System

Having pre-computed the integrals in the preprocessing stage, we take the linear combination of these values, weighted by the values of α_1, α_2, and β on the different grid cells to get the coefficients of the system matrix and constraint vector. In particular, if $\mathscr{C}(I)$ are the eight grid cells in the support of test function B_I, and $\mathscr{I}(c)$ are the indices of the eight test functions supported on c, we set:

$$\vec{\mathbf{b}}_I^{\mathbf{A},\beta} = \sum_{\substack{c\in\mathscr{C}(I)\\ J\in\mathscr{I}(c)}} \alpha_1(c)\alpha_2(c)\left(\rho_{IJ}(c)+\beta(c)\left(\frac{\ell^1_{IJ}(c)}{\alpha_1^2(c)}+\frac{\ell^2_{IJ}(c)}{\alpha_2^2(c)}\right)\right)\vec{\mathbf{c}}^{\circ}_J$$

$$S^{\mathbf{A}}_{IJ} = \sum_{\substack{c\in\mathscr{C}(I)\\ J\in\mathscr{I}(c)}} \alpha_1(c)\alpha_2(c)\left(\rho_{IJ}(c)+\frac{\ell^1_{IJ}(c)}{\alpha_1^2(c)}+\frac{\ell^2_{IJ}(c)}{\alpha_2^2(c)}\right).$$

Though it may seem that this computation incurs a heavy cost (both in terms of time and in terms of space) this turns out not to be the case. Using the fact that the test-functions are derived from first-order B-splines, it follows that the support of B_I overlaps with the support of (at most) 27 other test functions – the test functions centered at corners $J = I+(\delta_1,\delta_2,\delta_3)$ with $\delta_i\in\{-1,0,1\}$. Of these, the support of one, $|\delta_1|+|\delta_2|+|\delta_3|=0$, will overlap with the support of B_I on all eight cells. The support of six, $|\delta_1|+|\delta_2|+|\delta_3|=1$, will overlap on four cells. The support of twelve, $|\delta_1|+|\delta_2|+|\delta_3|=2$, will overlap on two cells. And, the support of eight, $|\delta_1|+|\delta_2|+|\delta_3|=3$, will overlap on one cell.

Thus, on average, a non-zero matrix coefficient $S^{\mathbf{A}}_{IJ}$, is expressed as the linear combination of only $64/27\approx 2.4$ values in ρ, ℓ^1, and ℓ^2.

Down-sampling the Matrices

To support a multigrid solver, we also need to compute the lower resolution system matrices when the values α_1 and α_2 are changed.

Although the Galerkin conditions can be directly satisfied by using the prolongation operator P_l^{l+1} to express the matrix at level l in terms of the matrix at level $l+1$:

$$S^{\mathbf{A}(l)} = \left(P_l^{l+1}\right)^T S^{\mathbf{A}(l+1)}\left(P_l^{l+1}\right),$$

we have found that it is more efficient to first compute the linear combination of the integrals at the finest resolution:

$$\iota^{\mathbf{A}}_{IJ}(c) = \alpha_1(c)\alpha_2(c)\left(\rho_{IJ}(c)+\frac{\ell^1_{IJ}(c)}{\alpha_1^2(c)}+\frac{\ell^2_{IJ}(c)}{\alpha_2^2(c)}\right).$$

Then, using the fact that the integrals at level l can be expressed as linear combinations of integrals at level $l+1$, we iteratively down-sample the integrals to the coarser levels and combine the down-sampled integrals to get the corresponding system matrices. (This is more efficient because the integral tables have a regular structure, with each cell contained in the support of exactly eight test-functions, so down-sampling can be easily performed in parallel, using SSE vectorization, and without fine-level branching.)

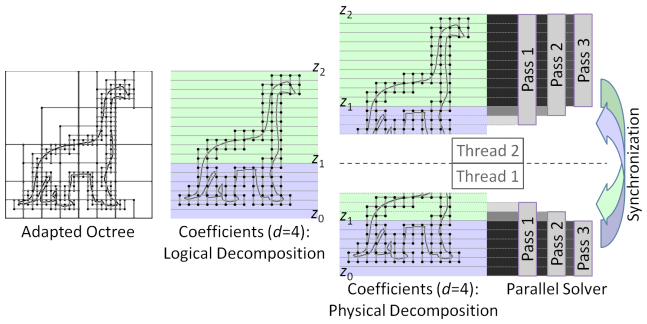


Figure 4: *Parallelization of Gauss-Seidel Relaxation: By decomposing the solution coefficients into overlapping blocks and shrinking the vertical extent of relaxed coefficients on subsequent updates, threads can perform multiple updates in parallel, without having to synchronize coefficient values between passes.*

Solving the System

To support an interactive solver, we exploit multi-threading to solve the screened Poisson equation. Using multigrid, we perform four basic operations in solving the system: (1) We relax the linear system using Gauss-Seidel iterations. (2) We compute the residual, corresponding to the constraints that have not been satisfied by relaxation. (3) We restrict the residual to obtain the constraints at the next coarser resolution. And, after solving at the coarser resolution, (4) we prolong the coarser solution to obtain the correction term for the current resolution.

While the last three steps only require a matrix multiplication, which is readily parallelizable, relaxation requires more attention.

The key to our implementation lies in the observation that although the mesh we are processing may be unstructured, our test functions inherit the regularity of the 3D lattice on which the B-splines are centered. This allows us to leverage parallelization and temporal blocking techniques designed for regular grids.

Parallelization To implement multiple Gauss-Seidel relaxations in parallel, we use an approach inspired by domain decomposition [Smith et al. 1996] and the "safe-zones" of Weber *et al*. [2008]. The idea is to redundantly decompose the data into overlapping blocks, allowing each processor to perform multiple relaxations on the solution within its block, without requiring synchronization between relaxations.

An example visualization is shown in Figure 4, for two processors performing three Gauss-Seidel relaxations in parallel. The quadtree defining the coefficients is shown on the left. The coefficients at depth four, visualized as the corners of the quadtree cells, are shown next. These are grouped into sparsely populated vertical "slices" and the two logical partitions associated with the threads are highlighted in different colors. The physical decomposition is shown next, indicating how the coefficients are laid out in memory, with each thread maintaining a copy of the coefficients in its partition *plus* the coefficients from the three slices across its boundary.

Using this decomposition, the two threads perform three Gauss-Seidel updates in parallel. In the first pass, the coefficients in a thread's partition, plus those in the two slices beyond the boundary

are updated. In the next pass, coefficients in the thread's partition plus those in the slice immediately beyond the boundary are updated. And, in the last update, only the coefficients in the thread's partition are updated. Finally, the coefficients are synchronized by copying the last three slices of the first thread's partition into the overlapping slices in second thread and vice-versa.

Temporal Blocking In addition to providing an ordering that supports parallelization, the grouping of coefficients into slices also supports cache-friendly memory access through temporal blocking [Pfeifer 1963; Douglas et al. 2000; Kazhdan and Hoppe 2008].

As an example, we describe the case when two Gauss-Seidel relaxations are performed. In the direct implementation, after relaxing the coefficients in slice i, a thread would proceed to the relaxation of coefficients in slice $i+1$. As a result, when returning to update the coefficients in slice i a second time, all the required data is likely to have been evicted from the cache, requiring an expensive memory load before the update could proceed. In a temporally blocked solver the order of updates is modified, so that after updating coefficients in slice i for the first time, the thread returns to slice $i-1$ and updates its coefficients a second time. Only after performing the second update of coefficients in slice $i-1$ does it proceed to the update of coefficients in slice $i+1$.

Since the two updates of the coefficients in a slice are now temporally proximal, data loaded into the cache for the first update is likely to still be resident when the second update begins, allowing us to avoid having to re-load memory into cache redundantly.

Vertex Evaluation

After obtaining the coefficients $\vec{\mathbf{c}}$ of the new coordinate function, we evaluate the product $\vec{\mathbf{v}} = E\vec{\mathbf{c}}$ to set the positions of the vertices on the edited mesh. When the number of vertices is noticeably larger than the number of coefficients, $m \gg n$, this evaluation can become a bottleneck for two reasons: First, though the evaluation can be expressed as a matrix-vector multiply, the matrix is now of size $m \times n$ so evaluating the positions can become as expensive as solving the system. Second, even after evaluating the positions, we must still transfer the new vertices to the GPU, which can be prohibitive when the number of vertices is large.

We address this by performing the vertex evaluation on the GPU. After computing the evaluation matrix E in the preprocessing step, we transfer it to the GPU where it resides for the remainder of the editing session. Then, at run-time, after computing the coefficient vector $\vec{\mathbf{c}}$ on the CPU, we stream the coefficients to the GPU, where a CUDA kernel is invoked to evaluate the new vertex positions and release them to OpenGL via a shared vertex buffer object.

6 Specifying Editing Constraints

To support the interactive editing of surface geometry, our system provides interfaces for specifying the gradient scale and Riemannian metric.

6.1 Scaling the Gradients

To allow the user to specify how the gradients should be scaled, we provide two interfaces.

The first is a simple slider used to set a constant gradient scale β for the entire surface. This is shown at the tops of Figures 1-3, where dragging the slider to the left results in global smoothing and dragging it to the right results in global sharpening.

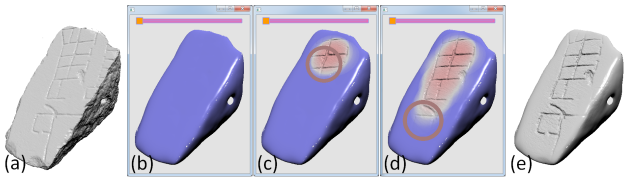


Figure 5: *Selective detail enhancement: Starting with the original model (a), a user applies global smoothing by specifying that all gradients should be dampened (b). The user then specifies that the top face of the tablet should be sharpened by selectively amplifying gradients in that area (c-d). The final results accentuates the floor plan and hides detail in the fracture region (e).*

The second is an airbrush interface for specifying the local gradient scale β. The implementation was designed to mimic the airbrush in typical image processing software. The user selects the radius of the circular brush and the gradient scale encoded as a color in the range from blue (smoothing) to red (sharpening). Our system interprets the spraying of gradient constraints by finding the ball in 3D whose silhouette corresponds to the outline of the brush and whose center is the surface point under the mouse.

Since β is represented as a function that is constant within each cell of the grid, we update its values by identifying the cells in the vicinity of the brush center. We then blend the brush's value with the values already encoded in these cell, using a blending function that decays as a function of distance from the brush center. We define the brush center by identifying the point under the mouse and finding its corresponding position in the original mesh.

Figure 5 show an example of using these local edits. After specifying global smoothing constraints (b), the user sharpens the detail on the face of the fragment by selecting a brush with gradient amplification (red) and spraying across the surface (c-d). This draws the viewer's attention to the floor plan by removing fracture details and selectively enhancing the engravings (e).

6.2 Adjusting the Metric

To allow the user to specify how the constraints should be weighted, we provide a simple profile-curve interface, motivated by the frequency transfer functions used by Guskov *et al.* [1999] and Vallet and Lévy [2008]. Dragging points on the curve, the user specifies a "transfer" function $\alpha(\kappa)$ giving the scaling of the inner product in principal curvature directions with principal curvature value κ:

$$\alpha_i(p) = \alpha(\kappa_i(p)).$$

Since α_1 and α_2 are piecewise-constant function, the implementation of this interface requires associating a minimum and maximum curvature value with each grid cell. In our implementation, we set these to the (area-weighted) average of the minimum and maximum curvatures of the triangles contained within the cell. Curvatures are computed using Trimesh [2009].

Figures 1-3 show snapshots from several editing sessions using our system. In these visualizations, the black curve shows the user-defined transfer function, with the horizontal axis corresponding to the curvature value κ and the vertical axis corresponding to the fidelity weight $\alpha(\kappa)$. Note that, in contrast to the transfer functions typically used for convolution, our function only specifies the importance of the gradient scaling constraint along the principal curvature directions. The actual type of gradient scaling (e.g. smoothing vs. sharpening) is dictated by the values of β.

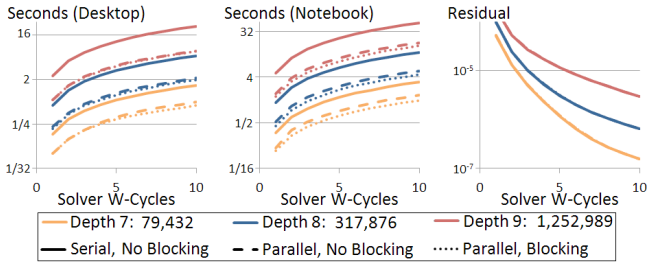


Figure 6: *Efficiency (left and center) and accuracy (right) of our solver: The plots show results for finite-elements systems defined by octrees at different depths and highlight the contributions of parallelization and temporal blocking in different architectures.*

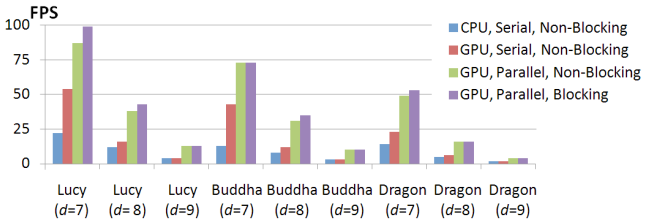


Figure 7: *Efficiency of our interactive solver: The chart shows the contribution, measured in increased interactivity, gained by performing the vertex evaluation on the GPU, leveraging parallelization, and using temporal blocking at different octree depths, d.*

7 Evaluation and Discussion

To evaluate our system, we study how accuracy and efficiency are affected by the different design choices. We begin with an analysis of the solver itself and then look at the system as a whole. We conclude with a discussion of our method.

7.1 Solver Evaluation

To analyze the performance of our solver outside the context of an editing system, we would like to determine how running time and solution accuracy behave as functions of (1) problem size, (2) number of iterations, (3) parallelization, and (4) temporal blocking. For these experiments, we assume no prior knowledge so we instantiate the solution to zero and solve using a W-cycle multigrid solver. We evaluate the accuracy of our solver as the ratio of the norm of the residual to the norm of the initial constraints.

We run the experiments on two architectures, a desktop with a 2.66 GHz Intel Core i7 920 processor and a notebook with a 2.54 GHz Intel Core 2 Extreme Q9300 processor. Both have four cores. The desktop shares an 8MB L3 cache between the cores and the notebook splits a 6 MB L2 cache in a two-by-3 MB configuration.

The results of our experiments with the Forma Urbis Romae fragment, using double-precision arithmetic, can be seen in Figure 6.

The plots show the running time (left and center) and accuracy (right) of the solvers and compare the performance of serial implementations without temporal blocking (solid line), parallel implementations without temporal blocking (dashed line), and parallel implementations with temporal blocking (dotted line). The plots also compare the performance across systems of different resolutions, using a function basis derived from an octree of depth 7 (orange), depth 8 (blue), and depth 9 (red), with 7.9×10^4, 3.2×10^5, and 1.3×10^6 leaf nodes at the finest level, respectively. The accuracy and running time are given as a function of the number of W-cycles performed, with five Gauss-Seidel relaxations performed at each level of the multigrid hierarchy.

The results highlight different features of our implementation. For experiments run on the desktop, the cache is sufficiently large that memory access does not bottleneck the system. As a result, the solver shows a 200% improvement with parallelization, but only an additional 5% improvement with temporal blocking. In contrast, on the cache-constrained notebook, parallelization across four processors only shows a 150% improvement in running time, while the improvement due to temporal blocking increases to roughly 20%.

In the accuracy plots, we see that the parallelization and temporal blocking give errors that are indistinguishable from the serial, non-blocking solver. This confirms that our system correctly implements Gauss-Seidel relaxations and that parallelization and temporal blocking do not affect the precision of the solver.

7.2 System Evaluation

In the context of an interactive editing system, we assume that the solution to the screened Poisson equation in one frame should be a good initial guess for the relaxation in the next. As a result, we use a less aggressive solver, performing a V-cycle to relax the solution, with just two Gauss-Seidel relaxation steps per level.

Since this solver exposes different performance characteristics from those exhibited by the stand-alone solver, we re-evaluate the running time as a function of (1) problem size, (2) parallelization, and (3) temporal blocking. Additionally, we look at the performance gains obtained by using the GPU to evaluate the vertex positions.

For these experiments, we use the desktop described above, equipped with an NVIDIA GeForce GTX 260 video card.

The results of these experiments are shown in Figure 7, which gives the frames-per-second achieved with our solver, for three different models, using the finite-element systems defined by octrees of different depths. The charts compare the performance of different implementations of the solver, allowing us to explore the benefits gained by relegating the vertex evaluation to the GPU, parallelizing the solver, and using temporal blocking. As expected, when the depth of the octree is low, the solution of the screened Poisson equation is less computationally expensive than the evaluation of the vertex positions, and using the GPU to evaluate the vertex positions provides the greatest gains. However, as the depth is increased, the solution of the system starts playing a more significant role, and the contributions of an efficient solver, measured in increased frames-per-second, become more apparent.

			Frames/Second			Solver
	Vertices	DoFs	Solve	RHS	Matrix	Speed-Up
Lucy	$2.6\cdot10^5$	$3\times1.5\cdot10^5$	40	20	8	×2.7
Buddha	$5.4\cdot10^5$	$3\times2.1\cdot10^5$	31	15	6	×2.6
Armadillo	$1.7\cdot10^5$	$3\times2.6\cdot10^5$	30	13	5	×3.3
Dragon	$4.4\cdot10^5$	$3\times2.7\cdot10^5$	27	12	5	×3.0
Isidore	$1.1\cdot10^6$	$3\times2.8\cdot10^5$	27	12	4	×3.0
Forma	$1.0\cdot10^6$	$3\times3.2\cdot10^5$	24	11	4	×3.0
David	$2.0\cdot10^6$	$3\times4.2\cdot10^5$	20	9	3	×3.3

Table 1: *Model summary: Statistics of the geometric complexity, numbers of degrees of freedom, frame-rate, and speed-up due to parallelization and temporal blocking for the models in this work.*

A summary of the performance for the different models discussed in this work can be seen in Table 1. The statistic reflect the performance when the finite-elements system is defined by an octree of depth 8 and the speed-up is measured over a non-blocking, serial, single-precision, implementation that performs the vertex evaluation on the GPU.

The three running times correspond to the three states of the system:

1. **Solve**: When the user has not specified any edits, the system performs a multigrid solve at each frame and then updates the vertex positions.
2. **RHS**: When the user modifies the gradient scales, the right-hand-side of the system is computed before solving and updating the vertex positions.
3. **Matrix**: When the user modifies the filter by adjusting the metric, both the system matrices and right-hand-side are computed before solving and updating the vertex positions.

As the table shows, our solver exhibits a roughly three-fold improvement in performance for all of the models and supports interactive modification of both the gradient scale and the underlying Riemannian metric for high-resolution models.

Evaluating the accuracy of our interactive solver, we find that even after significant modifications to both the gradient scale and the transfer function, the residual norm drops below 10^{-4} after just a few frames. In practice this means that we can process models like the David head, which define a 4×10^5-dimensional system, in a fraction of a second and obtain an artifact free visualization of our editing choices. In contrast, the recent solver of Shi *et al*. [2006] takes almost a second and a half to converge to a residual norm that is smaller than 10^{-3} for a 4.1×10^5-dimensional system.

7.3 Discussion

Smoothness and Aliasing To implement our interactive system, we use an octree-based finite-elements system with first-order B-splines defining the test functions. The use of the octree allows us to bound the dimensionality of the system independent of the number of vertices in the mesh, while the piecewise-tri-linearity of the functions reduces their support, resulting in a sparser system. Though our choices enable an efficient solver, the limitations of these choices become apparent as the resolution of the octree is reduced to push efficiency to its limit.

An example of the limitations of these choices is visualized in Figure 8, which shows the result of applying gradient dampening to the "Isidore Horse" model. The top row shows the original model, the model obtained as a result of gradient dampening at depth 8, and a close-up of the detail on the horse's head. The middle row shows close-ups from the results of gradient dampening at different octree depths, using first-order B-splines.

The figure shows that as the depth of the octree is reduced from eight, to six, to four, the support of the B-splines is more densely sampled by the vertices in the mesh, revealing the C^0 nature of the test functions. To validate that these visual artifacts are the results of the piecewise-linear nature of the first-order B-splines and not aliasing, we re-ran the gradient dampening with second-order elements, using an offline solver. As the images in the bottom row of Figure 8 indicate, the switch to smoother functions successfully removes the first-order discontinuity artifacts.

Examining the figures more carefully, we find that the expected aliasing artifacts do arise at lower depths, though they are more subtle. They are particularly evident in the results obtained using an octree of depth four, and are manifest as detail that could not be smoothed out through gradient dampening, such as the eye, the nostril, and the reigns. Informally, we have found that the aliasing artifacts become more prevalent as the gradient scale deviates from $\beta = 1$, but we are still exploring a more concrete relationship between the resolution of features on the mesh, support-size of the B-splines, magnitude of the gradient scale, and aliasing artifacts.

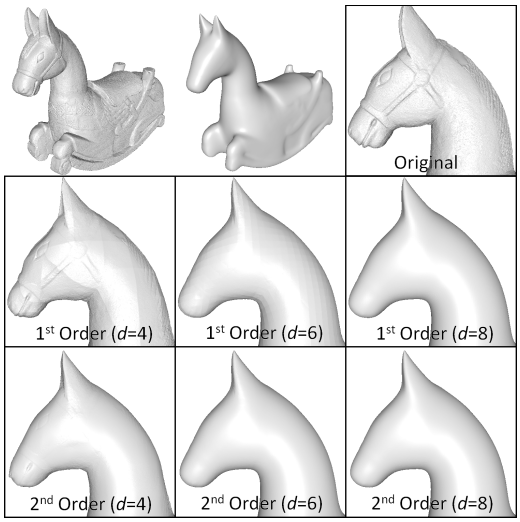


Figure 8: *Limitation of our approach: Due to the use of first-order B-splines in defining our finite-elements, the piecewise-linear nature of the functions becomes apparent when the element resolution is much coarser than the mesh resolution. Though transitioning to second-order B-splines is computationally more expensive, it can effectively alleviate these limitations by providing C^1 continuity.*

GPU-Based Implementation In our work, we have chosen to pursue a CPU-based implementation of the solver. In the future, we would like to adapt the GPU-based Poisson solver of Zhou *et al*., designed for surface reconstruction [2010], to our context. One challenge of doing so is that the GPU-based implementation was designed for solving a (stationary) volumetric Poisson equation. In that context, the value of the IJ-th matrix coefficient only depends on the relative position of cells I and J, allowing for the memory-efficient representation of the Laplace operator by a small $(3 \times 3 \times 3)$ stencil. In contrast, both the anisotropy of our system, and the fact that coefficients depend on how the surface passes through the grid cells, require the explicit computation and storage of each matrix coefficient.

We also note that though the construction of our linear system is implemented on the CPU and the system is non-stationary, our performance is still within a factor of about three of the performance of the stationary, GPU-based implementation of Zhou *et al*. For example, our method constructs and solves the matrix for a 4.2×10^5-dimensional system in roughly 0.3 seconds. By comparison, the GPU implementation of Zhou *et al*. constructs and solves the matrix for a 6.7×10^5-dimensional system in roughly 0.15 seconds.

8 Conclusion

In this work, we have described a general framework for performing curvature-aware geometry filtering through the solution of a screened Poisson equation. We have shown that the system can be efficiently adapted to a changing Riemannian metric and have described a parallel and streaming multigrid implementation for solving the system. Combining these, we develop the first interactive

system that supports the exploration of a rich landscape of mesh filters for real-time geometry processing.

Though our presentation has focused on the real-time editing of geometry, our approach can be generalized in a number of ways. The system can easily be adapted to the editing of other per-vertex quantities, such as colors and normals. The temporal blocking used to stream through memory can be leveraged to develop an out-of-core solver. And, the decomposition of coefficients enabling coarse-grained parallelization can be used to develop a distributed solver.

Finally, we would also like to enrich our system by supporting the use of a palette of profile-curves. This would allow users to apply different curvature transfer functions to different parts of the mesh, further expanding the types of filtering operations supported.

References

AKSOYLU, B., KHODAKOVSKY, A., AND SCHRÖDER, P. 2003. Multilevel solvers for unstructured surface meshes. *SIAM Journal of Scientific Computing 26*.

BAJAJ, C. L., XU, G. L., BAJAJ, R. L., AND T, G. X. 2002. Anisotropic diffusion of subdivision surfaces and functions on surfaces. *ACM Transactions on Graphics 22*, 4–32.

BHAT, P., CURLESS, B., COHEN, M., AND ZITNICK, L. 2008. Fourier analysis of the 2D screened Poisson equation for gradient domain problems. In *Proceedings of the 10th European Conference on Computer Vision*, 114–128.

BHAT, P., ZITNICK, L., COHEN, M., AND CURLESS, B. 2010. Gradientshop: A gradient-domain optimization framework for image and video filtering. *ACM Transactions on Graphics 29*, 10:1–10:14.

BURT, P., AND ADELSON, E. 1983. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communication 31*, 532–540.

CHUANG, M., LUO, L., BROWN, B., RUSINKIEWICZ, S., AND KAZHDAN, M. 2009. Estimating the Laplace-Beltrami operator by restricting 3D functions. *Computer Graphics Forum (Symposium on Geometry Processing)*, 1475–1484.

CLARENZ, U., DIEWALD, U., AND RUMPF, M. 2000. Anisotropic geometric diffusion in surface processing. *Visualization Conference, IEEE 0*, 70.

COWPER, C. 1973. Gaussian quadrature formulas for triangles. *International Journal of Numerical Methods in Engineering 7*, 405–408.

DESBRUN, M., MEYER, M., SCHRÖDER, P., AND BARR, A. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *ACM SIGGRAPH Conference Proceedings*, 317–324.

DOUGLAS, C., HU, J., KOWARSCHIK, M., RÜDE, U., AND WEISS, C. 2000. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis 10*, 21–40.

ECKSTEIN, I., PONS, J.-P., TONG, Y., KUO, C., AND DESBRUN, M. 2007. Generalized surface flows for mesh processing. In *Symposium on Geometry Processing*, 183–192.

GUSKOV, I., SWELDENS, W., AND SCHRÖDER, P. 1999. Multiresolution signal processing for meshes. In *ACM SIGGRAPH Conference Proceedings*, 325–334.

KAZHDAN, M., AND HOPPE, H. 2008. Streaming multigrid for gradient-domain operations on large images. *ACM Transactions on Graphics (SIGGRAPH '08) 27*.

KOBBELT, L., CAMPAGNA, S., VORSATZ, J., AND SEIDEL, H. 1998. Interactive multi-resolution modeling on arbitrary meshes. In *ACM SIGGRAPH Conference Proceedings*, 105–114.

MCCANN, J., AND POLLARD, N. 2008. Real-time gradient-domain painting. *ACM Transactions on Graphics (SIGGRAPH '08) 27*.

MEYER, M., DESBRUN, M., SCHRÖDER, P., AND BARR, A. 2002. Discrete differential-geometry operators for triangulated 2-manifolds. *Visualization and Mathematics 3*, 34–57.

OHTAKE, Y., BELYAEV, A., AND BOGAEVSKI, I. 2000. Polyhedral surface smoothing with simultaneous mesh regularization. In *Proceedings of the Geometric Modeling and Processing (GMP '00)*, 229–237.

PFEIFER, C. 1963. Data flow and storage allocation for the PDQ-5 program on the Philco-2000. *Communications of the ACM 6*, 365–366.

PINKALL, U., AND POLTHIER, K. 1993. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics 2*, 15–36.

SHI, L., YU, Y., BELL, N., AND FENG, W. 2006. A fast multigrid algorithm for mesh deformation. *ACM Transactions on Graphics (SIGGRAPH '06) 25*, 1108–1117.

SMITH, B., BJORSTAD, P., AND GROPP, W. 1996. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

TASDIZEN, T., WHITAKER, R., BURCHARD, P., AND OSHER, S. 2002. Geometric surface smoothing via anisotropic diffusion of normals. In *VIS '02: Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA, 125–132.

TAUBIN, G. 1995. A signal processing approach to fair surface design. In *ACM SIGGRAPH Conference Proceedings*, 351–358.

TRIMESH 2.9, 2009. `www.cs.princeton.edu/gfx/proj/trimesh2/`.

VALLET, B., AND LÉVY, B. 2008. Spectral geometry processing with manifold harmonics. *Computer Graphics Forum (Proceedings Eurographics) 27*, 251–160.

WEBER, O., DEVIR, Y., BRONSTEIN, A., BRONSTEIN, M., AND KIMMEL, R. 2008. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Transactions on Graphics 27*.

WITKIN, A. 1983. Scale-space filtering. In *International Joint Conference on Artificial Intelligence*, 1019–1022.

ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2010. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*.