Interpolating and Approximating Implicit Surfaces from Polygon Soup

Chen Shen James F. O'Brien Jonathan R. Shewchuk

University of California, Berkeley

Abstract

This paper describes a method for building interpolating or approximating implicit surfaces from polygonal data. The user can choose to generate a surface that exactly interpolates the polygons, or a surface that approximates the input by smoothing away features smaller than some user-specified size. The implicit functions are represented using a moving least-squares formulation with constraints integrated over the polygons. The paper also presents an improved method for enforcing normal constraints and an iterative procedure for ensuring that the implicit surface tightly encloses the input vertices.

Keywords: Implicit surfaces, polygon soup, physically based animation, surface smoothing, topological simplification, simulation envelopes, point-based surfaces, surface representation, surface reconstruction.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations; G.1.2 [Numerical Analysis]: Approximation—Approximation of surfaces and contours.

1 Introduction

Polygonal models occur ubiquitously in graphics applications. They are easy to render, easy to compute with, and a vast array of tools have been developed for creating and manipulating polygon data. Unfortunately, polygonal data sets often contain problems, such as holes, gaps, T-junctions, self-intersections, and non-manifold structure, that make them unsuitable for many purposes other than rendering. Even when a polygonal data set does define a closed, manifold surface, other difficulties such as excessive detail or bad-aspect-ratio polygons, can preclude many uses. Data sets containing these problems are so common that the term "polygon soup" has evolved for describing arbitrary collections of polygons that carry no warranties concerning their structure.

This paper provides a tool that can transform arbitrary polygon data into a more useful form. We address this task with a method for generating implicit surfaces that can interpolate or approximate a set of polygons. The user controls how closely the surface approximates the input by selecting a minimum feature size. Geometric details or topological structures below this size tend to be smoothed away. Setting the minimum feature size to zero forces exact interpolation of the polygons. Additionally, if desired, we can cause an approximating surface to fit tightly around the input polygons while still ensuring that the input vertices are completely enclosed by the implicit surface. Figure 1 shows interpolating and approximating surfaces genrated from a complex polygonal model with sharp edges and many small features.

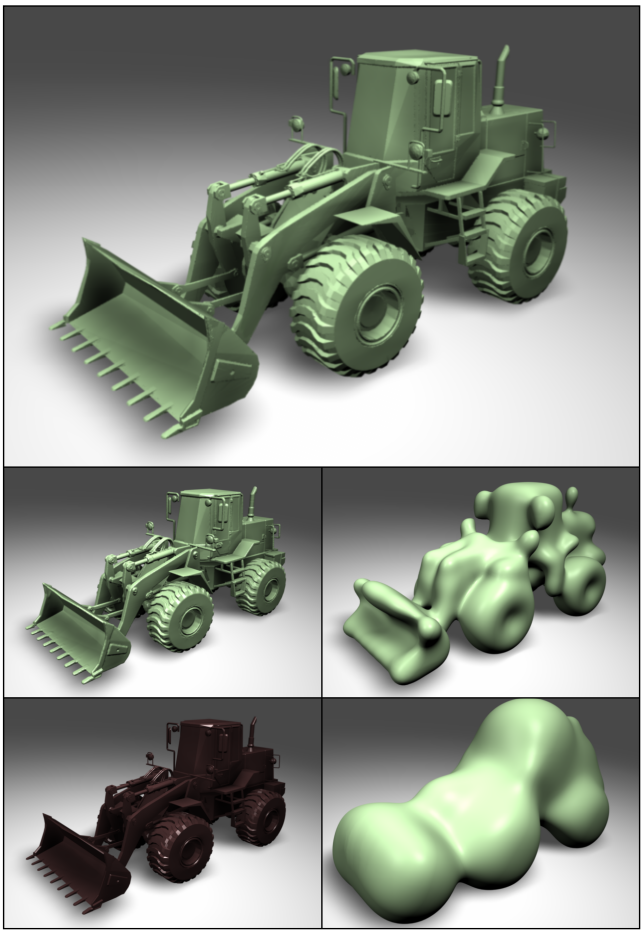

Figure 1: Interpolating and approximating surfaces (green) generated from polygonal original (brown).

An interpolating surface will exactly interpolate the input polygons, but it will also extend to fill gaps and holes so that the resulting surface will be "watertight." This implicit function can then be used directly for a variety of applications, such as inside-outside tests, that are better suited to implicit representations. Alternatively, a clean polygonal model can be extracted and used for applications that require such clean polygonal input.

Approximating surfaces will naturally smooth out geometric features of the input data. Because we are using an implicit representation, topological structures of the input surfaces can also be smoothed away. This behavior makes the method suitable as part of a model simplification process when combined with an appropriate polygonization algorithm.

We can also force the approximating surface to stay "tight" around the original polygons while still smooth-

E-mail: {csh,job,jrs}@eecs.berkeley.edu

ing away details and ensuring that all the original vertices fall inside the approximating surface. This capacity allows us to generate a family of increasingly smooth approximations that eventually converge to a circumscribing ellipsoid. Among other uses, these simplified shapes can be used for easily generating efficient simulation envelopes.

Our algorithm makes use of a scattered-data interpolation method known as *moving least-squares*, commonly abbreviated MLS. The function defining our implicit surfaces is specified by the moving least-squares solution to a set of constraints that would force the function to a given value over the surface region of each polygon, and that would over the same region also force the function's upward gradient to match the polygon's outward normal. Neither condition is specified by simple point constraints: integrated constraints are used over each polygon, and normals constraints directly affect the function's gradient. The degree of approximation is controlled by simply adjusting the least-squares weighting function, but the tightness of the surface and the requirement that the input vertices fall inside the implicit surface both depend on an iterative procedure for adjusting the constraint values over each polygon.

The moving least-squares method has been used by other researchers to define a surface as the fixed-point of an iterative parametric fit procedure—for example, see [Alexa et al., 2001]. Other than using the same general mathematical tool, that approach and this one are unrelated. Unfortunately, those surfaces are often referred to simply as *MLS Surfaces* which may cause some confusion with the method described here. We suggest that the term *implicit moving least-squares surface*, or *IMLS Surface* be used to describe our method.

Our approach is, however, closely related to implicit methods based on partition-of-unity interpolants. (For example see [Ohtake et al., 2003a].) Partition-of-unity and moving least-squares interpolants use different notation, but they are fundamentally alike. One key difference between our formulation and prior ones is that our integrated constraints differ significantly from collections of point constraints. We also use improved normal and approximation procedures, which are applicable to point constraints as well as to our integrated constraints.

Our algorithm has five primary components:

- A scattered data interpolation scheme that, in addition to simple point constraints, allows integrated constraints over polygons.
- A method for enforcing true normal constraints that does not produce undesirable oscillatory behavior.
- An adjustment procedure that causes the implicit surface to fit tightly around the input polygons while still ensuring that the input vertices are completely enclosed by the implicit surface.
- A hierarchical fast evaluation scheme that makes the method practical for large data sets.
- Optional preprocessing to remove unwanted geometry and enforce consistency among the input normals.

2 Background

The work most closely related to ours appears in [Ohtake et al., 2003a]. They use a partition-of-unity method to build a function whose zero-set passes through, or near, a set of input points. Using a procedure originally proposed by [Turk and O'Brien, 1999], they place zero-constraints at each input point, and they also place a pair of additional non-zero point constraints offset in the inward and outward normal directions. To keep the method feasible for large data sets, they use a fast hierarchical evaluation scheme. The partition-of-unity formulation they use and the moving least-squares formulation that we start with are essentially identical: they both belong to a family of meshless interpolation methods that also includes the element-free Galerkin method and smoothed particle hydrodynamics. We refer the reader to [Belytschko et al., 1996] for a discussion of the relationships between these different formulations. The two most significant differences between our work and [Ohtake et al., 2003a] are that we use integrated polygon constraints, and that we use a significantly improved method for enforcing normal constraints. We also describe a different hierarchical evaluation scheme and an iterative method for generating useful approximating surfaces.

Moving least-squares interpolation is also a part of the non-linear projection method used in [Alexa et al., 2001], [Alexa et al., 2003], and [Fleishman et al., 2003]. This projection method defines a surface as a function of a set of points, but the moving least-squares fit is used as part of a non-linear projection that differs substantially from the implicit-surface based method described here.

The technique of defining a surface implicitly using a function constrained to match a set of input points is fairly widespread. In [Savchenko et al., 1995], [Turk and O'Brien, 1999], [Carr et al., 2001], and [Turk and O'Brien, 2002] the function is represented using globally supported radial splines. This class of functions has the nice property that one can make definite statements about a solution's global behavior. These radial splines have also been used to match polygon data by [Yngve and Turk, 2002]. While they were able to achieve results that roughly matched the input polygons, the resulting implicit surfaces still deviated substantially from the input. Different, locally supported functions were used in both [Muraki, 1991], [Morse et al., 2001], and [Ohtake et al., 2003b] for fitting an implicit surface to clouds of point data. In addition to representing function as sums of continuous basis functions, [Museth et al., 2002] and [Zhao et al., 2001] have used level-set methods for fitting surfaces to point clouds. Other function representations include signed-distance functions [Cohen-Or et al., 1998], and medial axes [Bittar et al., 1995]. The text, [Bloomenthal, 1997], also describes several other methods for representing implicit surfaces.

Some of the applications that can be addressed with our method have also been addressed with other methods. An enormous amount of work has been done on smoothing explicit representations of polygonal models, two early examples of which include [Taubin, 1995] and [Desbrun et al., 1999]. Work in that subarea is now quite advanced and methods are available that can preserve sharp features while still smoothing away noise. (For a single recent example, see [Jones et al., 2003].) We can also generate envelopes around input objects and similar ideas have been explored in [Cohen et al., 1996] and [Keren and Gotsman, 1998]. The problem of rectifying polygonal models has been investigated in [Nooruddin and Turk, 2003]. In [Nooruddin and Turk, 2000] the same researchers also looked at methods for removing unwanted interior structure from a polygon model.

3 Methods

The primary tool we work with is a scattered data interpolation method known as moving least-squares. With this method we can create an implicit surface that either interpolates or approximates a given polygonal surface. In this section, we describe how we set up and apply constraints that allow us to generate and control the behavior of the implicit surface.

For the sake of clear exposition, we will start by describing a moving least-squares method for defining implicit surfaces using simple point constraints. We will then describe how that method can be extended to include integrated constraints defined over polygonal regions. Once we specify the

framework we use for defining our functions, we will describe how we enforce normal constraints, adjust the tightness of the surface around the input, and preprocess the data to avoid unwanted internal structures.

During our discussion of the implicit moving-least squares formulation, we keep the description of basis and weighting functions general. However, although our implementation supports a wide range of function choices, we have found that simple weighting functions and constant basis functions are computationally inexpensive, yet they produce results just as good as more expensive choices. For other problems, different choices of weighting and basis functions may be useful.

3.1 Value Constraints at Points

Assume that we have N points located at positions $\boldsymbol{p}_i$, $i \in [1 \dots N]$, and we would like to build a function, $f(\boldsymbol{x})$, that approximates the values ϕ_i at those points. For a standard least-squares fit we would solve

$$\begin{bmatrix} \boldsymbol{b}^\mathsf{T}(\boldsymbol{p}_1) \\ \vdots \\ \boldsymbol{b}^\mathsf{T}(\boldsymbol{p}_N) \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_N \end{bmatrix} \quad , \tag{1}$$

where $\boldsymbol{b}(\boldsymbol{x})$ is the vector of basis functions we use for the fit, and $\boldsymbol{c}$ is the unknown vector of coefficients. Unless this system is under-constrained, it can be resolved efficiently using the method of normal equations and solving an $M \times M$ linear system, where M is the number of basis functions (*i.e.*, the lengths of $\boldsymbol{b}$ and $\boldsymbol{c}$). For example, if we wished to fit a plane we would choose $\boldsymbol{b}(\boldsymbol{x}) = [1, x, y, z]$, or simply $\boldsymbol{b}(\boldsymbol{x}) = [1]$ if we just wished to fit a constant. The resulting function is

$$f(\boldsymbol{x}) = \boldsymbol{b}^\mathsf{T}(\boldsymbol{x})\, \boldsymbol{c} \quad . \tag{2}$$

For the moving least-squares formulation, we allow the fit to change depending on where we evaluate the function so that $\boldsymbol{c}$ varies with $\boldsymbol{x}$. We do so by weighting each row of Equation (1) by $w(\|\boldsymbol{x} - \boldsymbol{p}_i\|)$, where $w(r)$ is some distance weighting function, which gives us

$$\begin{bmatrix} w(\boldsymbol{x}, \boldsymbol{p}_1) & & \\ & \ddots & \\ & & w(\boldsymbol{x}, \boldsymbol{p}_N) \end{bmatrix} \begin{bmatrix} \boldsymbol{b}^\mathsf{T}(\boldsymbol{p}_1) \\ \vdots \\ \boldsymbol{b}^\mathsf{T}(\boldsymbol{p}_N) \end{bmatrix} \boldsymbol{c} = \begin{bmatrix} w(\boldsymbol{x}, \boldsymbol{p}_1) & & \\ & \ddots & \\ & & w(\boldsymbol{x}, \boldsymbol{p}_N) \end{bmatrix} \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_N \end{bmatrix} \tag{3}$$

where $w(\boldsymbol{x}, \boldsymbol{p}_i) = w(\|\boldsymbol{x} - \boldsymbol{p}_i\|)$.

By selecting an appropriate weight function, a variety of interpolating or approximating behaviors can be achieved, even with low-order basis functions. In general, a weight function that approaches $+\infty$ at zero will cause interpolation. We use the weight function

$$w(r) = \frac{1}{(r^2 + \epsilon^2)} \quad . \tag{4}$$

The parameter ϵ allows a degree of control over the function's behavior which we discuss later.

Giving matrices names and explicitly noting their dependence on $\boldsymbol{x}$, Equation (3) becomes

$$\boldsymbol{W}(\boldsymbol{x})\, \boldsymbol{B}\, \boldsymbol{c}(\boldsymbol{x}) = \boldsymbol{W}(\boldsymbol{x})\, \boldsymbol{\phi} \quad . \tag{5}$$

The resulting normal equations are

$$\boldsymbol{B}^\mathsf{T}\ (\boldsymbol{W}(\boldsymbol{x}))^2\ \boldsymbol{B}\, \boldsymbol{c}(\boldsymbol{x}) = \boldsymbol{B}^\mathsf{T}\ (\boldsymbol{W}(\boldsymbol{x}))^2\ \boldsymbol{\phi} \tag{6}$$

and we can evaluate the fit function's value using

$$f(\boldsymbol{x}) = \boldsymbol{b}^\mathsf{T}(\boldsymbol{x})\, \boldsymbol{H}^{-1}\, \boldsymbol{B}^\mathsf{T}\ (\boldsymbol{W}(\boldsymbol{x}))^2\ \boldsymbol{\phi} \quad , \tag{7}$$

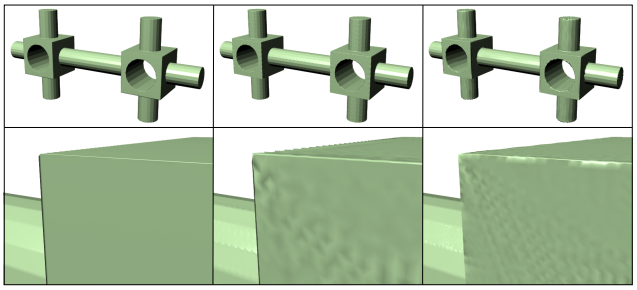

Figure 2: The column on the left shows the results generated using integrated polygonal constraints. The middle and right columns show the results generated with different densities of scattered point constraints.

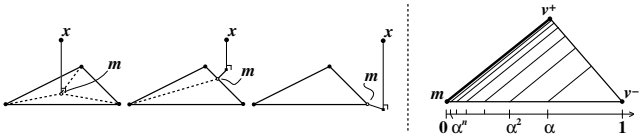

Figure 3: The quadrature scheme used over a triangle.

where

$$\boldsymbol{H} = \boldsymbol{B}^\mathsf{T}\ (\boldsymbol{W}(\boldsymbol{x}))^2\ \boldsymbol{B} \quad . \tag{8}$$

The derivatives with respect to $\boldsymbol{x}$ of the fit function can be evaluated using

$$\begin{aligned} f'(\boldsymbol{x}) = (\boldsymbol{b}^\mathsf{T})'(\boldsymbol{x})\, \boldsymbol{H}^{-1}\, \boldsymbol{B}^\mathsf{T}\ (\boldsymbol{W}(\boldsymbol{x}))^2\ \boldsymbol{\phi}\ - \\ \boldsymbol{b}^\mathsf{T}(\boldsymbol{x})\, \boldsymbol{H}^{-1} \boldsymbol{H}' \boldsymbol{H}^{-1}\, \boldsymbol{B}^\mathsf{T}\ (\boldsymbol{W}(\boldsymbol{x}))^2\ \boldsymbol{\phi}\ + \\ \boldsymbol{b}^\mathsf{T}(\boldsymbol{x})\, \boldsymbol{H}^{-1}\, \boldsymbol{B}^\mathsf{T}\ ((\boldsymbol{W}(\boldsymbol{x}))^2)'\ \boldsymbol{\phi} \quad , \end{aligned} \tag{9}$$

where

$$\boldsymbol{H}' = \boldsymbol{B}^\mathsf{T}\ ((\boldsymbol{W}(\boldsymbol{x}))^2)'\ \boldsymbol{B} \quad , \tag{10}$$

and the derivative of $(\boldsymbol{W}(\boldsymbol{x}))^2$ is obtained by simply taking the derivative of the squared weighting function along the matrix's diagonal.

3.2 Value Constraints Integrated over Polygons

Although the formulation in the previous section works well for point constraints, the input data we are concerned with consists of polygons, and for each of these polygons we want to constrain the fit function over its entire surface. If we were not interested in interpolating the polygons, we could approximate the desired effect with point constraints scattered over the surface of each polygon. Aside from potentially requiring a very large number of points, scattered point constraints work reasonably well for approximating surfaces. However, interpolating surfaces and surfaces that approximate closely show undesirable bumps and dimples corresponding to the point locations. (See Figure 2.) In particular, bumps and dimples occur unless ϵ is substantially larger than the spacing between points.

To achieve good results, what we would like to do is to scatter an infinite number of points continuously across the surface of each polygon. Notice that Equation (6) can be rewritten as an explicit summation over a set of point constraints,

$$\left(\sum_{i=1}^{N} w^2(\boldsymbol{x}, \boldsymbol{p}_i)\, \boldsymbol{b}(\boldsymbol{p}_i)\, \boldsymbol{b}^\mathsf{T}(\boldsymbol{p}_i) \right) \boldsymbol{c}(\boldsymbol{x}) = \sum_{i=1}^{N} w^2(\boldsymbol{x}, \boldsymbol{p}_i)\, \boldsymbol{b}(\boldsymbol{p}_i)\, \phi_i \tag{11}$$

In this form it becomes clear how we can apply constraints continuously over each polygon's surface.

For a data set of K polygons, let Ω_k, $k \in [1 \dots K]$, be the kth input polygon. The parenthesized term of Equation (11)

and the term on the right are replaced by integrals over the polygons and we have

$$\left(\sum_{k=1}^{K} \boldsymbol{A}_k\right)\ \boldsymbol{c}(\boldsymbol{x}) = \sum_{k=1}^{K} \boldsymbol{a}_k \tag{12}$$

where $\boldsymbol{A}_k$ and $\boldsymbol{a}_k$ are defined by

$$\boldsymbol{A}_k = \int_{\Omega_k} w^2(\boldsymbol{x},\boldsymbol{p})\, \boldsymbol{b}(\boldsymbol{p})\, \boldsymbol{b}^{\mathsf{T}}(\boldsymbol{p})\ \mathrm{d}\boldsymbol{p}\quad , \tag{13}$$

$$\boldsymbol{a}_k = \int_{\Omega_k} w^2(\boldsymbol{x},\boldsymbol{p})\, \boldsymbol{b}(\boldsymbol{p})\, \phi_k\ \mathrm{d}\boldsymbol{p}\quad , \tag{14}$$

$\boldsymbol{p}$ is the integration variable ranging over the polygon, and ϕ_k is the constraint value. We can choose ϕ_k to be constant, or we can choose ϕ_k to vary polynomially over each polygon. For later use, it is convenient to define terms with the weighting function omitted:

$$\widetilde{\boldsymbol{A}}_k = \int_{\Omega_k} \boldsymbol{b}(\boldsymbol{p})\, \boldsymbol{b}^{\mathsf{T}}(\boldsymbol{p})\ \mathrm{d}\boldsymbol{p}\quad , \tag{15}$$

$$\tilde{\boldsymbol{a}}_k = \int_{\Omega_k} \boldsymbol{b}(\boldsymbol{p})\, \phi_k\ \mathrm{d}\boldsymbol{p}\quad . \tag{16}$$

The integrals will be infinite when $\epsilon = 0$ and the evaluation point $\mathbf{x}$ lies precisely on a polygon. In this case, $f(\mathbf{x})$ has a removable singularity at $\mathbf{x}$; we can skip the least-squares step and simply set $f(\mathbf{x})$ to the value ϕ_k dictated by the polygon. It is possible that two polygons intersect at a point where their constraints disagree, in which case f has an essential singularity at that point. Evaluating at or near such points in a numerically stable fashion is difficult. However, we can sidestep the issue by setting ϵ to an extremely small number, far below the smallest feature size relevant to a given application.

Computing these integrals is conceptually straightforward. Each entry of the matrix $\boldsymbol{b}\ \boldsymbol{b}^{\mathsf{T}}$ and the vector $\boldsymbol{b}$ is a polynomial in $\boldsymbol{p}$, the weight function we have chosen is a rational polynomial in $\boldsymbol{p}$, and each of the components of the matrices can, of course, be computed independently. For a one-dimensional integral (*i.e.*, constraints over edges) the integrals have closed form solutions. (See Appendix A.) Unfortunately, we have not been able to find closed-form solutions of the two-dimensional integrals.

The obvious solution to this problem would simply approximate the integrals using a standard quadrature method. Unfortunately, this solution performs poorly for the same reason that scattering point constraints does: unless the distance between quadrature points is significantly less than ϵ the resulting surface will have dimples and bumps. The culprit responsible for this behavior is the weighting function. Its singularity, or near singularity, at zero, causes severe problems for standard quadrature schemes. These difficulties extend to Monte-Carlo schemes, which explains the problems encountered with scattered points. The method we use is aware of the singular nature of the weighting function and it accounts for that contribution without underweighting the contribution from the rest of the triangle.

Let $\boldsymbol{m}$ be the point in Ω_k that is closest to the evaluation point, $\boldsymbol{x}$. (See Figure 3.) If this point is on the interior of Ω_k, we split the triangle into three triangles each of which has $\boldsymbol{m}$ as one of its vertices. If the point lies on an edge, the triangle is split into two triangles. If the point lies on an existing vertex, the triangle is not split. The integral over the original triangle is the sum of integrals over each of these sub-triangles. Each sub-triangle has $\boldsymbol{m}$ as one of its vertices, and the other two vertices are denoted $\boldsymbol{v}^+$ and $\boldsymbol{v}^-$ such that $w(\boldsymbol{x},\boldsymbol{v}^+) \geq w(\boldsymbol{x},\boldsymbol{v}^-)$.

To compute the sub-triangle area integral we separate it into two successive one-dimensional integrals as shown in Figure 3. The outer one integrates along the edge from $\boldsymbol{m}$ to $\boldsymbol{v}^-$ using a special numerical quadrature rule. The inner one integrates along the barycentric iso-lines that are parallel to the edge from $\boldsymbol{m}$ to $\boldsymbol{v}^+$, using the one-dimensional analytical solution.

The outer, numerical integration uses the Newton-Cotes trapezoidal rule with irregularly spaced samples. If the edge from $\boldsymbol{m}$ to $\boldsymbol{v}^-$ is parameterized from zero to one with zero corresponding to $\boldsymbol{m}$, the samples occur at $0, \alpha^n, \ldots, \alpha^1, \alpha^0$. We arbitrarily use $\alpha = 2/3$, and n is proportional to the logarithm of the edge length. The integral should be appropriately scaled by the sub-triangle area. This scheme captures the behavor near the potentially singular location, $\boldsymbol{m}$, without neglecting the rest of the triangle.

3.3 Normal Constraints

The two previous sections describe how we can implement constraints on the value of the moving least-squares function at discrete points and over polygonal patches. However, if we attempt to define a surface by only requiring it to take a given value on its surface, we will not obtain useful results. Previous researchers, for example [Ohtake et al., 2003a], have implemented pseudo-normal constraints with a technique originally suggested by [Turk and O'Brien, 1999]. This technique places a zero constraint at a point on the surface, a positive constraint offset sightly outside the surface, and a negative one slightly inside.

Unfortunately, this approach does not work as well as one might like. The additional constraints influence the function's gradient only crudely, and they can cause undesirable oscillatory behavior as the evaluation point moves away from the surface. This behavior is illustrated in the lower half of Figure 4. It occurs because when the distance between the evaluation point and the surface point is much larger than the offset distance, the inside and outside constraints effectively cancel each other out. Even if only outside (or only inside) constraints are used, they will still effectively merge to a single average valued constraint far away. Heuristics, such as those described by [Ohtake et al., 2003a], can suppress some of the spurious behavior, but the value of the function far from the surface will not be useful. Furthermore, these quasi-normal constraints cause severe problems when used with the approximation procedure described in the next section.

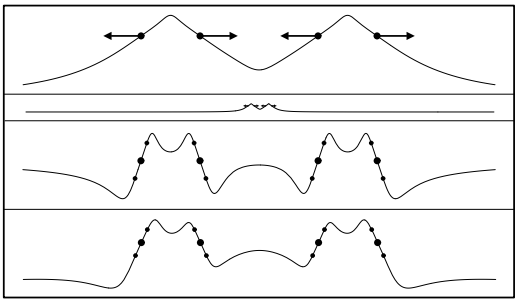

Figure 4: A one-dimensional example showing the height field generated from four position and normal constraints. The first (top) image shows the result with our method, and the arrows indicate the outward normal directions. The second shows an expanded view demonstrating far-field behavior. The third and fourth images show the results generated by pseudo-normal constraints with linear and quadratic basis functions. The small dots indicate the placement of the inside and outside pseudo-normal constraints.

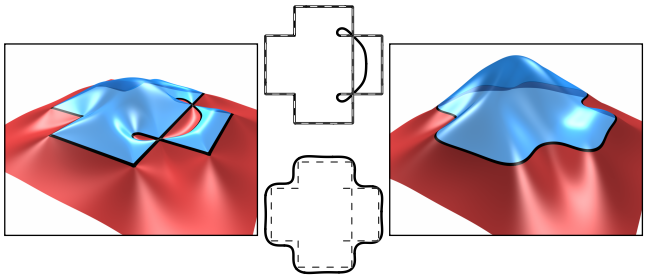

Figure 5: A two-dimensional example comparing interpolating and approximating results. The center images show input constraints as dotted lines and the contour as a solid line. The outer images show the resulting function as a height-field.

One of our key innovations is to impose normal constraints by forcing the interpolating function to behave like a prescribed function (in the neighborhood of a polygon), as opposed to a prescribed constant value. In other words, instead of using the moving least-squares method to blend between *constant values* associated with each polygon (or point), we blend between *functions* associated with them. This method exhibits little undesirable oscillation.

If $\hat{\boldsymbol{n}}_k$ is the normal associated with polygon Ω_k, we define the function $S_k(\boldsymbol{x})$ that describes how that polygon wants the interpolant to behave as

$$S_k(\boldsymbol{x}) = \phi_k + (\boldsymbol{x} - \boldsymbol{q}_k)^{\mathsf{T}} \hat{\boldsymbol{n}}_k \quad (17)$$

$$= \psi_{0k} + \psi_{xk}\, x + \psi_{yk}\, y + \psi_{zk}\, z \quad , \quad (18)$$

where $\boldsymbol{q}_k$ is an arbitrary point on the polygon Ω_k, and ψ_{0k}, ψ_{xk}, ψ_{yk}, and ψ_{zk} are resulting polynomial coefficients. Interpolating between these functions reduces to simply interpolating the ψ coefficients just as we would normally interpolate a constant value ϕ_k.

In the special case where $\hat{\boldsymbol{n}}_k = 0$, the normal constraints are exactly equivalent to the original value constraints. As a result we can easily mix constraints with and without normals.

In the case where we only use the constant basis function, so that $\boldsymbol{b}(\boldsymbol{x}) = [1]$, the fit from Equation (5) simplifies to

$$\begin{bmatrix} w(\boldsymbol{x}, \boldsymbol{p}_1) \\ \vdots \\ w(\boldsymbol{x}, \boldsymbol{p}_i) \end{bmatrix} c_1 = \begin{bmatrix} w(\boldsymbol{x}, \boldsymbol{p}_1) & & \\ & \ddots & \\ & & w(\boldsymbol{x}, \boldsymbol{p}_N) \end{bmatrix} \begin{bmatrix} S_1(\boldsymbol{x}) \\ \vdots \\ S_N(\boldsymbol{x}) \end{bmatrix} \quad (19)$$

which has the very intuitive interpretation that the interpolating function's value at $\boldsymbol{x}$ is simply the weighted average of the values at $\boldsymbol{x}$ predicted by each of the $S_k(\boldsymbol{x})$.

We have found this approach to work well. Figure 4 illustrates that the undesirable behavior that occurs with quasi-normal constraints does not occur with this method. Further, this approach causes the surface normals to actually take on the desired value at constraint points, whereas offset constraints do not. For polygonal constraints, the normals are interpolated so long as they are consistent with the polygon's plane. In addition to being useful with moving least-squares, this normal constraint approach should also work with other interpolation methods such as the radial splines used in [Turk and O'Brien, 1999]. Because the magnitude of the normal constraint grows linearly as the evaluation point moves away, we must choose a weighting function that falls off faster than linearly.

3.4 Interpolation and Approximation

When the weighting function parameter, ϵ, is set to zero, the moving least-squares function will exactly interpolate constraint values. If we follow the general approach described in [Turk and O'Brien, 1999] and [Ohtake et al., 2003a] of constraining the function to be zero at input points or polygons, supplying appropriate normal constraints, and extracting the iso-surface $f(\mathbf{x}) = 0$, then all the input polygons will be parts of the resulting implicit surface.

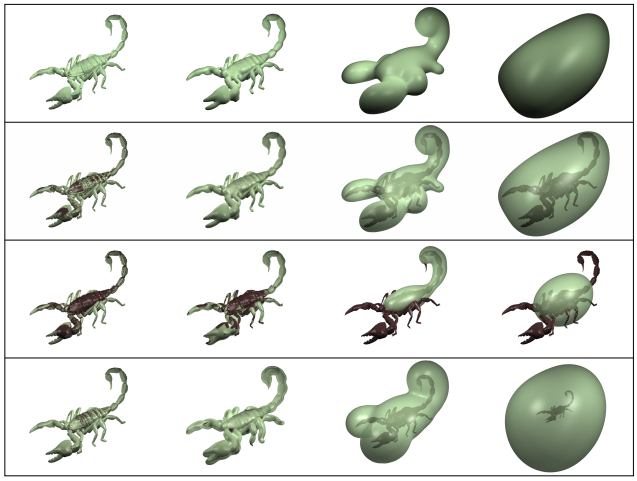

Figure 6: The first (top) row shows the result of applying our iterative adjustment algorithm with different values of ϵ to a polygonal scorpion model. The second row shows the original and constructed surfaces together. The third row shows the result of only adjusting the surface to average values (no iterative adjustment). The fourth row shows the result generated when no correction is applied.

If the polygonal surface contains gaps or holes, then the implicit surface will extend beyond the input polygons to generate a closed surface. As with previous methods that accomplish hole filling using some form of implicit surface, there is no guarantee that the results will satisfy any particular criteria. However, we generally find that these extensions close gaps and holes in a useful fashion that produces results simular to what a human might have selected.

If the polygon surface self-intersects, then the interpolating surface will have some form of saddle at the intersections. This behavior is illustrated for a two-dimensional example in Figure 5.

When ϵ is set to a non-zero value the weighting function is no longer singular at zero, and the moving least-squares function interpolates constraint values only approximately. Examination of Equation (4) reveals that ϵ has the same units as distance. It corresponds to a feature size parameter: structures smaller than ϵ tend to be smoothed away by the approximation.

While generating an approximate surface by simply setting ϵ to some non-zero value works well to a limited extent, it suffers from two problems. The first is that as epsilon is set to larger values, the approximating surface has the tendency to move away from the input data (Figure 6, bottom row). For example, very large values of ϵ will smooth an object to a simple sphere-like shape, but the sphere radius may be several times the original object's circumradius. The second problem is that we cannot ensure that all the object's original vertices fall inside the implicit surface, and for some applications this guarantee is important.

To correct the first problem we simply build a moving least-squares function with the desired ϵ, sample its average value over the input polygons, and then extract a surface at that iso-value (Figure 6, third row). Although this procedure may at first appear to require substantial extra work, the additional work is actually not particularly significant. The majority of computation is spent extracting the iso-surface, and that task still only needs to be done once.

By adjusting the iso-value we achieve a surface that, on average, stays close to the input data, but with this construction we expect that roughly half the original vertices will fall outside the surface. To ensure that original vertices lie inside the surface, we iteratively adjust the ϕ values assigned to the vertices.

Initially, the ϕ values associated with each vertex are all zero and the ϕ associated with each triangle is the constant zero as well. If a vertex, $\boldsymbol{v}$, protrudes outside the iso-surface (*i.e.*, $f(\boldsymbol{v}) > 0$), we adjust its ϕ value by $-\gamma\, f(\boldsymbol{v})$ where γ is an adjustment rate parameter between zero and one (typically close to one). Once the vertices of a triangle have been assigned different values, we linearly interpolate ϕ over the triangle when computing integrals. This adjustment process is done iteratively until no original vertex falls outside the iso-surface. The final surface is guaranteed to enclose all input vertices, as illustrated in the top two rows of Figure 6. As with adjusting the iso-value, the majority of computation is still spent extracting the iso-surface, and that task still only needs to be done once.

Variations on this iterative procedure for adjusting the ϕ values could also be used to enforce other conditions. For example, it could be used to guarantee that all points are within some set distance of the iso-surface. Conditions could be tested at points other than the initial vertices, and the iterative procedure could also adjust the normal direction or magnitude associated with each constraint.

3.5 Fast Evaluation

Naïve implementation of the moving least-squares function would require work linear in the number of constraints for each function evaluation. For large data sets, this naïve approach is completely infeasible. A similar problem arises with the partition-of-unity method used in [Ohtake et al., 2003a]. They address the problem using a hierarchical evaluation scheme that caches approximations based on local neighborhoods. Because partition-of-unity and moving least-squares methods are essentially equivalent methods, their hierarchical evaluation scheme could be used with our method as well. We have, however, implemented a different evaluation scheme which we describe briefly.

We observe that the primary expense for evaluating the moving least-squares function is the cost of computing the sums and integrals for Equation (12). Were it not for the weighting function's dependence on $\boldsymbol{x}$, the terms would be constant and the summation would only need to be computed once.

For terms that peak near the evaluation point, the weighting function changes rapidly. However, the weight function changes only slowly for far terms. We can approximate groups of the slowly changing far terms by first summing them and then multiplying by their average weight.

For our hierarchical scheme, we first store the input triangles in a K-D tree where each triangle is stored at one of the leaf nodes. We then compute the unweighted integrals, Equations (15) and (16), for each triangle and store them, along with the triangle's axis-aligned bounding box, in the leaf nodes. The interior nodes store the unweighted sums of their children's integrals/sums, and a bounding box that encloses the union of their children's bounds. We also store an area-weighted "center of mass" for each node.

To evaluate the contribution of a subtree, we test the evaluation point to see if it falls outside the subtree's bounding box by a distance greater than λ times the box's diameter. If it does, we use the sums stored at the subtree's root node with a weight computed using the distance between the node's center of mass and the evaluation point. If the evaluation point is not sufficiently distant, we recursively test the node's children. Only when we find that a leaf node fails our distance test do we need to compute the weighted integral terms for that node.

This scheme was easy to implement using existing K-D tree collision detection code, and it allows us to work with models consisting of several hundred thousand triangles. The user can make a trade-off between speed and accuracy by adjusting λ. Our examples were generated with λ between 0.01 (when $\epsilon = 0$) and 0.1 (when ϵ is large).

Figure 7: The top left image shows a polygonized version the Utah teapot which contains holes (around lid and tip of spout), and intersecting parts (handle and spout with body). The top right image is a near-interpolating surface which fills the holes and removes intersecting surfaces. The bottom row contains photographs of physical models built on a fused deposition machine. The bottom right image shows a physical cutaway model.

3.6 Preprocessing

Although the methods we have described in previous sections cope reasonably well with intersecting geometry and layers of internal structure, it may still be useful to first remove some of these polygons. In particular, our algorithm will happily produce surfaces corresponding to internal structures, even if only an exterior shell was desired. In these cases, we can pre-process the input to remove polygons that are not visible from the exterior using methods such as those in [Nooruddin and Turk, 2003] and [Nooruddin and Turk, 2000].

The normal constraints depend on consistently oriented normals. Unfortunately, many polygon models may have normals that randomly point inward or outward. We force normals on topological surfaces to point in a consistent direction. We also orient the normals of any exterior-visible polygon to point outward. If both sides of a triangle are exterior-visible then we set that triangle's normal to zero.

4 Results and Discussion

Figures 1, 10 and 11 show the result of applying our algorithm to a variety of models using different values of ϵ. Animations showing continuous variation of ϵ from interpolating to extreme smoothing appear on the proceedings DVD. Most of these models contain holes, self-intersections, non-manifold structure, and other defects. The objects in brown are the original polygonal models. Green objects are output from our algorithm. For sufficiently large ϵ, all objects converge to a circumscribing ellipsoid-like shape. As Figure 8 shows, the interpolating surfaces can reproduce small features and sharp edges.

As Figure 7 shows, we can use this algorithm as an effective preprocessor before sending a model to a rapid prototyping machine. The Utah teapot contains holes and self-intersections that would cause the machine to produce garbage output. A tightly approximating implicit surface does not contain those problems and allows a successful

Model	Fig.	P. In	ϵ	V. Out	Time	ϵ	V. Out	Time	ϵ	V. Out	Time	ϵ	V. Out	Time
Heavy Loader	1	37	0	2000	11:42	0.05	800	64:06	5	62	72:48	30	30	92:34
Teapot	10	6.3	0	1000	5:50	0.8	300	10:28	10	53	22:02	60	26	42:31
Cow	10	5.8	0	1000	5:37	0.8	300	8:04	7	61	23:35	120	23	58:19
Bunny	10	69	0	1500	8:13	0.4	400	19:34	10	50	41:36	60	28	72:23
Dragon	10	870	0	2000	12:23	0.6	400	82:21	7	46	89:54	30	21	97:04
Scorpion	10	78	0	1500	9:54	0.6	400	67:50	5	65	61:06	30	26	80:55
Intersecting Star	11	0.05	0	1000	4:44	1	300	5:18	10	48	7:03	110	28	8:20
Machine Part	11	0.8	0	1000	4:45	0.8	300	5:54	7	60	8:50	120	29	20:21
Deck Chair	11	3.9	0	1000	5:01	0.8	300	8:29	10	55	27:06	120	30	52:36
Armchair	11	3.4	0	1000	4:56	0.4	300	7:53	7	62	28:28	120	28	41:09
Cube Shape	11	0.01	0	1000	4:44	0.8	300	5:01	10	45	3:06	80	25	1:52

Table 1: This table lists the computation times and ϵ parameter for the examples used in this paper. The columns P. In and V. Out list the number (in thousands) of polygons in the input model and the number (in thousands) of vertices in the output surface. We measure ϵ in thousandths of the diagonal length of the object's bounding box. Computation times (minutes:seconds) measure total time on a 3 GHz P4 beginning with reading the input model and ending with writing the polygonized output surface. Column groups match the ϵ values used for the examples shown in the figures.

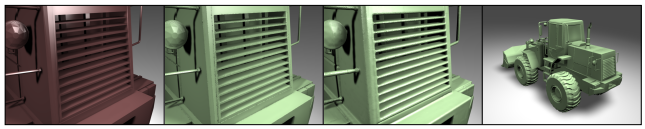

Figure 8: The far-left image shows a closeup view of the original polygons for the heavy-loader's back grill. The center-left image shows the resulting interpolating surface, and the center-right a slightly approximating one. The far-right image shows a rear view of the interpolating surface for the entire loader. The dented appearance near sharp edges is a polygonization artifact.

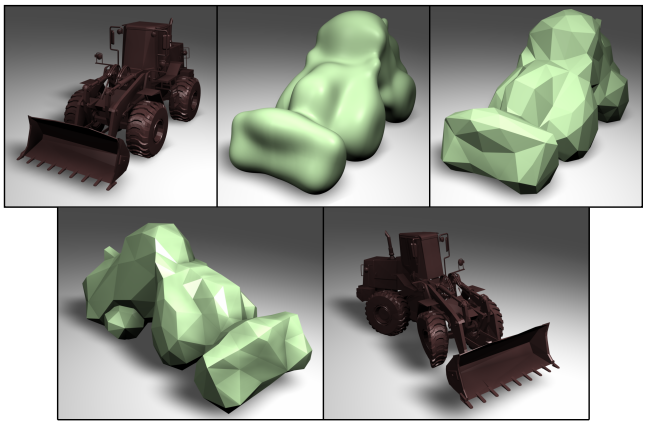

Figure 9: The heavy-loader shown top left contains many defects that make it unsuitable for simulation as a deformable object. The approximating surface, top center, fully encloses the original model. The tetrahedral finite-element model, top right, can be used as a simulation envelope to model the effect of an impact, lower left and lower right.

build. Additionally, because building a solid teapot would waste material, it is desirable to include an inner surface. We generated the inner surface of the cutaway teapot by taking the same MLS function used to create the outer surface and computing another iso-surface for a lower iso-value. These photographs demonstrate that our method produces surfaces that can be used to generate structurally sound physical models.

Deformable object simulations based on the finite element method have found widespread use in video games and film production. Unfortunately, self-intersections, topological inconsistencies, holes, and triangles with bad aspect ratios render most graphics models ill-suited for use as a finite element mesh. Even meshes that are free of these problems may contain far too many elements to be practical for simulation. We can still animate these objects by embedding them in a suitable, enclosing, deformable mesh. As demonstrated by Figure 9, the tight, smooth, enclosing surfaces that can be generated with our method make excellent simulation envelopes.

Currently, we are using the polygonizer described in [Bloomenthal, 1994] for extracting iso-surfaces. It works well for smooth surfaces, but extracting small features requires a very fine resolution and produces models with an inordinate number of polygons. Our polygonal models produce useful envelopes after being passed though surface simplification software (see Figure 9), but extracting them is time consuming and requires substantial storage. (See Table 1.) We are currently considering better methods for surface extraction based on the algorithm from [Boissonnat and Oudot, 2003]. The surfaces for the heavy-loader shown in Figures 1 and 8 were extracted using a partial implementation of that algorithm.

Acknowledgments

We thank the other members of the Berkeley Graphics Group for their helpful criticism and comments. We especially thank Ravi Kolluri for his help with polygonization, Adam Bargteil for help with the heavy-loader simulation, Carlo Séquin for his help with the fused deposition machine used to make the teapots, and Okan Arikan for help rendering. Images in this paper were rendered with Pixie. This work was supported in part by NSF CCR-0204377, California MICRO 02-055, and by generous support from Pixar Animation Studios, Intel Corporation, Sony Computer Entertainment America, the Okawa Foundation, and the Alfred P. Sloan Foundation.

References

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. T. 2001. Point set surfaces. In *IEEE Visualization 2001*, 21–28.

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. T. 2003. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics 9*, 1 (Jan.), 3–15.

Belytschko, T., Krongauz, Y., Organ, D., Fleming, M., and Krysl, P. 1996. Meshless methods: An overview and recent developments. *Computer Methods in Applied Mechanics and Engineering 139*, 3–47. Special issue on meshless methods.

Bittar, E., Tsingos, N., and Gascuel, M.-P. 1995. Automatic reconstruction of unstructured 3d data: Combining a medial axis and implicit surfaces. *Proceedings of Eurographics 95*, 457–468.

Bloomenthal, J. 1994. An implicit surface polygonizer. In *Graphics Gems IV*. 324–349.

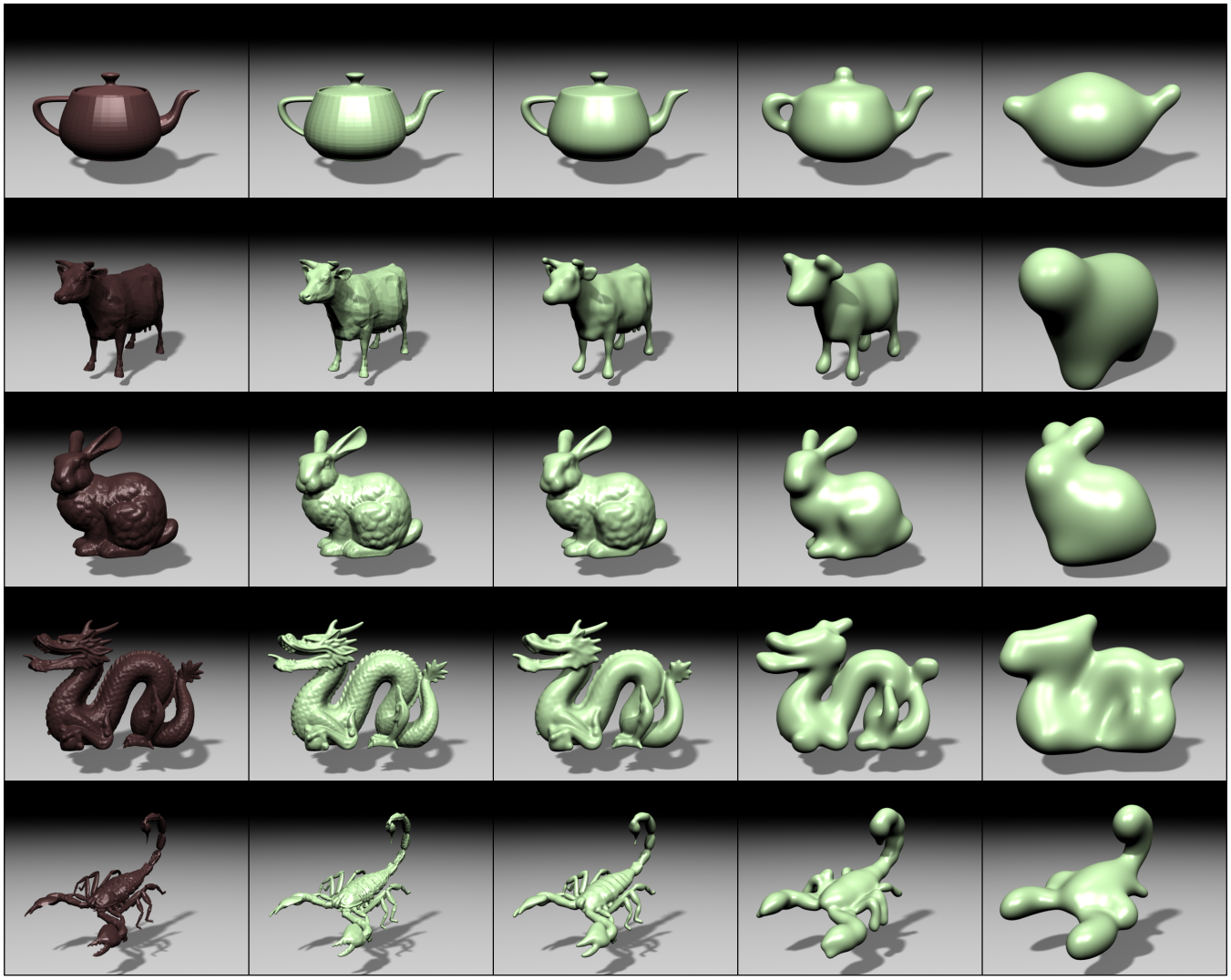

Figure 10: A collection of polygonal models processed with our algorithm. [Continued on next page.]

Bloomenthal, J., Ed. 1997. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, Inc., San Francisco, California.

Boissonnat, J. D., and Oudot, S. 2003. Provably good surface sampling and approximation. In *Proceedings of the ACM SIGGRAPH Symposium on Geometry Processing*, 9–18.

Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C., and Evans, T. R. 2001. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of ACM SIGGRAPH 2001*, 67–76.

Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Jr., F. P. B., and Wright, W. 1996. Simplification envelopes. In *Proceedings of ACM SIGGRAPH 1996*, 119–128.

Cohen-Or, D., Solomovici, A., and Levin, D. 1998. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics 17*, 2 (Apr.), 116–141.

Desbrun, M., Meyer, M., Schröder, P., and Barr, A. H. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of ACM SIGGRAPH 1999*, 317–324.

Fleishman, S., Alexa, M., Cohen-Or, D., and Silva, C. T. 2003. Progressive point set surfaces. *ACM Transactions on Graphics 22*, 4 (Oct.), 97–1011.

Jones, T. R., Durand, F., and Desbrun, M. 2003. Non-iterative, feature-preserving mesh smoothing. *ACM Transactions on Graphics 22*, 3 (July), 943–949.

Keren, D., and Gotsman, C. 1998. Tight fitting of convex polyhedral shapes. *International Journal of Shape Modeling*, 111–126.

Morse, B., Yoo, T. S., Rheingans, P., Chen, D. T., and Subramanian, K. 2001. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *Proceedings of Shape Modelling International*, 89–98.

Muraki, S. 1991. Volumetric shape description of range data using "blobby model". In *Proceedings of ACM SIGGRAPH 1991*, 227–235.

Museth, K., Breen, D. E., Whitaker, R. T., and Barr, A. H. 2002. Level set surface editing operators. *ACM Transactions on Graphics 21*, 3 (July), 330–338.

Nooruddin, F. S., and Turk, G. 2000. Interior/exterior classification of polygonal models. In *IEEE Visualization 2000*, 415–422.

Nooruddin, F. S., and Turk, G. 2003. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (Apr.), 191–205.

Ohtake, Y., Belyaev, A., Alexa, M., Turk, G., and Seidel, H.-P. 2003. Multi-level partition of unity implicits. *ACM Transactions on Graphics 22*, 3 (July), 463–470.

Ohtake, Y., Belyaev, A., and Seidel, H.-P. 2003. A multi-scale approach to 3d scattered data interpolation with compactly supported basis functions. In *Proceedings of Shape Modelling International*, 292–300.

Savchenko, V. V., Pasko, A. A., Okunev, O. G., and Kunii, T. L. 1995. Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum 14*, 4 (Oct.), 181–188.

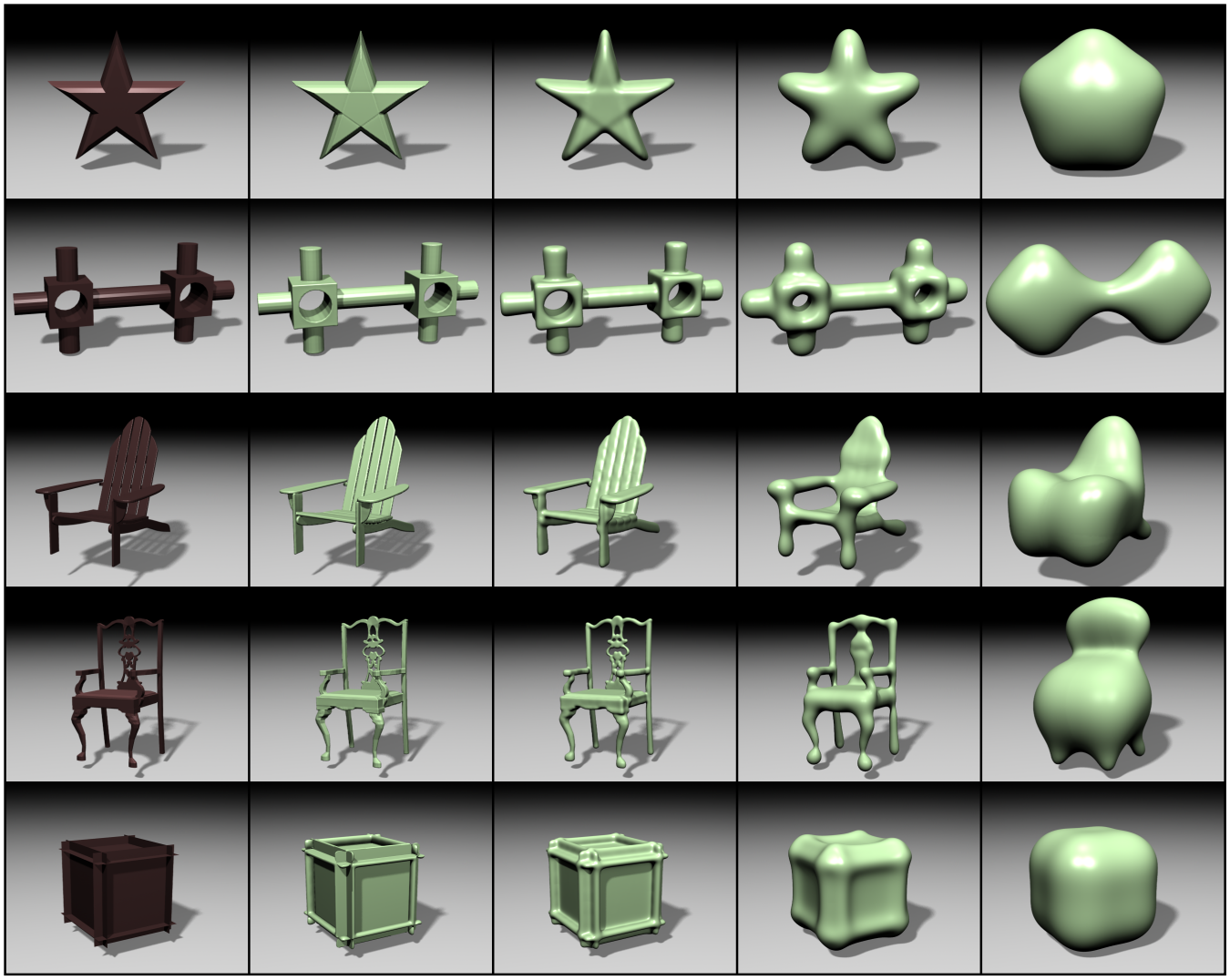

Figure 11: Additional polygonal models processed with our algorithm. [Continued from previous page.]

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

Turk, G., and O'Brien, J. F. 1999. Shape transformation using variational implicit functions. In *Proceedings of ACM SIGGRAPH 1999*, 335–342.

Turk, G., and O'Brien, J. F. 2002. Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics 21*, 4 (Oct.), 855–873.

Yngve, G., and Turk, G. 2002. Robust creation of implicit surfaces from polygonal meshes. *IEEE Transactions on Visualization and Computer Graphics 8*, 4 (Oct.), 346–359.

Zhao, H.-K., Osher, S., and Fedkiw, R. 2001. Fast surface reconstruction using the level set method. In *IEEE Workshop on Variational and Level Set Methods*, 194–202.

A Analytical Line Integrals

Our integrated constraints require solving integrals of the form

$$\int \frac{P(\boldsymbol{p})}{R(\boldsymbol{p})} \mathrm{d}\boldsymbol{p} \tag{20}$$

where $P(\boldsymbol{p})$ and $R(\boldsymbol{p})$ are functions in $\boldsymbol{p}$. The functions P and R are respectively determined by the basis functions and the weighting function. For our choices, P is a constant or linear polynomial, and R is a quadratic polynomial with restricted form. We cannot do the two-dimensional integral analytically, but the one-dimensional line integral one does have an analytic solution.

Once we have selected a direction for the line integration, the integrals for the constant and linear terms of P appear in the following forms:

$$\int_0^a \frac{1}{\left((x+k_1)^2+k_2\right)^2} \mathrm{d}x \tag{21}$$

$$\int_0^a \frac{x}{\left((x+k_1)^2+k_2\right)^2} \mathrm{d}x \tag{22}$$

where k_1 and k_2 are constant with respect to the integration variable, x.

The solutions to these integrals are

$$\beta \frac{-a\sqrt{k_2}(k_1(a+k_1)-k_2)-(k_1^2+k_2)((a+k_1)^2+k_2)}{2(k_2)^{\frac{3}{2}}(k_1^2+k_2)((a+k_1)^2+k_2)} \tag{23}$$

and

$$\beta \frac{a\sqrt{k_2}(a+k_1)+k_1((a+k_1)^2+k_2)}{2(k_2)^{\frac{3}{2}}((a+k_1)^2+k_2)} \tag{24}$$

respectively, where

$$\beta = \left(\tan^{-1}\left(\frac{k_1}{\sqrt{k_2}}\right) - \tan^{-1}\left(\frac{a+k_1}{\sqrt{k_2}}\right)\right) \quad . \tag{25}$$