As-Rigid-As-Possible Shape Manipulation

Takeo Igarashi[1, 3]   Tomer Moscovich[2]   John F. Hughes[2]

[1]The University of Tokyo   [2]Brown University   [3]PRESTO, JST

Abstract

We present an interactive system that lets a user move and deform a two-dimensional shape without manually establishing a skeleton or freeform deformation (FFD) domain beforehand. The shape is represented by a triangle mesh and the user moves several vertices of the mesh as constrained handles. The system then computes the positions of the remaining free vertices by minimizing the distortion of each triangle. While physically based simulation or iterative refinement can also be used for this purpose, they tend to be slow. We present a two-step closed-form algorithm that achieves real-time interaction. The first step finds an appropriate rotation for each triangle and the second step adjusts its scale. The key idea is to use quadratic error metrics so that each minimization problem becomes a system of linear equations. After solving the simultaneous equations at the beginning of interaction, we can quickly find the positions of free vertices during interactive manipulation. Our approach successfully conveys a sense of rigidity of the shape, which is difficult in space-warp approaches. With a multiple-point input device, even beginners can easily move, rotate, and deform shapes at will.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques – Interaction Techniques; I.3.3 [Computer Graphics]: Picture/Image Generation – Display algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling – Geometric algorithms.

Keywords: Shape Manipulation, Deformation, Image Editing, Mesh Editing, Animation, Interaction

1 Introduction

With a 2D image or drawing at hand, a user might want to manipulate it—move, rotate, stretch, and bend it. The primary application we have in mind is an editing tool for drawing or image-editing systems, but our interactive shape manipulation technique is also useful in various applications such as real-time live performance [Ngo et al. 2000] and enriching graphical user interfaces [Bruce and Calder 1995].

One popular approach for shape manipulation is to use a pre-defined skeleton. The user manipulates the skeleton configuration and the system adjusts the overall shape relative to the skeleton. However, defining a skeleton structure for a shape is not a trivial task [Lewis et al. 2000] and is not effective for objects, such as jellies, that lack an obvious jointed structure. Another popular method is free-form deformation (FFD) [MacCracken and Joy 1996] in which the user explicitly divides the space into several domains and manipulates each domain by moving control points defining it. But setting FFD domains is tedious and the user must laboriously manipulate many control vertices.

This paper presents an interactive system that allows the user to manipulate a shape without using a skeleton or FFD. The user chooses several points inside the shape as handles and moves each handle to a desired position. The system then moves, rotates, and deforms the overall shape to match the given handle positions while minimizing distortion. By taking the interior of the shape into account, our approach can model its rigidity (i.e., internal resistance to deformation), making the result much closer to the behavior of real-world objects than in space-warp approaches as in [Barrett and Cheney 2002; Llamas et al. 2003].

We use a two-step closed-form algorithm for finding the shape configuration that minimizes distortion. The typical approach is to use a physically based simulation or nonlinear optimizations [Sheffer and Kraevoy 2004], but these techniques are too slow for interactive manipulation. A key aspect of our approach is the design of a quadratic error metric so that the minimization problem is formulated as a set of simultaneous linear equations. Our system solves the simultaneous equations at the beginning, and can therefore quickly find a solution during interaction. Ideally we would like a single quadratic error function that handles all properties of a shape, but no such function exists (see Appendix A). We therefore split the problem into a rotation part and a scale part. This divides the problem into two least-squares minimization problems that we can solve sequentially. This method can be seen as a variant of the method proposed by Sorkine et al. [2004].

Our technique can be useful in standard dragging operations with a mouse, but it is particularly interesting when using a multiple-point input device such as a SmartSkin touchpad [Rekimoto 2002] (Figure 1). With such a device, one can interactively move, rotate, and deform an entire shape as if manipulating a real object using both hands. This is difficult with existing shape deformation tools because most allow only local modification while the overall position and orientation of the shape remain fixed.

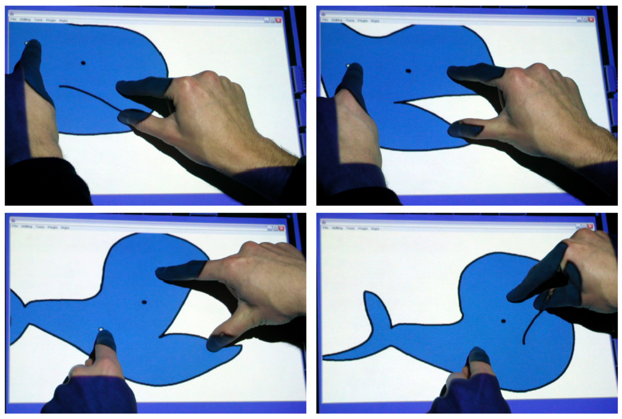

Figure 1: Shape manipulation using a SmartSkin touchpad. The user can interactively move, rotate, and deform the shape using both hands as if manipulating a real object.

2 Related Work

Shape manipulation techniques fall roughly into two categories. One is to deform the space in which the target shape is embedded; the other is to deform the shape while taking its structure into account.

Deformation using skeletons, FFD, and other space-warp approaches belong to the first category. With skeletons, each point in the shape is associated with a coordinate frame defined by a bone [Lewis et al. 2000]. In FFD, each point is associated with a closed region in a FFD grid [MacCracken and Joy 1996]. Other space warp techniques deform the global space [Milliron et al. 2002]. Beier and Neely used space deformation for morphing [1992]. Twister deforms the global space according to two 6-DOF controls [Llamas et al. 2003], and Barrett and Cheney [2002] used space-warp deformation for digital image editing. Brookstein [1989] used thin-plate splines to find a space deformation that is defined by several control points. A drawback of these approaches is that they model the rigidity of the ambient space, rather than that of the shape itself, and thus the resulting deformation differs greatly from the behavior of real objects.

The second category includes physically based methods, the most popular of which are mass-spring models [Gibson and Mirtich 1997]. These are very easy to implement, but their behavior is too elastic for many applications and they often converge slowly. In addition, careful parameter tuning is required to make them really work. More physically accurate simulation is possible with finite-element methods [Celniker and Gossard 1991], but these are very complicated and expensive to solve, making them inappropriate for interactive manipulation of simple drawings. The ArtDefo system [James and Pai 1999] achieved physically accurate, interactive shape deformation using boundary-elements, but it is limited to very small deformations such as poking the surface and is not applicable to large deformations like bending an arm.

The work presented here belongs to the second category. Our goal is to introduce internal model rigidity into shape manipulation. However, instead of using physically based models, we use simple geometric approach similar to a technique used in [Alexa et al. 2000]. They obtain an as-rigid-as-possible interpolation between shapes by computing a rigid transformation for each triangle element geometrically and stitching them together. Similarly, we achieve as-rigid-as-possible manipulation by geometrically minimizing the distortion associated with each triangle in a mesh. Sheffer and Kraevoy [2004] introduce a similar deformation tool, but use an iterative computation that is too expensive for interactive manipulations, especially when the control vertices move quickly.

The algorithm we use can be seen as a variant of the Laplacian surface-editing method proposed by Sorkine et al. [2004]. They achieved fast detail-preserving deformation by using rotation- and scale-invariant Laplacian coordinates. They also proposed scaling the Laplacians of the deformed shape back to their original scale and re-solving. Similarly, we add a scale-preserving effect to the initial deformation process. We show that this scale-preservation effect makes possible more dynamic manipulation than is seen in their paper's examples, where the user fixes most of the shape and moves only a specific region of interest.

3 Overview

We start with an overview of the system to establish a context for the core algorithms described in the next section.

The user first imports a 2D shape—represented either by vector graphics or a bitmap image—into the system. The only requirement is that the boundary of the shape can be represented as a simple closed polygon. For bitmap images, we currently manually remove backgrounds and apply automatic silhouette tracing using the marching squares algorithm. The system then generates a triangulated mesh inside the boundary. Various triangulation methods are available [Shewchuk 1996], but better manipulation results are achieved by using near-equilateral triangles of similar sizes across the region. We use a particle-based algorithm to obtain such a mesh [Markosian 1999]. Starting with a standard constrained Delaunay triangulation, the system iteratively refines the mesh by adjusting vertex positions and mesh connectivity. To work at interactive rates, it is important that the mesh not be too large. Our current implementation generates meshes with 100-300 vertices within a few seconds. The resulting triangulation is registered as the "rest shape," and the system performs a pre-computation (which we call "registration") to accelerate the computation during the interaction (Figure 2a).

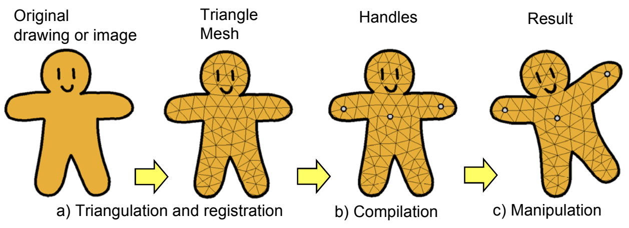

Figure 2: Overview of the system. The system first triangulates the original shape, and performs some pre-computation. The user adds handles. Moving the handles results in a fast deformation.

The user manipulates the shape by indicating handles on the shape and then interactively moving the handles (Figure 2b,c). The user clicks on the shape to place handles and drags the handles to move them. We currently let the user place handles only at existing mesh vertices. Ideally, the system would allow the user to put handles at arbitrary locations and modify the mesh structure to include the handle. We plan to incorporate such a re-meshing mechanism into the system in the future.

Our system also supports multiple-point input devices. We currently use SmartSkin [Rekimoto 2002], which can track multiple fingers touching its surface. By projecting the drawing onto the SmartSkin we bring the user's fingers into direct correspondence with the constraint points (Figure 1). This lets the user grasp and manipulate the drawing as if manipulating a real-world object. We are also testing a Wacom tablet with two orientation-sensitive mice. In this case, each mouse is associated with a couple of vertices which are moved and rotated by the mouse (Figure 16). The interface has some similarity to Twister [Llamas et al. 2003], but the operation using our system is more like manipulation of physical objects while their system is designed for model construction.

Shape manipulation is first applied to the triangle mesh; the system then maps the original drawing or image from the original mesh to the deformed mesh. When manipulating vector graphics, we use the barycentric coordinates of each vertex within the corresponding triangle of the mesh. When manipulating a bitmap image, we simply use standard linear texture mapping.

The system performs additional pre-computations when new handles are added or removed (Figure 2b). We call this process "compilation" because this process actually prepares a function that takes the handle configuration as input and returns the resulting shape as output. During interaction, the system repeatedly sends the updated handle configuration to this function.

4. Algorithm

The input to the algorithm is the set of all *xy*-coordinates of the constrained mesh vertices (Figure 3a) and the output is the *xy*-coordinates of the remaining free vertices that minimize the distortion associated with all triangles in the resulting mesh (Figure 3d). The central challenge is to find an appropriate definition for the distortion of an individual triangle. Our strategy is to design an error metric that is quadratic in its free variables so the system can solve the minimization problem as a simple matrix computation.

Ideally we would like a single quadratic error function that appropriately represents overall distortion. We have examined various possibilities, but finally concluded that it is impossible to design such a function (see Appendix A). Our solution is to split the problem into a rotation part and a scale part so that each part is handled by an independent quadratic error function. With this decomposition, we can obtain the final result by sequentially solving two least-squares problems.

Given the coordinates of the constrained vertices, the first step generates an intermediate result by minimizing an error metric that prevents shearing and non-uniform stretching but permits rotation and uniform scaling (Figure 3a). The second step takes this result and adjusts the scale of each triangle. This second step is further decomposed into two sequential processes. The system first fits each original triangle to the corresponding intermediate triangles without changing scale (Figure 3b), and then computes the final result by minimizing an error metric that represents the difference between the fitted triangle and the resulting triangles (Figure 3c). The following subsections describe each step in detail.

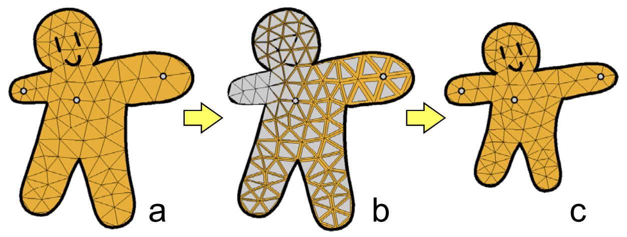

Figure 3: Overview of the algorithm. For the given handle configuration, the system first generates an intermediate result by minimizing conformal (i.e., scale-independent) distortion (a). The system then fits triangles from the rest shape to corresponding triangles in the intermediate result (b). The system generates a final result (c) by minimizing the difference between the fitted triangles and the corresponding triangles.

4.1 Step one: scale-free construction

Step one generates an intermediate result by minimizing an error function that allows rotation and uniform scaling. The input is the *xy*-coordinates of the constrained vertices and the output is the *xy*-coordinates of the remaining free vertices. Note that this algorithm does not use the previous result as an initial configuration, as do physically-based simulation or relaxation methods. Instead, we provide a closed-form solution for the problem.

This step corresponds to the 2D case in Laplacian editing [Sorkine et al. 2004]. Our formulation is slightly different in that we use a triangle mesh rather than the boundary and we assign quadratic error functionals to each individual triangle rather than each vertex. We believe that our formulation is slightly easier to implement but their formulation can certainly be used instead of ours in this step.

The error function for a deformed triangle $\{v_0', v_1', v_2'\}$ is defined as follows (Figure 4). For the corresponding triangle in the rest shape $\{v_0, v_1, v_2\}$, the system first computes relative coordinates $\{x_{01}, y_{01}\}$ of v_2 in the local coordinate frame defined by v_0 and v_1 (R_{90} denotes rotation counterclockwise by 90 degrees):

$$v_2 = v_0 + x_{01}\overrightarrow{v_0v_1} + y_{01}R_{90}\overrightarrow{v_0v_1} \tag{1}$$

Given v_0', v_1', x_{01}, and y_{01}, the system can compute the desired location for v_2'.

$$v_2^{desired} = v_0' + x_{01}\overrightarrow{v_0'v_1'} + y_{01}R_{90}\overrightarrow{v_0'v_1'} \text{ where } R_{90} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{2}$$

The error associated with v_2' is then represented as

$$E_{\{v_2\}} = \left\| v_2^{\text{desired}} - v_2' \right\|^2 \tag{3}$$

We can define v_0^{desired} and v_1^{desired} similarly, and we define the error associated with the triangle as

$$E_{\{v_0,v_1,v_2\}} = \sum_{i=1,2,3} \left\| v_i^{\text{desired}} - v_i' \right\|^2 \tag{4}$$

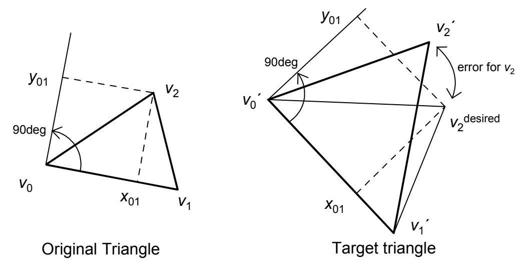

Figure 4: Error metric used in step one. v_2^{desired} is obtained by fitting the original triangle to the target triangle by translation, rotation, and scaling so that v_0' and v_1' match v_0 and v_1.

The error for the entire mesh is simply the sum of errors for all triangles in the mesh. Since the error metric is quadratic in $\mathbf{v}' = (v_{0x}', v_{0y}', \ldots, v_{nx}', v_{ny}')^{\mathrm{T}}$, we can express it in matrix form:

$$E_{1\{\mathbf{v}'\}} = \mathbf{v}'^{\mathrm{T}}\mathbf{G}\mathbf{v}' \tag{5}$$

The minimization problem is solved by setting the partial derivatives of the function $E_{1\{\mathbf{v}'\}}$ with respect to the free variables $\mathbf{u} = (u_{0x}, u_{0y}, \ldots, u_{mx}, u_{my})^{\mathrm{T}}$ in $\mathbf{v}'$ to zero. By reordering $\mathbf{v}'$ to put the free variables first we can write $\mathbf{v}'^{\mathrm{T}} = (\mathbf{u}^{\mathrm{T}}\,\mathbf{q}^{\mathrm{T}})$ where $\mathbf{q}$ represents the constrained vertices. This gives us

$$E_1 = \mathbf{v}'^{\mathrm{T}}\mathbf{G}\mathbf{v}' = \begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix}^{\mathrm{T}} \begin{bmatrix}\mathbf{G_{00}} & \mathbf{G_{01}}\\ \mathbf{G_{10}} & \mathbf{G_{11}}\end{bmatrix} \begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix} \tag{6}$$

$$\frac{\partial E_1}{\partial \mathbf{u}} = (\mathbf{G}_{00} + \mathbf{G}_{00}^{\mathrm{T}})\mathbf{u} + (\mathbf{G}_{01} + \mathbf{G}_{10}^{\mathrm{T}})\mathbf{q} = \mathbf{0} \tag{7}$$

We rewrite this as

$$\mathbf{G}'\mathbf{u} + \mathbf{B}\mathbf{q} = \mathbf{0} \tag{8}$$

Note that $\mathbf{G}'$ and $\mathbf{B}$ are fixed and only $\mathbf{q}$ changes during manipulation. Therefore, we can obtain $\mathbf{u}$ by simple matrix multiplication by pre-computing $\mathbf{G}'^{-1}\mathbf{B}$ at the beginning. $\mathbf{G}'$ is a $2m\times2m$ sparse, symmetric matrix with approximately 12 entries per column, because of the near-equilateral structure of the mesh.

Computing the solution in step one, as shown in Figure 3a, is very fast; it requires only one matrix multiplication during interaction. Step one generates reasonable results as long as the distances between handles are close to their distances in the rest shape, as shown in [Sorkine et al. 2004]. For example, one can successfully translate or rotate the shape using this step alone. However, since the error function does not capture changes in scale, the shape inflates as the handles move away from each other and shrinks as they approach each other. We fix this problem in step two.

4.2 Step two: scale adjustment

This step takes the intermediate result from step one (the *xy*-coordinates of all vertices) as input and returns the final result (updated *xy*-coordinates of the free vertices) by adjusting the scale of the triangles in the mesh (Figure 3b, c).

4.2.1 Fitting the original triangle to the intermediate triangle

The system first fits each triangle in the rest shape to the corresponding triangle in the intermediate result, allowing rotation and translation but not shearing or scaling (Figure 3b). There are a couple of methods for this sort of fitting; we use the following method in our current implementation.

Given a triangle $\{v_0', v_1', v_2'\}$ in the intermediate result and corresponding triangle in the rest shape $\{v_0, v_1, v_2\}$, the first problem is to find a new triangle $\{v_0^{\text{fitted}}, v_1^{\text{fitted}}, v_2^{\text{fitted}}\}$ that is congruent to $\{v_0, v_1, v_2\}$ and minimizes the following functional (Figure 5):

$$E_{\text{f}\{v_0^{\text{fitted}}, v_1^{\text{fitted}}, v_2^{\text{fitted}}\}} = \sum_{i=1,2,3} \left\| v_i^{\text{fitted}} - v_i' \right\|^2 \tag{9}$$

Since it is difficult to obtain such a result directly, we approximate it by first minimizing the error allowing uniform scaling and then adjusting the scale afterwards.

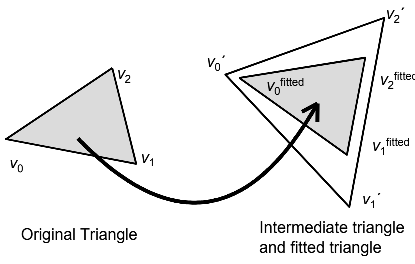

Figure 5: Fitting the original triangle to the intermediate triangle by translation and rotation.

Using the coordinates x_{01} and y_{01} defined in Section 4.1, we can express v_2^{fitted} using v_0^{fitted} and v_1^{fitted}:

$$v_2^{\text{fitted}} = v_0^{\text{fitted}} + x_{01}\overrightarrow{v_0^{\text{fitted}} v_1^{\text{fitted}}} + y_{01}R_{90}\overrightarrow{v_0^{\text{fitted}} v_1^{\text{fitted}}} \tag{10}$$

so the fitting functional becomes a function of just the coordinates of v_0^{fitted} and v_1^{fitted}, a quadratic in the four free variables of $\mathbf{w}=(v_{0\text{x}}^{\text{fitted}}, v_{0\text{y}}^{\text{fitted}}, v_{1\text{x}}^{\text{fitted}}, v_{1\text{y}}^{\text{fitted}})^{\text{T}}$. We can minimize E_{f} by setting the partial derivatives of E_{f} over the four free variables to zero. The result is an easily-solved 4×4 linear system. In matrix form,

$$\frac{\partial E_{\text{f}}}{\partial \mathbf{w}} = \mathbf{F}\mathbf{w} + \mathbf{C} = \mathbf{0} \tag{11}$$

$\mathbf{F}$ is fixed for a given mesh and $\mathbf{C}$ is defined by the result of step one. Therefore, we compute F and invert it during registration. By solving this equation, we obtain a newly fitted triangle $\{v_0^{\text{fitted}}, v_1^{\text{fitted}}, v_2^{\text{fitted}}\}$ that is similar to the original triangle $\{v_0, v_1, v_2\}$. We make it congruent simply by scaling the fitted triangle by the factor of $\|v_0^{\text{fitted}}\text{-}v_1^{\text{fitted}}\| / \|v_0\text{-}v_1\|$. We apply this fitting operation to all triangles in the mesh. Note that each vertex of the original mesh appears in several triangles and hence corresponds to multiple vertices in the fitting triangles (gray triangles in Figure 3b). Reconciling these distinct locations is the sole remaining task.

4.2.2 Generating the final result using the fitted triangles

The system now computes the final *xy*-coordinates of the free vertices for given *xy*-coordinates of the constrained vertices by minimizing the difference between the resulting triangle in the mesh and the fitted triangle (Figure 2d). Note that we use only the fitted triangles here and no longer need the intermediate mesh. This process is very similar to the assembly process in [Alexa et al. 2000; Sumner and Popovic 2004; Yu et al. 2004].

We again begin the explanation with the single triangle $\{v_0, v_1, v_2\}$ (Figure 6). Given the corresponding fitted triangle $\{v_0^{\text{fitted}}, v_1^{\text{fitted}}, v_2^{\text{fitted}}\}$, we define a quadratic error function by

$$E_{2\{v_0'', v_1'', v_2''\}} = \sum_{(i,j)\in\{(0,1),(1,2),(2,0)\}} \left\| \overrightarrow{v_i'' v_j''} - \overrightarrow{v_i^{\text{fitted}} v_j^{\text{fitted}}} \right\|^2 \tag{12}$$

Note that we associate an error with each edge, not each vertex. That is, we use the rotation of the fitted triangle and ignore its position. The translation is solved for as a side effect only. The error is clearly minimized when the triangles $\{v_0'', v_1'', v_2''\}$ and $\{v_0^{\text{fitted}}, v_1^{\text{fitted}}, v_2^{\text{fitted}}\}$ are identical. But since the vertex v_0'', for instance, may lie in several triangles, the optimal position for v_0'' will be some average of the positions desired by each triangle in which it appears.

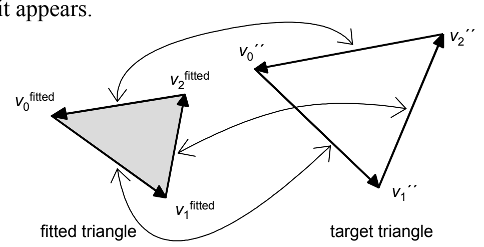

Figure 6: Error metric used in step two. This metric measures the difference between the edge vectors of the fitted triangle and those of the target triangle.

The error for the entire mesh can be represented in a matrix form:

$$E_{2\{v''\}} = \mathbf{v}''^{\text{T}}\mathbf{H}\mathbf{v}'' + \mathbf{f}\mathbf{v}'' + c \tag{13}$$

Note that $\mathbf{H}$ is defined by the connectivity of the original mesh and is independent of the fitted triangles, while $\mathbf{f}$ and $\mathbf{c}$ are determined by the fitted triangles and thus change during interaction.
We minimize E_2 by setting the partial derivatives of E_2 over free vertices $\mathbf{u}$ to zero. By reordering $\mathbf{v}''$, we can write $\mathbf{v}''^{\text{T}} = (\mathbf{u}^{\text{T}}\ \mathbf{q}^{\text{T}})$. This gives us

$$E_2 = \mathbf{v}''^{\mathrm{T}}\mathbf{H}\mathbf{v}'' + \mathbf{f}\mathbf{v}'' + c = \begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix}^{\mathrm{T}} \begin{bmatrix}\mathbf{H}_{00} & \mathbf{H}_{01}\\ \mathbf{H}_{10} & \mathbf{H}_{11}\end{bmatrix} \begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix} + (\mathbf{f}_0\mathbf{f}_1)\begin{pmatrix}\mathbf{u}\\ \mathbf{q}\end{pmatrix} + c \qquad (14)$$

$$\frac{\partial E_2}{\partial \mathbf{u}} = (\mathbf{H}_{00} + \mathbf{H}_{00}{}^{\mathrm{T}})\mathbf{u} + (\mathbf{H}_{01} + \mathbf{H}_{10}{}^{\mathrm{T}})\mathbf{q} + \mathbf{f}_0 = \mathbf{0} \qquad (15)$$

We rewrite this as

$$\mathbf{H}'\mathbf{u} + \mathbf{D}\mathbf{q} + \mathbf{f}_0 = \mathbf{0} \qquad (16)$$

H′ and **D** are fixed but **q** and $\mathbf{f}_0$ change during manipulation. We therefore pre-compute LU factorization of **H′** at the beginning and solve the equation using it during interaction. Actually, the *x* and *y* components are mutually independent in **H′**, so we can perform the above computation for each component separately. For each component, **H′** is an $m \times m$ sparse, symmetric matrix with approximately 6 entries for each column.

4.3 Algorithm summary

Our algorithm can be summarized as follows.

1 Registration (when a new rest shape is defined)
- 1-1 Construct matrices **G** and **H** using the vertex coordinates in the rest shape.
- 1-2 Construct **F** and invert it for each triangle.

2 Compilation (when handles are added or removed)
- 2-1 Construct **G′** and **B** from **G** and compute $\mathbf{G}'^{-1}\mathbf{B}$.
- 2-2 Construct **H′** and **D** from **H** and construct LU factorization of **H′**.

3 During manipulation (when handles are moved)
- 3-1 Obtain intermediate coordinates for the free vertices as $-\mathbf{G}'^{-1}\mathbf{B}\mathbf{q}$ where **q** represents the coordinates of handles.
- 3-2 Construct **C** for each triangle using the intermediate vertex coordinates. Multiplying $\mathbf{F}^{-1}$ and **C** and adjust its scale to obtain each fitted triangle.
- 3-3 Construct $\mathbf{f}_0$ using the fitted triangles and obtain the final result by solving $\mathbf{H}'\mathbf{u}+\mathbf{D}\mathbf{q}+\mathbf{f}_0=0$ using pre-computed LU factorization.

5. Extensions

This section discusses various adjustments necessary to make the system work in practice, as well as other enhancements.

5.1 Collision detection and depth adjustment

We must be careful when different parts of the shape overlap. If we assign depths inappropriately, the overlapping parts can interpenetrate (Figure 7 middle). One problem is that one cannot assign static consistent depth values that work for all possible deformations. Figure 8 shows a simple case. Suppose we had continuous depth values across the shape. Given three points *a*, *b*, and *c*, we can assume that *a*'s depth < *b*'s depth < *c*'s depth. If we bring vertex *b* between *a* and *c* by deformation, there must exist an edge where one vertex is deeper than *b* and the other is shallower. This produces the undesirable artifact in Figure 8.

Our approach is to dynamically adjust depth during interaction (Figure 7 right). We continuously monitor the mesh for self-intersection and assign appropriate depth values to the overlapping parts. This process is similar to the hidden-line removal algorithm in [Hornung 1984]. We first locate boundary intersections and then search for overlapping regions starting from those boundary intersections. Determining the depth order of the overlapping regions is still an open problem. We currently use a statically predefined order as a starting point. We have also implemented a very simple mechanism to achieve frame-to-frame coherence, but it is still in preliminary form.

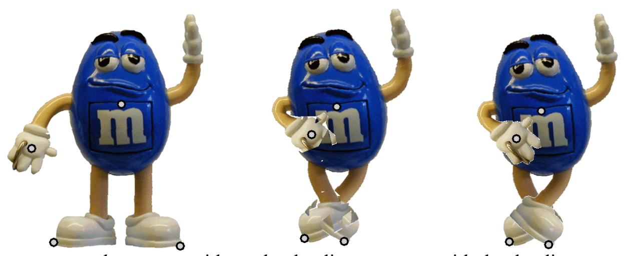

Figure 7: Collision detection and depth adjustment. Without appropriate depth assignment, one can see interpenetration (center). We detect overlapping regions and adjust depth on the fly.

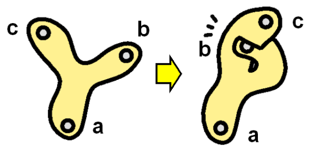

Figure 8: Limitation of static depth assignment.

5.2 Weights for controlling rigidity

Because our algorithm minimizes a simple error functional, adding different weights to different parts of the mesh is straightforward. We do this to control the local rigidity of the shape. We provide a painting interface in which the user can make certain parts stiffer than others. We currently use a weight of 10000 for the painted triangles and 1 for the others. This is useful in preventing important features such as a head from being distorted (Figure 9). It is possible to enforce perfect rigidity by reducing the number of free variables in the minimization, thus obtaining similar results, but we found that weighting is more flexible, and produces more pleasing results under extreme distortions..

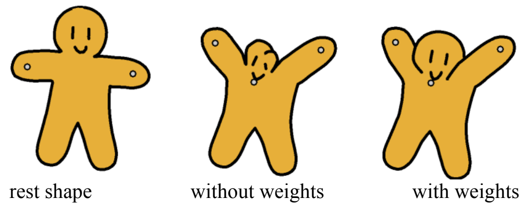

Figure 9: Adjusting rigidity with weights. By adding extra weights to important regions, one can prevent undesirable distortion.

5.3 Animations

Our shape manipulation technique is useful in making 2D animations, for example by setting the character shape at each key frame. This is especially helpful when one wants to animate drawings with detailed surface decoration, since drawing them manually is tedious. It also makes it practical to make animations by deforming photographed objects.

Another approach is to use it for performance-driven animation. By manipulating multiple control vertices simultaneously, one can make interesting animations by recording live performances. We are currently testing a technique similar to Ngo et al.'s system [2000] and facial animation example in [Lewis et al. 2000]. The user first sets a pose by manipulating control vertices and associates the control vertex configuration with a specific point on the canvas. After setting several such key poses, the user can direct multiple control vertices simultaneously by simply moving a control cursor. The system computes an appropriate control vertex configuration by interpolating nearby key poses using a radial basis function (Figure 10).

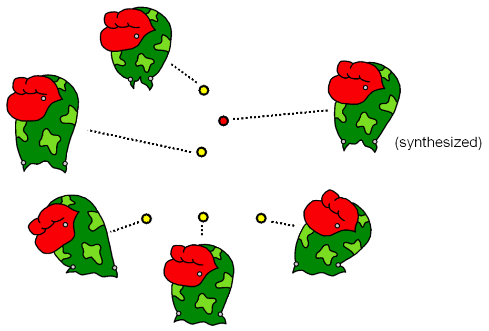

Figure 10: Designing performance-driven animation using spatial keyframing. The users specify a set of key poses (yellow mark) by manipulating handles. They can then manipulate the entire body by dragging a control cursor (red mark). The system blends nearby poses using a radial basis function.

The resulting motion is very smooth and lively because the user's own hand movement appears in the resulting animation. In addition, it is much easier and faster than traditional temporal keyframing using existing shape deformation techniques.

5.4 As-rigid-as-possible curve editing

We have also applied our two-step algorithm to curve editing. For curve editing, we take a polyline instead of a triangle mesh as input. We first apply 2D Laplacian curve editing [Sorkine et al. 2004] and then apply our scale adjustment procedure to the result. Figure 11 shows an example operation. Without scale adjustment, the curve grows and shrinks freely. With scale adjustment, the curve behaves as if it is rigid.

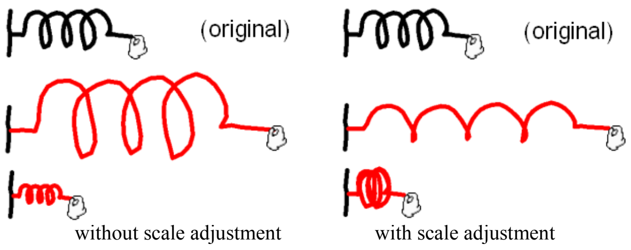

Figure 11: Curve editing with and without scale-adjustment procedure. Without scale adjustment, the stretched region grows and the squashed region shrinks.

To let the user adjust the influence region dynamically during interaction, we introduce a peeling interface in which the influence region grows as the user drags the curve father away (Figure 12). This frees the user from specifying an influence region beforehand and makes the interaction very intuitive. As an option, we also allow the user to explicitly specify the influence region by putting virtual pushpins along the curve before dragging. A similar grab-and-pull curve editing tool is used in Macromedia Flash, but it allows only local changes and does not let the user change the influence region.

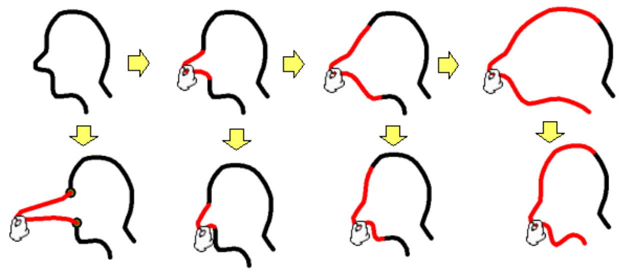

Figure 12: Curve editing with a peeling interface. As the user pulls the curve further away, the influence region grows (left to right). The user can also explicitly specify the region beforehand (bottom left).

6. Results

We have applied our technique to various drawings and images. Figure 13 shows the manipulation of arms and legs by controlling the end points. This is similar to the pin-and-drag interface for articulated characters [Yamane and Nakamura 2003], but our system works with no explicit skeleton structures. Figure 14 shows manipulation by controlling the internal points. Note that the arms and legs are displaced appropriately due to the rigidity of the body. By contrast, a mass-spring model can take some time to propagate the effect to the entire body. Figure 15 shows manipulation of a shape that lacks articulated structure; note that the shape is stretched and squashed appropriately. Figure 1 and 16 show applications of our technique to images. The natural deformation effects give the feel of manipulating real 3D objects.

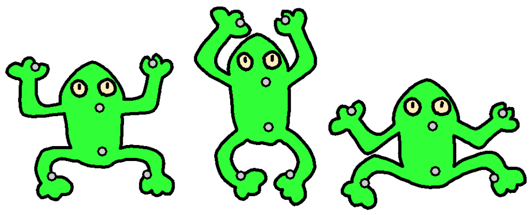

Figure 13: Manipulation of a shape by controlling the end points.

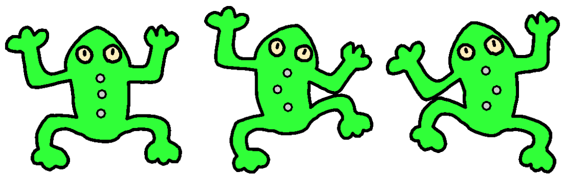

Figure 14: Manipulation of a shape by controlling the internal points. The user moves the handle at the center horizontally and the entire body is deformed appropriately.

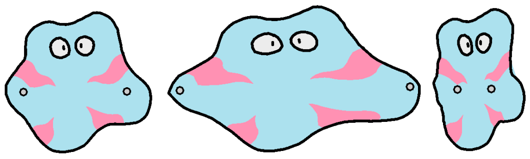

Figure 15: Stretch and squash of a non-articulated shape.

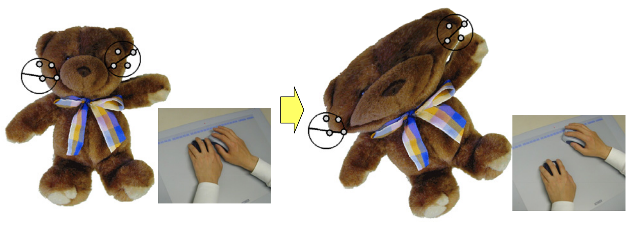
Figure 16: Manipulation of an image using two rotation sensitive mice.

Figure 1 shows a snapshot of operation using a SmartSkin multi-point touchpad [Rekimoto 2002]. Test users found it easy to bend and stretch the shapes, and enjoyed experimenting with the shapes and the movements they could produce. Since the deformation is updated in real time and is easy to control, several fingers can be used to steer different parts of the shape and perform simple animations. Two users can work together to create more complex motion.

Table 1 summarizes the performance of the current implementation. This data is for a system running Java 1.4 on a Windows XP notebook PC (Pentium III 1GHz processor and 756 MB of memory). We use a native sparse LU solver [Davis 2003] for matrix computations. We obtained the data by running the corresponding routines 100 times, so these are very rough estimates severely affected by garbage collection and CPU cache. The result indicates that step two is the bottleneck during manipulation. In our experience, one can obtain quite nice results with a very coarse mesh with fewer than 100 vertices (Figure 17 left). All examples in this paper are obtained with similar mesh sizes. In this range, the system shows completely real-time performance and the user experiences no delay. One can obtain smoother results by using a finer mesh but the interaction eventually becomes choppy. The delay becomes obvious at a vertex count of around 300 on our notebook PC (Figure 17 right).

Table 1: Sample running times (milliseconds) for the meshes in Figure 17

# of vertices	Registration		Compilation		Update	
	Step1	Step2	Step1	Step2	Step1	Step2
93	16	18	14	4	0.06	2.2
150	42	38	29	8	0.09	3.5
287	160	107	72	19	0.16	7.5

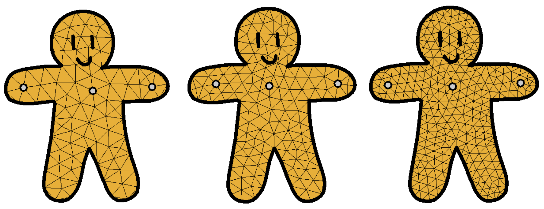
Figure 17: Example triangulations. The number of vertices is 85, 156, 298 respectively and three handles are attached to each.

We have experimentally examined the effect of uneven triangulation on our algorithm and found it to be fairly robust against irregularly spaced mesh. Figure 18 shows an example. The dense and sparse regions are evenly squashed and stretched, and similar behavior is observed for bending. This can be explained as follows. If triangle size is reduced by a factor of 1/n, the distortion associated with each triangle decreases by a factor of $1/n^2$. At the same time, the density of triangles becomes n^2 times higher. As a result, triangle size does not affect the total cost. This does not apply for curve editing, where the density increases only linearly. The error in a dense region is estimated as smaller, thus making the region softer. The simplest solution is to approximate the curve by an evenly spaced polyline and apply our algorithm to the resampled curve.

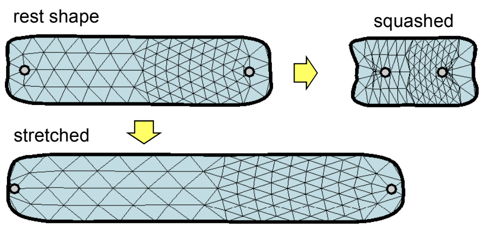

Figure 18: The effect of uneven triangulation. Our algorithm is not strongly affected by the mesh density.

7. Limitations and Future Work

The two-step algorithm introduced here is merely a practical approximation to achieve interactive performance. It works surprisingly well in most cases as shown in our examples, but in some cases its limitations are revealed. Figure 19 shows an example. When the control handle is moved one might expect the result shown on the right, but our algorithm returns the result shown in the middle due to its inherent linear nature. The free vertices only move parallel to the line connecting the constrained vertices. To handle these cases, a more accurate distortion metric similar to the one in [Sheffer and Kraevoy 2004] is probably necessary.

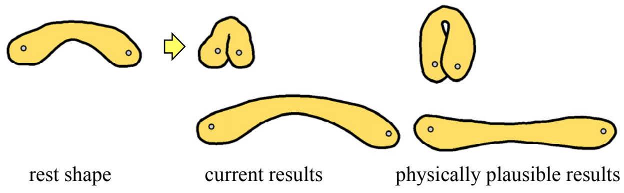

Figure 19: A limitation of our algorithm. If the user stretches a shape (left), the current algorithm returns the result in the middle; the results on the right would be more desirable.

As is discussed in [Alexa et al. 2000], a linear mapping from an original triangle to a deformed triangle ignoring the translation factor can be represented by a 2×2 affine transformation matrix A. Using singular value decomposition (SVD), the matrix A can be represented as a combination of a rotation part R_γ, a shearing part s_h, and a scaling part s_x, s_y [Shoemake and Duff 1992]:

$$A = R_\alpha D R_\beta = (R_\alpha R_\beta)(R_\beta{}^T D R_\beta) = R_\gamma \begin{pmatrix} s_x & s_h \\ s_h & s_y \end{pmatrix} \qquad (17)$$

Given this formulation, one can obtain as-rigid-as-possible mapping by minimizing $|s_h|$, $|s_x\text{-}1|$, and $|s_y\text{-}1|$. We would like to experiment with this by finding a way to minimize these errors directly.

Volume preservation is another feature that the current formulation cannot achieve [Angelidis et al. 2004]. With volume preservation, an object is squashed vertically when it is stretched horizontally. In the decomposition above, volume preservation is simulated by minimizing $|s_x s_y\text{-}1|$. We tried to implement this effect by adjusting the target triangle in the step 2, but the result was not very satisfactory. Our experience suggests that it is necessary to implement volume transfer between triangle elements to obtain a globally convincing result. We plan to explore this path in the future.

We very much want to extend the technique to 3D shapes. The ability to freely move, rotate, and deform a 3D object is very attractive in various applications such as object manipulation in virtual environments. Unfortunately, the extension is not straightforward. We have experimented with various formulations but we have yet to find quadratic error functions equivalent to those we use in 2D, and it seems that we must take a very different approach. We plan to continue our exploration.

Acknowledgements

We would like to thank Jun Rekimoto and Kentaro Fukuchi for providing SmartSkin. We also thank Daniel Keefe and Uruma Delvi for their lovely drawings. Finally we thank the user interface research group at the University of Tokyo and the Brown University graphics group for helpful discussions.

References

ALEXA, M., COHEN-OR, D., and LEVIN, D. 2000. As-Rigid-As-Possible Shape Interpolation. In *Proceedings of ACM SIGGRAPH 2000*, 157-164.

ANGELIDIS, A., CANI, M., WYVILL, G., and KING, S. 2004. Swirling-Sweepers: Constant-Volume Modeling. *Pacific Graphics 2004*, 10-15.

BARRETT, W. A., and CHENEY, A. S. 2002. Object-based Image Editing. *ACM Transactions on Graphics*, *21,* 3, 777-784.

Beier, T. and Neely, S. 1992 Feature-based image metamorphosis, In *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26, 2, 35-42.

Bookstein, F. L. 1989. Principal Warps: Thin-Plate Splines and the Decomposition of Deformations, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11, 6, 567-585.

BRUCE, H. T., and CALDER, P. 1995. Animating Direct Manipulation Interfaces. In *Proceedings of UIST '95*, 3-12.

CELNIKER, G., and GOSSARD, D. 1991. Deformable Curve and Surface Finite Elements for Free-form Shape Design. In *Computer Graphics (Proceedings of ACM SIGGRAPH 91), 25*, 4, 257-266.

DAVIS, T. A. 2003. *Umfpack Version 4.1 User Guide*. Technical report TR-03-008, University of Florida.

GIBSON, S. F., and MIRTICH, B. 1997. *A Survey of Deformable Models in Computer Graphics*. Technical report TR-97-19, Mitsubishi Electric Research Laboratories.

HORNUNG, C. 1984. A Method for Solving the Visibility Problem. *IEEE Computer Graphics and Applications*, *4*, 7, 26-33.

JAMES, D. L., and PAI, D. K. 1999. ArtDefo Accurate Real Time Deformable Objects, In *Proceedings of ACM SIGGRAPH 1999*, 65-72.

LEWIS, J. P., CORDNER, M., and FONG, N. 2000. Pose Space Deformations: A Unified Approach to Shape Interpolation and Skeleton-driven Deformation. In *Proceedings of ACM SIGGRAPH 2000*, 165-172.

LLAMAS, I., KIM, B., GARGUS, J., ROSSIGNAC, J., and SHAW, C. D. 2003. Twister: A Space-Warp Operator for the Two-Handed Editing of 3D Shapes. *ACM Transactions on Graphics*, *22,* 3, 663–668.

MACCRACKEN, R., and JOY, K.I. 1996. Free-form Deformations with Lattices of Arbitrary Topology. In *Proceedings of ACM SIGGRAPH 1996*, 181-188.

MARKOSIAN, L., COHEN, J.M., CRULLI, T., and HUGHES, J.F. 1999. Skin: a Constructive Approach to Modeling Free-form Shapes. In *Proceedings of ACM SIGGRAPH 1999*, 393-400.

MILLIRON, T., JENSEN, R., BARZEL, R. and FINKELSTEIN, A. 2002. A Framework for Geometric Warps and Deformations. *ACM Transactions on Graphics*, *21*, 1, 20-51.

NGO, T., CUTRELL, D., DANA, J., DONALD, B., LOEB, L. and ZHU, S. 2000. Accessible Animation and Customizable Graphics via Simplicial Configuration Modeling. In *Proceedings of ACM SIGGRAPH 2000*, 403-410.

REKIMOTO, J. 2002. SmartSkin: An Infrastructure for Freehand Manipulations on Interactive Surfaces. In *Proceedings of CHI'02,* 113-120.

SHEWCHUK, J.R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Proceedings of First Workshop on Applied Computational Geometry*, 124-133.

SHEFFER, A., and KRAEVOY, V. 2004. Pyramid Coordinates for Morphing and Deformation. In *Proceedings of 3D Data Processing, Visualization, and Transmission, 2nd International Symposium on (3DPVT'04),* 68-75.

Shoemake K. and Duff, T. 1992. Matrix Animation and Polar Decomposition. In *Proceedings of Graphics Interface '92*, 258-264.

SORKINE, O., COHEN-OR, D., LIPMAN, Y., ALEXA, M., ROSSL, C., and SEIDEL, H. 2004. Laplacian Surface Editing. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, 179-188.

SUMNER, R. W., and POPOVIC, J. 2004. Deformation Transfer for Triangle Meshes, *ACM Transaction on Graphics*, 23, 3, 399-405.

YAMANE, K., and NAKAMURA, Y. 2003. Natural Motion Animation Through Constraining and Deconstraining at Will. *IEEE Transaction on Visualization and Computer Graphics*, *9*, 3, 352-360.

YU, Y., ZHOU, K., XU, D., SHI, X., BAO, H., GUO, B. and SHUM, H. Y. 2004. Mesh Editing with Poisson-Based Gradient Field Manipulation. *ACM Transactions on Graphics*, 23, 3, 641-648.

A Proof of nonexistence of quadratic error metric

We claimed that there is no quadratic function of the locations of the deformed mesh vertices that measures the distortion from the original mesh, in the sense that it is minimized exactly when the deformed mesh is oriented congruent to the original.
The proof is by contradiction: Suppose there is such a measure, and apply it to a one-triangle mesh with an original triangle ((0,0), (1,0), (0,1)) and a deformed triangle ((0,0), (x,y), (u,v)). This gives a function $T(x,y,u,v)$ that is quadratic in the variables x,y,u,v. By subtracting the constant term, we can get a new quadratic with no constant term, but with exactly the same minima. Because rotation through 180 degrees is an orientation preserving congruence, we know that $T(x,y,u,v) = T(-x,-y,-u,-v)$ for all (x,y,u,v). This means that T can be written with no linear terms. We already removed the constant term, so T is "pure quadratic" (i.e., all terms have total exponent two).
The function T must be "positive semidefinite" (i.e., $T(x,y,u,v) >= 0$ for all (x,y,u,v)), because if $T(x_0,y_0,u_0,v_0) = K < 0$, then $T(tx_0,ty_0,tu_0,tv_0) = t^2K < 0$; as we increase t, the values of T become arbitrarily negative; hence T has no minimum value, which contradicts the hypothesis.
However, for such a function, the value at (0,0,0,0) is zero, and this is certainly the global minimum. But (0,0,0,0) represents a degenerate triangle that is not congruent to our chosen one. This is a contradiction.