Pliant Type

Development and temporal manipulation
of expressive, malleable typography

by Peter Cho

Submitted to the Department of Mechanical Engineering
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

at the Massachusetts Institute of Technology
May 1997

Signature of Author
Department of Mechanical Engineering
May 8, 1997

Certified by
John Maeda
Assistant Professor of Design and Computation
MIT Media Laboratory
Thesis Advisor

Accepted by
Peter Griffith
Professor of Mechanical Engineering
Chairman of the Undergraduate Thesis Committee

Pliant Type

Development and temporal manipulation
of expressive, malleable typography

by Peter Cho

Submitted to the Department of Mechanical Engineering
on May 9, 1997 in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Abstract

Text is not limited to static presentation in digital communication. Past research into temporal typography has focused on changing the size, orientation, color, and position of typographic forms while keeping the letterforms themselves intact. This thesis proposes *pliant type* as an area of study within temporal typography in which letterforms are malleable shapes which can move in expressive ways. This thesis presents a shape representation for type which allows the manipulation of typographic shapes on a basic level. Two shape representations have been implemented: outline and skeletal. In pliant type design experiments, letterforms are created using a *shape builder*. These shapes then are manipulated through interaction with the user and with computational engines. Pliant type suggests new ideas for expressive visual communication. The research contributes to a broader understanding of temporal typographic design.

Thesis Supervisor: John Maeda
Title: Assistant Professor of Design and Computation, MIT Media Lab

Acknowledgements

I would like to thank John Maeda, my thesis advisor, for his guidance and constant support. My thanks go to Glorianna Davenport, who first entrusted work to me as an undergraduate researcher, and Angelynn Grant, who encouraged my interest in typography. I would also like to thank my friends and colleagues at the Media Lab—Reed, Chloe, Sawad, Tom, Dave, Matt, Jared, Elise, and Phillip.

Table of Contents

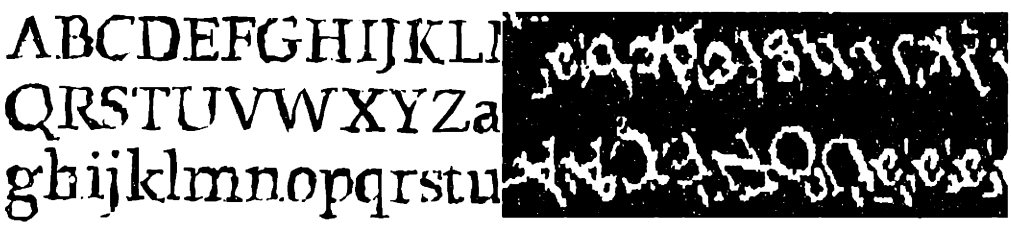
1 Introduction

Type is the building block of written communication. Typography evolved through history—at different points of time, type was carved into clay, chiseled into stone, and cast in metal to be printed onto paper. Type in the digital medium is represented as data. On the computer, the outline of each letter of a font is represented as points in space connected by line and curve segments. This research approaches type as computational shapes which can be manipulated expressively in real time. The treatment of typography as malleable shapes, *pliant type*, adds new possibilities and complexities to the design of temporal typography.

Temporal typography is the dynamic treatment of text—essentially, type that changes its form over time. Examples of temporal typography can be seen in different time-based media: television commercials, introductory film and TV sequences, computer games, and most recently, the web.

While these examples of temporal typography often make use of computers in the design process, the computer in these cases serves as a high-level designer's tool. In this research, as well as in other research conducted at the MIT Media Laboratory and elsewhere, computation is used to explore typographic expression on the computer through the use of motion and interactivity.

When type is represented digitally, it becomes relatively easy to manipulate the letter shapes. Graphics programs such as Adobe Illustrator allow the user to convert fonts into vector-based paths which can be modified point by point. Some experiments in manipulating letter shapes on a computational basis have been made: Just van Rossum and Erik van Blokland's font *Beowolf*, for example, has "randomness" coded into it so



Figures 1 and 2: Beowolf and Nimida represent attempts at "computational" type.

that the individual letterforms change every time they are printed. Van Blokand's font *Nimida* randomly degenerates its letterforms. While facilities for manipulating the shape of type are available, the potential for developing expressive, computational deformation of letters in the context of words remains unexplored.

Motivation

Letters are abstract shapes. We have the ability to read a wide range of writing styles: serif, sans serif, script, and display typefaces, calligraphy, even bad handwriting. Similarly, we can recognize letters in the shapes of the physical world. If letters are abstract shapes, typefaces represent only specific instances in a wide spectrum of letter shape possibilities. This research proposes that new expressive design solutions and possibilities become available in the digital medium when letterforms are mutable.

This thesis examines some of the issues which arise in temporal typography when letterforms themselves change shape. These issues include devising a flexible shape representation which is responsive to the specific needs of typography, developing a method for building pliant type, and exploring the kinds of expression pliant type makes possible.

In order to examine these issues, two letter shape representation schemes, outline and skeletal, were implemented, a tool was developed for building pliant type, and several design experiments were made to explore the possibilities for dynamic and interactive pliant type.

Organization of this thesis

The next chapter contains background information about expressive typography in print and digital media, including related work in temporal typography. Chapter 3 discusses the design issues involved in developing pliant type. Chapter 4 describes the implementation of the two shape representation schemes and the shape building tool. Chapter 5 discusses the design experiments. Chapter 6 draws conclusions from the research and suggests future research in the area.

2 Background and Related Work

Expressive typography

Examples of expressive typography, type designed to communicate emotion and meaning, can be found in print, motion pictures, and digital media.

Letters on a page have a tremendous opportunity to evoke expression, through contrasts of size, weight, structure, form, color, direction, texture, and other attributes. The German typographer Jan Tschichold advocated in *Die Neue Typographie* (1928) that a dynamic force should be present in each design. Tschichold, arguing that type should be placed in motion rather than at rest, favored kinetic asymmetrical design of contrasting elements.

Examples of dynamic and expressive typography can be found in Bauhaus works. Lazlo Moholy-Nagy's poster for Pneumatik tires, for instance, gives the letterforms a perspective treatment (Fig. 3). This is an example of conveying motion on a static page by manipulating and distorting the typography—placing the type on a plane other than the plane of the page.

Figure 3: An example of type distortion. Lazlo Moholy-Nagy, **Pneumatik**, 1923

In his design of Eugène Ionesco's absurdist play, *The Bald Soprano*, Robert Massin treats the page as a stage for type (Fig. 4). Photographs and lines of text are orchestrated to represent the inflections of voice, the awkward silences, and the commotion of multiple voices in a visual manner. The characters' movement on the stage, their vocal expressions, and their emotions are conveyed through the typography.

Adding the dimension of time creates new possibilities for expressive typography. Many examples of temporal typorgraphy can be seen in

Figure 4: An example of expressive typography in print. Robert Massin, **La Cantatrice Chauve**, 1964.

motion picture titles, television commercials, and other venues. Saul Bass pioneered the use of temporal typography in the introductory titles for many films. In the opening titles for "Psycho," for example, horizontal lines move across the picture and reveal broken letterforms, making the viewer feel the unease associated with the word "psycho" (Fig. 5).

Interactive temporal typographic examples are not common in part because the technology needed to develop such pieces was not available until recently. Furthermore, most authoring environments which allow for complex user interaction and animation of typography require programming skills most designers and artists do not possess.

The dynamic, interactive work of John Maeda, however, serves as an example of what can result when a creative designer has a full grasp of programming. In his interactive book, *Flying Letters*, for instance, Maeda uses creative typographical experiments to amuse and to suggest new ways of thinking about letters and the ways we can interact with the computer.

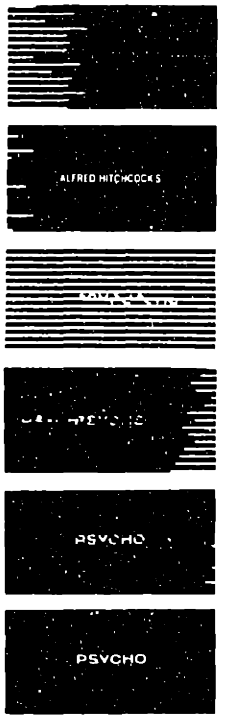


Figure 5: A temporal typography sequence. Saul Bass, titles for Hitchock's **Psycho**, 1960.

Related research

Much of the research in expressive temporal typography has come from the MIT Media Laboratory. The work of David Small and Yin Yin Wong in particular

has contributed to this study. Small (1987) develops the notion of "expressive typography," creating examples in which object dynamics are applied to typography, treating type as if it were part of the physical world. Wong (1995) presents a framework for thinking about and designing temporal typography. In the characterization scheme she develops, pliant type can be thought of a complex visual technique.

Jason Lewis' work (1996) at the Royal College of Art focuses on exploring expressive typography within the context of poetry. He stresses the need for devising new methods of expressing content in the digital medium.

In a previous design experiment, I looked at a single letterform—the A—as a playful interactive element. As the user moves the mouse around the screen, the letter appears to dance and smile. Pliant type builds on the idea that letterforms are shapes which can move expressively. This research applies this idea to words, using expressive motion in this larger context.

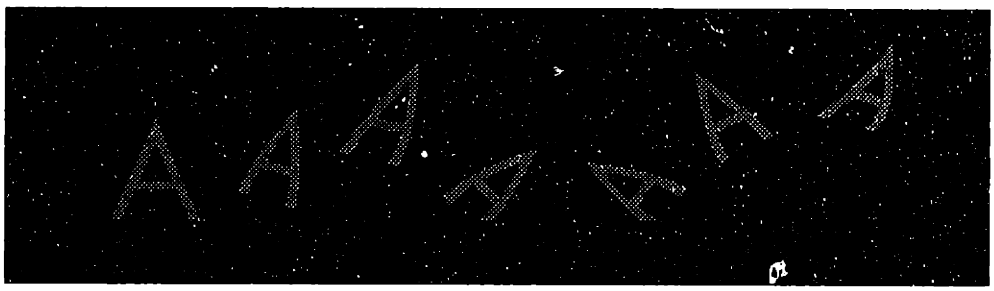


Figure 6: The dancing **A**. Cho, 1996.

3 Design Issues

Digital typography and letterform design

Type design is a painstaking process. In developing a font, typographers must craft each letter so that each character works both on its own and with every other letter. I have great respect for this process. While I created my own letterforms in this research, I do not profess to be highly experienced in type design.

In developing pliant type, I considered using the data from actual fonts in my design experiments. While this could be a future step, I decided instead to create the type from scratch using the tools I built. This gave me control over the letterforms and thus the most relevant typeface experiments.

Legibility

The deformation of type brings up the issue of legibility. At a certain point during deformation, a letter becomes no longer recognizable as that letter. If the letter is part of a word, the surrounding letters may provide enough context so that the word's meaning remains intact. Sometimes during type deformation, a letter can change shape and be perceived as a different letter, or as both the original letter and the new letter, as when the ascender of a lowercase *h* is low enough so that the *h* can be seen as a lowercase *n* (Fig. 7).

In this research, I became interested in looking at the boundary at which a form is still recognizable, but distorted in ways that make it a less pleasing form. This is related to another issue of temporal typographic design. Many tools for dynamic design allow the author to set keyframes at specific times, and the system interpolates the frames in between. In some cases, the designer may create coherent designs at the keyframes

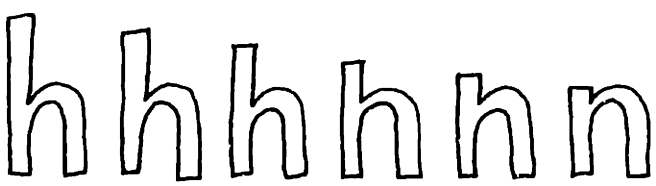


Figure 7: Ambiguity in transition from lowercase h to lowercase n.

while the interpolated frames are a mess. This issue is raised: does the author want to have control so that every frame, including every intervening frame, is well-crafted? How does one devise a well-designed motion?

This issue of temporal design is related to pliant type deformations. Some of the user-controlled deformations of pliant type are open-ended, meaning the type begins in a certain state designed by the author, then the reader is free to deform the type in certain ways. In this case, does the author want to make every possible outcome be well-designed? One solution is to develop the reader's interactions carefully so that control over what he sees is maintained. Another solution is to develop pliant type which holds a memory of its form so that regardless of user interaction, the type shapes remain intelligible.

Scope of pliant type expression

In the pliant type design experiments, my goal was to use the meaning of words to inform the kind of deformations which take place. The issue of scope was brought up. Deformation can occur at any number of levels, from the parts of letters to entire bodies of text. The dancing *A* showed that typographic distortion can be effective on the level of a single letter. One could imagine deformation applied to just part of a letter—a serif which expands and shrinks, for example. At a much higher level, pliant type ideas could be used to enhance the expression of a temporal typography poem.

While this research considers pliant type expression at the level of single words, the deformation actually occurs at the level of individual letters.

Letter shape representation

In this research, I was faced with the issue of how to represent letter shapes. The two basic options—outline and skeletal representation—each have advantages and disadvantages. The two representations differ in flexibility and complexity.

Outline representation involves describing a letter as a closed shape. The data points for such a shape lie on the outline of the shape; the points are connected by segments, and the shape is filled. Basically, this is the current implementation of digital fonts.

Skeletal representation, on the other hand, involves conceptualizing a letter as a path with a stroke width. The data points for a skeletal letter lie

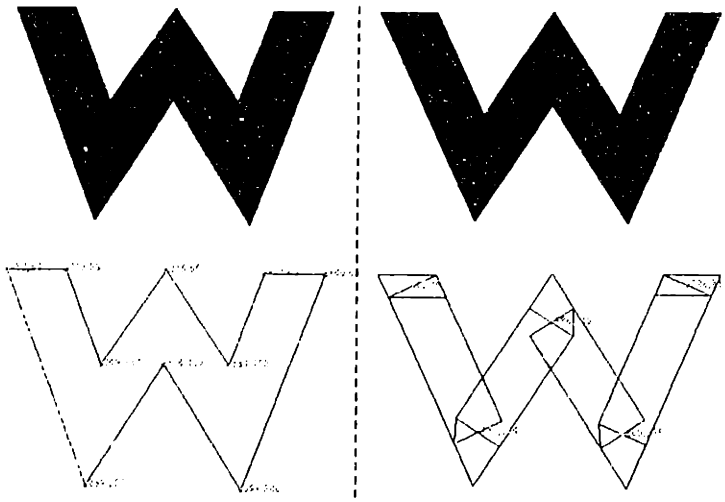


Figure 8: Shape representations compared. Letters on the left are represented by the outline shape model. Letters on the right, by the skeletal model.

on the inside of the shape. Each point has a stroke width. The points are connected, and the shape is filled from the inside out to the specified stroke width at each point. This representation is closer to how we write—we draw letters as paths, where the stroke width is the width of our pen.

While the outline representation is more flexible than the skeletal representation in terms of the letter shapes which can be created, outline fonts are harder to manipulate in an intuitive manner. Since an outline letter has no knowledge of its "stroke width," it is difficult to perform manipulations in which the stroke remains constant. In contrast, a skeletal letter can be easily manipulated so that the stroke either remains constant or changes in a controlled manner.

One drawback of the skeletal representation is that it is more complex and therefore more computationally intensive than the outline representation. Whereas the data points of an outline letter can easily be converted into a filled polygon to be drawn to the screen, the filled segments of a skeletal letter must be calculated before the letter can be displayed. In addition, "corner points," or points where line segments intersect, must be calculated (see Chapter 4 for details). These complexities make real-time manipulation of complicated skeletal type less feasible than outline type manipulation on slower systems.

In this thesis, I implemented both outline and skeletal type representations and created pliant type experiments to test the features of both models.

4 Implementation

The pliant type design process involves two steps: designing letterforms using the shape builder, then designing the computational methods that control the pliant type motion. This section describes the shape representations, the shape manipulation methods available to these representations, and the shape building tool. This research is implemented in Java.

Outline shapes

The outline shape representation was developed first. In this representation, a list of coordinates represents the perimeter of the letter shape. These points are connected in order by straight line segments which create an open-path shape. A line segment from the last point to the first point closes the shape. Each point has a weight value which specifies that point's tendency to stay where it is during deformation. Though designating weights for each point was an implemented feature, I did not make use of it in my design experiments.

Once I created this outline shape representation, I developed the methods by which the shape could be manipulated on a basic level. Individual points of the shape can be translated in x and y. Additionally, points of the shape can be moved so that neighboring points also move, depending either on their distance in the chain or their absolute distance in the xy plane. A point can also move toward or away from another point in the shape or a specified xy point outside of the shape. Line segments between two neighboring points can be moved, lengthened or shortened.

Skeletal shapes

The skeletal type representation was developed next. In this representation, a letter shape, as in the outline representation, consists of a list of coordinates. However, these points make up the "backbone" of the letter rather than the outline. The skeletal shape is given an overall stroke width, and each point is assigned a stroke value which is a fraction of the overall stroke width.

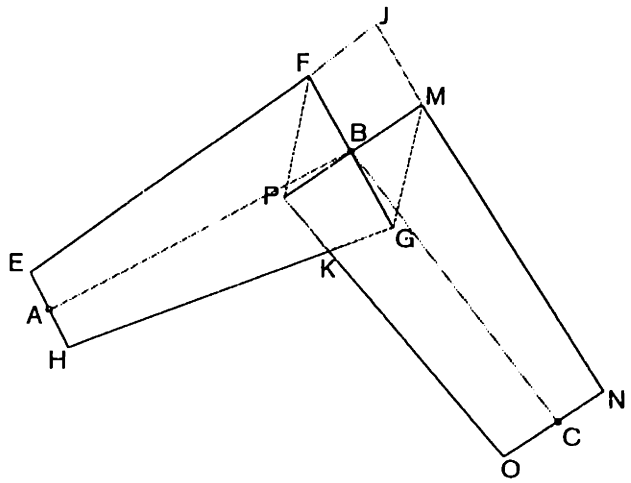


Figure 9: Construction of skeletal shape segments.

This representation involves some geometric calculations. Two adjacent points in the skeletal shape representation are connected by a quadrilateral as shown in Figure 9. If two adjacent points are called A and B, a quadrilateral (EFGH) is constructed between these points which has the following properties:

- A bisects EH
- B bisects FG
- EH is perpendicular to AB
- FG is perpendicular to AB
- The length of EH is determined by the stroke width at A, and the length of FG is determined by the stroke width at B.

If B is not an end point of the shape, a quadrilateral (MNOP) is constructed connecting points B and C. A polygon (FJMGKP) is also constructed at point B to provide a "corner" at this point. Point J is at the intersection of EF and MN; point K is at the intersection of GH and OP. This polygon is constructed instead of the quadrilateral FJMB because, in the current system, displaying this second quadrilateral leaves undesirable artifacts along the line segments FB and MB. If the angle of ABC is below a certain "mitre" value (20°), the quadrilateral FMGP is constructed instead of FJMGKP.

Skeletal shapes have the same basic methods for manipulations as outline shapes—methods for moving points, groups of points, and lines—in addition to operations for changing line widths.

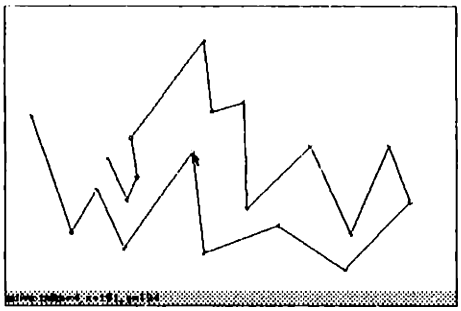


Figure 10: The outline shape builder.

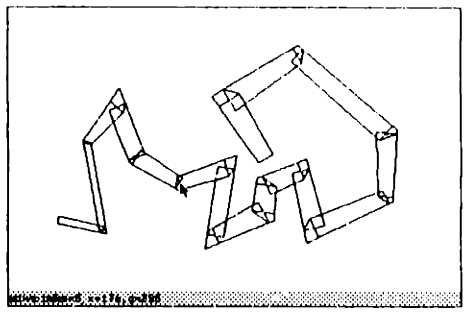


Figure 11: The skeletal shape builder.

Both outline shapes and skeletal shapes allow saving and reverting shapes—storing the current state of a shape, performing some manipulations, then returning to the saved state.

Shape builder

I created a simple tool for building outline and skeletal letter shapes. The shape builder runs as a Java applet. Using this tool, the author can input shape points interactively using the mouse. Points can be moved by selecting and dragging with the mouse, added, or deleted. The shape builder for skeletal shapes also allows the author to change the stroke width at each point by using the keyboard. Line segments can be moved or changed in length. The shape builder permits the user to see information about the shape such as point coordinates, line lengths, and angles formed by line segments. In the shape builder, letter shapes can be viewed in either filled or outline mode. Figures 10 and 11 show sample screens from the outline and skeletal shape builders.

5 Design Experiments

I designed several pliant type experiments to see the possibilities of malleable, moving type. In each experiment, I built a simple word using the shape builder, then made it dynamic and interactive. I chose short words to make the shape building process simpler. In choosing words for these design experiments, I avoided words which describe concrete physical actions—words such as bulge, melt, wiggle—because I wanted to develop more abstract motions. I also felt action words that were too specific would give the reader preconceptions about what sort of movements should take place.

Sleepy

In the first pliant type experiment, the word *sleepy* is animated over a period of about 40 seconds. The letters, created using the outline shape representation, begin to droop, or elongate vertically, then jolt back to the original shape. While drooping, the letters also change slowly from white against a black background to a gray color, returning to the white color when they jolt back. They begin to droop a second time, again changing color, then jolt back again. Finally the letters begin to droop and continue to droop, while slowly becoming darker. The letters move downward until they appear to rest on a common horizon. Once all the letters reach this resting state, the letters shrink and expand slowly until the gray color transitions to black.

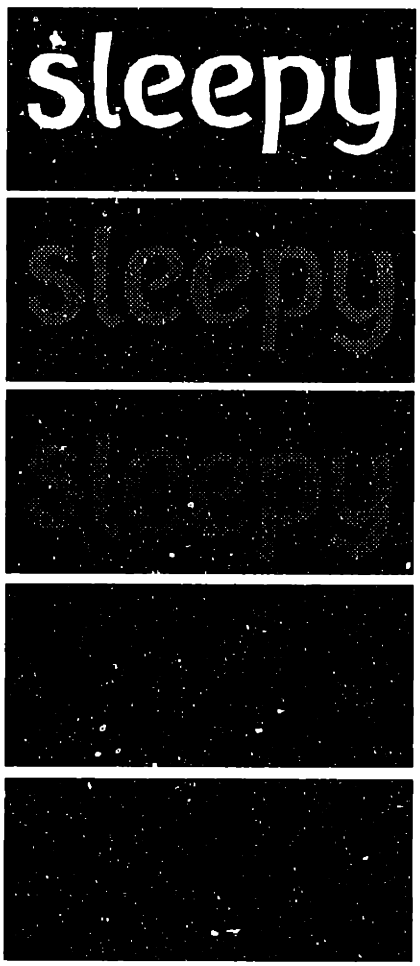


Figure 12: Pliant type experiment, **Sleepy**.

In this experiment, I attempted to portray the feeling of being sleepy. The drooping

downward and jolting back of the letters represents the physical motion of the head or eyelids as a drowsy person dozes off, then jolts back awake. The third time the letters begin to droop represents the action of drifting off to sleep. The expanding and contracting of the letters depicts a sleeping person's slow breathing, the motion of the lungs.

In this first experiment, pliant type animation is used to tell a story. The shape of the word becomes a character who tries to stay awake, then finally falls asleep. In essence, the motion of the letters in this example describes the word itself. This is a case in which the motion is integrated with the definition of the word.

Stitch

Next I created an animation of the word *stitch*. In this piece, the word is drawn in cursive by a single continuous line. The animation begins with a horizontal line, then adds points in the line sequentially, spelling out the word as if the letters were being stitched like a thread into cloth. Once the word is complete, points are removed from the beginning of the word, as if the thread were being unravelled.

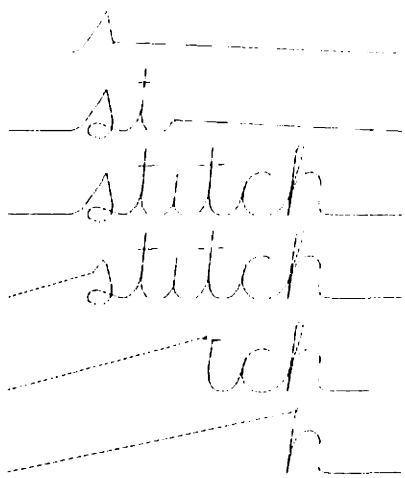


Figure 13: Pliant type example, **Stitch**.

While this experiment does not make use of the manipulation features of the outline shape representation, this example does demonstrate an interesting effect.

Aim

The next design experiment involved an interactive, dynamic treatment of the word *aim*. Instead of playing out a pre-programmed animation like the two previous pieces, this example responds to input from the mouse in real time. When the mouse button is pressed and the cursor moves around the screen, the letter shapes deform, moving toward and following the mouse cursor. When the mouse button is released, the word slowly returns to its original shape.

The implementation of this interaction is straightforward. Each of the letters is composed of many data points. When the mouse is moved, each

data point moves toward the mouse point by a certain amount, depending on its distance from the mouse.

The user interaction is important in the effectiveness of this piece. In this example, the word seems to be "sucked in" by the mouse or recede into the distance due to the way the letters become smaller during the mouse motion. This type manipulation seems to work well with the word choice, although the motion illustrates the word's meaning only in an abstract way. The example is helped visually by the fact that the letters *A*, *I*, and *M* have only straight line segments. A letter with curves, approximated by a few straight line segments, might be distracting in this example.

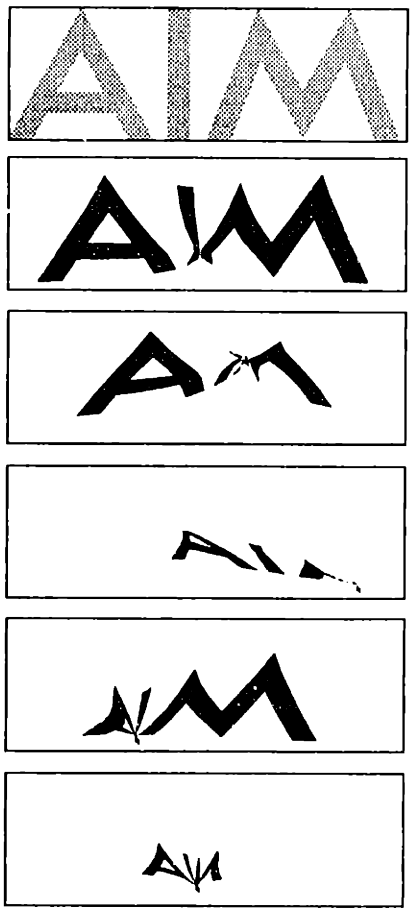


Figure 14: Interactive pliant type expample, **Aim.**

Typographic toolkit

This piece was constructed as an example of using pliant type for a more commercial application. The words *typographic toolkit* were built crudely using the outline shape builder.

When the mouse moves near the letters, the letter shapes bulge and distort. This effect was created by directing the shape points away from the mouse point.

This piece was used to make a short animated sequence for a video about this project. In the animation, the words appear, then are "pushed off" of the screen by some unseen force.

Figure 15: Application of pliant type, **Typographic toolkit.**

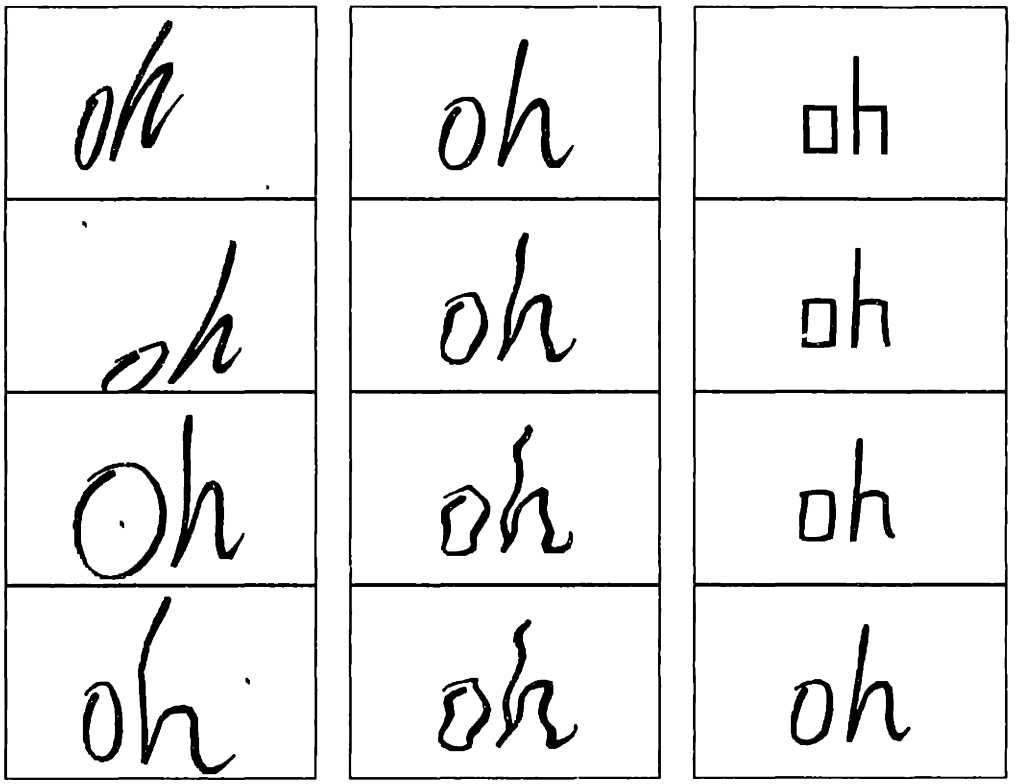


Figure 16: Pliant type experiments using the skeletal shape representation, **Oh 1, 2, 3**.

Oh 1, 2 and 3

In the final experiment, three examples of pliant type manipulation were created using the same word, *oh*. A calligraphic *oh* was created using the skeletal shape representation. In the first example, the letters distort according to the mouse position. The letters appear to be "pushed away" by the mouse. In the second, the letters undulate, or animate in a "wavy" manner. The amount of wavy action is determined by the horizontal position of the mouse. In the third example, the letter shapes become boxy when the mouse button is pressed, then slowly return to the calligraphic style on release.

These three examples demonstrate some of the features made easier by the skeletal shape representation. However, the third of the three experiments is the only manipulation that would be difficult to repeat using the outline shape representation. In the third example, it is the actual paths of the letters and the stroke widths that change. The real benefit of using the skeletal shape representation in this case occurred in the type design process. Variable stroke, brush-like characters are much easier to create using this shape model.

6 Conclusion and Discussion

This thesis represents a new way of thinking about temporal typographic expression. The computational treatment of typography as malleable shapes is a design concept which deserves further study.

Type representation

In this research, type is represented in two ways, outline and skeletal. Each was found to have its own benefits. Outline representation tends to lend itself toward more drastic deformations, while the skeletal representation allows letter shapes to remain more intact. In general, outline representation seems to be the more flexible of the two in terms of the kind of shape manipulations which can take place. Skeletal representation however is more appropriate for deformations in which the path of the letter is important and the stroke width of the letter stays constant.

The current letter shape representations suffer visually because curved parts of letters have to be approximated by multiple straight line segments. A future development would involve implementing cubic spline or Bezier curve capabilities so that the letter *o*, for example, can appear smooth while being represented by only a few points.

Another possibility for future development is to build the capability of importing actual fonts and using them in pliant type designs. This step would likely be appreciated by graphic designers. This would also be a step towards incorporating pliant type concepts into other temporal typography designs.

Steve Strassmann's computer graphics paper Hairy Brushes (1986), which I received late in the process of this research, focuses on representing brush strokes on the computer with four variables: the brush shape, the stroke, the dip in ink, and the paper. Since his brush stroke representation is similar to the skeletal letter shape model, his work suggests different ways type shapes could be expressed.

Metafont, developed by computer programming guru Donald Knuth, also suggests different ways typography could be represented. This system for

describing fonts allows for creating type on a high-level computation basis.

Level of pliant type manipulation

In the pliant type experiments, words are manipulated at the level of individual letters. Pliant type expression at different levels is possible. For example, it may be useful to group points of a letter shape together so that part of the letter acts as a unit. This unit may have certain behaviors depending on its position with respect to the mouse or other units. A parts-based letter representation may have advantages over the current point-based letter representation.

Computation allows objects to have behaviors and an understanding of certain rules. Since the objects which are created in pliant type experiments are letters, these letters could be given an understanding of typographic rules. For example, the letter shapes could be encoded with knowledge about x-height of letters, ascenders and descenders, kerning, even ligatures. Since these letters are also elements of written and spoken language, the pliant type letters could conceivably have information about vowels and consonants, phonetics, and syllable stress. Pliant type expression could potentially draw from many levels, from letters to lexical.

Other applications

Pliant type imparts letters with character through computational methods. When designed carefully, pliant type has the potential to create expressions of concepts and emotions difficult to achieve with other methods. Pliant type ideas may be applied in different contexts. One could imagine dynamic concrete poetry in which the movement of the typography evokes images or emotions. An interactive and dynamic logo could also use type which changes in shape. Pliant type could be used in those media where temporal typography can be seen—motion picture titles and television commercials, to name the most prevalent.

Design and computation

Before the computer, designers had many techniques for playing with type. Using photo type, they experimented with using different lenses, printing onto glass and other surfaces, and controlling lighting to get innovative and expressive typographic effects. Massin (Fig. 4), for instance, experimented with printing type onto a rubber surface, then

stretching the sheet to get different effects. Now that the computer is the the tool of choice for many designers, these techniques for the most part are no longer being developed.

This research has shown that type effects along the same lines as techniques by manual methods can be achieved on the computer. Reproducing photo type effects exactly through computational methods, however, would be difficult if not impossible. The question comes to mind: how does working by computer to create this sort of design differ from working by hand? In general, how does computation enhance and detract from the design process?

Certainly, computation makes some tasks, especially those involving complex calculations, easier. Computation allows for complex interactive design. It also facilitates trying many different options; changes can always be undone. On the other hand, traditional methods of design often have a proclivity toward giving character to crafted pieces. Designs which are created by traditional methods often convey a sense of handicraft—a sense that someone's hands have crafted the design. This sense is difficult to achieve with works on the computer.

Both computational and traditional methods have benefits and drawbacks. The two can be used together. Paul Rand notes, "The conflict between design and technology, like the conflict between form and content, is not an either/or problem, it is one of synthesis" (Rand, p. 41). It is hoped that design on the computer is not perceived as its own category, "multimedia," but that it is realized the computer may be anywhere in the broad spectrum of design, co-existing with traditional methods at every level of design.

Appendix A Code

This appendix includes the Java implementation of several parts of the pliant type research: the skeleton shape representation class SkeletonShape, the set of utilities in the class Utils, and the Oh applet running the final pliant type experiment described in Chapter 5.

SkeletonShape.java

```
package pcho.typotool;

import java.awt.*;
import java.lang.*;
import pcho.util.*;

public class SkeletonShape extends Object {
   static final int maxPts = 200;
   protected int numPoints;
   protected int strokeWidth;
   protected double[] s; // fraction of strokeWidth. must be greater than 0
   protected double[] x, y; // arrays to hold x,y coordinates of shape points
                     // stored as doubles for accuracy, converted to ints for screen

   protected int SnumPoints = 0; // saved number of points
   protected int SstrokeWidth;
   protected double[] Ss;  // saved strokes
   protected double[] Sx, Sy; // these hold saved x,y coordinates

   // these determine how skeletonshapes are "finished" at the begin and end
   public static final int NORM = 0;
   public static final int HORIZ = 1;
   public static final int VERT = 2;

   public int begin = NORM;
   public int end = NORM;

   // only one item can be active at a time
   //   activeItem=POINT:  point at active
   //   activeItem=LINE:   line i,i+1
   //   activeItem=ANGLE:  angle formed by lines i-1,i and i,i+1
   public static final int POINT = 0;
   public static final int LINE = 1;
   public static final int ANGLE = 2;

   public static final int NONE = -1;

   public int activeItem;
   public int active; // index of x,y arrays

   public boolean showPointInfo = false;
   public boolean showLineInfo = false;
   public boolean showAngleInfo = false;
   public boolean showIndexInfo = false;
```

```
public boolean drawPoints = false;

Polygon poly; // use getPolygon() to get a Polygon describing Shape

public SkeletonShape() {
   x = new double[maxPts];
   y = new double[maxPts];
   s = new double[maxPts];
   numPoints = 0;
   strokeWidth = 1;
   activeItem = POINT;
   active = NONE;
}

public SkeletonShape(int width) {
   // Shape with specified stroke width for all points
   x = new double[maxPts];
   y = new double[maxPts];
   s = new double[maxPts];
   numPoints = 0;
   strokeWidth = width;
   activeItem = POINT;
   active = NONE;
}

/* saving and restoring SkeletonShapes */

public void save() {
   // save info on SkeletonShape
   Sx = new double[numPoints];
   Sy = new double[numPoints];
   Ss = new double[numPoints];
   SnumPoints = numPoints;
   SstrokeWidth = strokeWidth;
   for (int i=0; i<numPoints; i++) {
      Sx[i] = x[i];
      Sy[i] = y[i];
      Ss[i] = s[i];
   }
}

public void restore() {
   // reassign values of x,y,strokeWidth,stroke,numPoints from last saved
   // do nothing if no shape is saved
   if (SnumPoints > 0) {
      x = new double[maxPts];
      y = new double[maxPts];
      s = new double[maxPts];
      for (int i=0; i<SnumPoints; i++) {
         x[i] = Sx[i];
         y[i] = Sy[i];
         s[i] = Ss[i];
      }
      numPoints = SnumPoints;
      strokeWidth = SstrokeWidth;
   }
}

public void restore(double frac) {
   // move toward the values of x,y,w,numPoints from last saved
   //   by the fraction frac, 0<frac<1
   // do nothing if no shape is saved
   // delete any extra points in current shape
```

```
  if (SnumPoints > 0) {
     if (numPoints > SnumPoints) {
        for (int i=SnumPoints; i< numPoints; i++)
           deletePoint(i);
     }
     for (int i=0; i<SnumPoints; i++) {
        x[i] = x[i] + frac*(Sx[i] - x[i]);
        y[i] = y[i] + frac*(Sy[i] - y[i]);
        s[i] = s[i] + frac*(Ss[i] - s[i]);
     }
     numPoints = SnumPoints;
     strokeWidth = SstrokeWidth;
  }
}

/* procs on stroke */

public void addStroke(int index, double amt) {
  if ((index>=0)&&(index<numPoints)) {
     s[index] += amt;
  }
}

public void multStroke(int index, double amt) {
  if ((index>=0)&&(index<numPoints)) {
     s[index] *= amt;
  }
}

/* procs on points */

public void addPoint(int mouseX, int mouseY) {
  // add a point to the end of shape
  if (numPoints<maxPts) {
     x[numPoints] = (double)mouseX;
     y[numPoints] = (double)mouseY;
     if (numPoints == 0)
        s[numPoints] = 1.0;
     else
        s[numPoints] = s[numPoints-1];
     numPoints++;
     active = numPoints-1;
  }
} // addPoint, int x,y

public void addPoint(double inputX, double inputY) {
  // add a point to the end of shape
  if (numPoints<maxPts) {
     x[numPoints] = inputX;
     y[numPoints] = inputY;
     s[numPoints] = s[numPoints-1];
     if (numPoints == 0)
        s[numPoints] = 1.0;
     else
        s[numPoints] = s[numPoints-1];
     numPoints++;
     active = numPoints-1;
  }
} // addPoint, double x,y

public void addPoint(int index, int mouseX, int mouseY) {
  // add a point to shape after index
  if ((index<numPoints)&&(index>=0)&&(numPoints<maxPts)) {
```

```
      // move x,y points to "make room" for point after active point
      for(int i=numPoints-1; i>index; i—) {
         x[i+1] = x[i];
         y[i+1] = y[i];
         s[i+1] = s[i];
      }
      // add a point after the point at index
      x[index+1] = (double)mouseX;
      y[index+1] = (double)mouseY;
      s[index+1] = s[index];
      numPoints++;
      active = index+1;
   }
} // addPoint after index, int x,y

public void addPoint(int index, double inputX, double inputY) {
   // add a point to shape after index
   if ((index<numPoints)&&(index>=0)&&(numPoints<maxPts)) {
      // move x,y points to "make room" for point after active point
      for(int i=numPoints-1; i>index; i—) {
         x[i+1] = x[i];
         y[i+1] = y[i];
         s[i+1] = s[i];
      }
      // add a point after the point at index
      x[index+1] = inputX;
      y[index+1] = inputY;
      s[index+1] = 1.0;
      numPoints++;
      active = index+1;
   }
} // addPoint after index, double x,y

public void deletePoint(int index) {
   // delete point at index
   if ((index<numPoints)&&(index>=0)) {
      for (int i=index; i<numPoints-1; i++) {
         x[i] = x[i+1];
         y[i] = y[i+1];
         s[i] = s[i+1];
      }
      //x[numPoints-1] = 0;
      //y[numPoints-1] = 0;
      //s[numPoints-1] = 1.0;
      numPoints—;
   }
   active = NONE;
} // deletePoint

public void movePoint(int index, int mouseX, int mouseY) {
   // set point at index to coordinates mouseX, mouseY
   if ((index<numPoints)&&(index>=0)) {
      x[index] = (double)mouseX;
      y[index] = (double)mouseY;
   }
} // movePoint, int x,y

public void movePoint(int index, double inputX, double inputY) {
   // set point at index to coordinates inputX, inputY
   if ((index<numPoints)&&(index>=0)) {
      x[index] = inputX;
      y[index] = inputY;
   }
```

```
} // movePoint, double x,y

public void translatePoint(int index, int xAmt, int yAmt) {
   // translate point at index by xAmt, yAmt
   if ((index<numPoints)&&(index>=0)) {
      x[index] += xAmt;
      y[index] += yAmt;
   }
} // translatePoint, int xamt,yamt

public void translatePoint(int index, double xAmt, double yAmt) {
   // translate point at index by xAmt, yAmt
   if ((index<numPoints)&&(index>=0)) {
      x[index] += xAmt;
      y[index] += yAmt;
   }
} // translatePoint, double xamt,yamt

public void changePoint(int index, int radius, int mouseX, int mouseY) {
   // move the point at index, pulling along neighboring points
   //  x[index],y[index] moves to mouseX,mouseY
   if ((index<numPoints)&&(index>=0)) {
      double Xchange = mouseX - x[index];
      double Ychange = mouseY - y[index];
      // move index point
      y[index] += Ychange;
      x[index] += Xchange;
      double length, frac;
      for (int i=0; i<numPoints; i++) {
         if (i != index) {
            length = this.getLength(index, i);
            if ( length<radius ) {
               if (Math.abs(i-index) <= 5) {
                  frac = 1/Math.pow(2.0,(double)Math.abs(i-index));
                  x[i] += Xchange*(1-(1-frac)*length/radius);
                  y[i] += Ychange*(1-(1-frac)*length/radius);
               } else {
                  x[i] += Xchange*(1-length/radius);
                  y[i] += Ychange*(1-length/radius);
               }
            }
         }
      } // for
   } // if
} // changePoint

public void changePointInRadius(int index, int radius, int mouseX, int mouseY) {
   // move the point at index, pulling along neighboring points
   //  x[index],y[index] moves to mouseX,mouseY
   if ((index<numPoints)&&(index>=0)) {
      double Xchange = mouseX - x[index];
      double Ychange = mouseY - y[index];
      // move index point
      y[index] += Ychange;
      x[index] += Xchange;
      double length, frac;
      for (int i=0; i<numPoints; i++) {
         if (i != index) {
            length = this.getLength(index, i);
            if ( length<radius ) {
               x[i] += Xchange*(1-length/radius);
               y[i] += Ychange*(1-length/radius);
            }
```

```
        }
      } // for
    } // if
  } // changePointInRadius

  public void changePointInChain(int index, int howFar, int mouseX, int mouseY) {
    // move the point at index, also moving neighboring points in chain
    // movement propagates howFar number of points up and down the chain
    //  x[index],y[index] moves to mouseX,mouseY
    if ((index<numPoints)&&(index>=0)) {
      double Xchange = mouseX - x[index];
      double Ychange = mouseY - y[index];

      // move index point
      y[index] += Ychange;
      x[index] += Xchange;

      double frac;
      int start=index-howFar, end=index+howFar;
      if (start<0)
        start = 0;
      if (end>=numPoints)
        end = numPoints-1;
      for (int i=start; i<=end; i++) {
        if (i != index) {
          frac = 1/Math.pow(2.0,(double)Math.abs(i-index));
          x[i] += Xchange*frac;
          y[i] += Ychange*frac;
        }
      }
    }
  } // changePointInChain, absolute, for ints

  public void changePointInChain(int index, int howFar, double Xchange, double Ychange) {
    // move the point at index, also moving neighboring points in chain
    // movement propagates howFar number of points up and down the chain
    //  x[index],y[index] moves by Xchange,Ychange
    if ((index<numPoints)&&(index>=0)) {
      // move index point
      y[index] += Ychange;
      x[index] += Xchange;

      double frac;
      int start=index-howFar, end=index+howFar;
      if (start<0)
        start = 0;
      if (end>=numPoints)
        end = numPoints-1;
      for (int i=start; i<=end; i++) {
        if (i != index) {
          frac = 1/Math.pow(2.0,(double)Math.abs(i-index));
          x[i] += Xchange*frac;
          y[i] += Ychange*frac;
        }
      }
    }
  } // changePointInChain, relative, for doubles

  public void collapsePoint(int index, int mouseX, int mouseY) {
    // move point at index to mouseX, mouseY, pulling bordering points along
    if ((index<numPoints)&&(index>=0)) {
      x[index] = (double)mouseX;
      y[index] = (double)mouseY;
```

```
        int i = 1;
        double amt = 1.0;
        // move points after directly moved point
        while ( ((index+i)<numPoints) && (amt >=1.0) ) {
            // amt = distance to previous pt * 1/i^2 * weight of point
            amt = getLength(index+i-1)*(1/Math.pow(2.0,(double)i))*(0.5);
            //Utils.print("index="+(index+i)+" amt="+amt);
            moveIndex1TowardIndex2(index+i, index+i-1, amt);
            i++;
        }
        i = 1; amt = 1.0;
        // move points before directly moved point
        while ( ((index-i)>=0) && (amt >=1.0) ) {
            // amt = distance to next pt * 1/i^2 * weight of point
            amt = getLength(index-i)*(1/Math.pow(2.0,(double)i))*(0.5);
            //Utils.print("index="+(index+i)+" amt="+amt);
            moveIndex1TowardIndex2(index-i, index-i+1, amt);
            i++;
        }
    }
} // collapsePoint for ints

public void collapsePoint(int index, double Xchange, double Ychange) {
    // move point at index by Xchange, Ychange pulling bordering weighted points along
    // for floats
    if ((index<numPoints)&&(index>=0)) {
        x[index] += Xchange;
        y[index] += Ychange;
        int i = 1;
        double amt = 1.0;
        // move points after directly moved point
        while ( ((index+i)<numPoints) && (amt >=1.0) ) {
            // amt = distance to previous pt * 1/i^2 * weight of point
            amt = getLength(index+i-1)*(1/Math.pow(2.0,(double)i))*(0.5);
            //Utils.print("index="+(index+i)+" amt="+amt);
            moveIndex1TowardIndex2(index+i, index+i-1, amt);
            i++;
        }
        i = 1; amt = 1.0;
        // move points before directly moved point
        while ( ((index-i)>=0) && (amt >=1.0) ) {
            // amt = distance to next pt * 1/i^2 * weight of point
            amt = getLength(index-i)*(1/Math.pow(2.0,(double)i))*(0.5);
            //Utils.print("index="+(index+i)+" amt="+amt);
            moveIndex1TowardIndex2(index-i, index-i+1, amt);
            i++;
        }
    }
} // collapsePoint for doubles

public void moveIndexTowardPt(int index, double amt, Point pt) {
    // move point at index toward Point pt by a distance of amt
    //  positive amt -> moving towards
    if ((index< numPoints)&&(index>=0)) {
        double theta = Math.atan2(pt.y-y[index], pt.x-x[index]);
        y[index] += (float)(amt*Math.sin(theta));
        x[index] += (float)(amt*Math.cos(theta));
    }
} // moveIndexTowardPt

public void moveIndex1TowardIndex2(int index1, int index2, double amt) {
    // move point at index1 toward point at index2 by a distance of amt
    if ( (index1<numPoints)&&(index1>=0)&&(index2<numPoints)&&(index2>=0)
```

```
      &&(index1!=index2) ) {
      double theta = Math.atan2(y[index2]-y[index1], x[index2]-x[index1]);
      y[index1] += (float)(amt*Math.sin(theta));
      x[index1] += (float)(amt*Math.cos(theta));
   }
} // moveIndex1TowardIndex2

/* procs on lines */

public void setLength(int index, double length) {
   // move each end point of line from index to index+1 an equal amount
   //  so that line length = length
   if ((index<numPoints-1)&&(index>=0)) {
      double curYLength = y[index+1]-y[index];
      double curXLength = x[index+1]-x[index];
      double theta = Math.atan2(curYLength,curXLength);
      double newYLength = length*Math.sin(theta);
      double newXLength = length*Math.cos(theta);
      double Ychange = (newYLength - curYLength)/2.0;
      double Xchange = (newXLength - curXLength)/2.0;

      y[index+1] += Ychange;
      y[index] -= Ychange;
      x[index+1] += Xchange;
      x[index] -= Xchange;
   }
} // setLength

public void changeLength(int index, double length) {
   // move each end point of line from index to index+1 an amt based on weights
   //  so that line length = length and pull neighboring weighted points along
   if ((index<numPoints-1)&&(index>=0)) {
      // move end points at index and index+1
      double curYLength = y[index+1]-y[index];
      double curXLength = x[index+1]-x[index];
      double theta = Math.atan2(curYLength,curXLength);
      double newYLength = length*Math.sin(theta);
      double newXLength = length*Math.cos(theta);
      double Ychange = newYLength - curYLength;
      double Xchange = newXLength - curXLength;

      y[index+1] += Ychange;
      y[index] -= Ychange;
      x[index+1] += Xchange;
      x[index] -= Xchange;

      // now move points after directly moved point at index+1
      int i = 1; double frac = (float)1.0;
      while ( ((index+i+1)<numPoints) && (frac >=0.1) ) {
         frac = 1/Math.pow(2.0,(double)i);
         x[index+i+1] += Xchange*frac;
         y[index+i+1] += Ychange*frac;
         i++;
      }

      // now move points before directly moved point at index
      i = 1; frac = (float)1.0;
      while ( ((index-i)>=0) && (frac >=0.1) ) {
         frac = 1/Math.pow(2.0,(double)i);
         x[index-i] -= Xchange*frac;
         y[index-i] -= Ychange*frac;
         i++;
      }
```

```
    }
  } // changeLength

  public void moveLine(int index, int mouseX, int mouseY) {
    // move the line segment from index to index+1, keeping slope & length constant
    //  x[index],y[index] moves to mouseX,mouseY
    if ((index<numPoints-1)&&(index>=0)) {
      double curYLength = y[index+1]-y[index];
      double curXLength = x[index+1]-x[index];
      y[index] = mouseY;
      x[index] = mouseX;
      y[index+1] = mouseY+curYLength;
      x[index+1] = mouseX+curXLength;
    }
  } // moveLine

  public void changeLine(int index, int mouseX, int mouseY) {
    // move the line segment from index to index+1, using weights of end points
    //  and pulling along neighboring points
    //  x[index],y[index] moves to mouseX,mouseY
    if ((index<numPoints-1)&&(index>=0)) {
      double Xchange = mouseX - x[index];
      double Ychange = mouseY - y[index];

      // move points at index and index+1
      y[index] += Ychange;
      x[index] += Xchange;
      y[index+1] += Ychange;
      x[index+1] += Xchange;

      // now move points after directly moved point at index+1
      int i = 1; double frac = 1.0;
      while ( ((index+i+1)<numPoints) && (frac >=0.1) ) {
        frac = 1/Math.pow(2.0,(double)i);
        x[index+i+1] += Xchange*frac;
        y[index+i+1] += Ychange*frac;
        i++;
      }

      // now move points before directly moved point at index
      i = 1; frac = 1.0;
      while ( ((index-i)>=0) && (frac >=0.1) ) {
        frac = 1/Math.pow(2.0,(double)i);
        x[index-i] += Xchange*frac;
        y[index-i] += Ychange*frac;
        i++;
      }
    }
  } // changeLine

  public void changeLine(int index, double Xch, double Ych) {
    // move the line segment from index to index+1, using weights of end points
    //  and pulling along neighboring points
    //  x[index],y[index] moves to mouseX,mouseY
    if ((index<numPoints-1)&&(index>=0)) {
      // move points at index and index+1
      y[index] += Ych;
      x[index] += Xch;
      y[index+1] += Ych;
      x[index+1] += Xch;

      // now move points after directly moved point at index+1
      int i = 1; double frac = 1.0;
```

```
        while ( ((index+i+1)<numPoints) && (frac >=0.1) ) {
          frac = 1/Math.pow(2.0,(double)i);
          x[index+i+1] += Xch*frac;
          y[index+i+1] += Ych*frac;
          i++;
        }

        // now move points before directly moved point at index
        i = 1; frac = (float)1.0;
          frac = 1/Math.pow(2.0,(double)i);
          x[index-i] += Xch*frac;
          y[index-i] += Ych*frac;
          i++;
        }
      }
} // changeLine for doubles

public void deleteLine(int index) {
    // delete line from index to index+1
    if ((index<numPoints-1)&&(index>=0)) {
        numPoints--;
        x[active] = (x[active] + x[active+1])/2;
        y[active] = (y[active] + y[active+1])/2;
        s[active] = (s[active] + s[active+1])/2;
        for (int i=active+1; i<numPoints; i++) {
            x[i] = x[i+1];
            y[i] = y[i+1];
            s[i] = s[i+1];
        }
        active = NONE;
    }
}

/* these five procs are for setting active items,
   used for interactively creating shapes */

public void makePointActive(int index) {
    if ((index<numPoints)&&(index>=0)) {
        active = index;
        activeItem = POINT;
    }
}

public void makeLineActive(int index) {
    if ((index<numPoints-1)&&(index>=0)) {
        active = index;
        activeItem = LINE;
    }
}

public void makeAngleActive(int index) {
    if ((index<numPoints-1)&&(index>=0)) {
        active = index;
        activeItem = ANGLE;
    }
}

public void deactivate() {
    active = NONE;
}

public void reset() {
    showPointInfo = false;
```

```
    showLineInfo = false;
    showAngleInfo = false;
    showIndexInfo = false;
    drawPoints = false;
    active = NONE;
  }

  /* two procs for checking mouse input */

  public int isNearPoint(int mouseX, int mouseY) {
    // takes in x and y of mouse and returns the index
    // which x,y is "near" to (within a certain number of pixels)
    // if x,y is not near any point, returns -1

    for (int i=0; i<numPoints; i++) {
      if ((Math.abs(mouseX-x[i])<4.0) && (Math.abs(mouseY-y[i])<4.0))
        return i;
    }

    return -1;
  } // isNearPoint

  public int isNearLine(int mouseX, int mouseY) {
    // takes in x and y of mouse and returns the index i of the line
    // connecting i and j which x,y is "near" to
    // if x,y is not near any connecting line, returns -1

    // fast 2D point-on-line test from graphics gems p50
    int Tx = mouseX, Ty = mouseY;
    int Qx, Qy, Px, Py;
    for (int i=0; i<numPoints-1; i++) {
      Px = (int)Math.round(x[i]); Py = (int)Math.round(y[i]);
      Qx = (int)Math.round(x[i+1]); Qy = (int)Math.round(y[i+1]);
      if (Math.abs((Qy-Py)*(Tx-Px)-(Ty-Py)*(Qx-Px))
        >= Math.max(Math.abs(Qx-Px), Math.abs(Qy-Py)))
        continue;
      if ( ((Qx<Px)&&(Px<Tx)) || ((Qy<Py)&&(Py<Ty)) )
        continue;
      if ( ((Tx<Px)&&(Px<Qx)) || ((Ty<Py)&&(Py<Qy)) )
        continue;
      if ( ((Px<Qx)&&(Qx<Tx)) || ((Py<Qy)&&(Qy<Ty)) )
        continue;
      if ( ((Tx<Qx)&&(Qx<Px)) || ((Ty<Qy)&&(Qy<Py)) )
        continue;
      return i;
    }
    return -1;
  } // isNearLine

  /* procs for getting useful info on shape */

  public Point getPoint(int i) {
    // return a Point describing x[i],y[i]
    if ((i<numPoints) && (i>=0))
      return (new Point((int)x[i],(int)y[i]));
    else
      return (new Point(0,0));
  } // getPoint

  public Point getAverage() {
    // returns a Point representing the average of all points in shape
    if (numPoints>0) {
      double Xtotal=0, Ytotal=0;
```

```
    for (int i=0; i<numPoints; i++) {
      Xtotal += x[i];
      Ytotal += y[i];
    }
    return (new Point((int)(Xtotal/numPoints), (int)(Ytotal/numPoints)));
  } else
    return (new Point(0,0));
} // getAverage

public double getLength(int i) {
  // return length of line between x[i],y[i] and x[i+1],y[i+1]
  if ((i<numPoints-1) && (i>=0))
    return (Math.sqrt((x[i+1]-x[i])*(x[i+1]-x[i]) + (y[i+1]-y[i])*(y[i+1]-y[i])));
  else
    return 0;
} // getLength

public double getLength(int i1, int i2) {
  // return length of line between x[i1],y[i1] and x[i2],y[i2]
  if ( (i1<numPoints) && (i1>=0) && (i2<numPoints) && (i2>=0) && (i1 != i2) )
    return (Math.sqrt((x[i2]-x[i1])*(x[i2]-x[i1]) + (y[i2]-y[i1])*(y[i2]-y[i1])));
  else
    return 0;
} // getLength

public double getAngle(int i) {
  // return angle of /_(x[i-1],y[i-1])(x[i],y[i])(x[i+1],y[i+1]) in degrees
  if ((i<numPoints-1)&&(i>0)) {
    double aSqrd = (x[i-1]-x[i])*(x[i-1]-x[i]) + (y[i-1]-y[i])*(y[i-1]-y[i]);
    double bSqrd = (x[i+1]-x[i])*(x[i+1]-x[i]) + (y[i+1]-y[i])*(y[i+1]-y[i]);
    double cSqrd = (x[i+1]-x[i-1])*(x[i+1]-x[i-1]) + (y[i+1]-y[i-1])*(y[i+1]-y[i-1]);
    double angle = Math.acos( (aSqrd+bSqrd-cSqrd)
                    /(2*Math.sqrt(aSqrd)*Math.sqrt(bSqrd)) );
    return (angle*180.0/Math.PI);
  } else
    return 0.;
} // getAngle

public void fill(Graphics g) {
  Polygon a, b, wedge;
  for (int i=0; i<numPoints-1; i++) {
    a = this.shapeSegment(x[i], y[i], x[i+1], y[i+1],
                  strokeWidth*s[i], strokeWidth*s[i+1]);
    // draw normal shape segment
    g.fillPolygon(a);
    if (i<numPoints-2) {
      // draw connecting wedge shape between segments
      b = this.shapeSegment(x[i+1], y[i+1], x[i+2], y[i+2],
                    strokeWidth*s[i+1], strokeWidth*s[i+2]);
      if ( (getAngle(i+1) > 20.)  // mitre value
        && (getLength(i+1,i+2)>5.0) // don't draw wedge if points are too close
        && (getLength(i,i+1)>5.0) ) {
        // find corner points
        double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                    a.xpoints[3], a.ypoints[3]);
        double[] b1 = Utils.getLineThrough2Pts(b.xpoints[0], b.ypoints[0],
                                    b.xpoints[3], b.ypoints[3]);
        Point int1 = Utils.intersection(b1[0], b1[1], b1[2], b1[3],
                                a1[4], a1[5], a1[6]);
        double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                    a.xpoints[2], a.ypoints[2]);
        double[] b2 = Utils.getLineThrough2Pts(b.xpoints[1], b.ypoints[1],
                                    b.xpoints[2], b.ypoints[2]);
```

```
            Point int2 = Utils.intersection(b2[0], b2[1], b2[2], b2[3],
                                            a2[4], a2[5], a2[6]);
            wedge = new Polygon();
            wedge.addPoint(a.xpoints[2], a.ypoints[2]);
            if ((int2.x != -1) && (int2.y != -1))
               wedge.addPoint(int2.x, int2.y);
            wedge.addPoint(b.xpoints[1], b.ypoints[1]);
            wedge.addPoint(a.xpoints[3], a.ypoints[3]);
            if ((int1.x != -1) && (int1.y != -1))
               wedge.addPoint(int1.x, int1.y);
            wedge.addPoint(b.xpoints[0], b.ypoints[0]);
         } else {
            wedge = new Polygon();
            wedge.addPoint(a.xpoints[2], a.ypoints[2]);
            wedge.addPoint(b.xpoints[1], b.ypoints[1]);
            wedge.addPoint(a.xpoints[3], a.ypoints[3]);
            wedge.addPoint(b.xpoints[0], b.ypoints[0]);
         }
         g.fillPolygon(wedge);
      }
   }

   if (numPoints >= 2) { // at least 2 points in shape
      a = this.shapeSegment(x[0], y[0], x[1], y[1],
                            strokeWidth*s[0], strokeWidth*s[1]);
      // draw end cap at beginning of shape
      if (begin == HORIZ) {
         double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                a.xpoints[3], a.ypoints[3]);
         double[] h1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                a.xpoints[0]+10.0, a.ypoints[0]);
         double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                a.xpoints[2], a.ypoints[2]);
         double[] h2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                a.xpoints[1]+10.0, a.ypoints[1]);
         Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
         Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
         Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
         Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);
         wedge = new Polygon();
         if ((int1.x != -1) && (int1.y != -1))
            wedge.addPoint(int1.x, int1.y);
         if ((int2.x != -1) && (int2.y != -1))
            wedge.addPoint(int2.x, int2.y);
         if ((int3.x != -1) && (int3.y != -1))
            wedge.addPoint(int3.x, int3.y);
         if ((int4.x != -1) && (int4.y != -1))
            wedge.addPoint(int4.x, int4.y);
         g.fillPolygon(wedge);
      } else if (begin == VERT) {
         double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                a.xpoints[3], a.ypoints[3]);
         double[] h1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                a.xpoints[0], a.ypoints[0]+10.0);
         double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                a.xpoints[2], a.ypoints[2]);
         double[] h2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                a.xpoints[1], a.ypoints[1]+10.0);
         Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
         Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
         Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
         Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);
         wedge = new Polygon();
```

```
    if ((int1.x != -1) && (int1.y != -1))
      wedge.addPoint(int1.x, int1.y);
    if ((int2.x != -1) && (int2.y != -1))
      wedge.addPoint(int2.x, int2.y);
    if ((int3.x != -1) && (int3.y != -1))
      wedge.addPoint(int3.x, int3.y);
    if ((int4.x != -1) && (int4.y != -1))
      wedge.addPoint(int4.x, int4.y);
    g.fillPolygon(wedge);
  } // begin cap of shape

  a = this.shapeSegment(x[numPoints-2], y[numPoints-2], x[numPoints-1], y[numPoints-1],
                strokeWidth*s[numPoints-2], strokeWidth*s[numPoints-1]);
  // draw end cap at end of shape
  if (end == HORIZ) {
    double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                a.xpoints[3], a.ypoints[3]);
    double[] h1 = Utils.getLineThrough2Pts(a.xpoints[3], a.ypoints[3],
                                a.xpoints[3]+10.0, a.ypoints[3]);
    double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                a.xpoints[2], a.ypoints[2]);
    double[] h2 = Utils.getLineThrough2Pts(a.xpoints[2], a.ypoints[2],
                                a.xpoints[2]+10.0, a.ypoints[2]);
    Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
    Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
    Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3] h2[4], h2[5], h2[6]);
    Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);

    wedge = new Polygon();
    if ((int1.x != -1) && (int1.y != -1))
      wedge.addPoint(int1.x, int1.y);
    if ((int2.x != -1) && (int2.y != -1))
      wedge.addPoint(int2.x, int2.y);
    if ((int3.x != -1) && (int3.y != -1))
      wedge.addPoint(Int3.x, int3.y);
    if ((int4.x != -1) && (int4.y != -1))
      wedge.addPoint(int4.x, int4.y);
    g.fillPolygon(wedge);
  } else if (end == VERT) {
    double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                a.xpoints[3], a.ypoints[3]);
    double[] h1 = Utils.getLineThrough2Pts(a.xpoints[3], a.ypoints[3],
                                a.xpoints[3], a.ypoints[3]+10.0);
    double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                a.xpoints[2], a.ypoints[2]);
    double[] h2 = Utils.getLineThrough2Pts(a.xpoints[2], a.ypoints[2],
                                a.xpoints[2], a.ypoints[2]+10.0);
    Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
    Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
    Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
    Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);

    wedge = new Polygon();
    if ((int1.x != -1) && (int1.y != -1))
      wedge.addPoint(int1.x, int1.y);
    if ((int2.x != -1) && (int2.y != -1))
      wedge.addPoint(int2.x, int2.y);
    if ((int3.x != -1) && (int3.y != -1))
      wedge.addPoint(int3.x, int3.y);
    if ((int4.x != -1) && (int4.y != -1))
      wedge.addPoint(int4.x, int4.y);
    g.fillPolygon(wedge);
  } // end cap of shape
```

```
    } // if numPoints >= 2
} // fill

public void draw(Graphics g) {
    Polygon a, b, wedge;
    for (int i=0; i<numPoints-1; i++) {
        a = this.shapeSegment(x[i], y[i], x[i+1], y[i+1],
                              strokeWidth*s[i], strokeWidth*s[i+1]);

        if (i < numPoints-2) {
            // draw connecting wedge shape between segments
            b = this.shapeSegment(x[i+1], y[i+1], x[i+2], y[i+2],
                                  strokeWidth*s[i+1], strokeWidth*s[i+2]);
            if ( (getAngle(i+1) > 20.)  // mitre value
                && (getLength(i+1,i+2)>5.0) // don't draw wedge if points are too close
                && (getLength(i,i+1)>5.0) ) {
                double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                       a.xpoints[3], a.ypoints[3]);
                double[] b1 = Utils.getLineThrough2Pts(b.xpoints[0], b.ypoints[0],
                                                       b.xpoints[3], b.ypoints[3]);
                Point int1 = Utils.intersection(b1[0], b1[1], b1[2], b1[3], a1[4], a1[5], a1[6]);
                double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                       a.xpoints[2], a.ypoints[2]);
                double[] b2 = Utils.getLineThrough2Pts(b.xpoints[1], b.ypoints[1],
                                                       b.xpoints[2], b.ypoints[2]);
                Point int2 = Utils.intersection(b2[0], b2[1], b2[2], b2[3], a2[4], a2[5], a2[6]);

                wedge = new Polygon();
                wedge.addPoint(a.xpoints[2], a.ypoints[2]);
                if ((int2.x != -1) && (int2.y != -1))
                    wedge.addPoint(int2.x, int2.y);
                wedge.addPoint(b.xpoints[1], b.ypoints[1]);
                wedge.addPoint(a.xpoints[3], a.ypoints[3]);
                if ((int1.x != -1) && (int1.y != -1))
                    wedge.addPoint(int1.x, int1.y);
                wedge.addPoint(b.xpoints[0], b.ypoints[0]);
            } else {
                wedge = new Polygon();
                wedge.addPoint(a.xpoints[2], a.ypoints[2]);
                wedge.addPoint(b.xpoints[1], b.ypoints[1]);
                wedge.addPoint(a.xpoints[3], a.ypoints[3]);
                wedge.addPoint(b.xpoints[0], b.ypoints[0]);
            }
            g.drawPolygon(wedge);
        }
    }

    if (numPoints >= 2) { // at least 2 points in shape
        a = this.shapeSegment(x[0], y[0], x[1], y[1],
                              strokeWidth*s[0], strokeWidth*s[1]);
        // draw end cap at beginning of shape
        if (begin == HORIZ) {
            double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                   a.xpoints[3], a.ypoints[3]);
            double[] h1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                                   a.xpoints[0]+10.0, a.ypoints[0]);
            double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                   a.xpoints[2], a.ypoints[2]);
            double[] h2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                                   a.xpoints[1]+10.0, a.ypoints[1]);
            Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
            Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
            Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
```

```
    Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);

    wedge = new Polygon();
    if ((int1.x != -1) && (int1.y != -1))
      wedge.addPoint(int1.x, int1.y);
    if ((int2.x != -1) && (int2.y != -1))
      wedge.addPoint(int2.x, int2.y);
    if ((int3.x != -1) && (int3.y != -1))
      wedge.addPoint(int3.x, int3.y);
    if ((int4.x != -1) && (int4.y != -1))
      wedge.addPoint(int4.x, int4.y);
    g.drawPolygon(wedge);
  } else if (begin == VERT) {
    double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                a.xpoints[3], a.ypoints[3]);
    double[] h1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                a.xpoints[0], a.ypoints[0]+10.0);
    double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                a.xpoints[2], a.ypoints[2]);
    double[] h2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                a.xpoints[1], a.ypoints[1]+10.0);
    Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
    Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
    Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
    Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);

    wedge = new Polygon();
    if ((int1.x != -1) && (int1.y != -1))
      wedge.addPoint(int1.x, int1.y);
    if ((int2.x != -1) && (int2.y != -1))
      wedge.addPoint(int2.x, int2.y);
    if ((int3.x != -1) && (int3.y != -1))
      wedge.addPoint(int3.x, int3.y);
    if ((int4.x != -1) && (int4.y != -1))
      wedge.addPoint(int4.x, int4.y);
    g.drawPolygon(wedge);
  } // begin cap of shape

  a = this.shapeSegment(x[numPoints-2], y[numPoints-2], x[numPoints-1], y[numPoints-1],
                strokeWidth*s[numPoints-2], strokeWidth*s[numPoints-1]);
  // draw end cap at end of shape
  if (end == HORIZ) {
    double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                a.xpoints[3], a.ypoints[3]);
    double[] h1 = Utils.getLineThrough2Pts(a.xpoints[3], a.ypoints[3],
                                a.xpoints[3]+10.0, a.ypoints[3]);
    double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                a.xpoints[2], a.ypoints[2]);
    double[] h2 = Utils.getLineThrough2Pts(a.xpoints[2], a.ypoints[2],
                                a.xpoints[2]+10.0, a.ypoints[2]);
    Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
    Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
    Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
    Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);

    wedge = new Polygon();
    if ((int1.x != -1) && (int1.y != -1))
      wedge.addPoint(int1.x, int1.y);
    if ((int2.x != -1) && (int2.y != -1))
      wedge.addPoint(int2.x, int2.y);
    if ((int3.x != -1) && (int3.y != -1))
      wedge.addPoint(int3.x, int3.y);
    if ((int4.x != -1) && (int4.y != -1))
```

```
            wedge.addPoint(int4.x, int4.y);
          g.drawPolygon(wedge);
        } else if (end == VERT) {
          double[] a1 = Utils.getLineThrough2Pts(a.xpoints[0], a.ypoints[0],
                                    a.xpoints[3], a.ypoints[3]);
          double[] h1 = Utils.getLineThrough2Pts(a.xpoints[3], a.ypoints[3],
                                    a.xpoints[3], a.ypoints[3]+10.0);
          double[] a2 = Utils.getLineThrough2Pts(a.xpoints[1], a.ypoints[1],
                                    a.xpoints[2], a.ypoints[2]);
          double[] h2 = Utils.getLineThrough2Pts(a.xpoints[2], a.ypoints[2],
                                    a.xpoints[2], a.ypoints[2]+10.0);
          Point int1 = Utils.intersection(h1[0], h1[1], h1[2], h1[3], a1[4], a1[5], a1[6]);
          Point int2 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h1[4], h1[5], h1[6]);
          Point int3 = Utils.intersection(a2[0], a2[1], a2[2], a2[3], h2[4], h2[5], h2[6]);
          Point int4 = Utils.intersection(a1[0], a1[1], a1[2], a1[3], h2[4], h2[5], h2[6]);

          wedge = new Polygon();
          if ((int1.x != -1) && (int1.y != -1))
            wedge.addPoint(int1.x, int1.y);
          if ((int2.x != -1) && (int2.y != -1))
            wedge.addPoint(int2.x, int2.y);
          if ((int3.x != -1) && (int3.y != -1))
            wedge.addPoint(int3.x, int3.y);
          if ((int4.x != -1) && (int4.y != -1))
            wedge.addPoint(int4.x, int4.y);
          g.drawPolygon(wedge);
        } // end cap of shape
      } // if numPoints >= 2
    } // draw

    public void drawInfo(Graphics g) {
      for (int i=0; i<numPoints; i++) {
        if (drawPoints) {
          // draw filled 3 pixel circle at points
          g.setColor(Color.gray);
          g.fillOval((int)x[i]-1, (int)y[i]-1, 3,3);
          g.setColor(Color.black);
        }
        if ((i == active)&&(activeItem == POINT)) {
          // draw red outline 3 pixel circle for active point
          g.setColor(Color.white); // background color
          g.fillOval((int)x[i]-1, (int)y[i]-1, 3,3);

          g.setColor(Color.red);
          g.drawOval((int)x[i]-1, (int)y[i]-1, 3,3);

          g.setColor(Color.black); // foreground color
        }
        if ((i == active)&&(activeItem == LINE)&&(i<numPoints-1)) {
          // draw red line for active line
          Polygon seg = this.shapeSegment(x[i], y[i], x[i+1], y[i+1],
                        strokeWidths[i], strokeWidths[i+1]);
          g.setColor(Color.red);
          g.drawPolygon(seg);
          g.setColor(Color.black);
        }
        // put info about points
        if (showPointInfo) {
          g.setColor(Color.gray);
          g.drawString((int)x[i]+","+(int)y[i], (int)x[i]+3, (int)y[i]+2);
          g.setColor(Color.black);
        }
        if (i != numPoints-1) {
```

```
            // put info about lines
            if (showLineInfo) {
               g.setColor(Color.gray);
               g.drawString(""+(int)getLength(i), (int)((x[i]+x[i+1])/2), (int)((y[i]+y[i+1])/2+3));
               g.setColor(Color.black);
            }
            if (i != 0) {
               // put info about angles
               if (showAngleInfo) {
                  g.setColor(Color.gray);
                  g.drawString(""+(int)getAngle(i), (int)x[i]-5, (int)y[i]+10);
                  g.setColor(Color.black);
               }
            }
         }
      }
   } // drawInfo

   public static Polygon shapeSegment(double x1, double y1, double x2, double y2,
                                      double stroke1, double stroke2) {
   // based on Phillip Tiongson's fillLine
      // to be filled when get 4 points
      // 4 points ordered in clockwise direction
      Polygon segment = new Polygon();

      // stroke is half of real stroke
      stroke1 /= 2.0;
      stroke2 /= 2.0;

      // slope calulation
      double dx = x2 - x1;
      double dy = y2 - y1;
      double Px, Py;

      // must handle special cases
      if (dx == 0) {
         // if slope of line is zero, vertical
        if (y2<y1) {
          stroke1 *= -1;
          stroke2 *= -1;
        }
          Px = x1 - stroke1;
          Py = y1;
          segment.addPoint((int) Px, (int) Py);

          Px = x1 + stroke1;
          Py = y1;
          segment.addPoint((int) Px, (int) Py);

          Px = x1 + stroke2;
          Py = y1 + dy;
          segment.addPoint((int) Px, (int) Py);

          Px = x1 - stroke2;
          Py = y1 + dy;
          segment.addPoint((int) Px, (int) Py);

          Px = x1 - stroke1;
          Py = y1;
          segment.addPoint((int) Px, (int) Py);
      } else if (dy == 0) {
         // if horizontal slope
         if (x2<x1) {
```

```
            stroke1 *= -1;
            stroke2 *= -1;
          }
          Px = x1;
          Py = y1 + stroke1;
          segment.addPoint((int) Px, (int) Py);

          Px = x1;
          Py = y1 - stroke1;
          segment.addPoint((int) Px, (int) Py);

          Px = x1 + dx;
          Py = y1 - stroke2;
          segment.addPoint((int) Px, (int) Py);

          Px = x1 + dx;
          Py = y1 + stroke2;
          segment.addPoint((int) Px, (int) Py);

          Px = x1;
          Py = y1 + stroke1;
          segment.addPoint((int) Px, (int) Py);
        } else {
        // the usual case
          double slope = -1 * dx/dy;

          // the increment in the normal vector
          double n1 = Math.sqrt(stroke1*stroke1/(slope*slope + 1));
          double n2 = Math.sqrt(stroke2*stroke2/(slope*slope + 1));

          // points always ordered in clockwise direction
          if ((x2>x1)&&(slope > 0) || (x2<x1)&&(slope < 0)) {
            n1 *= -1;
            n2 *= -1;
          }
          Px = x1 - n1;
          Py = y1 - (n1 * slope);
          segment.addPoint((int) Px, (int) Py);

          Px = x1 + n1;
          Py = y1 + (n1 * slope);
          segment.addPoint((int) Px, (int) Py);

          Px = x2 + n2;
          Py = y2 + (n2 * slope);
          segment.addPoint((int) Px, (int) Py);

          Px = x2 - n2;
          Py = y2 - (n2 * slope);
          segment.addPoint((int) Px, (int) Py);

          Px = x1 - n1;
          Py = y1 - (n1 * slope);
          segment.addPoint((int) Px, (int) Py);
        } // else

        return segment;
      } // shapeSegment

      public void printPoints() {
        Utils.print("numPoints="+numPoints);
        for (int i=0; i<numPoints; i++)
          Utils.print(".addPoint("+(int)Math.round(x[i])+","+(int)Math.round(y[i])+"); // "+i);
```

```
    Utils.print("strokeWidth="+strokeWidth);
    for (int i=0; i<numPoints; i++)
      Utils.print("s["+i+"]="+s[i]);
  } // printPoints
} // SkeletonShape
```

Utils.java

```
package pcho.util;

import java.lang.System;
import java.util.Random;
import java.awt.Point;

public class Utils extends Object {
  public static final boolean DEBUG = true;

  public static void print(String s) {
    if ( DEBUG == true )
      System.out.println(s);
  } // print

  public static Point midPoint(Point p1, Point p2) {
    // returns a Point which is mid-point between p1 and p2
    return (new Point( (p2.x-p1.x)/2, (p2.y-p1.y)/2) );
  } // midPoint

  public static double getDistance(int x1, int y1, int x2, int y2) {
    // return length of line between x1,y1 and x2,y2
    return (Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1)));
  } // getDistance

  public static double[] getLineThrough2Pts(double x1, double y1, double x2, double y2) {
  // returns array: [Ux, Uy, Vx, Vy, Nx, Ny, c]
  // where N*P + c = 0 and P = U + V*t
  // p9 graphic gems
    double Ux, Uy, Vx, Vy, Nx, Ny, c;
    Ux = x1;
    Uy = y1;

    Vx = (x2-x1)/Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
    Vy = (y2-y1)/Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));

    Nx = -Vy; // <————————
    Ny = Vx;  // <————————
    c = -(Nx*Ux + Ny*Uy);  // <————————

    double[] line = {Ux, Uy, Vx, Vy, Nx, Ny, c};
    return line;
  } // getLineThrough2Pts

  public static Point intersection(double mUx, double mUy, double mVx, double mVy,
                   double lNx, double lNy, double lC) {
  // p11 graphic gems
    double d = lNx*mVx + lNy*mVy;

    if (Math.abs(d)>0.1) {
      double x, y;
      x = mUx - ((((lNx*mUx + lNy*mUy) + lC)/d)*mVx;
      y = mUy - ((((lNx*mUx + lNy*mUy) + lC)/d)*mVy;
      return (new Point((int)Math.round(x), (int)Math.round(y)));
    } else
```

```
        return (new Point(-1,-1));
    } // intersection
} // Utils
```

Oh.java

```
package pcho.typotool;

import java.awt.*;
import java.applet.Applet;
import java.lang.*;
import pcho.util.*;

public class Oh extends Applet implements Runnable {
    Label info = new Label();
    Font infoFont;

    int gX = 550, gY = 380;
    SkeletonShape o, h1, h2;
    boolean fill = true;
    int lastX=0, lastY=0, curX=gX/5, curY=0;
    int mode = 1;
    int frameDelay = 50;

    public String getAppletInfo() {
        return "oh applet, by Peter Cho (c) Copyright 1997 MIT Media Lab";
    }

    Thread Motion;

    public void start() {
        Motion.start();
    } // start

    public void stop() {
        Motion.stop();
    } // stop

    public void destroy() {
        Motion.stop();
    } // destroy

    public void init() {
        Utils.print("initializing...");
        this.setBackground(Color.white);
        this.setForeground(Color.black);
        infoFont = new Font("Helvetica", Font.PLAIN, 10);
        Motion = new Thread(this);
        setLayout (new BorderLayout());
        Panel p = new Panel();
        p.setBackground(Color.lightGray);
        p.setForeground(Color.black);
        p.setLayout(new GridLayout(0, 2));

        add("South", p);
        info.setText("select points");
        p.add("West", info);
        validate();

        o = new SkeletonShape(10);
        o.addPoint(159,135); // 0
        o.addPoint(154,139); // 1
```

```
o.addPoint(149,144); // 2
o.addPoint(144,152); // 3
o.addPoint(140,160); // 4
o.addPoint(135,170); // 5
o.addPoint(131,182); // 6
o.addPoint(127,197); // 7
o.addPoint(126,210); // 8
o.addPoint(128,226); // 9
o.addPoint(132,233); // 10
o.addPoint(141,241); // 11
o.addPoint(153,242); // 12
o.addPoint(166,237); // 13
o.addPoint(174,230); // 14
o.addPoint(180,223); // 15
o.addPoint(186,213); // 16
o.addPoint(190,203); // 17
o.addPoint(195,188); // 18
o.addPoint(198,172); // 19
o.addPoint(199,158); // 20
o.addPoint(198,145); // 21
o.addPoint(192,132); // 22
o.addPoint(188,124); // 23
o.addPoint(180,119); // 24
o.addPoint(168,119); // 25
o.addPoint(158,121); // 26
o.addPoint(149,125); // 27
o.addPoint(139,132); // 28
o.addPoint(132,139); // 29
o.s[0]=1;
o.s[1]=1.4;
o.s[2]=1.4;
o.s[3]=1.4;
o.s[4]=1.4;
o.s[5]=1.4;
o.s[6]=1.4;
o.s[7]=1.2;
o.s[8]=1.1;
o.s[9]=0.8;
o.s[10]=0.8;
o.s[11]=0.8;
o.s[12]=0.9;
o.s[13]=0.9;
o.s[14]=1.1;
o.s[15]=1.2;
o.s[16]=1.3;
o.s[17]=1.3;
o.s[18]=1.3;
o.s[19]=1.3;
o.s[20]=1.1;
o.s[21]=1.1;
o.s[22]=0.9;
o.s[23]=0.8;
o.s[24]=0.8;
o.s[25]=0.7;
o.s[26]=0.7;
o.s[27]=0.7;
o.s[28]=0.5;
o.s[29]=0.5;
for (int i=0; i<o.numPoints; i++) {
  o.translatePoint(i, 40, 50);
}
o.deactivate();
o.save();
```

```
h1 = new SkeletonShape(10);
h1.addPoint(274,10); // 0
h1.addPoint(274,17); // 1
h1.addPoint(272,26); // 2
h1.addPoint(270,37); // 3
h1.addPoint(267,49); // 4
h1.addPoint(265,63); // 5
h1.addPoint(262,77); // 6
h1.addPoint(258,100); // 7
h1.addPoint(254,122); // 8
h1.addPoint(249,147); // 9
h1.addPoint(246,163); // 10
h1.addPoint(242,183); // 11
h1.addPoint(239,198); // 12
h1.addPoint(237,211); // 13
h1.addPoint(235,221); // 14
h1.addPoint(231,237); // 15
h1.s[0]=1;
h1.s[1]=1.3;
h1.s[2]=1.5;
h1.s[3]=1.5;
h1.s[4]=1.5;
h1.s[5]=1.3;
h1.s[6]=1.2;
h1.s[7]=1.15;
h1.s[8]=1.1;
h1.s[9]=1.05;
h1.s[10]=1;
h1.s[11]=1;
h1.s[12]=1;
h1.s[13]=1.1;
h1.s[14]=1.1;
h1.s[15]=1;

h2 = new SkeletonShape(10);
h2.addPoint(231,237); // 0
h2.addPoint(242,214); // 1
h2.addPoint(250,199); // 2
h2.addPoint(257,184); // 3
h2.addPoint(266,166); // 4
h2.addPoint(274,150); // 5
h2.addPoint(287,133); // 6
h2.addPoint(295,127); // 7
h2.addPoint(302,124); // 8
h2.addPoint(308,126); // 9
h2.addPoint(314,139); // 10
h2.addPoint(313,162); // 11
h2.addPoint(313,179); // 12
h2.addPoint(312,192); // 13
h2.addPoint(312,202); // 14
h2.addPoint(314,211); // 15
h2.addPoint(317,223); // 16
h2.addPoint(322,232); // 17
h2.addPoint(327,235); // 18
h2.addPoint(335,234); // 19
h2.addPoint(344,226); // 20
h2.addPoint(352,216); // 21
h2.addPoint(360,205); // 22
h2.addPoint(363,199); // 23
h2.s[0]=1;
h2.s[1]=1.1;
h2.s[2]=0.9;
```

```
    h2.s[3]=0.9;
    h2.s[4]=0.9;
    h2.s[5]=0.9;
    h2.s[6]=1.1;
    h2.s[7]=1.5;
    h2.s[8]=1.4;
    h2.s[9]=1.4;
    h2.s[10]=1.4;
    h2.s[11]=1.4;
    h2.s[12]=1.4;
    h2.s[13]=1.4;
    h2.s[14]=1.4;
    h2.s[15]=1.5;
    h2.s[16]=1.5;
    h2.s[17]=1.5;
    h2.s[18]=1.5;
    h2.s[19]=1.4;
    h2.s[20]=1;
    h2.s[21]=0.6;
    h2.s[22]=0.4;
    h2.s[23]=0.4;

    for (int i=0; i<h1.numPoints; i++) {
      h1.translatePoint(i, 40, 50);
    }
    for (int i=0; i<h2.numPoints; i++) {
      h2.translatePoint(i, 40, 50);
    }
    h1.deactivate();
    h1.save();
    h2.deactivate();
    h2.save();
     Utils.print("finished loading.");
  } // init

  boolean restore = false, mouseDown = false;
  int counter=0;
  double waveAmt = 0;

  public void run() {
     System.out.println("running...");
     double amt = 2.0;
     double odist, h1dist, h2dist;
     Point mouse;

     while(true) {
       if (mode==1) {
           mouse = new Point(curX,curY);
           if (restore) {
             if (counter<21) {
                o.restore( (0.25 + (double)counter/28.0) / (21.0-(double)counter) );
                h1.restore( (0.25 + (double)counter/28.0) / (21.0-(double)counter) );
                h2.restore( (0.25 + (double)counter/28.0) / (21.0-(double)counter) );

                //Utils.print("counter="+counter);
                counter++;
             } else if (counter==21) {
                restore = false;
                counter = 0;
             }

               repaint();
           } else if (mouseDown) { // mouse is down
```

```
        if ( (lastX != curX) && (lastY != curY) ) { // mouse has moved
          for (int i=0; i<o.numPoints; i++) {
            odist = Utils.getDistance((int)o.x[i], (int)o.y[i], curX, curY);
            h1dist = Utils.getDistance((int)h1.x[i], (int)h1.y[i], curX, curY);
            h2dist = Utils.getDistance((int)h2.x[i], (int)h2.y[i], curX, curY);
            if ((odist<400)&&(odist!=0.0)) {
              o.s[i] *= 1-1/odist;
              o.moveIndexTowardPt(i, -(10.0-odist/40.0), mouse);
            }
            if ((h1dist<400)&&(h1dist!=0.0)) {
              h1.s[i] *= 1-1/h1dist;
              h1.moveIndexTowardPt(i, -(10.0-h1dist/40.0), mouse);
            }
            if ((h2dist<400)&&(h2dist!=0.0)) {
              h2.s[i] *= 1-1/h2dist;
              h2.moveIndexTowardPt(i, -(10.0-h2dist/40.0), mouse);
            }
          }
        } else { // mouse is still in same place
          for (int i=0; i<o.numPoints; i++) {
            odist = Utils.getDistance((int)o.x[i], (int)o.y[i], curX, curY);
            h1dist = Utils.getDistance((int)h1.x[i], (int)h1.y[i], curX, curY);
            h2dist = Utils.getDistance((int)h2.x[i], (int)h2.y[i], curX, curY);
            if ((odist<400)&&(odist!=0.0)) {
              o.s[i] *= 1-0.2/odist;
              o.moveIndexTowardPt(i, -(10.0-odist/40.0), mouse);
            }
            if ((h1dist<400)&&(h1dist!=0.0)) {
              h1.s[i] *= 1-0.2/h1dist;
              h1.moveIndexTowardPt(i, -(10.0-h1dist/40.0), mouse);
            }
            if ((h2dist<400)&&(h2dist!=0.0)) {
              h2.s[i] *= 1-0.2/h2dist;
              h2.moveIndexTowardPt(i, -(10.0-h2dist/40.0), mouse);
            }
          }
        }
        lastX = curX; lastY = curY;
        repaint();
      }
    } // if (mode==1)

    else if (mode==2) {
      if ( (Math.abs(o.x[0]-o.Sx[0]) <= 1.0) // change wave amt when letters are in place
          &&(Math.abs(h1.x[0]-h1.Sx[0]) <= 1.0)
          &&(Math.abs(h2.x[0]-h2.Sx[0]) <= 1.0) )
        waveAmt = 5*((double)curX/(double)gX);
      for (int i=0; i< o.numPoints ; i++) {
        o.translatePoint(i, waveAmt*Math.sin(counter+i), 0.0);
        h1.translatePoint(i, waveAmt*Math.sin(counter+i), 0.0);
        h2.translatePoint(i, waveAmt*Math.sin(counter+15-i), 0.0);

        o.s[i] += 0.1*Math.cos(counter+i);
        h1.s[i] += 0.1*Math.cos(counter+i);
        h2.s[i] += 0.1*Math.cos(counter+15-i);
      }
      counter++;
      repaint();
      //Utils.print("counter="+counter+" amt="+waveAmt*Math.sin(counter));
    } // else if (mode==2)

    else if (mode==3) {
      if (counter==0) {
```

```
for (int i=0; i<o.numPoints; i++) {
   o.s[i] = 1.0;
   h1.s[i] = 1.0;
   h2.s[i] = 1.0;

   if (i==0) {
      o.x[i] = 200; o.y[i] = 182; // top left
   } else if (i<9) {
      o.x[i] = 200;
      o.y[i] = o.y[0] + ((double)i/9)*(257-182);//(o.y[9]-o.y[0]);
   } else if (i==9) {
      o.x[i] = 200; o.y[i] = 257; // bot left
   } else if (i<16) {
      o.x[i] = o.x[9] + (((double)i-9)/(16-9))*(250-o.x[9]);
      o.y[i] = o.y[9];
   } else if (i==16) {
      o.x[i] = 250; o.y[i] = 257; // bot right
   } else if (i<22) {
      o.x[i] = o.x[16];
      o.y[i] = o.y[16] + (((double)i-16)/(22-16))*(182-o.y[16]);
   } else if (i==22) {
      o.x[i] = 250; o.y[i] = 182; // top right
   } else if (i<29) {
      o.x[i] = o.x[22] + (((double)i-22)/(29-22))*(195-o.x[22]);
      o.y[i] = o.y[22];
   } else if (i==29) {
      o.x[i] = 195; o.y[i] = 182; // top left
   }

   if (i==0) {
      h1.x[i] = 288; h1.y[i] = 90; // top
   } else if (i==15) {
      h1.x[i] = 288; h1.y[i] = 263; // bot
   } else if (i<15) {
      h1.x[i] = h1.x[0];
      h1.y[i] = h1.y[0] + ((double)i/15)*(262-h1.y[0]); // mid
   }

   if (i==0) {
      h2.x[i] = 288; h2.y[i] = 263; // bot left
   } else if (i<5) {
      h2.x[i] = h2.x[0];
      h2.y[i] = h2.y[0] + ((double)i/5)*(182-h2.y[0]); // mid left
   } else if (i==5) {
      h2.x[i] = 288; h2.y[i] = 182; // top left
   } else if (i<11) {
      h2.x[i] = h2.x[5] + (((double)i-5)/(11-5))*(338-h2.x[5]); // mid top
      h2.y[i] = h2.y[5];
   } else if (i==11) {
      h2.x[i] = 338; h2.y[i] = 182; // top right
   } else if (i<21) {
      h2.x[i] = h2.x[11]; // mid right
      h2.y[i] = h2.y[11] + (((double)i-11)/(21-11))*(256-h2.y[11]);
   } else if (i==21) {
      h2.x[i] = 338; h2.y[i] = 256;
   } else if (i==22) {
      h2.x[i] = 338; h2.y[i] = 259;
   } else if (i==23) {
      h2.x[i] = 338; h2.y[i] = 264; // bot right
   }
}
//restore = true;
counter++;
```

```
        } else if (restore) {
          if (counter<21) {
            o.restore( (0.25 + (double)counter/28.0) / (21.0-(double)counter) );
            h1.restore( (0.25 + (double)counter/28.0) / (21.0-(double)counter) );
            h2.restore( (0.25 + (double)counter/28.0) / (21.0-(double)counter) );
            counter++;
          } else if (counter==21) {
            restore = false;
            counter = -1;
          }
        }
        repaint();
      } // else if (mode==3)
      try { Motion.sleep(frameDelay); }
      catch (Exception e) {};
    }
  } // run

  public void update(Graphics g) {
    // to prevent flicker
    paint(g);
  } // update

  Image offImage;
  Dimension offImageSize;
  Graphics offGraphics;

  public void paint(Graphics g) {
    // temp variable for current screen size
    Dimension d = size();

    // if no offscreen buffer image, create a new one
    if ((offImage == null) || (d.width != offImageSize.width)
        || (d.height != offImageSize.height )) {
      offImage = createImage(d.width, d.height );
      offImageSize = d;
      offImageSize.height = d.height ;
      offGraphics = offImage.getGraphics();
      offGraphics.setFont(infoFont);
      offGraphics.setPaintMode();
      Utils.print("created new offscreen buffer.");
    }

    // clear the offscreen buffer
    offGraphics.setColor(getBackground());
    offGraphics.fillRect(0, 0, offImageSize.width, offImageSize.height);
    offGraphics.setColor(getForeground());

    // draw shape to offscreen image
    if (fill) {
      o.fill(offGraphics);
      h1.fill(offGraphics);
      h2.fill(offGraphics);
    } else {
      o.draw(offGraphics);
      h1.draw(offGraphics);
      h2.draw(offGraphics);
    }
    o.drawInfo(offGraphics);
    h1.drawInfo(offGraphics);
    h2.drawInfo(offGraphics);

    // blast offImage to screen
```

```
    g.drawImage(offImage, 0, 0, null);
  }

  public boolean handleEvent(Event e) {
    switch(e.id) {
    case Event.ACTION_EVENT:
      break;

    case Event.MOUSE_DOWN:
      Utils.print("mouse down> x=" + e.x + " y=" + e.y);
      info.setText("");
      mouseDown = true;
      if (mode==1) {
        restore = false;
      } else if (mode==3) {
        counter = 0;
        restore = false;
      }
      lastX = curX; lastY = curY;
      curX = e.x; curY = e.y;

      repaint();
      break;

    case Event.MOUSE_DRAG:
      if (mode==1) {
        mouseDown = true;
        lastX = curX; lastY = curY;
        curX = e.x; curY = e.y;
      }
      repaint();
      break;

    case Event.MOUSE_UP:
      mouseDown = false;
      if (mode==1) {
        restore = true;
      }
      if (mode==3) {
        restore = true;
      }
      repaint();
      break;

    case Event.KEY_PRESS:
      char c1 = (char) e.key;
      if (e.id == Event.KEY_PRESS) {
        Utils.print("key down: " + c1);
        switch (c1) {
          case '1': // mode 1
            Utils.print("change to mode 1");
            info.setText("1");
            mode = 1;
            counter = 0;
            o.restore();
            h1.restore();
            h2.restore();
            repaint();
            return true;
          case '2': // mode 2
            Utils.print("change to mode 2");
            info.setText("2");
            mode = 2;
```

```
        counter = 0;
        o.restore();
        h1.restore();
        h2.restore();
        repaint();
        return true;
      case '3': // mode 3
        Utils.print("change to mode 3");
        info.setText("3");
        mode = 3;
        counter = -1;
        restore = false;
        o.restore();
        h1.restore();
        h2.restore();
        repaint();
        return true;
      case 'S': // Save
        o.save();
        h1.save();
        h2.save();
        info.setText("saving shape");
        return true;
      case 'R': // Restore
        o.restore();
        h1.restore();
        h2.restore();
        info.setText("restoring shape");
        repaint();
        return true;
      case 'r': // reset
        o.reset();
        h1.reset();
        h2.reset();
        info.setText("reset");
        repaint();
        return true;
      case 'f': // toggle filled/outline mode
        fill = !fill;
        info.setText("toggle fill mode");
        repaint();
        return true;
      case 'p': // toggle point info
        o.showPointInfo = !o.showPointInfo;
        h1.showPointInfo = !h1.showPointInfo;
        h2.showPointInfo = !h2.showPointInfo;
        info.setText("toggle point coordinates");
        repaint();
        return true;
      case 'l': // toggle line info
        o.showLineInfo = !o.showLineInfo;
        h1.showLineInfo = !h1.showLineInfo;
        h2.showLineInfo = !h2.showLineInfo;
        info.setText("toggle line lengths");
        repaint();
        return true;
      case 'a': // toggle angle info
        o.showAngleInfo = !o.showAngleInfo;
        h1.showAngleInfo = !h1.showAngleInfo;
        h2.showAngleInfo = !h2.showAngleInfo;
        info.setText("toggle angles in degrees");
        repaint();
        return true;
```

```
                case 'h': // toggle drawing point handles
                    o.drawPoints = !o.drawPoints;
                    h1.drawPoints = !h1.drawPoints;
                    h2.drawPoints = !h2.drawPoints;
                    repaint();
                    return true;
                }
            }
            break;
        }
        return true;
    } // handleEvent
} // Oh applet
```

References

Cho, Peter. "Dancing Letters: Movement and Expression in Typographic Elements." ACG memo, 1996.

Dair, Carl. *Design with Type*. University of Toronto Press, Toronto. 1967.

Elam, Kimberly. *Expressive Typography*. Van Nostrand Reinhold, New York. 1990

Ionesco, Eugène, Massin, and Cohen. *La Cantatrice Chauve*. Grove Press. 1956.

Knuth, Donald. *Computer Modern Typefaces*. Addison Wesley, Reading, Massachusetts. 1986.

Lewis, Jason. "Dynamic Poetry: Introductory Remarks to a Digital Medium." Masters thesis, Royal College of Art, 1996.

Maeda, John. *Flying Letters*. Digitalogue, Tokyo. 1996.

Meggs, Philip B. *A History of Graphic Design*. Van Nostrand Reinhold, New York. 1992.

Rand, Paul. *From Lascaux to Brooklyn*.. Yale University Press, New Haven and London. 1996.

Small, David. "Expressive Typography: High Quality Dynamic and Responsive Typography in the Electronic Environment." Masters thesis, Massachusetts Institute of Technology, 1987.

Soo, Douglas. "Implementation of a Temporal Typography System." Master's thesis, Massachusetts Institute of Technology, 1997.

Strassmann, Steve. "Hairy Brushes." *ACM* 20, no. 4 (August 1986): pp 225–232.

Tschichold, Jan. *Die neue Typographie*. Brinkmann & Bose, Berlin. 1987.

Wong, Yin Yin. "Temporal Typography: Characterication of Time-Varying Typographic Forms." Masters thesis, Massachusetts Institute of Technology, 1995.