A Gestural Interface to Free-Form Deformation

Geoffrey M. Draper
Brigham Young University
draperg@byu.edu

Parris K. Egbert
Brigham Young University
egbert@cs.byu.edu

Abstract

We present a gesture-based user interface to Free-Form Deformation (FFD). Traditional interfaces for FFD require the manipulation of individual points in a lattice of control vertices, a process which is both time-consuming and error-prone. In our system, the user can bend, twist, and stretch/squash the model as if it were a solid piece of clay without being unduly burdened by the mathematical details of FFD. We provide the user with a small but powerful set of gesture-based "ink stroke" commands that are invoked simply by drawing them on the screen. The system automatically infers the user's intention from the stroke and deforms the model without any vertex-specific input from the user. Both the stroke recognition and FFD algorithms are executed in real-time on a standard PC.

Key words: Computer graphics, user interfaces, pen-based interfaces, free-form deformation

1 Introduction

3D computer graphics is an active area of research in computer science today. The explosive increase in power of modern 3D graphics hardware, coupled with the equally dramatic decrease in cost, has made 3D modeling well within the reach of the average user's desktop PC. Unfortunately, much of this power remains unharnessed. Perhaps the scarcity of 3D modeling applications for the average user can be partially explained by the steep learning curve generally associated with such systems.

One very nice step towards simplifying the interface of 3D modeling was Teddy [8]. Teddy allows users to create and edit 3D geometric models through a simple set of commands that can be drawn on the screen with a mouse or stylus. 3D modeling is thus cast into the universally appreciable paradigm of pen-and-paper sketching. The work presented in this paper expands the Teddy system to include Free-Form Deformation, or FFD [13], a very powerful mechanism for deforming 3D objects. Our system, nicknamed Freddy, allows users to bend, twist, and otherwise contort 3D geometric objects as easily as they would a lump of clay, without any prior experience with 3D modeling or free-form deformation. This concept of using mouse or stylus strokes as input is called a "gestural interface" (also known as a "pen-based" or "sketching" interface).

Free-Form Deformation [13] is a technology that has a history entirely separate from that of gestural interfaces. Many researchers have extended or otherwise improved the original algorithm [3, 4, 6, 7, 11] since its introduction, but the basic idea has remained the same. A 3D object, usually represented as a polygonal mesh, is surrounded by a lattice of control vertices. As the user displaces the points in the control lattice, the vertices of the polygon mesh are proportionally displaced as well. This proven technique works very well for many applications. However, setting up the FFD lattice and moving its control points to the desired positions can be a cumbersome, time-consuming process.

Freddy combines the efficiency and expressiveness of gestural interfaces with the power of free-form deformation. We have taken three of the most common functions of FFD: the ability to twist, bend, and stretch/squash [9] 3D geometric models, and created a gestural interface for each. By sliding the cursor over the object, the user can create complex deformations that would otherwise require the precise placement of many individual control points.

2 Related Work

This paper merges two previously separate technologies: gestural interfaces and free-form deformation. We now present a brief overview of the research previously conducted in both of these areas.

2.1 Gesture-Based Interfaces

While the traditional command-line interface (CLI) dominated much of early computing, the advent of the Macintosh and Windows operating systems popularized the now-ubiquitous graphical user interface (GUI). CLIs often have great power and extensibility, whereas GUIs are arguably easier to learn. Neither interface, however, completely succeeds at intuitively mimicking people's own perception of how things are done in the real world. Gesture-based interfaces are an attempt to duplicate, to some extent, the motions people commonly use to accomplish certain tasks.

The notion of a gestural interface has been around since the 1960's, when Coleman [2] created a system for editing text using proofreader's marks. However, the majority of work in this area began in the early 1990's, as inexpensive graphics hardware became more readily available. Rubine [12] did much to popularize the study of gestural interfaces by defining a formal set of geometric "features" that one could extract from the user's input strokes, and thereby differentiate between them.

An early example of a gestural interface applied to 3D modeling is Zeleznik, Herndon, and Hughes' SKETCH [14] system. In SKETCH, users can create a wide variety of 3D objects by drawing simple shapes on the screen. Teddy [8] leveraged off of SKETCH's strengths and corrected many of its weaknesses. Rather than building an extensive collection of recognizable strokes, Teddy utilizes a small but powerful set of gestures that allows the rapid construction and editing of rotund 3D objects. As mentioned earlier, our system, Freddy, extends the functionality of Teddy.

In Freddy, the operations of twisting, bending, and stretching 3D objects can be done as easily as drawing lines on the screen. The following section presents a review of the FFD algorithm and describes some extensions to it that have been proposed since its introduction.

2.2 Free-Form Deformation

Barr [1] was the first to introduce the current notion of geometric deformation, although it was constrained to operations about a single axis, and the deformable space was limited to modification by only a few parameters. FFD improves upon this early research in solid modeling, both in expressive power and flexibility.

FFD as we know it today was introduced by Sederberg and Parry [13]. In FFD, a geometric model is enclosed within a parallelepiped lattice of control vertices. Any point X on the model with Cartesian coordinates in world space has corresponding *(s,t,u)* coordinates in lattice space. When the control vertices are displaced, the Cartesian coordinates of the model are re-computed based on their previously calculated *(s,t,u)* values. This simple technique is quite effective at producing dramatic deformations, even for lattices of relatively few control vertices.

Various researchers have enhanced the original algorithm in subsequent years. Coquillart [3] permits not only parallelepiped lattices, but prismatic and cylindrical ones as well. MacCracken and Joy [11] extend Coquillart's idea by allowing lattices of arbitrary topology, using an extension of Catmull-Clark subdivision. Crespin [4] unites many of the previous approaches to FFD into a generalized method called Implicit FFD, or IFFD. In IFFD, rather than using a *single* lattice, the deformation tool consists of a number of "deformation primitives" which can be used in tandem.

The FFD methods discussed thus far require the explicit positioning of the control lattice. Hsu [7] simplifies the interface somewhat by allowing the user to directly reposition one or more vertices of the object itself, rather than manipulating the control lattice. While this does make it easier for novices, it still retains the necessity of dealing with the model on a per-vertex level.

Di Fiore and Van Reeth [5] have proposed one approach to combining gestural interfaces with FFD. Their 3D sketching tool for animators uses free-form strokes like Teddy, but represents them internally as cubic Bézier splines. Their system also supports a simple two-stroke "bend" operation similar to Teddy's. (See section 3.1 for our discussion of Freddy's "bend" function.)

Although other deformation algorithms are available [10], we have chosen to implement our system using traditional FFD due to its simplicity and near-universal familiarity. Our approach simplifies the FFD interface by replacing manual vertex displacement by a series of high-level gesture-based modeling operations. This technique significantly decreases the learning curve traditionally associated with FFD, while retaining much of its power and flexibility.

3 Freddy Overview

Teddy has proven to be a very successful tool for use in creating and editing 3D objects. A complete review of Teddy's interface is given by Igarashi et al. [8], so we will not discuss it in detail here. Freddy maintains all of the features of Teddy, and in addition adds significant functionality and usability. The user interaction in Freddy is accomplished entirely through the drawing of strokes on the screen with a mouse or stylus. The gestures used to initiate particular deformations are intended to be suggestive of their real-world counterparts. For example, to bend an object, the user draws a curved stroke similar in appearance to the object's desired final shape. To twist an object, the user draws a roughly spiral-shaped curve indicative of the direction and axis of the desired twist. To stretch or squash an object, the user simulates the action of pushing or pulling on the object by drawing a line on the screen and dragging it closer to or farther away from the object. The system's gestural interpretation algorithms extract specific features [12] from the user's stroke, and displace the vertices of the FFD lattice accordingly.

Once the vertices of the FFD lattice are displaced, the deformation of the object is computed using the well-known FFD algorithm [13] summarized above. Our primary goal in designing this system was to shield the user from the details of FFD, so the FFD lattice is invisible by default. When the program starts, the FFD lattice is constructed with the same dimensions as the width, height, and depth of the object.

Each operation in Freddy can be broken down into four steps: stroke recognition, stroke interpretation, FFD lattice displacement, and object deformation. We will now discuss each of these in more detail.

3.1 Stroke Recognition

Most of the time a user spends using Freddy will be in its default "idle" state. While the system is in this state, the user can view, rotate, and zoom in and out of the current object. Whenever the user draws a stroke on the screen, however, Freddy exits the idle state and enters the stroke recognition state. The stroke is then transformed into world coordinates via a trivial projection of their 2D pixel positions into their corresponding locations in 3D space. Once this is done, the system extracts a few basic "features" of the stroke [12] in order to determine which of Freddy's stroke interpretation methods should be used. These features include:

- *initial location* (does the stroke start inside or outside of the object?)
- *silhouette intersection count* (how many times does the stroke pass inside or outside of the object?)
- *closure* (does this stroke form a closed loop?)
- *modality* (is the system currently in a special mode, such as creation, extrusion, bend, stretch/squash?).

Freddy's first-pass gesture-recognition algorithm uses these basic features to categorize the stroke. The gesture-recognition state represents only a preliminary sorting; no modeling or deformation action is taken until control is passed to the "stroke interpretation" state, described next.

3.2 Stroke Interpretation and Lattice Displacement

Once a stroke has been categorized using Freddy's gesture-recognition capabilities, control is then passed to the stroke interpretation module, which specifies how objects are selected, bent, twisted, or stretched/squashed.

Selection

By default, the FFD operations described in the following sections affect the entire object. This can be changed, however, by selecting specific regions of the object. To select a sub-region, the user draws a closed stroke that passes both inside and outside the object. The new lattice is formed from the bounding box of the input stroke. This does not, however, imply that *every* vertex of the model that is inside the new lattice will be affected by subsequent deformation operations. Only those vertices of the object whose (x,y) coordinates were inside the polygon formed by the selection stroke are truly "selected," and only these vertices will be subject to deformation in future FFD operations. (An example is shown later on in Figure 13.)

Bending

The *bend* function is performed by drawing a stroke that begins inside the object and crosses the border of the object only once, as shown in Figure 1. While the original Teddy program did support a "bend" operation, it was a two-stroke operation initiated by entering a special mode. Our FFD-based bend function differs in that it is invoked with only one stroke, and requires no special modality. The basic idea behind the bend operation is to fit the shape of the lattice as closely as possible to the shape of the input stroke. The deformed object will likewise be bent along the contours of the stroke.

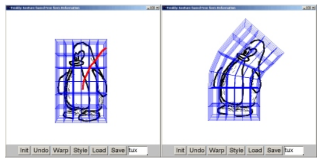


Figure 1: Example of Bend operation

This is accomplished by translating and rotating either the horizontal or vertical planes of the FFD lattice to follow the shape of the stroke as closely as possible. In the bend operation, the lattice's planes are rotated about the z (depth) axis. To determine whether the horizontal or vertical planes of the lattice are to be rotated, Freddy employs a simple heuristic which samples the first ten points of the input stroke. If the y-displacement of the first ten points exceeds the x-displacement, we rotate the horizontal planes of the lattice. Otherise, we rotate the vertical planes of the lattice.

Let the planes of the lattice be numbered from 0 to n, and the plane which is closest to the stroke's starting point be numbered k, where $0 \leq k \leq n$. To determine the exact angle, θ_i, by which to rotate each plane of the lattice, we divide the user's input stroke into $n - k$ equidistant segments, and calculate the tangent of the y-displacement ("rise") and x-displacement ("run") between the endpoints of each segment. Each plane of the lattice is rotated about the z axis by θ_i. In addition to being rotated, each plane in the lattice is also translated in y and in x by the rise and the run. In this manner, the planes of the lattice truly follow the contours of the stroke, as shown in Figure 2.

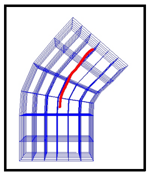


Figure 2: The lattice follows the contours of the input stroke

We treat the stroke as if it were a continuous curve rather than a connected set of discrete points. The system calculates the total length of the stroke by summing the Euclidean distance between each discrete point in the stroke. It then interpolates to find the virtual positions of $n - k + 1$ equally-spaced points along the stroke, regardless of how many or how few points were in the original stroke.

The system then proceeds with the lattice displacement state, as discussed earlier. A pseudocode representation of this algorithm is as follows:

```
let n = the number of planes in the lattice,
        minus 1
    k = the plane closest to the stroke's
        starting point; such that $0 \le k \le n$
    $p_k ... p_n$ = $n-k+1$ equidistant points along
        the stroke

    begin
      for every plane i in lattice from k to n
        rise = $p_i$.y - $p_{i-1}$.y
        run = $p_i$.x - $p_{i-1}$.x
        $\theta_i$ = $\tan^{-1}$(rise / run)
        rotate plane i by $\theta_i$
        translate plane i by rise and run
    end.
```

Figure 3 shows how the previous example would appear to the average user, with the lattice invisible.

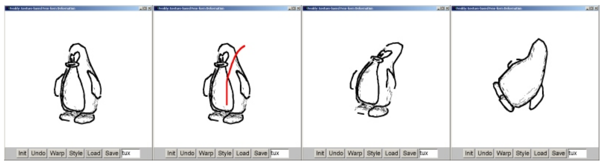


Figure 3: Overview of bending operation (a) initial state (b) bending stroke (c) result of bend (d) rotated view

Twisting

To visualize the basic idea behind Freddy's twist operation, imagine holding a slab of clay with both hands, and then twisting the clay by rotating your hands in opposing directions. The clay between your hands would be twisted, while the clay under your hands would be simply rotated. (See Figure 4.)

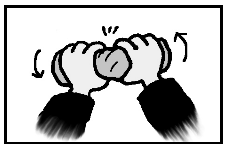


Figure 4: Physical analogy of Freddy's twist operation

The stroke required to initiate a twist operation is an attempt to simulate the physical act of twisting. To twist, the user draws a stroke that starts inside the object and then crosses the object's silhouette at least twice. The first and last points of the stroke represent the effective placement of each hand, and the direction the stroke is heading as it exits/enters the object represents the direction each hand would turn. The part of the object that lies between the endpoints of the stroke is twisted, while the rest of the object is merely rotated in opposite directions.

To determine which axis the lattice planes should be rotated about, the system samples the first ten points of the input stroke. If the x-displacement exceeds the y-displacement, we rotate the horizontal planes of the lattice about the vertical axis. If the y-displacement exceeds the x-displacement, we rotate the vertical planes of the lattice about the horizontal axis.

To illustrate the twist operation, we will step through the example shown in Figures 5 and 6. Suppose the lattice has $n+1$ horizontal planes along the y axis, numbered from 0 at the bottom to n at the top. In this example, $n = 6$. We calculate which two vertices in the lattice have the smallest linear distance to the first and last points of the input stroke, respectively. The planes containing these vertices are labeled k_1 and k_2. The planes at and below k_1 and the planes at and above k_2 are rotated uniformly about the y axis, while the planes between k_1 and k_2 are rotated incrementally, imparting a gradually twisted look to the object. In our current example, $k_1 = 1$ and $k_2 = 5$.

To determine the degree of rotation, θ, we first determine the y-displacement ("rise") and the x-displacement ("run") between the first and last points of the stroke, and divide the rise by the run. Let θ be one-half the arctangent of this value. We then rotate the planes below k_1 by $\theta/2$, and planes above k_2 by $-\theta/2$. Those planes between k_1 and k_2 are rotated incrementally to create a smooth blending between $\theta/2$ and $-\theta/2$. In our current example, the angle of the slope between the first and last points of the stroke is approximately 90°, so $\theta \approx 45°$. The planes are rotated as shown in Figure 6.

The algorithm can be summarized in the following pseudocode.

```
let $\theta$ = angle between first and last points in
        input stroke
    n = number of planes in lattice, minus 1
    k1,k2 = planes of lattice closest to
            first and last points in stroke;
            $0 \le k1 \le k2 \le n$
    begin
      for i = 0 to k1
        rotate plane i by $\theta/2$
      for i = k1 to k2
        rotate plane i by $\theta/2 - \theta*i/(1+k2-k1)$
      for i = k2 to n
        rotate plane i by $-\theta/2$
    end.
```

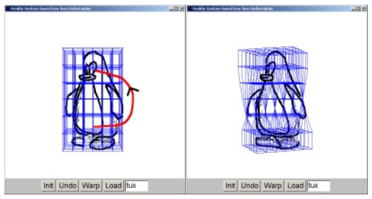


Figure 5: Example of twist operation

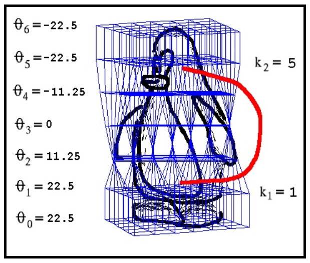


Figure 6: The twist operation, showing the amount of rotation applied to each plane

Figures 7 and 8 show examples of twist strokes of lesser and greater degrees of twist. The stroke in Figure 7 is acute, and would therefore result in a relatively small θ, while the stroke in Figure 8 is obtuse and produces a somewhat greater θ.

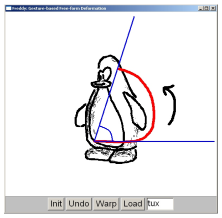


Figure 7

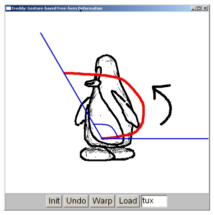


Figure 8

In Figure 9, we show an example similar to Figure 5, as it would appear with the lattice invisible:

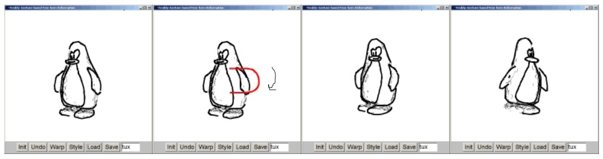


Figure 9: Overview of twisting operation (a) initial state (b) twist stroke (c) result of twist (d) rotated view

Stretch and Squash

This operation is initiated by drawing a stroke completely outside the boundary of the object, which the user can subsequently click and drag either closer to or farther from the object. If the user drags the stroke away from the object, the object will be "stretched" proportionally. Likewise, if the user drags the stroke closer to the object, it will be "squashed."

The system handles straight strokes slightly differently than it does curved strokes. The "straightness" of the stroke is measured by the length of the stroke's bounding-box diagonal divided by the total length of the stroke. For our purposes, a "straight" stroke is one whose straightness is greater than 0.95. A straight stroke deforms the object globally, while a curved stroke deforms a specific region of the object.

We will begin by describing how the system treats straight strokes. When the user has finished dragging the straight stroke to its final location, the system stores the overall displacement of the input stroke between its old and new locations. It also calculates and stores the distance between every point in the displaced input stroke and every point in the control lattice. Additionally, for every point in the lattice, the system stores the longest distance between that point and any point on the stroke. After this preprocessing, the system computes the deformation as described below:

```
let $d_{(x,y)}$ = x,y displacement between old and
         new stroke location
    $\ell_i$ = longest distance from point $i$ in
         lattice to any point on undisplaced
         stroke
    $\delta_{i,s}$ = distance from point $i$ in lattice to
         a given point $s$ in undisplaced
         stroke
    $P_i$ = an (x,y,z) point in the lattice

  begin
    for every point $s$ in the stroke:
      for every point $i$ in the lattice:
        $P_{i(x,y)} \mathrel{+}= d_{(x,y)}(\ell_i - \delta_{i,s}) / \ell_i$
    end.
```

Using this simple algorithm, the lattice is more greatly deformed in the areas nearest the input stroke. The input stroke acts as a sort of magnetic field, with its force to attract or repel gradually lessening as the distance from it increases. Consequently, those polygons which are nearer to the input stroke experience a greater degree of deformation, and those that are farther away receive less deformation. Notice how in Figure 10, the penguin's head and neck are much taller than before, while its feet are largely unchanged.

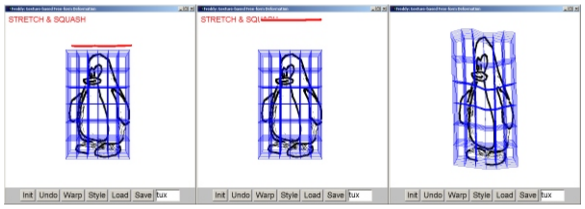


Figure 10: The object receives a greater degree of deformation in regions closer to the input stroke.

Curved strokes are treated somewhat differently. The primary difference is the notion of a "zone of influence." The "zone of influence" can be thought of as an invisible parallelogram that is created by projecting a line segment between the initial and final positions of the stroke's first point, and another line segment between the initial and final positions of the stroke's last point. The remaining two sides of the parallelogram are formed by connecting the first two segments together, as shown in Figure 11.

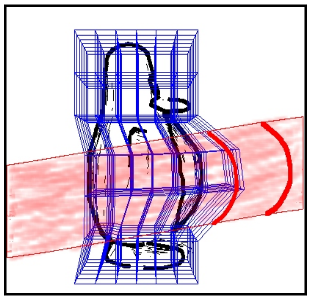


Figure 11: A close-up view of the "zone of influence" created by sweeping an area through the initial and final positions of the stroke

When the lattice deformation is computed, a standard polygonal "inside/outside" test is run on each vertex in the lattice to see if it lies within the parallelogram. Only those vertices of the lattice that fall within this zone are displaced. This allows localized deformations without explicitly selecting a sub-region by hand, as shown in Figure 12.

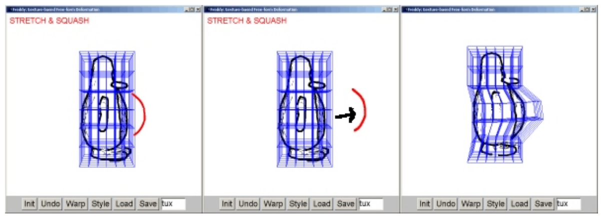
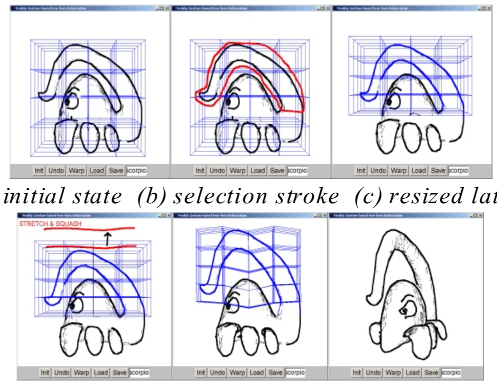


Figure 12: Deformation of FFD lattice for curved Stretch/Squash strokes

We mentioned above that when the user manually selects a sub-region of the object, subsequent deformation operations affect only the selected part of the object, even if the lattice happens to include other parts of the object. A practical example of this behavior is illustrated in Figure 13, using Freddy's stretch and squash operation. In that example, we attempt to deform the scorpion's tail. The lattice that Freddy constructs around the selected region (the tail) also happens to cover much of the scorpion's head. The deformation only affects the area that the user had selected.


(a) initial state (b) selection stroke (c) resized lattice

(d) stretch stroke (e) result (f) rotated view

Figure 13: Localized Stretch & Squash

We show in Figure 14 a few more examples of Freddy's stretch and squash operation, with the lattice invisible as a typical user would see them.

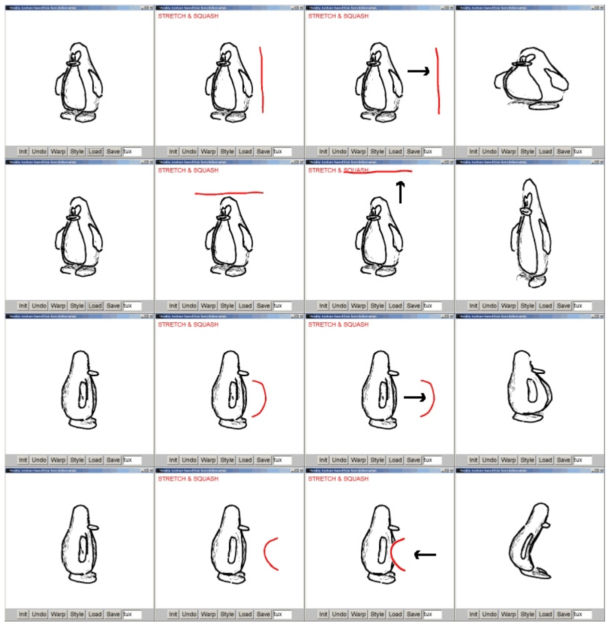


Figure 14: Overview of stretch and squash operation. (a) initial state (b) stretch/squash stroke (c) drag the stroke (d) deformed object

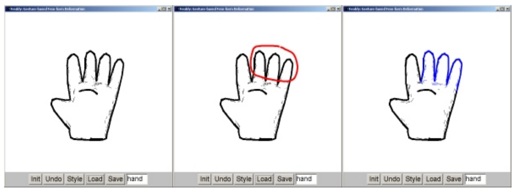
3.3 Object Deformation

Once the vertices of the FFD lattice have been displaced as directed by the appropriate stroke interpretation algorithm, Freddy enters the "deformation" state. In this state, the FFD transformation, as summarized in Section 2.2 above, is applied. The full details of this algorithm are masterfully described by Sederberg and Parry [13]; therefore we shall not dwell further on it here.

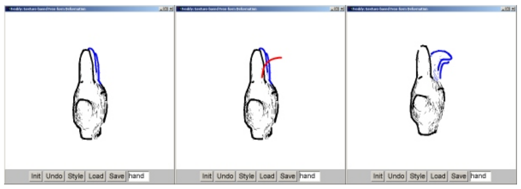
Upon completion of this step, Freddy clears its display and renders the newly deformed object into the viewport. The old lattice is discarded as well, and the system reconstructs a new parallelepiped lattice based on the object's new dimensions.

3.4 Putting it all Together

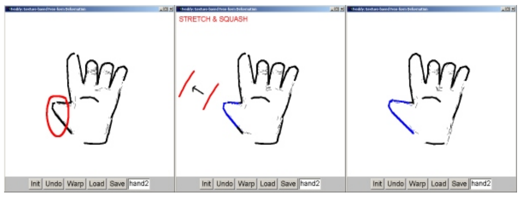
In Figure 15, we show an example that demonstrates the combined power of Freddy's FFD capabilities.



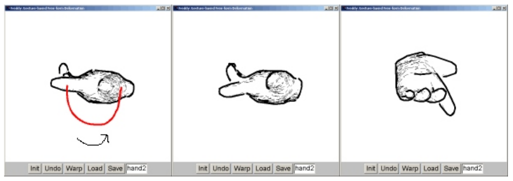
(a)initial position (b)selection stroke (c) selection in blue



(d) rotated view (e) bend stroke (f) resulting deformation



(g) selection stroke (h) stretch operation (i) result



(j) twist stroke (k) result (l) rotated view

Figure 15: A hybrid example of Freddy's FFD operations

4 Implementation

Our system is written entirely in Java, building upon the original Teddy code base. The new code implements the standard FFD algorithm, as well as algorithms to recognize and interpret the user's input strokes and deform the lattice accordingly. Freddy should run on any platform for which there exists a Java Virtual Machine (JVM). In particular, Freddy has been successfully tested on Linux, IRIX, and Windows platforms.

5 Results and Conclusion

In this paper, we have presented Freddy, a system which greatly simplifies the way people can use Free-Form Deformation. The common tasks of bending, twisting, and stretching 3D objects are accomplished via simple "ink stroke" gestures that are easy to learn and use. Traditional FFD user interfaces have required the user to manually set up and transform the vertices of the FFD lattice. Once the lattice is configured, however, the FFD algorithm executes quickly on today's machines. Thus, the limiting factor has commonly been the *user interface*, rather than the hardware or the algorithm itself. The examples shown in this paper were all performed in a matter of seconds, *including* user interaction time, on a mid-range PC. By removing the need to manipulate the FFD lattice by hand, we have given FFD the potential for true real-time interaction by users of varying levels of expertise.

We stated earlier that one of the fundamental ideas behind gestural interfaces is to mimic the actions one would use to perform tasks in the real world. We feel that we have accomplished this in the design of our interface for FFD. The input strokes required to initiate bending, twisting, and stretching were all inspired by the physical metaphor of clay modeling, and in our experience, people generally feel comfortable using and experimenting with Freddy's interface.

While the interface is admittedly not precise enough for refined, mission-critical modeling, it serves as a very convenient tool for brainstorming and rapid prototyping.

6 Future Work

In this paper, we unite two well-respected topics of research, gestural interfaces and free-form deformation. Previous work in both of these areas had been limited primarily to two separate domains of computer science: human/computer interaction and 3D geometric solid modeling.

While Freddy's algorithms and interface are robust enough for common deformation operations, they may produce erratic results if the user draws unexpected strokes. While this is acceptable for a research prototype, more robust recognition schemes will be required if this system is to be adapted for widespread use. In particular, the "twist" operation has perhaps the greatest room for improvement. The recognition and deformation algorithms for "twist," as well as the interface itself, are not as straightforward as for the "bend" and "stretch" functions. Hence, this feature tends to be the most difficult for new users to grasp.

The system currently does not scale well to large numbers of repeated deformations. For example, applying several "twist" operations in succession to a model can quickly render it unrecognizable. It is an open question as to whether or not the system should permit the user to do this; however, checks and balances could be put into place to prevent the model from deteriorating too rapidly.

Another potential area for enhancement would be to use FFD lattices of arbitrary topology [11]. Our current implementation supports only the traditional parallelepiped lattice described by Sederberg and Parry [13]. Lattices of arbitrary topology, combined with additional and more robust input strokes, may possibly allow for more precise control of the deformation by the user.

References

[1] A.H. Barr. Global and Local Deformations of Solid Primitives. *SIGGRAPH 84 Proceedings*, pages 21-30, 1984.

[2] M.L. Coleman. Text Editing on a Graphic Display Device Using Hand-Drawn Proofreader's Symbols. *Proceedings of the Second University of Illinois Conference on Computer Graphics*, pages 283-290, 1969.

[3] S. Coquillart. Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling. *SIGGRAPH 90 Conference Proceedings*, pages 187-196, 1990.

[4] B. Crespin. Implicit Free-Form Deformations. *Implicit Surfaces*, 1999.

[5] F. Di Fiore and F. Van Reeth. A Multi–Level Sketching Tool for "Pencil–and–Paper" Animation. *Proceedings of AAAI Spring Symposium on Sketch Understanding*, pages 32-36, 2002.

[6] G. Hirota, R. Maheshwari, and M.C. Lin. Fast Volume-Preserving Free Form Deformation Using Multi-Level Optimization. *Fifth Symposium on Solid Modeling*, pages 234-245, 1999.

[7] W.M. Hsu, J.F. Hughes, and H. Kaufman. Direct Manipulation of Free-Form Deformations. *SIGGRAPH 92 Conference Proceedings*, pages 177-184, 1992.

[8] T. Igarashi, S. Matsuoka, H. Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. *SIGGRAPH 99 Conference Proceedings*, pages 409-416, 1999.

[9] Lasseter, John. Principles of Traditional Animation Applied to 3D Computer Animation. *SIGGRAPH 87 Conference Proceedings*, pages 35-44, 1987.

[10] F. Lazarus, S. Coquillart, P. Jancène. Axial Deformations: An Intuitive Deformation Technique. *Computer-Aided Design*, 26(8), pages 607-613, 1994.

[11] R. MacCracken and K.I. Joy. Free-Form Deformations With Lattices of Arbitrary Topology. *SIGGRAPH 96 Conference Proceedings*, pages 181-188, 1996.

[12] D. Rubine. Specifying Gestures by Example. *SIGGRAPH 91 Conference Proceedings*, pages 329-337, 1991.

[13] T.W. Sederberg and S.R. Parry. Free-Form Deformation of Solid Geometric Models. *SIGGRAPH 86 Conference Proceedings*, pages 151-160, 1986.

[14] R.C. Zeleznik, K.P. Herndon, and J.F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. *SIGGRAPH 96 Conference Proceedings*, pages 163-169, 1996.