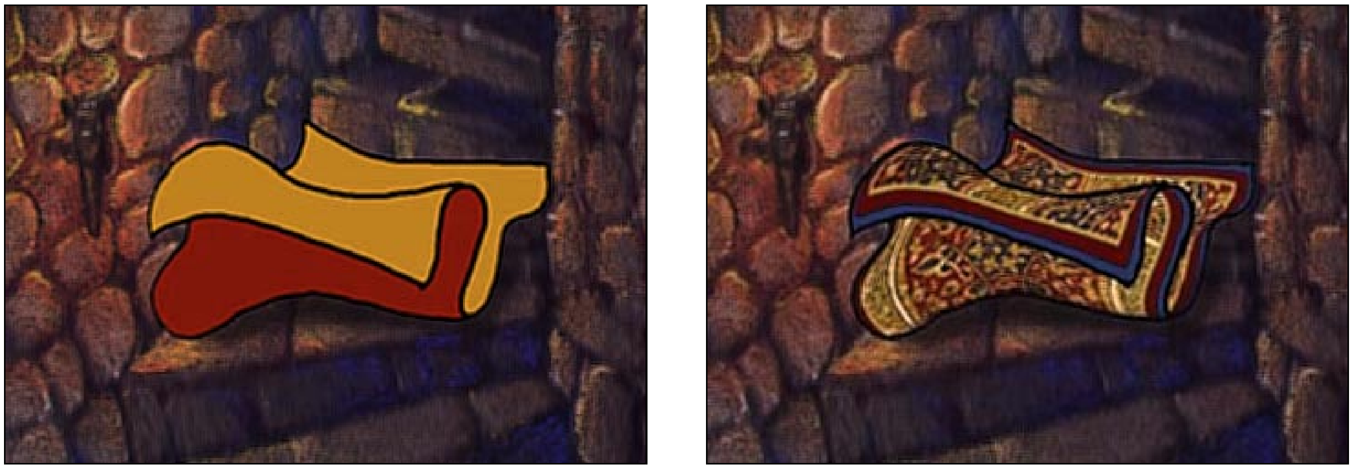
Texture Mapping for Cel Animation

Wagner Toledo Corrêa[1] Robert J. Jensen[1] Craig E. Thayer[2] Adam Finkelstein[1]

[1] Princeton University
[2] Walt Disney Feature Animation



(a) Flat colors (b) Complex texture

Figure 1: A frame of cel animation with the foreground character painted by (a) the conventional method, and (b) our system.

Abstract

We present a method for applying complex textures to hand-drawn characters in cel animation. The method correlates features in a simple, textured, 3-D model with features on a hand-drawn figure, and then distorts the model to conform to the hand-drawn artwork. The process uses two new algorithms: a silhouette detection scheme and a depth-preserving warp. The silhouette detection algorithm is simple and efficient, and it produces continuous, smooth, visible contours on a 3-D model. The warp distorts the model in only two dimensions to match the artwork from a given camera perspective, yet preserves 3-D effects such as self-occlusion and foreshortening. The entire process allows animators to combine complex textures with hand-drawn artwork, leveraging the strengths of 3-D computer graphics while retaining the expressiveness of traditional hand-drawn cel animation.

CR Categories: I.3.3 and I.3.7 [Computer Graphics].

Keywords: Cel animation, texture mapping, silhouette detection, warp, metamorphosis, morph, non-photorealistic rendering.

1 INTRODUCTION

In traditional cel animation, moving characters are illustrated with flat, constant colors, whereas background scenery is painted in subtle and exquisite detail (Figure 1a). This disparity in rendering quality may be desirable to distinguish the animated characters from the background; however, there are many figures for which complex textures would be advantageous. Unfortunately, there are two factors that prohibit animators from painting moving characters with detailed textures. First, moving characters are drawn differently from frame to frame, requiring any complex shading to be replicated for every frame, adapting to the movements of the characters—an extremely daunting task. Second, even if an animator were to re-draw a detailed texture for every frame, temporal inconsistencies in the painted texture tend to lead to disturbing artifacts wherein the texture appears to “boil” or “swim” on the surface of the animated figure. This paper presents a method for applying complex textures to hand-drawn characters (Figure 1b). Our method requires relatively little effort per frame, and avoids boiling or swimming artifacts.

In recent years, the graphics community has made great progress in 3-D animation, leading up to full-length feature animations created entirely with 3-D computer graphics. There are advantages to animating in 3-D rather than 2-D: realism, complex lighting and shading effects, ease of camera motion, reuse of figures from scene to scene, automatic in-betweening, and so forth. Furthermore, it is easy to apply complex textures to animated figures in 3-D. Thus, one might consider using 3-D computer graphics to create *all* animated figures, or at least the characters that have interesting textures. However, it turns out that there are several reasons why hand-drawn 2-D animation will not be replaced with computer-generated 3-D figures. For traditional animators, it is much easier to work in 2-D rather than in 3-D. Hand-drawn animation enjoys an *economy of line* [14], where just a few gestures with a pen can suggest life and emotion that is difficult to achieve by moving 3-D models. Finally, there exists an entire art form (and industry) built around hand-drawn animation, whose techniques have been refined for more than 80 years [27]. While the industry is increasingly using computer-generated elements in animated films, the vast majority of characters are hand-drawn in 2-D, especially when the figure should convey a sense of life and emotion.

In this project, we begin with hand-drawn characters created by a traditional animator. Next, a computer graphics animator creates a crude 3-D model that mimics the basic poses and shapes of the hand-drawn art, but ignores the subtlety and expressiveness of the character. The 3-D model includes both a texture and the approximate camera position shown in the artwork. Our algorithm distorts the model within the viewing frustum of the camera, in such a way that the model conforms to the hand-drawn art, and then renders the model with its texture. Finally, the rendered model replaces the flat colors that would be used in the ink-and-paint stage of traditional cel animation. This process combines the advantages of 2-D and 3-D animation. The critical aspects of the animation (gestures, emotion, timing, anticipation) are created in 2-D with the power and expressiveness of hand-drawn art; on the other hand, effects that give shape to the texture occur in 3-D, yielding plausible motion for the texture. The hand-drawn line art and the texture are merged for the final animation.

To implement the process outlined above, this paper offers two new algorithms: a silhouette detection scheme and a depth-preserving warp. The silhouette detector (which is based on the frame buffer) is efficient, simple to implement, and can be used for any application where visible silhouette detection is necessary. The warp has two main advantages over previous warps that make it appropriate for our problem: it works with curved features, and it provides 3-D effects such as wrapping, foreshortening, and self-occlusion. It also has other potential applications: texture acquisition, and manipulation of 3-D objects using hand-drawn gestures. (These applications are described in Section 9.)

The remainder of this paper is organized as follows. Section 2 surveys previous work related to this project. Section 3 presents an overview of our process. In Sections 4, 5, and 6, we describe in detail how we correlate features on the model with features on the art, how our system warps the model to conform to the art, and how to control the warp. In Section 7, we present some resulting animations. Section 8 discusses some limitations of our technique, while Section 9 describes several applications for this technology other than its use in traditional cel animation. Finally, Section 10 outlines some possible areas for future work.

2 RELATED WORK

A variety of previous efforts have addressed the use of computer graphics for animation, though to our knowledge nobody has successfully solved the specific problem described here. Researchers have largely automated the image processing and compositing aspects of cel animation [6, 18, 25, 28], wherein the conventional ink-and-paint stage could be replaced with the textures resulting from our system. Wood *et al.* [35] demonstrate the use of 3-D computer graphics in the design of static background scenery; in contrast, this paper addresses animated foreground characters. Sabiston [20] investigates the use of hand-drawn artwork for driving 3-D animation, which, although not the main focus of our work, is similar to the application we describe in Section 9.2. Finally, animators at Disney actually applied a computer graphics texture to color a hand-drawn magic carpet in the film *Aladdin* [30], but their process involved meticulously rotoscoping a 3-D model to match each frame of artwork—an arduous task that would have benefitted from a system such as the one we describe.

The heart of our method is a new warp. Critical for visual effects such as metamorphosis (or *morphing*), image warps have been extensively studied. Beier and Neely's warp [2] distorts images in 2-D based on features marked with line segments; it was the inspiration for the warp that we describe, which works with curved feature markers and provides 3-D effects such as occlusion and foreshortening. Litwinowicz and Williams [13] describe a warp (based on a thin-plate smoothness functional) that behaves more smoothly than that of Beier and Neely in the neighborhood of feature markers; perhaps a hybrid approach could combine these smoothness properties into the warp that we describe. Lee *et al.* [10] have described a user-interface based on *snakes* [9] that is useful for feature specification, as well as a new warp based on free-form deformations [24]. Warps have been applied in other domains as well, such as the work of Sederberg *et al.* [23] on 2-D curves, Witkin and Popović [33] on motion curves for 3-D animation, and Lerios *et al.* [11] on volumes.

This paper also presents a scheme for silhouette detection based on rendering the 3-D model into a frame buffer. In general, silhouette detection is closely-related to hidden surface removal, for which there are a host of methods [7]. Markosian *et al.* [14] present some improvements and simplifications of traditional algorithms, and are able to trade off accuracy for speed. Most algorithms dealing with polyhedral input traverse the mesh tagging edges of the model as silhouettes. In our work, we are only interested in *visible* silhouettes (since they correspond to features that appear in the drawing). Furthermore, we want our algorithm to produce smooth, continuous silhouette curves on the actual model. As described in Section 4, our method generates this kind of output, and solves the problem of bifurcation along the silhouette edge by rendering the 3-D model colored with texture coordinates into a frame buffer. Saito and Takahashi [21] employed a similar technique for highlighting edges in technical illustrations, and Wallach *et al.* [29] used this idea for finding frame-to-frame coherence in 3-D animations.

This project shares much in spirit with the recent progress of the computer graphics community toward non-photorealistic rendering (NPR), although the actual NPR aspects of our work (the shape of the animated figure and often its texture) are created by an artist. For the researchers who have investigated animation and video in simulated media (oil paint for Meier [15] and Litwinowicz [12]; pen and ink for Markosian *et al.* [14] and Winkenbach *et al.* [32]; and watercolor for Curtis *et al.* [4]) a challenge has been to maintain temporal coherence in the individual strokes of the artwork to avoid "boiling." In our project, this challenge is circumvented because we use a single texture throughout the animation.

3 THE PROCESS

In this section we present our system from the user's point of view. The details of the algorithms mentioned here are explained in later sections.

For each shot in the animation, we follow these steps (Figure 2):

(a) A person scans in the cleaned-up hand-drawn artwork.

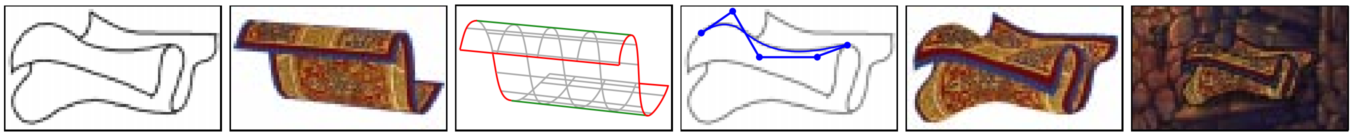
(b) A person creates a simple 3-D model that approximates roughly the shape of the hand-drawn character.

(c) The computer finds border and silhouette edges in the model.

(d) A person traces over edges of the line art that correspond to border and silhouette features of the 3-D model.

(e) The computer warps the 3-D model to match the shape of the line art, and then renders the model.

(f) The computer composites the rendered model with the hand-drawn line art and background scenery.



(a) Hand-drawn art (b) 3-D model (c) Edges in model (d) Edges in art (e) Warped model (f) Final frame

Figure 2: The process of creating one texture mapped frame.

Our method fits into the existing production pipeline for cel animation [6, 18]. Steps (a) and (f) are stages in the current digital production process, with the ink-and-paint stage between them. We are offering, as an alternative to the constant colors of the ink-and-paint stage, a process that applies complex textures to the drawings.

The problem of applying textures to hand-drawn artwork poses a challenge: the line art must be interpreted as some kind of shape. Given a set of black lines on white paper, the computer must acquire at least a primitive model for the 3-D forms conveyed by the art. This information is necessary if we are to provide 3-D effects for the texture such as self-occlusion and foreshortening. (See, for example, the difference in occlusion between Figures 2b and 2e or the foreshortening shown in Figure 7.) Note that with the constant colors of the traditional ink-and-paint stage, these 3-D effects are unnecessary. The viewer does not expect the constant orange color of the front of the carpet in Figure 1a to appear to recede as it crosses over a silhouette; however the texture of the carpet in Figure 1b must recede. Thus, some form of 3-D information must be available to the algorithm. Since interpreting hand-drawn line art as a 3-D figure is tantamount to the computer vision problem (which has not as yet been solved), we resort to human intervention for steps (b) and (d) above. These phases of our process can be labor-intensive, and we believe that partial automation of these tasks through heuristic methods is a critical area for future work.

In step (b) above, we create a simple 3-D model that corresponds to the animated figure. As seen from a set of specific camera perspectives, the model should have the approximate form of the hand-drawn figure. By "approximate form" we mean that the model projected into screen space should have a similar set of features in a similar arrangement as the features of the line art. For example, the artwork and 3-D model and in Figures 2a and 2b both have four border edges and an upper and lower silhouette edge. Note that in this example the warp succeeds even though the order of the upper silhouette edge and the backmost edge of the carpet is reversed between the hand-drawn artwork and the 3-D model. For the animations shown in this paper, our models were represented by tensor product B-spline patches [5]. However, before performing the warp described in Section 5, we convert our models to polygon meshes. Thus, the method should be applicable to any model that can be converted to polygons, provided that the model has a global parameterization, which is generally necessary for texture mapping.

4 SPECIFYING MARKERS

In this section we show how we specify *marker curves*—curves that identify features on the 3-D model and on the 2-D artwork. We call feature curves on the model *model markers* and feature curves on the drawing *drawing markers*. These curves will be used by the warp described in Section 5 to deform the 3-D model so that it matches the drawing.

Section 4.1 explains how we automatically find model markers by detecting visible border edges and silhouette edges on the model. Section 4.2 explains how these edges are converted to form smooth curves on the model. Section 4.3 shows how to guarantee that these edges can be safely used later as the input to the warp. Section 4.4 shows how to specify the edges on the 2-D drawing that correspond to the edges found on the 3-D model.

4.1 Silhouette Detection

In this section we describe a scheme for finding visible silhouette and border edges in a 3-D model represented by a polygon mesh. These features are likely to correspond to features in the hand-drawn line art; such correspondences are the primary input to the warp we describe in Section 5. We also allow the user to specify model markers by drawing them directly on the 3-D model, but it would be cumbersome to have to specify *all* model marker curves this way. Thus, we automatically construct model markers for all visible border and silhouette edges, and allow the user to pick the useful marker curves (often, all of them).

To get started, we need to define some terminology, consistent with that of Markosian *et al.* [14]. A *border edge* is an edge adjacent to just one polygon of the mesh. A *silhouette edge* is an edge shared by a front-facing polygon and a back-facing polygon (relative to the camera).

Standard silhouette detection algorithms will identify the subset of edges in the mesh that are silhouette edges. Treated as a group, these edges tend to form chains. Unfortunately, in regions of the mesh where many of the faces are viewed nearly edge-on, the chains of silhouette edges can bifurcate and possibly re-merge in a nasty tangle. For our warp, we are interested in finding a smooth, continuous curve that traces the silhouette on the model, rather than identifying the exact, discrete set of silhouette edges in the mesh. Furthermore, we are only interested in *visible* silhouettes because they tend to correspond to features that appear in the drawing. Finally, we want to distinguish border edges from other silhouette edges.

To detect the border and silhouette edges of a 3-D model, we proceed as follows. Using Gouraud shading (*without* lighting effects or antialiasing) we render the 3-D model over a black background as a polygon mesh whose vertices are colored $(R, G, B) = (u, v, \mathrm{ID})$, where u and v are the parametric coordinates of each vertex, and ID identifies the texture (Figure 3b). Let us call the resulting image the *uv-image*. The method accommodates models with multiple texture maps, but so far in our animations all of our models have only used a single texture, whose ID is 1. The ID 0 is reserved for the background.

When a pixel on the uv-image corresponds to a point on the surface of the model, we say that the pixel is *covered by* the model. Also, a *pixel corner* is one of the four corners of a pixel, while a *pixel boundary* is the line segment joining two pixel corners shared by two adjacent pixels. For example, in Figure 4, p_1 and p_2 are pixels, c_1 and c_2 are pixel corners, and e is a pixel boundary.

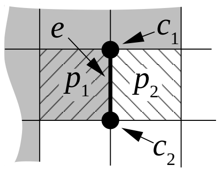


Figure 4: A few pixels.

Borders and silhouettes generate color discontinuities in the resulting image (Figure 3c). To find these discontinuities, we construct a directed graph $G = (V, E)$, where V are the vertices

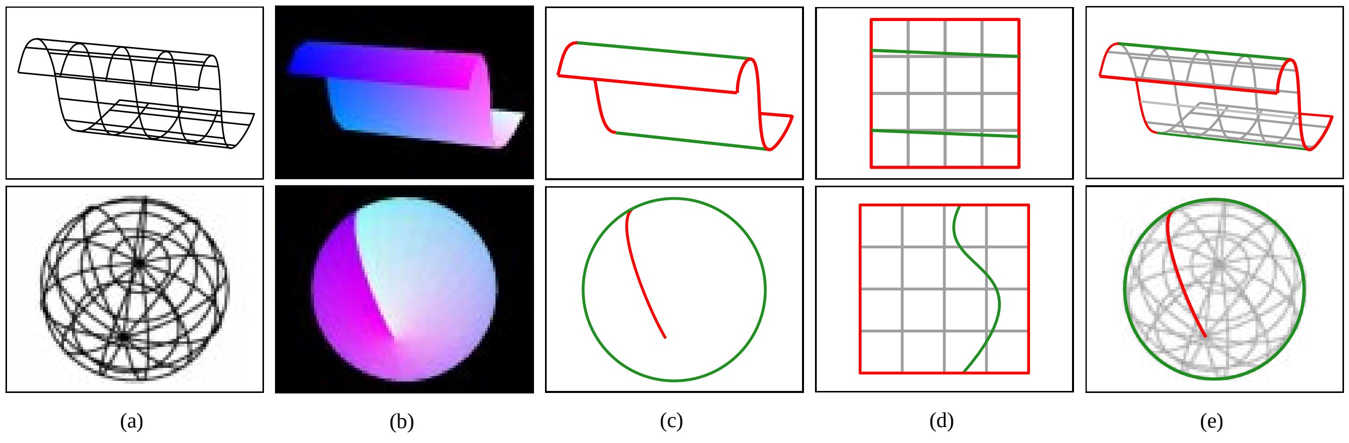


Figure 3: Detecting border and silhouette edges on a 3-D model. (a) Wireframe 3-D model. (b) 3-D model shaded with vertices colored (u, v, ID). (c) Discontinuities in color. (d) Model marker curves in parameter space. (e) Marker curves on model.

of the graph, and E are directed edges in the graph. V consists of the pixel corners in the uv-image, and E is constructed by the following classification process:

```
CLASSIFY($G$)
1  for every boundary between two neighboring pixels
2      $p_1 \leftarrow$ pixel closer to the camera
3      $p_2 \leftarrow$ pixel farther from the camera
4      if $p_1$.color $\not\approx$ $p_2$.color
5          $e \leftarrow$ ADD_EDGE($G, p_1, p_2$)
6          if $p_1$ is an extremum
7              $e$.type $\leftarrow$ corresponding kind of border edge
8          else
9              $e$.type $\leftarrow$ silhouette edge
```

In steps 2 and 3, we determine which of the two pixels of a boundary is closer to the camera, using one of three methods. First, if exactly one of the pixels is covered by the model, then it is the closer pixel (because the other pixel corresponds to background). Second, we can read from the depth buffer the z-values of the two pixels and compare them. Third, if the depth value is unavailable (as on some machines for which z-buffering is implemented in hardware) then we can read the u and v parameters of the part of the model that covered those pixels, evaluate the model at those parametric locations, and compare their distances from the camera.

In step 5, we add to G a directed edge e between the two corners shared by p_1 and p_2 in such a way that p_1 is on the left of e, and p_2 is on the right. If the parametric space is periodic then the border edges should be ignored. For example, in Figure 3, the carpet has two silhouette edges (upper and lower) and four border edges (at $u = 0$, $u = 1$, $v = 0$, and $v = 1$); the ball only has silhouette edges because the interior line in Figure 3c is ignored. In the actual implementation, we generate for each edge a confidence value for each of the possible kinds of edge. If, for example, the u parameter at p_1 was found to be very near 0, and u at p_2 was large, then we could say with high confidence that the edge between these pixels represents a border at $u = 0$. Finally, we choose the highest confidence value to represent the type of the edge.

After the classification process is finished, finding the edges on the model is equivalent to finding the connected components of G, which can be done efficiently using depth-first search [3, 26]. We traverse the graph, finding paths of edges that have the same edge type and the same color (within a tolerance). The running time of depth-first search is $\Theta(|V|+|E|)$ [3]. In our case, both $|V|$ and $|E|$ are linear in the number of pixels. Furthermore, the classification process is linear in the number of pixels because it makes a single pass over the entire image, adding a number of edges to the graph that is bounded by a constant multiple of the number of pixels. Thus, the silhouette detection scheme described here is linear in the number of pixels in the image. The image resolution dictates the accuracy of the resulting silhouette curves. We have found that a resolution of 512×512 is sufficient for the simple models we have used in our animations.

Having found the connected components of G, we have a set of lists of pixel boundaries that correspond to border edges and silhouette edges on the model. In the next section we will describe how to fit smooth curves to this data. These curves are the model marker curves (Figure 3e) used by the warp in Section 5.

The benefits of this silhouette detector are that it is simple to implement, it leverages existing support for hidden surface removal and texture mapping, it works for any object with a well-defined parameter space, and produces smooth, visible silhouette curves.

4.2 Curve Fitting

To represent each marker curve, we use a chord-length parameterized endpoint-interpolating uniform cubic B-spline [5, 19]. These curves have several desirable properties: they are smooth; they can be linked “head to tail” (without a break between them); the rate of change of the curve with respect to the parameter is uniform; and they are well understood. We obtain each curve by fitting it to a set of data. These data are either chains of pixels (as in the previous section) or the result of user input (as in Section 4.4).

To fit the curves, we typically have many more data points than degrees of freedom: a chain of hundreds of pixels generated by the silhouette detector (or generated by tracing on the drawing) may be smoothly approximated by a spline with only a few control points. To calculate the control points, we solve an overdetermined linear system using least squares data fitting techniques that minimize the root-mean-square error between the data and the resulting curve [1, 17, 19, 22]. Our fitting procedure attempts to use as few control points as possible, given a maximum error threshold.

4.3 Ghostbusting

The marker curves created in this section drive the warp described in Section 5. Beier and Neely [2] observe that image warps tend to fail if feature lines cross, producing what they call *ghosts*. Near the intersection between two feature lines, both features exert a strong influence on the warp. If crossed feature lines do not “agree” about the warp, then there tend to be sharp discontinuities in the resulting warp leading to unpleasant artifacts. Thus, conventional wisdom regarding image warps warns: “do *not* cross the streams.”

For the warp of Section 5, there are some configurations of crossed feature markers that are dangerous and others that are benign. Our warp distorts a 3-D model, so feature markers that cross *on the model* lead to sharp discontinuities in the warp. On the other hand, feature markers that cross in image space (but are distant on the model) do not cause any ill artifacts. An example of each kind of crossing is shown in Figure 5. Our application automatically detects dangerous crossings, and splits model marker curves where they cross. Since the silhouette detector described in this section builds feature markers based on finding discontinuities in parameter space, it is easy to use this information to distinguish dangerous crossings from benign crossings. At dangerous crossings we split marker curves so that they meet only at their endpoints. For example, after splitting, three curves in Figure 5 meet at the point labeled “dangerous.” Furthermore, our application trims the feature curves a small distance away from the crossing point, so the resulting curves do not touch. The splitting procedure ensures that the resulting marker curves “agree” about the warp at their intersection. The trimming step, though not strictly necessary, causes the warp to behave even more smoothly in the neighborhood.

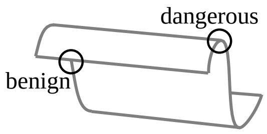


Figure 5: Crossed streams.

4.4 Specifying Markers on the Drawing

We have described the creation of model marker curves based on features in the 3-D model. Next the user picks model marker curves, and specifies corresponding drawing marker curves. To specify these curves, the user traces over features in the hand-drawn art—a time-consuming and tedious task. To reduce user-intervention, we use contour tracing techniques similar to those presented by Gleicher [8] and Mortensen and Barrett [16] in which the cursor automatically snaps onto nearby artwork. For each drawing marker, tracing results in a list of pixels that we subsequently approximate with a smooth curve, as described in Section 4.2.

We now have a collection of model and drawing markers. These curves identify features on the model that correspond to features on the hand-drawn artwork. In the following section, we describe a warp that uses this information to guide the distortion of the 3-D model so that it matches the hand-drawn artwork.

5 THE WARP

This section describes our method for warping a given 3-D model to match hand-drawn line art. The inputs to the warp are the model, camera parameters, and a set of pairs of (model and drawing) marker curves. The warp is applied to points on the model *after* they have been transformed into the screen space for the given camera but before the model has been rendered. The warp, which maps a point in screen space to a different point in screen space, has the following important properties:

- Each warped model marker lies on its associated drawing marker as seen from the camera.
- The surface warped between two markers varies smoothly, avoiding puckering or buckling.
- The warp preserves approximate area in image space, so that foreshortening effects are still apparent.
- The warp leaves depth information (relative to the camera) unchanged.

5.1 The Warp for One Pair of Markers

To begin, let us assume we only have a single pair of marker curves in screen space: a model marker curve $M(t)$ (generated by the silhouette detector in Section 4.1) and a drawing marker curve $D(t)$ (traced on the drawing by the user as described in Section 4.4). In the next section we will describe how this works with multiple pairs of marker curves.

For a given point p in the screen-space projection of the model, we want to find the corresponding point q on the drawing. For a particular value of t, we define two coordinate systems: one on the model marker curve and one on the drawing marker curve (Figure 6). The model coordinate system has origin at $M(t)$, and abscissa direction $\hat{x}_m(t)$ given by the tangent of M at t. Likewise, the drawing coordinate system has origin at $D(t)$, and abscissa direction $\hat{x}_d(t)$ given by the tangent of D at t. For now, the ordinate directions $\hat{y}_m(t)$ and $\hat{y}_d(t)$ are oriented to be perpendicular to the respective abscissa. However, in Section 6.4 we will modify the ordinate direction to get a smoother warp.

We find the x and y coordinates of p in relation to the model coordinate system in the usual way:

$$\begin{aligned} x(t) &= (p - M(t)) \cdot \hat{x}_m(t) \\ y(t) &= (p - M(t)) \cdot \hat{y}_m(t) \end{aligned}$$

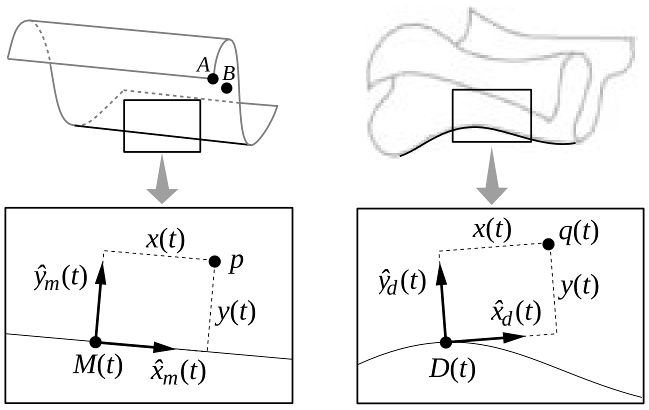
Next we define a tentative drawing point $q(t)$ corresponding to $p(t)$:

$$q(t) = D(t) + x(t)\,\hat{x}_d(t) + y(t)\,\hat{y}_d(t) \quad (1)$$

This is the location to which we warp $p(t)$, taking into account *only* the coordinate systems at parameter t. Of course, we have a continuum of coordinate systems for all parameters t, and in general they do not agree about the location of $q(t)$. Thus, we take q to be a weighted average of $q(t)$ for all t, using a weighting function $c(t)$:

$$q = \frac{\int_0^1 c(t)\,q(t)\,dt}{\int_0^1 c(t)\,dt} \quad (2)$$

We want the contribution $c(t)$ to fall off with the growth of the distance $d(t)$ between the 3-D points that project to p and $M(t)$. Intuitively, we want nearby portions of the marker curves to have more influence than regions of the markers that are far away. We compute $d(t)$ in one of two ways. We either approximate the



(a) Model marker curve (b) Drawing marker curve

Figure 6: Coordinate systems for the warp.

distance along the surface of the model, or we compute world-space distance on an undeformed reference mesh. This is an important difference between our warp and traditional warps: traditional warps would use the 2-D distance between the origins of the coordinate systems. Although our warp happens in 2-D space, it takes into account 3-D information. Thus, the marker curves at point A in Figure 6a have little influence on the warp of point B, even though they are very near each other in the image plane. Thus, we choose the contribution to be:

$$c(t) = \frac{1}{\epsilon + d(t)^f} \tag{3}$$

where ϵ is a small constant to avoid singularities when the distance is very near zero, and f is a constant that controls how fast the contribution falls off with distance. We discuss the nature of these parameters in greater detail in Section 6.1.

5.2 The Warp for Multiple Pairs of Markers

The warp in equation (2) only considers a single pair of marker curves. Here we generalize the warp to handle multiple pairs of curves. We assign a user-specified weight w_i to each pair i of curves, $i = 1, 2, \ldots, m$, balancing the relative contribution of each pair to the overall warp. Finally, we compute the drawing point q as a weighted average using $q_i(t)$ and $c_i(t)$ from equations (1) and (3), for each marker pair i:

$$q = \frac{\sum_{i=1}^{m} \left(w_i \int_0^1 c_i(t)\, q_i(t)\, dt \right)}{\sum_{i=1}^{m} \left(w_i \int_0^1 c_i(t)\, dt \right)} \tag{4}$$

The main features of our warp are: it uses curved features; it is visually smooth; it is scale, translation, and screen-space rotation independent; and, most importantly, it maps model markers to drawing markers while preserving 3-D effects.

5.3 Computing the Warp

In practice, we evaluate the integrals in equation (4) numerically, by uniformly sampling the marker curves. This can be computed quickly for all vertices in the model by caching the coordinate systems at sample locations.

Our warp uses *forward mapping* [34]: for each point in the source texture, it finds where the point is mapped in the destination. The motivation in this case is that we get self-occlusion and hidden-surface removal for free using normal z-buffered rendering. Also, this permits us to compute the warp at low resolution for the interactive parts of the process, and later perform the final rendering at high-resolution. Litwinowicz and Williams [13] also use forward mapping, whereas Beier and Neely [2] use *inverse mapping*: for each point in the destination, they find the location at which to sample the input image. We sample the map by calculating the warp only at vertices in the 3-D model; thus, we can render the warped model using conventional texture mapping simply by modifying the coordinates of vertices in the mesh and then rendering the usual way.

6 CONTROLLING THE WARP

Many factors influence the behavior of the warp described in Section 5. In this section we describe some of these influences in better detail.

6.1 Contribution Parameters

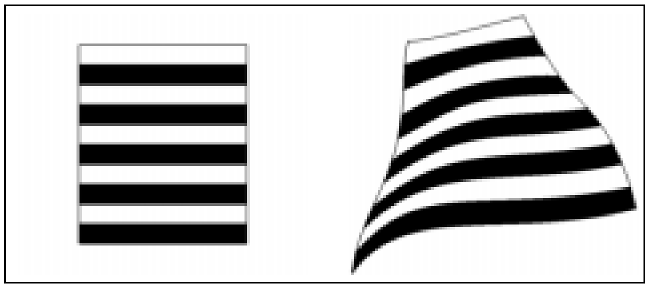
The factors ϵ and f of equation (3) can be modified to achieve different effects. By varying ϵ, we can get different levels of smoothness and precision. If ϵ is large, the warp is smoother. If ϵ is small, the warp is more precise. By varying f, we control how fast the contribution falls off with distance. If f is large, distant points will have almost no influence. (The factors ϵ and f are similar to the factors a and b of the warp of Beier and Neely [2].) The figures in this paper use $\epsilon = 10^{-4}$ and f between 2 and 3.

6.2 Modeling and Viewing Parameters

Our warp is based on the projection of the 3-D model into screen space. By varying the modeling and viewing parameters we produce different projections of the model, and obtain different results. Figure 7 shows the results of the warp for two different camera views. In both cases, the hand-drawn art and the 3-D model are the same. Notice that without the texture the drawing would be ambiguous. We do not know whether the drawing is smaller at the top or recedes into the distance. The foreshortening effect helps resolve this ambiguity. Thus, use of a texture in a figure may provide the viewer with information about spatial arrangement that is not available in conventional flat-colored figures. Since it only takes a few seconds to rotate the model and re-render the frame, it is easy to interactively switch between the two views like those of Figure 7 to choose one that has the desired effect.

6.3 Extra Markers

So far we have only mentioned model marker curves detected automatically. In the actual implementation, the user can specify extra markers on the model and the corresponding ones on the drawing, to match features that were not detected automatically. For example, Figure 8 shows how extra markers can be used to tell the carpet how to fold over a set of stairs. Also, one could add *marker points*— control points that add to the formula like marker curves, but have a single coordinate system that is embedded in the


(a) Upright model warped to the art

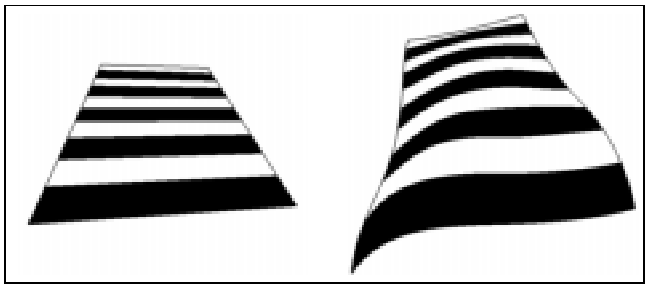
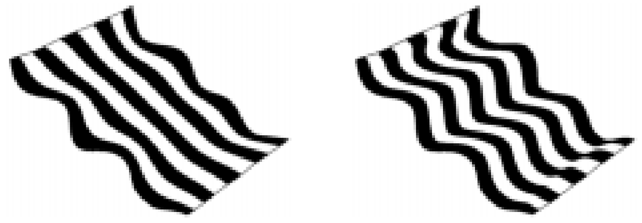

(b) Tilted model warped to the art

Figure 7: Influence of modeling and viewing parameters.



(a) Without extra markers (b) With extra markers

Figure 8: Effect of defining extra markers on the warp.

plane—although we have not yet found it necessary to use marker points for any of our animations.

6.4 Ordinate Direction Adjustment

In Section 5 we described a warp that uses coordinate systems whose ordinate and abscissa directions are perpendicular to each other. This approach leads to puckering when there is a great disparity between the orientation of nearby drawing coordinate systems (Figure 9a). Instead, we adjust the ordinate direction of the drawing coordinate systems so that they conform better to the actual artwork. The algorithm produces drawing coordinate systems whose axes are linearly independent, but not necessarily orthogonal. The result is a smoother and more intuitive warp (Figure 9b).

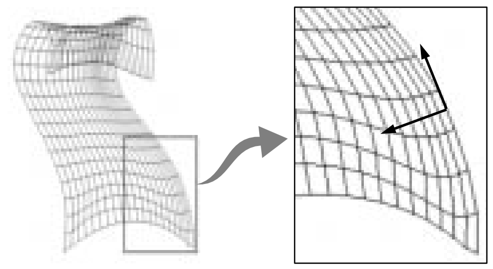
To understand how our method works, let us suppose we have a drawing coordinate system with axes $(\hat{x}_d, \hat{y}_d)$, and a corresponding model coordinate system with axes $(\hat{x}_m, \hat{y}_m)$ as shown in Figures 10a and 10b. Also, let us assume that we have a single drawing marker curve D and a corresponding model marker curve M. For a given value of t, we want to find the ordinate direction $\hat{y}'_d$ taking into account the drawing coordinate system with origin at $D(t)$ and axes $(\hat{x}_d(t), \hat{y}_d(t))$, as well as the corresponding model coordinate system with origin at $M(t)$ and axes $(\hat{x}_m(t), \hat{y}_m(t))$. We find the rotation that maps $\hat{y}_m(t)$ to $\hat{y}_m$, and apply this same rotation to $\hat{y}_d(t)$ to obtain $\hat{y}'_d(t)$, which is where the coordinate system at $D(t)$ "thinks" that $\hat{y}_d$ should be. Using an approach similar to the one we used for the warp, we find $\hat{y}'_d$ as a weighted combination of $\hat{y}'_d(t)$, using a weighting function $c'(t)$:

$$\hat{y}'_d = \frac{\displaystyle\int_0^1 c'(t)\, \hat{y}'_d(t)\, dt}{\displaystyle\int_0^1 c'(t)\, dt} \tag{5}$$

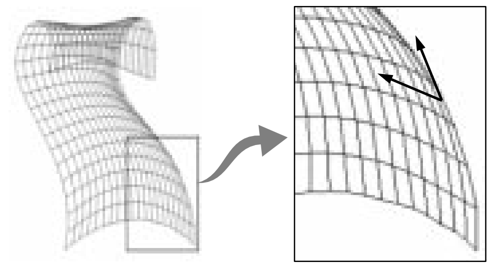
We want the contribution $c'(t)$ to have two properties. First, it should fall off with the distance $d'(t)$ between the model coordinate systems, computed as in Section 5. Second, it should be larger when the corresponding coordinate systems in parameter space (Figure 10c) are perpendicular to each other. Intuitively, the bottom edge of the carpet in Figure 9 should have a strong influence over the ordinate direction along the right edge of the carpet, because they are perpendicular in parameter space. This leads to "isoparameter lines" in the drawing space that follow the shape of the lower boundary curve. The contribution is then:

$$c'(t) = \frac{1 - \hat{y}_p \cdot \hat{y}_p(t)}{\epsilon' + d'(t)^{f'}}$$

where $\hat{y}_p$ and $\hat{y}_p(t)$ are the vectors in parameter space that correspond to $\hat{y}_m$ and $\hat{y}_m(t)$, respectively. The parameters ϵ' and f', and the distance function $d'(t)$ have the same roles as in equation (3). As with the warp in Section 5, we compute the final ordinate direction by promoting equation (5) to include *all* marker curves, but here we do not use weights w_i. When all coordinate systems are parallel to each other in parameter space, the result of this algorithm is undefined. In this special case, we simply use the original orthogonal $\hat{y}_d$.

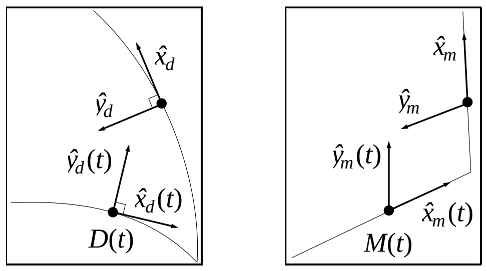


(a) Without adjustment

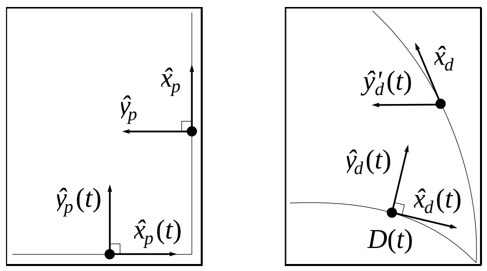


(b) With adjustment

Figure 9: Effect of ordinate direction adjustment on the warp.



(a) Drawing (b) Model



(c) Parameter (d) Adjusted

Figure 10: Coordinate systems for ordinate direction adjustment.

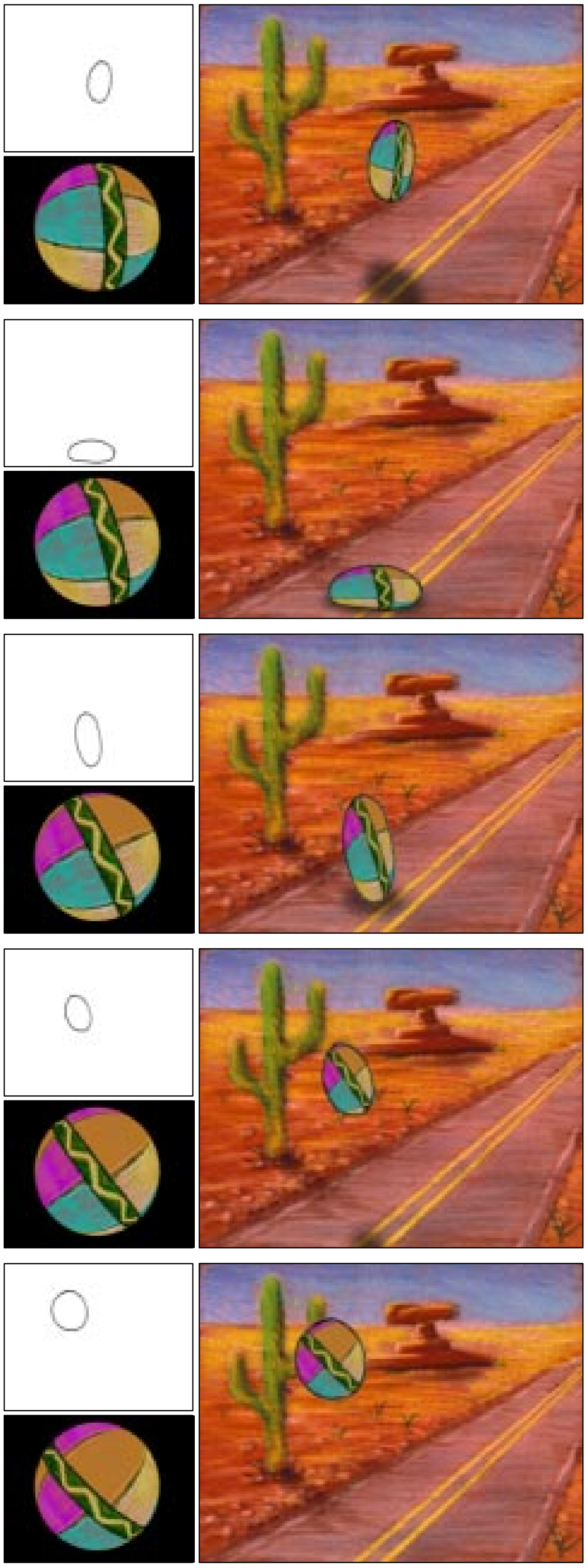
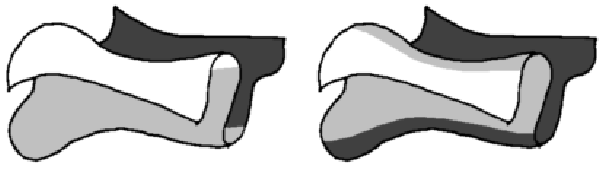


Figure 11: Bouncing ball animation.



(a) Without reparametrization (b) With reparametrization

Figure 12: Effect of reparameterizing the texture.

6.5 Reparameterizing the Texture

As we mentioned in Section 4, the marker curves are approximately chord length parameterized. This leads to an interpretation of which part of the overall texture should be visible that is often, but not always, correct. We provide a mechanism for pulling occluded texture back over a silhouette horizon, or pushing visible texture over the horizon so it becomes occluded. For example, for the carpet in Figure 12, the lower silhouette curve corresponds to two different regions in parameter space. We simply move the texture on the underformed model, essentially reparameterizing the texture on the surface. We can perform such reparameterization easily by warping the texture in parameter space prior to rendering, guided by feature curves such as the borders of the texture and the desired silhouette positions. However, we have not yet found it necessary to use this feature for any of our animations.

7 RESULTS

In this section we describe several animations we created with our application.

Figure 11 shows an animation of a ball bouncing along a desert highway. For each row, the upper left picture shows the hand-drawn artwork, the lower left picture shows the 3-D model with the texture applied to it, and the right picture shows the final frame from the movie: hand-drawn art composited with the warped model and the background. In these frames the ball exhibits “squash and stretch”, a fundamental principle of traditional cel animation [27]. The 3-D model for this case is a simple sphere with a hand-painted beach-ball texture applied to it. The sphere rotates with respect to the camera, so that it appears to be rolling. To guide the warp for this simple model, we used exactly one pair of marker curves per frame.

In Figure 14 we show the line art, model, and final frames for an animated carpet on a staircase. In these frames, the carpet “stands up” to look around. For this animation we used about 5 or 6 marker curves per frame. Section 6.3 describes a method for specifying extra marker curves that would cause the carpet to follow the contours of the steps more closely, but we did not use that feature in this animation. The frames demonstrate 3-D effects such as self-occlusion and foreshortening—as well as temporal coherence in a complex texture—that would be very difficult to produce with traditional cel animation. Also note that even though the character is computer-rendered using a highly-symmetric, repeating texture, it blends aesthetically with the background art due to the hand-drawn shape.

Finally, Figure 15 shows a hand-drawn animation of a fish. The 3-D model of the fish contains only the body of the fish (with a simple checkerboard texture). Not shown are the three fins of the fish (which are planar quadrilaterals) and the two eyes of the fish (which are simple spheres). These were modeled, rendered, and composited as separate layers—in the style of traditional, multi-layer animation [27]—although they all share the same hand-drawn art. The model is rigid throughout the animation, only rotating back and forth with respect to the camera. Nonetheless, in the final frames of the animation the fish appears to bend and flex as it swims. The fins and tail are semi-transparent (using a hand-painted matte embedded in the texture of the model) and thus it is possible to see the reeds of the background through transparent parts of the fish.

To create the hand-drawn artwork for the fish, it took several hours to draw 23 frames. Building the 3-D model was easier (approximately an hour) because the motion is simple rotation. The computer found the model markers in just a few seconds per frame. Drawing markers were specified by hand, requiring about two minutes per frame. Finally, rendering and compositing required tens of seconds per frame. For the ball and carpet animation (which have simpler art and simpler models) these steps required less time.

8 LIMITATIONS

There are several classes of line art for which our process does not work well. First, cel animation has developed a "vocabulary" for conveying texture by modifying the edges of characters. For example, tufts of fur on a character may be suggested or by drawing a few sharp wiggles rather than a smooth edge (Figure 13a). A character illustrated with *both* "hinted" texture (in its line art) and the kind of textured-fill described in this paper would probably suffer visually from this mixed metaphor; moreover, the texture near the sharp wiggles would be likely to stretch and pucker unpleasantly in order to conform to the wiggles of the line art. Our process also does not work well with figures for which it is difficult to generate the 3-D model—most notably, clothing. Cloth is typically drawn showing creases and folds, which would be difficult to replicate well in a 3-D model (Figure 13b). Other drawings use a few lines to suggest greater geometric complexity. The interior folds in the knotted carpet would have to be modeled explicitly if we expect the texture to look right. Figure 13c shows an example of how some drawings do not correspond to any reasonable 3-D geometry. Here the nose is drawn in profile, the eyebrows float off the surface, and the hair is a crude representation. There are other limitations inherent in using quadrilateral patches to represent complex shapes. The body, legs, and tail of the cat Figure 13d could not reasonably be represented with a single patch.

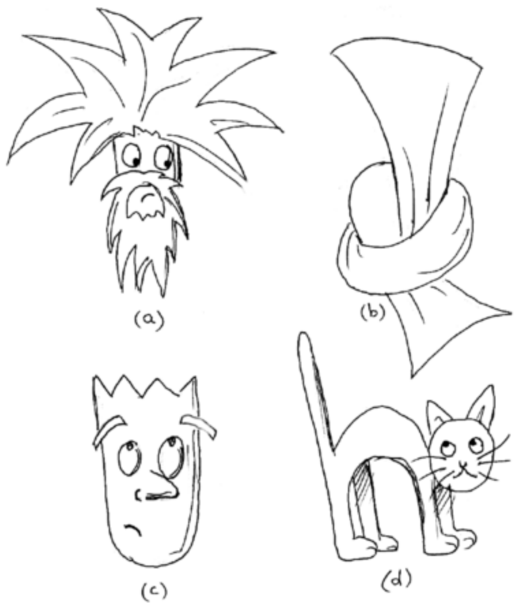


Figure 13: Line art that would be difficult to texture.

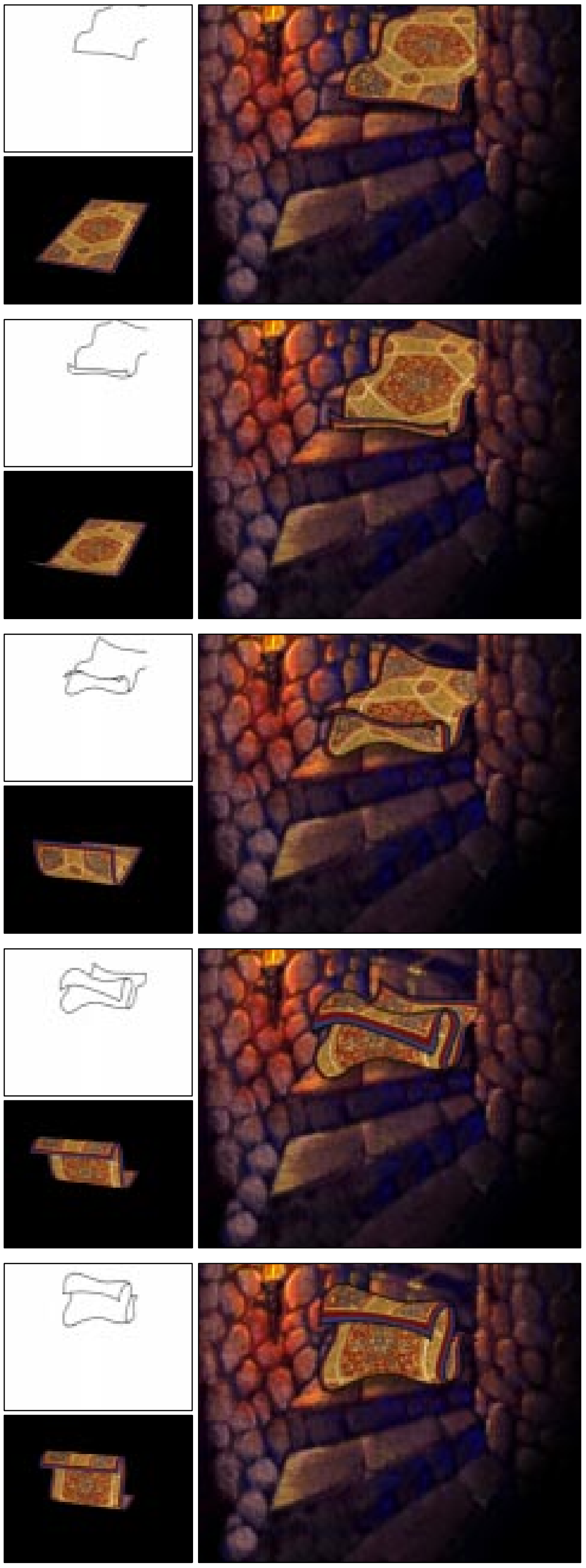


Figure 14: Carpet animation.

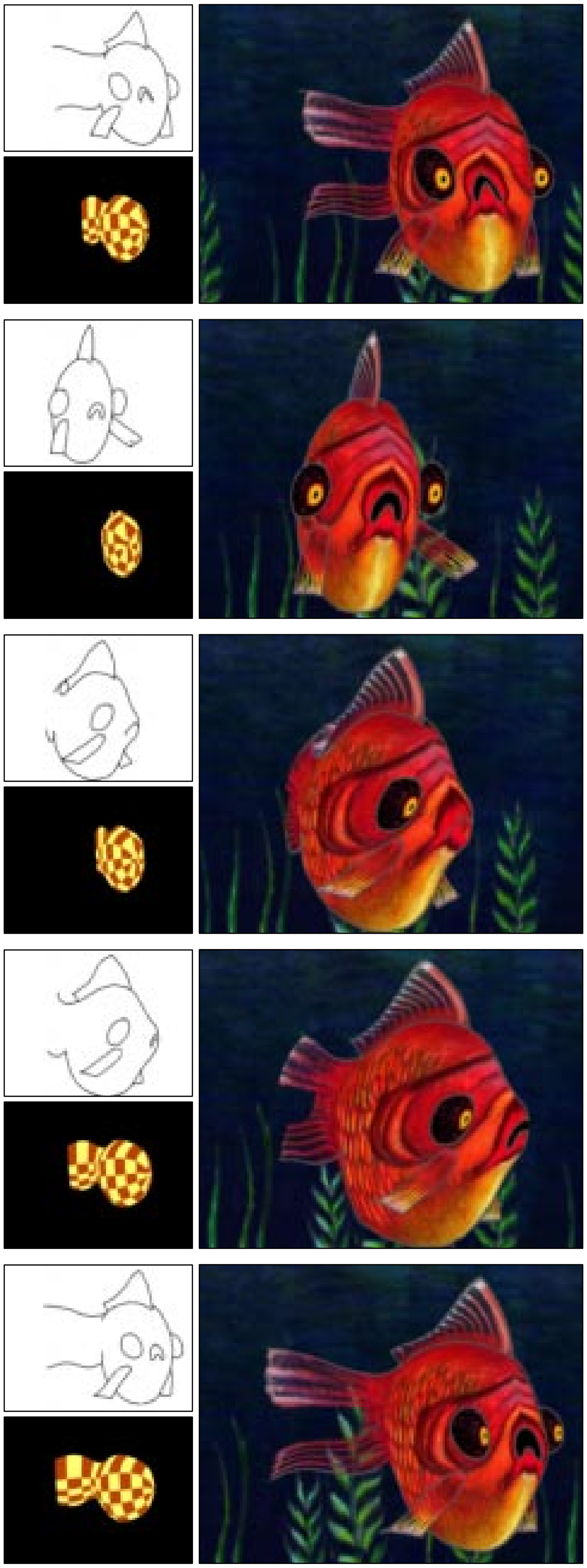


Figure 15: Fish animation.

To solve these problems we either need to use multiple patches and solve continuity issues, or switch to a more general surface representation. Subdivision surfaces could be used in the warp if we devised a method of representing surface curves and some concept of orthogonality in parameter space.

9 OTHER APPLICATIONS

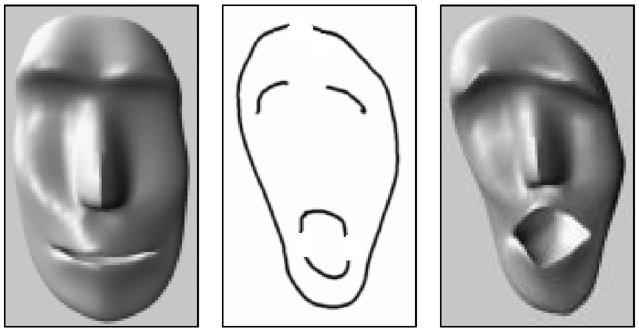
In this section we briefly talk about some other applications that might benefit from this technology.

9.1 Shading Effects

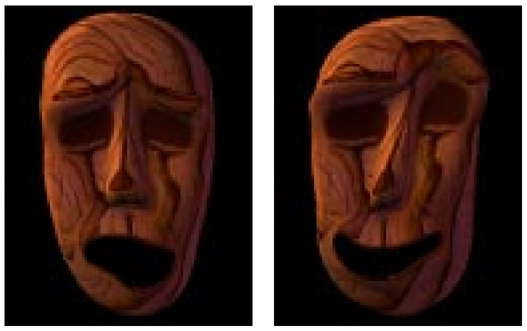
Once we have this correspondence between a 3-D model and the artwork, it is easy to incorporate many traditional computer graphics effects such as highlights, shadows, transparency, environment mapping, and so forth. Since this draws us further away from the look and feel of traditional animation, we have not investigated these effects in our work (except for the use of transparency in the fins of the fish in Figure 15).

9.2 3-D Shape Control for Animation

While it is not the focus of this work, we are currently developing a variation of this method as a new form of control in 3-D animation. It would fit into the 3-D animation pipeline just before rendering. An animator could add detail or deformations to the 3-D geometry by *drawing* on the image plane (Figure 16). This is better than distorting the final image because it affects the *actual* geometry, correctly modifying occlusion and shading. Note that after the warp, the model should only be viewed from the original camera position, as the figure may appear distorted when viewed from other directions.

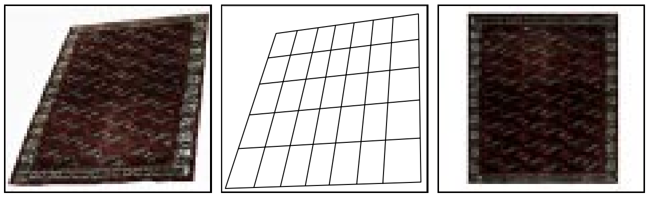


(a) Original 3-D model (b) Hand-drawn art (c) Warped 3-D model



(d) Perplexed face (e) Goofy face

Figure 16: Controlling a 3-D model by drawing.



(a) Texture photograph (b) 3-D model (c) Acquired texture

Figure 17: Texture acquisition.

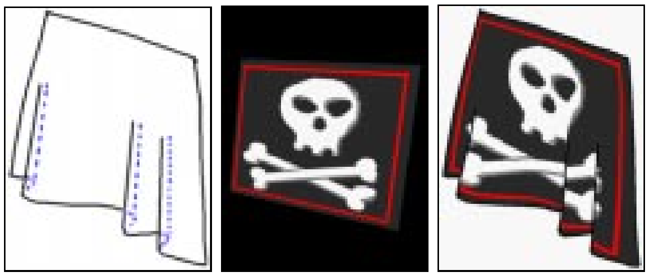
9.3 Texture Acquisition

Sometimes we find pictures of objects with textures that we would like to apply to other objects. Often these textures are not available in a perfect orthogonal view as it is necessary for texture mapping. Our technique can be used in reverse to acquire the texture. For example, shown in Figure 17 is a photograph of a carpet on the floor. Since the camera was not directly over the carpet, it appears in perspective; furthermore, since the carpet is not rigid the edges are not completely straight. Thus, the image of the carpet is not the kind of rectangle one would like to use for texture mapping. We build a 3-D model of the carpet (a rectangle), position and orient the model in space so its projection on screen space is similar to the picture, associate markers on the picture and on the model, and apply the inverse of our warp to extract the texture form the picture and apply it to the model. Of course, if parts of the figure were occluded in the original photograph this could lead to holes in the final texture map.

10 FUTURE WORK

This project suggests a number of areas for future work, several of which are described below.

Computer Vision. We would like to reduce the amount of effort required to construct and position the 3-D model. One strategy is to investigate the applicability of computer vision algorithms to reconstruct the 3-D geometry from the 2-D drawings. Perhaps the animator could draw hints in the artwork using Williams's scheme [31] for conveying depth information through line-drawings. Computer vision techniques would also be useful for discerning camera position, inferring model deformations, and applying kinematics constraints. The computer could orient and deform the 3-D model based on the 2-D drawings. Finally, perhaps the computer could also guess the correspondence between a curve in the drawing and a curve in the 3-D model using simple heuristics based on location, orientation and shape.



(a) Hand-drawn art (b) 3-D model (c) Resulting image

Figure 18: Applying texture to hand-drawn cloth.

Frame-to-Frame Coherence. In an animation sequence, two consecutive frames are likely to be similar. We would like to minimize user intervention by exploiting frame-to-frame coherence, reusing information such as association between drawing and approximating 3-D model, detection and association of feature curves, and model and camera adjustment.

Cloth. As mentioned in Section 8 there are some kinds of figures for which our process does not yet work. Perhaps the most challenging (and probably the most rewarding) class of figures would be those with complex surface textures such as cloth and hair. One of the difficulties with cloth is understanding and how it folds, based on the line art. Given the right set of marker curves, our warp can produce the right kind of behavior. For example, in Figure 18 we show how adding extra marker curves by hand (shown as dashed blue) can disambiguate the line art.

Acknowledgements

We thank Kiran Joshi for posing this problem, as well as Brent Burley, Aliza Corson, Mauro Maressa and Paul Yanover at Disney for their help and advice on this project. We also thank Cassidy Curtis for his guidance, and Tim Milliron for the 3-D examples. Finally, we are grateful to Ronen Barzel, John Hughes, Dan Wallach, and the anonymous reviewers for many improvements to the paper. This work was supported in part by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), Brazil.

References

[1] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. John Wiley & Sons, New York, 1988.

[2] Thaddeus Beier and Shawn Neely. Feature-Based Image Metamorphosis. In Edwin E. Catmull, editor, *SIGGRAPH 92 Conference Proceedings*, Annual Conference Series, pages 35–42. ACM SIGGRAPH, Addison Wesley, July 1992.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.

[4] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. Computer-Generated Watercolor. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 421–430. ACM SIGGRAPH, Addison Wesley, August 1997.

[5] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design: a Practical Guide*. Academic Press, 1997.

[6] Jean-Daniel Fekete, Érick Bizouarn, Éric Cournarie, Thierry Galas, and Frédéric Taillefer. TicTacToon: A Paperless System for Professional 2-D Animation. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 79–90. ACM SIGGRAPH, Addison Wesley, August 1995.

[7] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice.* Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[8] Michael Gleicher. Image Snapping. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 183–190. ACM SIGGRAPH, Addison Wesley, August 1995.

[9] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes: Active Contour Models. *International Journal of Computer Vision*, pages 321–331, 1988.

[10] Seung-Yong Lee, Kyung-Yong Chwa, Sung Yong Shin, and George Wolberg. Image Metamorphosis Using Snakes and Free-Form Deformations. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 439–448. ACM SIGGRAPH, Addison Wesley, August 1995.

[11] Apostolos Lerios, Chase D. Garfinkle, and Marc Levoy. Feature-Based Volume Metamorphosis. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 449–456. ACM SIGGRAPH, Addison Wesley, August 1995.

[12] Peter Litwinowicz. Processing Images and Video for an Impressionist Effect. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 407–414. ACM SIGGRAPH, Addison Wesley, August 1997.

[13] Peter Litwinowicz and Lance Williams. Animating Images with Drawings. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 409–412. ACM SIGGRAPH, Addison Wesley, July 1994.

[14] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time Nonphotorealistic Rendering. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 415–420. ACM SIGGRAPH, Addison Wesley, August 1997.

[15] Barbara J. Meier. Painterly Rendering for Animation. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 477–484. ACM SIGGRAPH, Addison Wesley, August 1996.

[16] Eric N. Mortensen and William A. Barrett. Intelligent Scissors for Image Composition. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 191–198. ACM SIGGRAPH, Addison Wesley, August 1995.

[17] Michael Plass and Maureen Stone. Curve Fitting with Piecewise Parametric Cubics. In Peter Tanner, editor, *SIGGRAPH 83 Conference Proceedings*, Annual Conference Series, pages 229–239. ACM SIGGRAPH, July 1983.

[18] Barbara Robertson. Disney Lets CAPS out of the Bag. *Computer Graphics World*, pages 58–64, July 1994.

[19] D. F. Rogers and J. A. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, New York, second edition, 1990.

[20] Walter Roberts Sabiston. Extracting 3D Motion from Hand-Drawn Animated Figures. M.Sc. Thesis, Massachusetts Institute of Technology, 1991.

[21] Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In Forest Baskett, editor, *SIGGRAPH 90 Conference Proceedings*, Annual Conference Series, pages 197–206. ACM SIGGRAPH, Addison Wesley, August 1990.

[22] Philip J. Schneider. An Algorithm for Automatically Fitting Digitized Curves. In Andrew S. Glassner, editor, *Graphics Gems*, number I, pages 612–626. Academic Press, 1990.

[23] Thomas W. Sederberg, Peisheng Gao, Guojin Wang, and Hong Mu. 2D Shape Blending: An Intrinsic Solution to the Vertex Path Problem. In James T. Kajiya, editor, *SIGGRAPH 93 Conference Proceedings*, Annual Conference Series, pages 15–18. ACM SIGGRAPH, Addison Wesley, August 1993.

[24] Thomas W. Sederberg and Scott R. Parry. Free-Form Deformation of Solid Geometric Models. In David C. Evans and Russell J. Athay, editors, *SIGGRAPH 86 Conference Proceedings*, Annual Conference Series, pages 151–160. ACM SIGGRAPH, August 1986.

[25] Michael A. Shantzis. A Model for Efficient and Flexible Image Computing. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 147–154. ACM SIGGRAPH, Addison Wesley, July 1994.

[26] Robert E. Tarjan and Jan van Leeuwen. Worst-Case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245–281, April 1984.

[27] Frank Thomas and Ollie Johnston. *Disney Animation: The Illusion of Life*. Walt Disney Productions, New York, 1981.

[28] B. A. Wallace. Merging and Transformation of Raster Images for Cartoon Animation. In Henry Fuchs, editor, *SIGGRAPH 81 Conference Proceedings*, Annual Conference Series, pages 253–262. ACM SIGGRAPH, August 1981.

[29] Dan S. Wallach, Sharma Kunapalli, and Michael F. Cohen. Accelerated MPEG Compression of Dynamic Polygonal Scenes. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 193–197. ACM SIGGRAPH, Addison Wesley, July 1994.

[30] Walt Disney Home Video. Aladdin and the King of Thieves. Distributed by Buena Vista Home Video, Dept. CS, Burbank, CA, 91521. Originally released in 1992 as a motion picture.

[31] Lance R. Williams. Topological Reconstruction of a Smooth Manifold-Solid from its Occluding Contour. Technical Report 94-04, University of Massachusetts, Amherst, MA, 1994.

[32] Georges Winkenbach and David H. Salesin. Computer–Generated Pen–and–Ink Illustration. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, Annual Conference Series, pages 91–100. ACM SIGGRAPH, Addison Wesley, July 1994.

[33] Andrew Witkin and Zoran Popović. Motion Warping. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 105–108. ACM SIGGRAPH, Addison Wesley, August 1995.

[34] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Washington, 1990.

[35] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective Panoramas for Cel Animation. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 243–250. ACM SIGGRAPH, Addison Wesley, August 1997.