TWO-DIMENSIONAL SHAPE BLENDING

by

Lorrita L. McKnight

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Mathematics
University of Manitoba
Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

Two-Dimensional Shape Blending

BY

Lorrita L. McKnight

A Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Shape blending is the process of taking two existing shapes and finding in-between shapes that provide a smooth transition from the first shape to the second. Shape blending can be divided into two main sub-problems: the vertex correspondence problem and the vertex path problem. This thesis looks at algorithms to solve these problems, and applies these algorithms to both polygons and Bézier curves.

Table of Contents

List of Figures

Acknowledgments

I would first like to express my appreciation to my advisor, Dr. W.D. Hoskins, for his guidance, support and patience, and to the members of my examining committee, Dr. P.W. Aitchison and Dr. D. Walton.

I owe a special debt of gratitude to Dr. T.G. Berry for his generous interest in my mathematical development, for always making time to listen to me and for his constant encouragement.

I am grateful to my parents, Ruth and Harry McKnight, for their support over the years, and to my sister Melody, for those much-needed ski-trips.

Thanks also go to Will Redekop, for the advice, the laughter and the late-night phone-calls.

Lastly, I would like to thank my fellow graduate students, especially Jackie Storen, Allan Hildebrand, Tracy Ewen, and Stephanie Olafson, for brightening so many of my otherwise long and tedious days at school.

Chapter 1: Introduction

1.1 Problem Statement and Background

Shape blending or shape interpolation is the process of taking two existing shapes or curves (known as key shapes or curves) and finding in-between shapes that provide a smooth transformation from one key shape to the other. Shape blending should not be confused with image morphing; shape blending changes the actual outline of the shape, whereas image morphing warps digital images.

Digital image morphing is comprised of two operations which take place at the same time: *dissolving*, in which one image gradually fades out as another images fades in, and *warping*, which moves points of the initial image to corresponding points of the final image. Despite the difference between shape blending and image morphing, some of the same techniques of shape blending are applicable to the warping operation of image morphing (for example,

determining a correspondence between points of the images and determining the path the points should follow during the morph).

Shape blending has application in areas such as animation, and computer-aided design and illustration.

Volino, N. Thalman, Jianhua, and D. Thalman [1996] have described a method for simulating clothes on virtual actors [17] using physics-based modeling, in which the cloth is modeled as planar garment panels. Physics-based modeling can be costly to compute for each frame; as a cost-cutting measure, physics-based models could be computed only for some of the frames, and shape blending of the panels could be used to compute the remainder of the frames.

Blending two images to simulate realistic motion is often a difficult task. In the past, this animation has been done manually by artists, who must draw thousands of frames in order to simulate a short sequence of motion. Clearly, this is a very time consuming and costly endeavour. Naturally, automation of the animation process is desirable.

Shape blending can generally be divided into two primary sub-problems: the vertex correspondence problem, and the vertex path problem.

Vertex correspondence determines a matching of the vertices of one key shape with the vertices of the other, so that if vertex P_a in shape 1 is matched with

vertex P_b in shape 2, then vertex P_a will follow a path to vertex P_b during the blend (see Fig. 1.1).

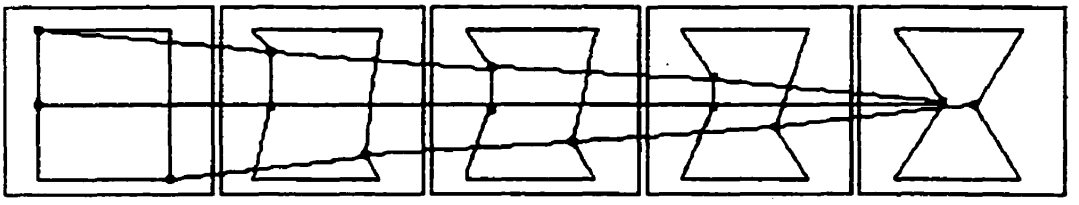


Fig. 1.1 – Correspondence of vertices

Adding additional vertices to one or both of the key shapes is often desirable, if not necessary, in order to provide a more appealing blend. The problem of where these additional vertices should be added is included in the vertex correspondence problem.

Vertex correspondence is an important problem to consider, since an inappropriate correspondence can lead to highly inaccurate and distorted in-between images. For example, consider the two shapes shown in Fig. 1.2a (and 1.3a), and blended in Figs. 1.2b and 1.3b. In Fig. 1.2b, an inadequate vertex matching has distorted that which should have been a trivial blend (Fig. 1.3b).

The vertex path problem determines the path along which a vertex of the first key shape will travel to arrive at its corresponding vertex in the second key shape. For example, a linear path is a simple approach to this problem, but one that often leads to unappealing results, as will be shown in Chapter 3 (see Figs. 3.1 and 3.2 for an example).

Fig. 1.2a – Two images to be blended

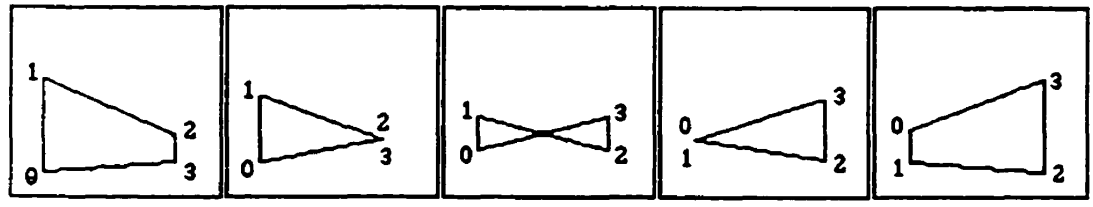


Fig. 1.2b – A distorted blend

Fig. 1.2 – Example of à blend with an inappropriate vertex correspondence

Fig. 1.3a – Two images to be blended

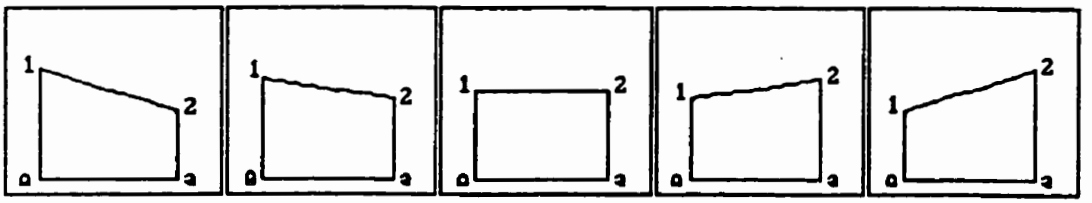


Fig. 1.3b – A good blend

Fig. 1.3 – Example of a blend with a more pleasing vertex correspondence

The purpose of this thesis is to present a detailed study of the vertex correspondence method known as "Least Work Matching" [1] and the vertex path method known as "Intrinsic Interpolation" [2]. This includes implementations of both, and comparisons with some simpler methods. These techniques will be applied to closed polygons and Bézier curves [9].

1.2 Preliminaries

This work considers only 2-dimensional geometric blending. No consideration has been given to 3-d blending, or to blending of other properties of an object (e.g. lighting, colour, etc.).

Hughes [1992] presented a method for interpolating between two volumetric models [18]. This method takes the Fourier transforms of the volumetric models, interpolates between the transformed models, and then transforms the results back. An interpolation scheme is used in which the high frequencies of the first model are gradually removed, interpolation between the low frequencies is performed, and then the high frequencies of the second model are gradually added back in.

Kent, Carlson, and Parent [1992] developed an algorithm to compute transformations between two 3D objects, as opposed to 2D images of the 3D objects [16]. The technique involves merging the topologies of the two objects and mapping this merged topology back onto each of the original objects.

Throughout this thesis, counter-clockwise angles are considered to be positive angles, and angles are given in radians.

1.3 Overview

This thesis begins by considering a solution, "Least Work Matching", to the vertex correspondence problem (Chapter 2). Section 2.2 and its subsections develop the algorithm and discuss the calculations required for the algorithm, and section 2.3 discusses the results of applying the algorithm to various polygons. Chapter 3 presents "Intrinsic Interpolation", a method used to solve the vertex path problem. This method is developed in section 3.2. A variation on the method, Edge Tweaking, is discussed in section 3.3, and section 3.4 gives a summary and results. Chapter 4 deals with the blending of Bézier curves. Section 4.2 discusses blending based on the control polygon of a curve, and section 4.3 discusses the Least Work Curve Matching algorithm. Section 4.5 gives the results of applying these methods to some examples. Chapter 5 gives conclusions and looks at future work. Appendix A.2 introduces the computer program that was coded for the implementation portion of this thesis. Appendix A.3 discusses some of the noteworthy aspects of the implementations, and appendix A.4 gives a listing of the code.

Chapter 2: Least Work Matching

2.1 Introduction

Least Work Matching, presented in [1], is a method for smoothly blending two 2-dimensional shapes. The general idea behind this solution is to consider the shapes to have edges made of bendable, stretchable wire, and then to bend and stretch these wires until the first shape is transformed into the second, while minimizing a quantity analogous to work (energy) used in bending and stretching the wires. If the energy expended in bending and stretching the wires is minimized, then the amount of bending and stretching is therefore minimized, resulting in a blend with minimal motion of the wires. Typically, minimal distortion of the wires is thought to be most visually pleasing.

The goal of the algorithm is to determine the vertex correspondence which results in the least amount of work required to transform the first shape to the second. "Work" refers to a measure of the effort expended in bending,

stretching, and shortening the "wires" of the polygon in order to transform themselves from shape 1 to shape 2. Sections 2.2.2 and 2.2.3 discuss the calculation of work for the wires of the polygons.

The algorithm finds the best vertex correspondence using only the existing vertices; that is, no additional distinct vertices are added to either of the polygons by the algorithm (although a user is certainly free to add vertices to the polygons during pre-processing). However, the algorithm *will*, at times, insert vertices at existing vertex locations, resulting in vertices with multiplicity greater than one.

2.2 Development

Since the algorithm must determine the amount of work required for all possible vertex correspondences, we must first determine which vertex correspondences are possible. Let the two polygons to be blended be designated P^0 and P^1, with vertices $P_0^0, P_1^0, ..., P_m^0$, and $P_0^1, P_1^1, ..., P_n^1$, respectively, where $P_0^0 = P_m^0$, and $P_0^1 = P_n^1$ (that is, the polygons are closed). All subscripts are defined modulo the number of vertices on the polygon in question (for example, $P_{m+1}^0 = P_1^0$).

The algorithm depends on the vertices of both shapes being numbered in the same direction. In this thesis, the convention of numbering the vertices in a clockwise direction is used.

In order to determine the best possible correspondence of the vertices of P^0 and P^1, a graph, in the form of an $(n+1)\times(m+1)$ rectangular grid, is used. The vertices of P^0 and P^1 are represented by the columns and the rows of the graph, respectively (Figs. 2.1 and 2.2b).

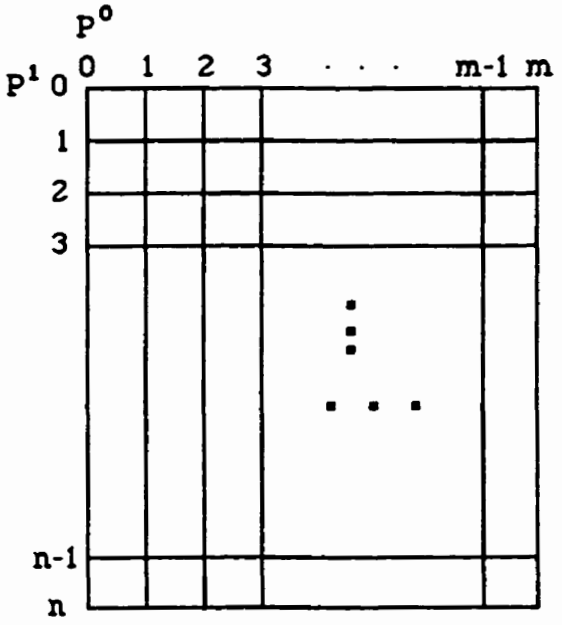


Fig. 2.1 – The grid used for a vertex correspondence graph

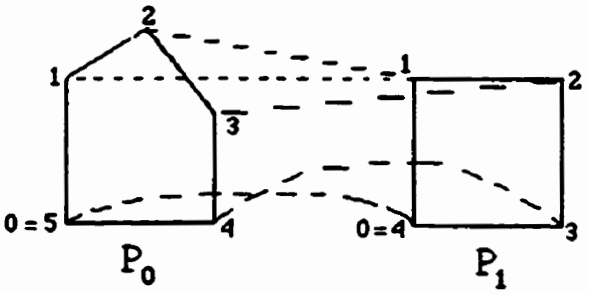


Fig. 2.2a

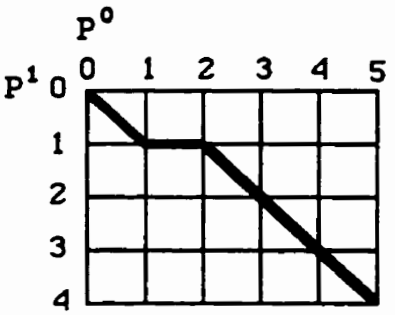


Fig. 2.2b

Fig. 2.2 – Example of a vertex correspondence graph

A correspondence between vertices P_i^0 and P_j^1 is denoted on the grid by a point at location $[i,j]$ (see Fig. 2.2), where, contrary to the general mathematical convention, i refers to the column and j refers to the row. Note that henceforth, a point on the graph will be referred to by the complete phrase "graph vertex" or "grid vertex", as the term "vertex" refers to a vertex of a polygon.

A vertex correspondence between two polygons is considered possible if the following two vertex correspondence conditions apply:

1. P_i^0 may correspond to P_j^1 only if one of the following three conditions holds (see Figs. 2.3 and 2.4):
   a) P_{i-1}^0 corresponds to P_j^1,
   b) P_i^0 corresponds to P_{j-1}^1, or
   c) P_{i-1}^0 corresponds to P_{j-1}^1.

2. Each vertex of a polygon must correspond to at least one vertex in the other polygon, and vice versa.

The first condition prevents the in-between polygons from breaking apart into pieces. The second condition is necessary since all vertices must follow some path from one image to the other (i.e. vertices cannot just vanish or appear out of nowhere).

Starting vertices are required for each polygon, and are labeled P_0^0 and P_0^1. These starting vertices correspond to one another. My implementation simply takes the first vertex in a file of polygon vertices (or the first vertex clicked if the user is drawing her own polygon) as the starting vertex. Therefore, pre-processing is necessary to ensure an appropriate first vertex matching.

Every possible correspondence that adheres to the rules set above will create a continuous path through the graph, starting at the top left corner, [0,0], and proceeding to the bottom right corner, $[m,n]$, and this path will move only to the right and down (or both), but never up or to the left.

Now, the problem of finding the least work vertex correspondence becomes the problem of finding the least work path through the graph.

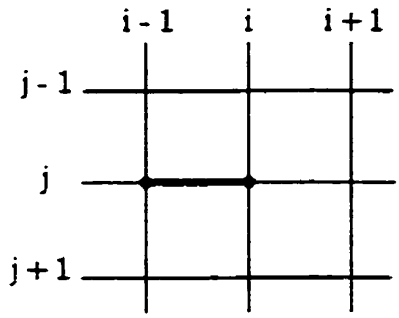


Fig. 2.3a

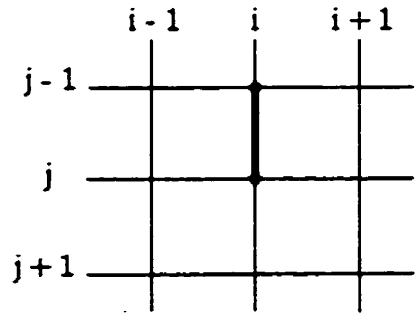


Fig. 2.3b

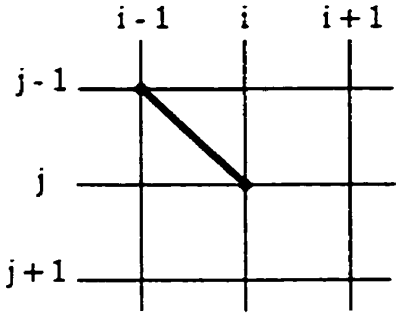


Fig. 2.3c

Fig. 2.3 -- The graph is not allowed to break into pieces

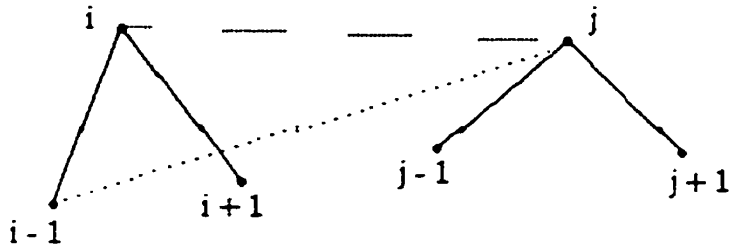


Fig. 2.4a – Corresponds with Fig. 2.3a

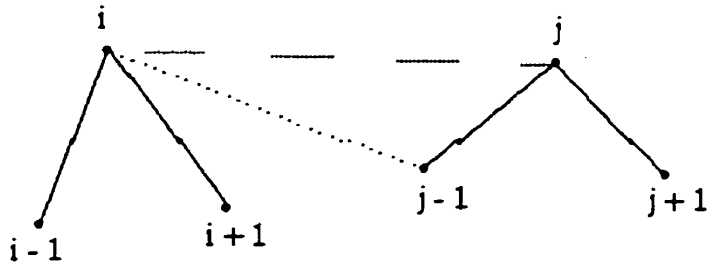


Fig. 2.4b – Corresponds with Fig. 2.3b

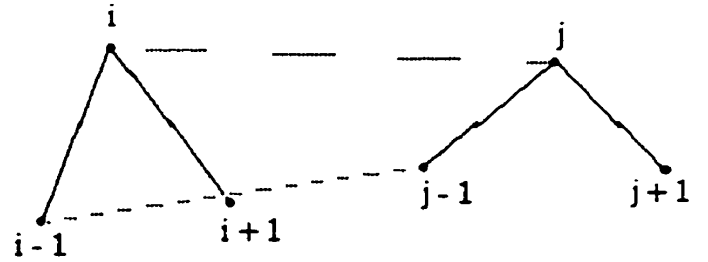


Fig. 2.4c – Corresponds with Fig. 2.3c

Fig. 2.4 – The only three possible vertex correspondences when vertex i corresponds to vertex j (vertex correspondence condition 1)

2.2.1 Finding the Least Work Path

To determine the least work solution, we look at a piece, or fragment, of the polygon P^0 consisting of vertices $P_0^0, P_1^0, ..., P_i^0$ and the edges of the polygon P^0 connecting them, and at a corresponding fragment of the polygon P^1, consisting of vertices $P_0^1, P_1^1, ..., P_j^1$ and the edges of the polygon P^1 connecting them. Call these fragments $P^0(i)$ and $P^1(j)$, respectively.

We define the work value of a graph vertex $[i,j]$ to be the amount of work required to transform fragment $P^0(i)$ to fragment $P^1(j)$. This work value is denoted by $W(i,j)$.

If one or both of fragments $P^0(i)$ and $P^1(j)$ were reduced in size by deleting the corresondence $[i,j]$, then the three following correspondences are possible: $[i-1,j]$, $[i,j-1]$, and $[i-1,j-1]$. In order to determine $W(i,j)$, we must know the work values of these three graph vertices that could precede graph vertex $[i,j]$, that is, $W(i-1,j)$, $W(i,j-1)$, and $W(i-1,j-1)$. $W(i-1,j)$ represents the amount of work required to transform fragment $P^0(i-1)$ to fragment $P^1(j)$. An example of a situation in which $P^0(i-1)$ must be transformed to $P^1(j)$ is shown in Fig. 2.4a. Similarly, $W(i,j-1)$ represents the amount of work required to transform fragment $P^0(i)$ to fragment $P^1(j-1)$

(Fig. 2.4b), and $W(i-1,j-1)$ represents the amount of work required to transform fragment $P^0(i-1)$ to fragment $P^1(j-1)$ (Fig. 2.4c).

If these work values are known, $W(i,j)$ is then equal to the work required to transform one of these preceding fragments plus the additional work required to transform the new part of the fragment. In terms of the graph, $W(i,j)$ is equal to the work required to arrive at a preceding graph vertex plus the amount of work required to travel from the preceding graph vertex to graph vertex $[i,j]$. That is, one of the following three formulae must hold (corresponding to the three possibilities in Fig. 2.4):

$$W(i,j) = W(i-1,j) + \text{the work to transform the edge between vertices } P^0_{i-1} \text{ and } P^0_i \text{ to vertex } P^1_j, \tag{2.1}$$

$$W(i,j) = W(i,j-1) + \text{the work to transform vertex } P^0_i \text{ to the edge between vertices } P^1_{j-1} \text{ and } P^1_j, \tag{2.2}$$

or

$$W(i,j) = W(i-1,j-1) + \text{the work to transform the edge between vertices } P^0_{i-1} \text{ and } P^0_i \text{ to the edge between vertices } P^1_{j-1} \text{ and } P^1_j. \tag{2.3}$$

Therefore, it is necessary that each of these three values of $W(i,j)$ be calculated for each pair of fragments $P^0(i)$ and $P^1(j)$. Denote the work of equation 2.1 as $W_{west}(i,j)$, since the preceding graph vertex on the path is $[i-1,j]$ (i.e. is

directly west of graph vertex $[i,j]$). Similarly, the work of equation 2.2 is denoted by $W_{north}(i,j)$, and the work of equation 2.3 by $W_{northwest}(i,j)$.

We make the requirement that if graph vertex $[i,j]$ is preceded by graph vertex $[i-1,j]$, then graph vertex $[i-1,j]$ must be preceded either by $[i-2,j]$ or by $[i-2,j-1]$, and not by $[i-1,j-1]$ (that is, we do not allow right angles in the graph path). Similarly, if graph vertex $[i,j]$ is preceded by graph vertex $[i,j-1]$, then $[i,j-1]$ must be preceded either by $[i,j-2]$ or by $[i-1,j-2]$.

An example illustrating the reasoning behind this requirement is outlined as follows: suppose we are given vertices a and b of P^0, and the edge between them, $\overline{ab}$, and vertices c and d of P^1, and the edge between them, $\overline{cd}$. If a corresponds with c, and b corresponds with d, intuitively, less work will be required to stretch or shorten edge $\overline{ab}$ into edge $\overline{cd}$ than would be required to stretch vertex a into edge $\overline{cd}$ and then collapse edge $\overline{ab}$ into vertex d.

2.2.2 Stretching Work

Two quantities which are used to measure the result of a force acting on a wire to stretch the wire are strain and stress.

Strain, denoted by ε, is defined as the elongation, ΔL, of the wire, divided by the initial length, L, of the wire:

$$\varepsilon = \frac{\Delta L}{L}. \tag{2.4}$$

Stress, denoted by σ, is defined to be the deforming force, F, acting on the wire, per unit of the wire's cross-sectional area, A:

$$\sigma = \frac{F}{A}. \tag{2.5}$$

When stress acts upon a wire, a strain is produced. Therefore, stress and strain can be plotted against one another to give a stress-strain diagram for a given material. Over the range of usefulness, stress and strain are proportional; over this range, the stress-strain diagram is linear with constant slope. This slope depends solely on the properties of the material of the wire; it does not depend at all on the length or cross-sectional area of the wire. This constant slope is known as Young's modulus of elasticity, E:

$$E = \frac{\sigma}{\varepsilon}. \tag{2.6}$$

A piece of wire that is being stretched will undergo either linear elastic stretching or plastic stretching, depending on the amount of stretching that is occurring in the wire. The yield stress, σ_{yield}, of a material is the elastic limit of the material. σ_{yield} is defined to be the amount of stress beyond which the material suffers permanent damage, and will not return to its original size or shape when the stress is removed (i.e. we say the material undergoes plastic deformation). For any amount of stress below the yield stress, the stretching will be a close approximation to linearly elastic. The modulus of elasticity given in equation 2.6 applies to the elastic stretching of a wire.

The amount of work done by a force F to displace a particle from point a to point b is defined to be

$$W = \int_a^b F(L)dL.$$

If we have a plot of force F vs. displacement from $L = a$ to $L = b$, work is therefore the area under the curve.

Thus, the work per unit volume, $\frac{W}{AL}$, required to stretch a wire by an amount ΔL is the area under the stress-strain curve (i.e. $\frac{W}{AL} = \frac{1}{2}\sigma\varepsilon$). Therefore, for elastic stretching,

$$\begin{aligned} W_{stretch} &= \frac{1}{2}\sigma\varepsilon AL \\ &= \frac{1}{2}\sigma A\frac{(\Delta L)^2}{\varepsilon L} \\ &= \frac{(\Delta L)^2 EA}{2L}. \end{aligned} \tag{2.7}$$

For a more comprehensive treatment of stress, strain and work, see [11].

We will start by examining equation 2.7, and making several changes to render it suitable for use here.

Since the "wire" polygon edges do not possess any real physical qualities, both A and E can be defined by the user to suit the specific needs of a particular blend. Replace AE by the constant $k_{stretch}$, whose value represents the stretchiness of the wire. A lower value of $k_{stretch}$ indicates a stretchier wire (a wire requiring less work to stretch), and a higher value indicates a wire that is more difficult to stretch.

In our application, we would like to require stretching of a wire to include both the lengthwise stretching and shrinkage of the wire. This condition is imposed to ensure that a blend between initial polygon 0 to final polygon 1 is the same blend (in reverse) as that between initial polygon 1 to final polygon 0. Therefore, the work involved in stretching a wire of length L_0 into a wire of length L_1 should be the same as the work involved in shortening a wire of length L_1 into a wire of

length L_0. Equation 2.7 does not satisfy this requirement. Furthermore, if a single vertex is stretched out to a line, its original length is 0, which results in an infinite amount of stretching work. The solution to these problems is to use a combination of the two lengths, as will be shown shortly.

In many situations, it is undesirable for an edge to collapse to a point, or for a point to be stretched out into an edge. Thus, a user-defined constant $c_{stretch}$, $0 \le c_{stretch} \le 1$, is introduced to penalize this behaviour, if the user should so choose. Lower values of $c_{stretch}$ indicate greater penalty. Hence, for a polygon with an edge of length L, the quantity $2L$ is replaced by the quantity $((1-c_{stretch})\min(L_0, L_1) + c_{stretch}\max(L_0, L_1))$.

The exponent of 2 in equation 2.7 is changed to be a user-defined constant, $e_{stretch}$. The exponent in the equation will vary, depending on how much stretching will occur in the blend. For example, if the wire does not stretch too much, the stretching will be linearly elastic, and an exponent of 2 will be sufficient. However, if the wire stretches quite a bit and undergoes plastic deformation, less work is required to stretch the wire, and an exponent of 1 would represent the situation more accurately. An exponent of 1 in equation 2.7 does not exactly represent the plastic deformation situation, but, rather, is an approximation. However, since the "wires" used for the polygons are not physical wires, this approximation is sufficient. Furthermore, also due to the fact that the wires are not real, the choice of $e_{stretch}$ is very subjective.

Employing these changes, the equation for the work required to stretch a segment of polygon 0 into a segment of polygon 1 is given by:

$$W_{stretch} = k_{stretch} \frac{|L_1 - L_0|^{e_{stretch}}}{(1 - c_{stretch})\min(L_0, L_1) + c_{stretch}\max(L_0, L_1)}, \tag{2.8}$$

where, again,

L_0 = length of the segment of polygon 0,

L_1 = length of the segment of polygon 1,

$k_{stretch}$ is an elasticity constant of the "wire" polygon edge,

$c_{stretch}$ is a constant that penalizes an edge if it collapses to a point, and

$e_{stretch}$ is a plastic deformation constant.

2.2.3 Bending Work

Bending work is the amount of work required to change an angle defined by vertices i_0, i_1, and i_2 of polygon 0 to the angle defined by vertices j_0, j_1, and j_2 of polygon 1. The amount of work required to change an angle is dependent on the change in the size of the angle, $\Delta\theta$, from one polygon to the other.

Many angles in a blend do not change monotonically from one shape to the other. Real elastic bending is unconcerned with non-monotonicity, since any work used to bend an angle an amount θ will be released if the angle unbends.

However, our application is concerned with the calculation of work solely for the purpose of minimizing the motion of the wires. Therefore, we will assume that any angle change, regardless of direction, is governed by the same work calculation. Hence, in the cases in which angles do not change monotonically, knowing only the value of $\Delta\theta$ will give an inaccurate description of the amount of work taking place. It is important here to also calculate the amount that the angle deviates from monotonicity, denoted $\Delta\theta^*$. Hence, bending work can now be defined by:

$$\Delta\theta + \Delta\theta^*.$$

The calculations of $\Delta\theta$ and $\Delta\theta^*$ are discussed in sections 3.3.2.1 and 3.3.2.2, respectively.

Non-monotonicity in an angle change is often not thought to be a pleasing or natural feature. Therefore, such behaviour, in some circumstances, should be penalized. Penalty is imposed by way of a multiplicative constant, m_{bend}, which can be chosen by the user. The choice of m_{bend} will depend on how undesirable non-monotonicity is in a particular blend. Higher values of m_{bend} indicate greater difficulty in bending, while lower values indicate greater ease of bending. This yields a bending work equation of

$$\Delta\theta + m_{bend}\Delta\theta^*.$$

Another problem that may arise when bending the angles is that of collapsing angles. In Fig. 2.5, the angles in the top right and bottom left corners become smaller and smaller until they collapse, after which the edges essentially "cross over" one another.

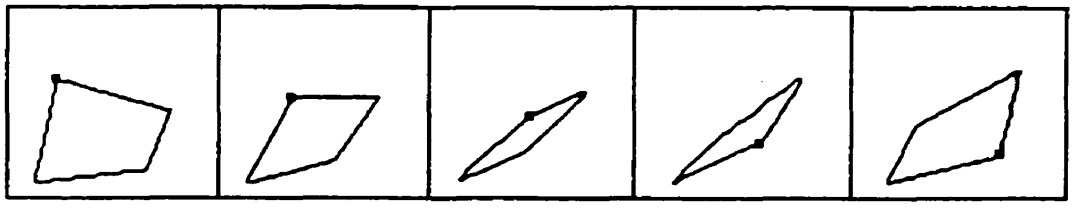


Fig. 2.5 – Collapsing angles

This sort of behaviour creates the appearance of a polygon turning inside out, or collapsing to a line and then reconstructing itself. Collapsing angles, or angles which go to zero, are penalized with the use of the user-defined, additive constant, p_{bend}:

$$\Delta\theta + m_{bend}\Delta\theta^* + p_{bend}.$$

If an angle collapses, the quantity p_{bend} is added to the work calculation. If an angle does not collapse, nothing is added. A discussion on how to determine which angles collapse is given in section 3.2.2.3.

As in stretching work, the user may choose the difficulty with which the angles can bend. This is done via a user-defined multiplicative constant k_{bend}, and a user-defined exponential constant, e_{bend}.

Thus, the final work equation for calculating work due to bending is given by

$$\begin{aligned} W_{bend} &= k_{bend}(\Delta\theta + m_{bend}\Delta\theta^*)^{e_{bend}}, && \text{if } \theta(t) \text{ does not go to zero} \\ &= k_{bend}(\Delta\theta + m_{bend}\Delta\theta^*)^{e_{bend}} + p_{bend}, && \text{if } \theta(t) \text{ does go to zero} \end{aligned} \qquad (2.9)$$

where, again,

$\Delta\theta$ = change in angle from P^0 to P^1,

$\Delta\theta^*$ = deviation from monotonicity of the angle change,

m_{bend} is a constant which penalizes non-monotonically changing angles,

p_{bend} is a constant which penalizes angles that go to zero,

e_{bend} is an exponential bending stiffness constant, and

k_{bend} is a multiplicative bending stiffness constant.

2.2.3.1 Calculating the Change in Angle Size

Denote the angle at vertex i, as it changes over time $t \in [0,1]$ by $\theta_i(t)$. $\theta_i(0)$ gives the angle at vertex i of the initial polygon, and $\theta_i(1)$ gives the angle at the corresponding vertex in the final polygon.

If a, b, and c are three consecutive vertices of a polygon, and if $\overline{ab}$ is the edge between vertices a and b, and $\overline{bc}$ is the edge between vertices b and c, then we use the notation $\angle[a,b,c]$ to denote the acute angle between $\overline{ab}$ and $\overline{bc}$.

If we let $\bar{i}$ denote the vertex of P^1 that corresponds with vertex i of P^0, and if we assume that the vertices follow a linear path from P^0 to P^1, then the path that vertex P_i^0 follows during the blend is given by

$$(1-t)P_i^0 + tP_{\bar{i}}^1, \quad t \in [0,1].$$

Therefore, the angle which initially is defined by the three vertices P_{i-1}^0, P_i^0, and P_{i+1}^0 is given by

$$\theta_i(t) = \angle[(1-t)P_{i-1}^0 + tP_{\overline{i-1}}^1, (1-t)P_i^0 + tP_{\bar{i}}^1, (1-t)P_{i+1}^0 + tP_{\overline{i+1}}^1], \tag{2.10}$$

for time $t \in [0,1]$.

This angle can easily be translated to the origin, giving

$$\theta_i(t) = \angle[(1-t)\tilde{P}_{i-1}^0 + t\tilde{P}_{\overline{i-1}}^1, \mathbf{0}, (1-t)\tilde{P}_{i+1}^0 + t\tilde{P}_{\overline{i+1}}^1], \tag{2.11}$$

where $\tilde{P}^0_{i-1} = P^0_{i-1} - P^0_i$,

$\tilde{P}^0_{i+1} = P^0_{i+1} - P^0_i$,

$\tilde{P}^1_{i-1} = P^1_{i-1} - P^1_i$, and

$\tilde{P}^1_{i+1} = P^1_{i+1} - P^1_i$.

Now the angles can be measured with respect to the positive x-axis; that is, as

$$\angle[(1,0),\mathbf{0},(1-t)\,\tilde{P}^0_{i-1} + t\,\tilde{P}^1_{i-1}] - \angle[(1,0),\mathbf{0},(1-t)\,\tilde{P}^0_{i+1} + t\,\tilde{P}^1_{i+1}].$$

For example, the angle $\angle[a,\mathbf{0},b]$ of Fig. 2.6 can be calculated as

$$\angle[(1,0),\mathbf{0},a] - \angle[(1,0),\mathbf{0},b].$$

It would be convenient to determine the point c such that

$$\angle[(1,0),\mathbf{0},c] = \angle[(1,0),\mathbf{0},a] - \angle[(1,0),\mathbf{0},b],$$

and then refer to the angle $\angle[a,\mathbf{0},b]$ in terms of the point c. We will call this point c the angle-defining point of $\angle[a,\mathbf{0},b]$.

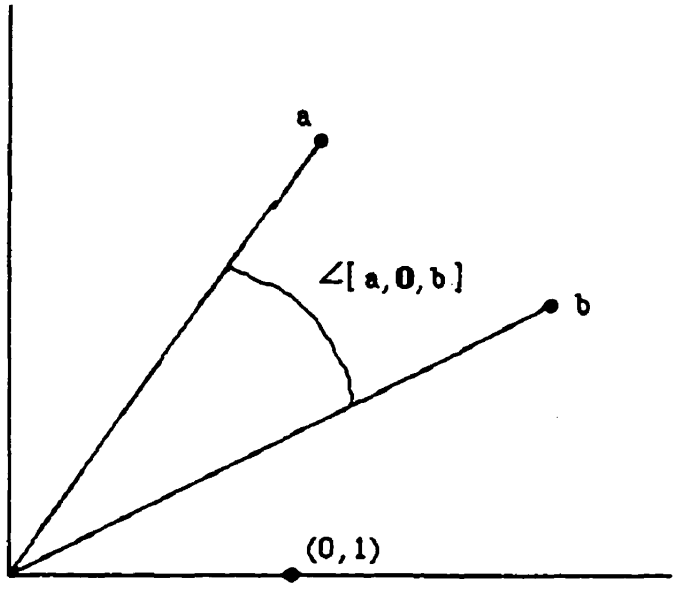


Fig. 2.6 – Measuring angles

The y-coordinate of this angle-defining point of the angle given by equation 2.11 is given by

$$\begin{aligned}\sin(\theta_i(t)) &= \sin(\angle[(1-t)\tilde{P}^0_{i-1} + t\tilde{P}^1_{i-1}, \mathbf{0}, (1-t)\tilde{P}^0_{i+1} + t\tilde{P}^1_{i+1}]) \\ &= \frac{[(1-t)\tilde{P}^0_{i-1} + t\tilde{P}^1_{i-1}] \times [(1-t)\tilde{P}^0_{i+1} + t\tilde{P}^1_{i+1}]}{\left\|(1-t)\tilde{P}^0_{i-1} + t\tilde{P}^1_{i-1}\right\| \cdot \left\|(1-t)\tilde{P}^0_{i+1} + t\tilde{P}^1_{i+1}\right\|},\end{aligned} \tag{2.12}$$

and the x-coordinate by

$$\begin{aligned}\cos(\theta_i(t)) &= \cos(\angle[(1-t)\tilde{P}_{i-1}^0 + t\tilde{P}_{i-1}^1, \mathbf{0}, (1-t)\tilde{P}_{i+1}^0 + t\tilde{P}_{i+1}^1]) \\ &= \frac{\left|[(1-t)\tilde{P}_{i-1}^0 + t\tilde{P}_{i-1}^1] \cdot [(1-t)\tilde{P}_{i+1}^0 + t\tilde{P}_{i+1}^1]\right|}{\left\|(1-t)\tilde{P}_{i-1}^0 + t\tilde{P}_{i-1}^1\right\| \cdot \left\|(1-t)\tilde{P}_{i+1}^0 + t\tilde{P}_{i+1}^1\right\|},\end{aligned} \tag{2.13}$$

where the operator $\times$ is defined as

$$P_i^k \times P_j^k = x_i y_j - x_j y_i,$$

where $P_i^k = (x_i, y_i)$ and $P_j^k = (x_j, y_j)$, and the operator $\cdot$ is the usual dot-product.

Disregarding the equal denominators of these equations, and expanding gives

$$y(t) = (1-t)^2\left|\tilde{P}_{i-1}^0 \times \tilde{P}_{i+1}^0\right| + 2t(1-t)\frac{\left|\tilde{P}_{i-1}^0 \times \tilde{P}_{i+1}^1\right| + \left|\tilde{P}_{i-1}^1 \times \tilde{P}_{i+1}^0\right|}{2} + t^2\left|\tilde{P}_{i-1}^1 \times \tilde{P}_{i+1}^1\right|, \tag{2.14}$$

and

$$x(t) = (1-t)^2(\tilde{P}_{i-1}^0 \cdot \tilde{P}_{i+1}^0) + 2t(1-t)(\frac{\tilde{P}_{i-1}^0 \cdot \tilde{P}_{i+1}^1 + \tilde{P}_{i-1}^1 \cdot \tilde{P}_{i+1}^0}{2}) + t^2(\tilde{P}_{i-1}^1 \cdot \tilde{P}_{i+1}^1). \quad (2.15)$$

A quadratic Bézier curve is a curve guided by three control points, $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$, and is given by

$$Q(t) = (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2, \ t \in [0,1].$$

Development of this formula can be found in any elementary computer graphics text (eg. [4], [5], et al.).

These two equations together have the form of a quadratic Bézier curve:

$$Q(t) = \mathbf{Q}_0(1-t)^2 + \mathbf{Q}_1 2t(1-t) + \mathbf{Q}_2 t^2, \quad (2.16)$$

where

$$\mathbf{Q}_0 = (\left|\tilde{P}_{i-1}^0 \times \tilde{P}_{i+1}^0\right|, (\tilde{P}_{i-1}^0 \cdot \tilde{P}_{i+1}^0)), \quad (2.17a)$$

$$\mathbf{Q}_1 = (\frac{\left|\tilde{P}_{i-1}^0 \times \tilde{P}_{i+1}^1\right| + \left|\tilde{P}_{i-1}^1 \times \tilde{P}_{i+1}^0\right|}{2}, \frac{\tilde{P}_{i-1}^0 \cdot \tilde{P}_{i+1}^1 + \tilde{P}_{i-1}^1 \cdot \tilde{P}_{i+1}^0}{2}), \quad (2.17b)$$

and

$$\mathbf{Q}_2 = (\left|\tilde{P}^1_{i-1} \times \tilde{P}^1_{i+1}\right|, (\tilde{P}^1_{i-1} \cdot \tilde{P}^1_{i+1})). \tag{2.17c}$$

As time changes, the coordinates (x, y) change (since $(x, y) = Q(t)$). Therefore, as a line through the origin follows this curve, the angle that this line makes with the x-axis changes exactly as the corresponding angle in the blend changes. That is,

$$\theta(t) = \angle[(1,0),(0,0),Q(t)]. \tag{2.18}$$

The possibilities for extreme values of the angle are $\theta(0)$, $\theta(1)$ and angles $\theta(t_r)$ such that the line through the origin and $Q(t_r)$ is the tangent line to $Q(t)$ at the point $t = t_r$. This property can be expressed by the equation

$$Q(t) \times Q'(t) = 0, \tag{2.19a}$$

where

$$Q'(t) = -2\mathbf{Q}_0(1-t) + \mathbf{Q}_1(1-2t) + 2\mathbf{Q}_2 t$$

(that is, $Q'(t)$ is the first derivative of $Q(t)$ with respect to t).

Expanding equation 2.19a and reducing gives

$$|\mathbf{Q}_0 \times \mathbf{Q}_1|(1-t)^2 + \frac{|\mathbf{Q}_0 \times \mathbf{Q}_2|}{2} 2t(1-t) + |\mathbf{Q}_1 \times \mathbf{Q}_2|t^2 = 0, \tag{2.19b}$$

which is a quadratic Bézier equation.

If the angle θ changes monotonically, then the only extreme values of θ occur at $t = 0$ and $t = 1$, and there are no values of $t \in (0,1)$ such that equation 2.19b holds.

If the angle does *not* change monotonically, then the extreme values need not occur at $t = 0$ and $t = 1$, so there are either one or two values of $t \in (0,1)$ which produce extreme values of θ (i.e. such that equation 2.19 holds). For an example, see Fig. 2.7. As the line from the origin to the curve follows the curve, it first swings counter-clockwise (which is a deviation from monotonicity) before moving in a clockwise direction toward $\mathbf{Q}_2$.

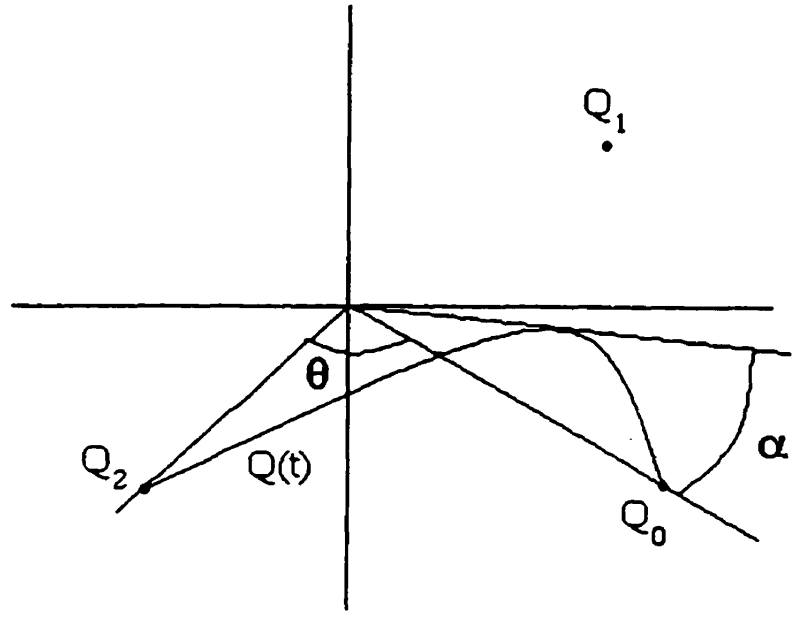


Fig. 2.7 – Deviation from monotonicity

The net change in angle, $\Delta\theta$, will be either:

1. $\angle[\mathbf{Q_0}, \mathbf{0}, \mathbf{Q_2}]$, if the angle changes less than π radians, or

2. $2\pi - \angle[\mathbf{Q_0}, \mathbf{0}, \mathbf{Q_2}]$, if the angle changes more than π radians (that is, if the angle that the line from $Q(t)$ to (0,0) makes with the x-axis changes more than π radians as t changes from 0 to 1).

Assertion: The angle changes more than π radians if and only if the following two conditions hold:

1. The triangle with vertices $\mathbf{Q}_0$, $\mathbf{Q}_1$, and $\mathbf{Q}_2$ contains the origin, and

2. Equation 2.19 has no solutions $t \in (0,1)$ (i.e. θ changes monotonically).

Proof of Assertion:

First, suppose $\Delta\theta > \pi$. We will show that the triangle $\mathbf{Q}_0\mathbf{Q}_1\mathbf{Q}_2$ must contain the origin, and that θ must change monotonically.

We first show that the triangle $\mathbf{Q}_0\mathbf{Q}_1\mathbf{Q}_2$ must contain the origin. If we extend the line between $\mathbf{Q}_0$ and (0,0), then $\mathbf{Q}_2$ must lie on one side of the line and $\mathbf{Q}_1$ on the other. This facilitates the rotation of more than π radians. (If both points lie on the same side of the line between $\mathbf{Q}_0$ and (0,0), then the entire curve $Q(t)$ would lie in the half plane defined by the line between $\mathbf{Q}_0$ and (0,0). This would mean that any line from the origin that follows the curve would lie entirely in that half plane, which would imply that the net change in angle was less than or equal to π radians. See Fig. 2.8.) Similarly, if we extend the line between $\mathbf{Q}_2$ and (0,0), then $\mathbf{Q}_0$ must lie on one side and $\mathbf{Q}_1$ on the other. Therefore, $\mathbf{Q}_1$ must lie in the region **A** (shown in Fig. 2.9). This implies that the triangle $\mathbf{Q}_0\mathbf{Q}_1\mathbf{Q}_2$ contains the point (0,0).

Secondly, we show that θ must change monotonically. Since we have assumed that $\Delta\theta > \pi$, $Q(t)$ must, at some point, pass through region **A** (Fig. 2.9). We will argue that $Q(t)$ passing through region **A** implies that θ must change monotonically.

If $Q(t)$ passes through region **A** and θ does *not* change monotonically, then one of the following two situations must occur:

1. The line from $Q(t)$ to (0,0) would start traveling from $\mathbf{Q_0}$ in the direction opposite of that which facilitates a rotation of more than π radians (i.e. in the direction of the smallest angle between the line from $\mathbf{Q_0}$ (0,0) and the line from $\mathbf{Q_2}$ to (0,0)) before changing directions and heading toward $\mathbf{Q_2}$ in the direction facilitating a rotation of more than π radians (in terms of Fig. 2.7, the line from $Q(t)$ to (0,0) would have to travel clockwise from $\mathbf{Q_0}$ and then change direction to travel counter-clockwise toward $\mathbf{Q_2}$), or

2. The line from $Q(t)$ to (0,0) would have to travel in the direction facilitating a rotation of more than π radians past $\mathbf{Q_2}$, before turning back and traveling in the opposite direction to end up at $\mathbf{Q_2}$. (To correspond with Fig. 2.7, the line from $Q(t)$ to (0,0) would have to travel counter-clockwise past $\mathbf{Q_2}$ and then clockwise back toward $\mathbf{Q_2}$.)

(Note that these are the only two possibilities for deviation from monotonicity: if θ were to deviate, say, part of the way through the angle change, the curve

$Q(t)$ would have inflection points, which is not possible with quadratic Bézier curves.)

Neither situation 1 nor 2 is possible here. Since at $\mathbf{Q}_0$, $Q(t)$ is tangent to the line between $\mathbf{Q}_1$ and $\mathbf{Q}_0$ (by the definition of a Bézier curve), and since $\mathbf{Q}_1$ is in region **A**, the line from $Q(t)$ to (0,0) starts out in the correct direction (that is, in the direction which facilitates a rotation of more than π radians). Similarly, the line from $Q(t)$ to (0,0) must end its travels in the correct direction. Since we have determined that deviation from the correct direction is not possible except at the beginning or the end of the curve, θ must change monotonically.

Now we will suppose that the triangle $\mathbf{Q}_0\mathbf{Q}_1\mathbf{Q}_2$ contains the origin, and that θ changes monotonically, and we will show that this implies that $\Delta\theta > \pi$.

Since the triangle $\mathbf{Q}_0\mathbf{Q}_1\mathbf{Q}_2$ contains the origin, the line from $\mathbf{Q}_0$ to (0,0) has $\mathbf{Q}_1$ on one side and $\mathbf{Q}_2$ on the other, and the line from $\mathbf{Q}_2$ to (0,0) has $\mathbf{Q}_0$ on one side and $\mathbf{Q}_1$ on the other. Draw a line through the origin that is parallel to the line through $\mathbf{Q}_2$ and $\mathbf{Q}_0$, and define a region **B** to be the region on the opposite side of this line as the points $\mathbf{Q}_2$ and $\mathbf{Q}_0$ (see Fig. 2.11). The start and end of the curve occur on the same side of this line. If the curve were to pass into region **B** (i.e. to the opposite side of this line), then the angle would have to change by more than π radians (since the angle change in region **B** is π radians in itself).

We will show that the curve must pass into this region **B.**

It is easy to see that region **A** is entirely contained by region **B**. We will therefore show that the curve must pass into region **A**, and thereby infer that the curve must pass through region **B**.

Suppose that $Q(t)$ does not cross into region **A**. Since $Q(t)$ is a Bézier curve, $Q(t)$ (at $t = 0$) is tangent to the line between $\mathbf{Q}_0$ and $\mathbf{Q}_1$. This line between $\mathbf{Q}_0$ and $\mathbf{Q}_1$ is on the opposite side of the line between $\mathbf{Q}_0$ and (0,0) as the line between $\mathbf{Q}_2$ and (0,0). Therefore the angle must first travel away from $\mathbf{Q}_2$ before traveling toward it, which means that θ deviates from monotonicity (see Fig. 2.10). However, our assumption states that θ must change monotonically. Therefore, $Q(t)$ must cross into region **A**.

Since $Q(t)$ crosses into region **A**, it also crosses into region **B**. As we showed above, if $Q(t)$ crosses into region **B**, the angle changes more than π radians.

This concludes the justification of the assertion.

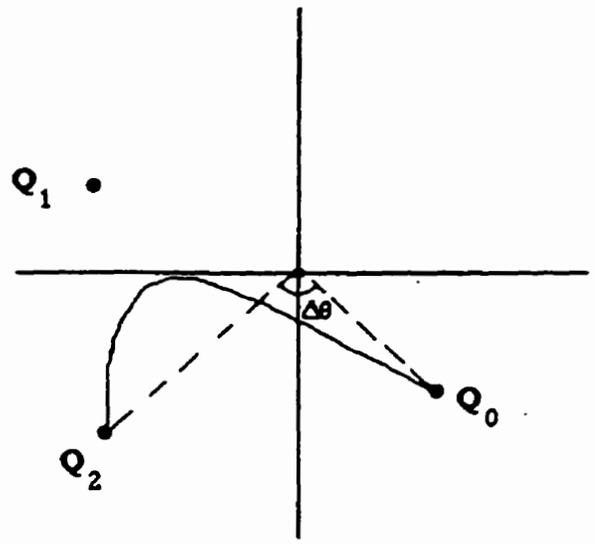


Fig. 2.8 – $\Delta\theta$ is less than π radians

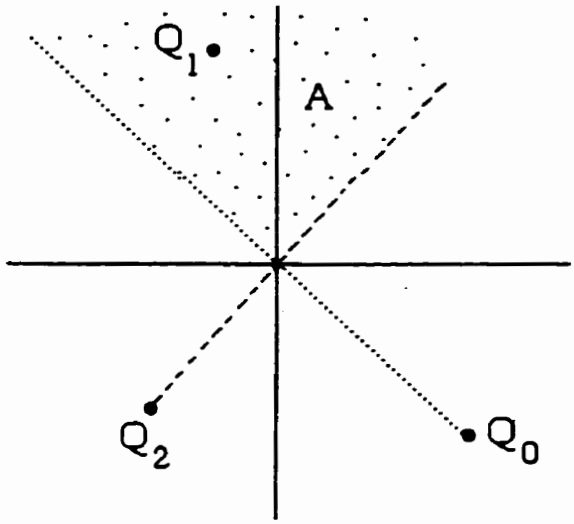


Fig. 2.9 – $\mathbf{Q_1}$ must lie in region **A**

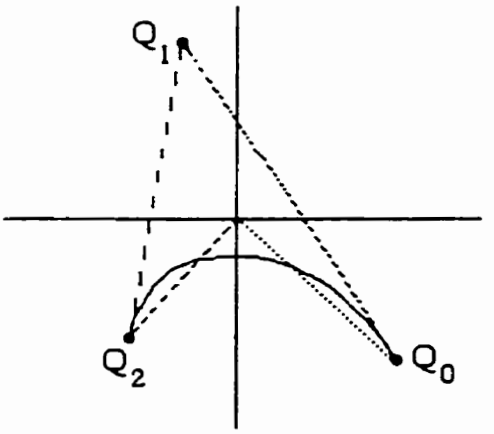


Fig. 2.10 – $Q(t)$ must start out along line segment between $\mathbf{Q}_1$ and $\mathbf{Q}_0$

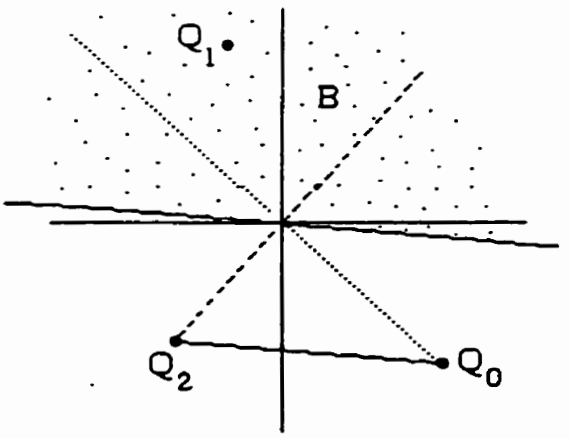


Fig. 2.11 – $Q(t)$ must pass through region **B**

2.2.3.2 Deviation from Monotonicity

If an angle does not change monotonically (i.e. if we find values of $t \in [0,1]$ such that equation 2.19 holds), then we must determine how far the angle deviates from monotonicity.

Deviation can occur in either direction; either the line from (0,0) to $Q(t)$ travels from $Q(0)$ (= $\mathbf{Q}_0$) away from $Q(1)$ (= $\mathbf{Q}_2$) before changing direction and heading back toward $Q(1)$, or the line travels past the angle $\angle[(1,0),\mathbf{0},Q(1)]$ before turning and heading back toward $\mathbf{Q}_2$ (see Figs. 2.7 and 2.12).

To calculate this deviation, solve equation 2.19 for t_1 and t_2. Then the deviation given by t_1, and denoted by α, is

$$\alpha = \angle[Q(t_1),\mathbf{0},\mathbf{Q}_0],$$

and the deviation given by t_2, and denoted by β, is

$$\beta = \angle[Q(t_2),\mathbf{0},\mathbf{Q}_2].$$

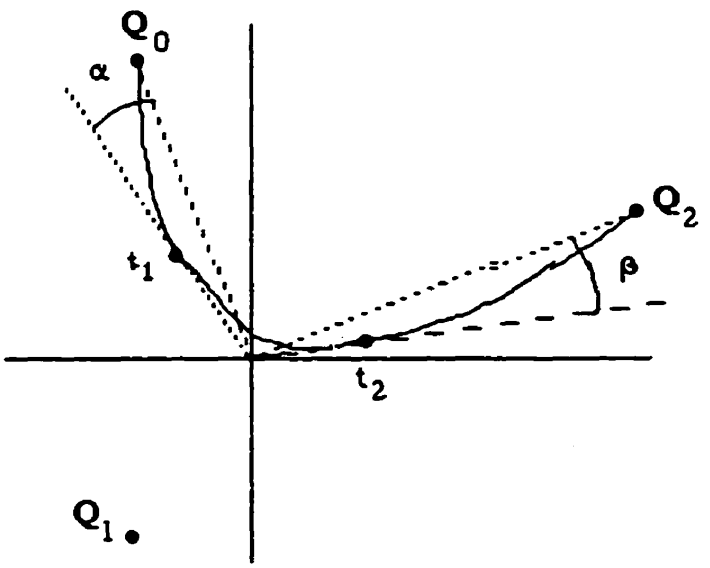


Fig. 2.12 – Values of t_1 and t_2

2.2.3.3 Collapsing Angles

We say an angle collapses if it goes to zero at some point during the transition. That is, $\theta(t) = 0$ for some $t \in (0,1)$. Clearly, this happens only when the curve $Q(t)$ crosses the positive x-axis. (Recall that in section 2.2.3.1 we manipulated the angles so that they are measured with respect to the positive x-axis.)

2.2.3.4 Multiple Vertices

Special problems arise when a polygon contains vertices of multiplicity greater than one (i.e. two or more distinct vertices of one polygon all map to a single vertex of the other polygon). Specifically, how does one calculate an angle defined by three points, when two, or perhaps all three, of the points are exactly the same? The solution to this problem is to pretend that the vertex of multiplicity n is actually n distinct vertices, spaced infinitely close together. These vertices lie along an infinitely short edge, inserted between the two edges incident to the vertex in question, in such a way that the angles between this new edge and each of the incident edges are equal to one another (each equal, in fact, to one half the angle between the two original edges, plus $\pi/2$ radians). Of course, the angles between any interior edges of this new infinitely short edge will be π radians. See Fig. 2.13.

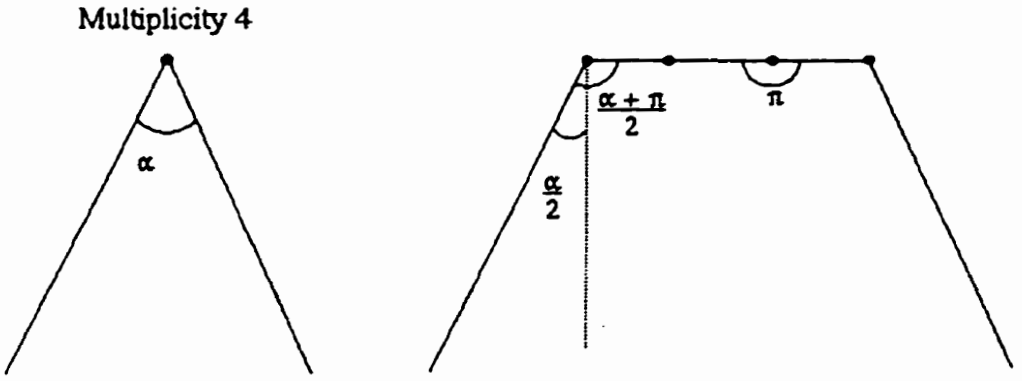


Fig. 2.13 – Magnified view of a vertex of multiplicity 4

2.2.4 The Least Work Path Revisited

We denote the amount of work required to stretch (or shorten) an edge between vertices P_a^0 and P_b^0 of polygon P^0 into an edge between vertices P_c^1 and P_d^1 of polygon P^1 (where vertex P_a^0 corresponds to vertex P_c^1, and vertex P_b^0 corresponds to vertex P_d^1) by

$$W_{stretch}([P_a^0, P_c^1], [P_b^0, P_d^1]).$$

Similarly, the amount of work required to change an angle defined by vertices P_a^0, P_b^0, and P_c^0 of polygon P^0 into an angle defined by vertices P_d^1, P_e^1, and P_f^1 of polygon P^1 (where P_a^0 corresponds to P_d^1, P_b^0 corresponds to P_e^1, and P_c^0 coresponds to P_f^1) is denoted by

$$W_{bend}([P_a^0, P_d^1], [P_b^0, P_e^1], [P_c^0, P_f^1]).$$

Now that we have described how to calculate bending and stretching work (sections 2.2.3 and 2.2.4), the pseudo-equations 2.1, 2.2 and 2.3 can be written more concisely:

$$W_{west}(i,j) = \min(W1, W2) + W_{stretch}([i-1,j],[i,j]), \tag{2.20}$$

$$W_{north}(i,j) = \min(W3, W4) + W_{stretch}([i,j-1],[i,j]), \tag{2.21}$$

and

$$W_{northwest}(i,j) = \min(W5, W6, W7) + W_{stretch}([i-1,j-1],[i,j], \tag{2.22}$$

where

$$W1 = W_{west}(i-1,j) + W_{bend}([i-2,j],[i-1,j],[i,j]), \tag{2.20a}$$

$$W2 = W_{northwest}(i-1,j) + W_{bend}([i-2,j-1],[i-1,j],[i,j]), \tag{2.20b}$$

$$W3 = W_{north}(i,j-1) + W_{bend}([i,j-2],[i,j-1],[i,j]), \tag{2.21a}$$

$$W4 = W_{northwest}(i,j-1) + W_{bend}([i-1,j-2],[i,j-1],[i,j]), \tag{2.21b}$$

$$W5 = W_{north}(i-1,j-1) + W_{bend}([i-1,j-2],[i-1,j-1],[i,j]), \tag{2.22a}$$

$$W6 = W_{northwest}(i-1, j-1) + W_{bend}([i-2, j-2],[i-1, j-1],[i, j]), \qquad (2.22b)$$

and

$$W7 = W_{west}(i-1, j-1) + W_{bend}([i-2, j-1],[i-1, j-1],[i, j]). \qquad (2.22c)$$

To better understand these equations, let us look at equation 2.20 (along with the corresponding equations 2.20a and 2.20b).

The first term of equation 2.20 is the lesser of the following:

1. The work at the graph vertex $[i-1, j]$), arrived at from the vertex directly west of $[i-1, j]$ (that is, graph vertex $[i-2, j]$), plus the amount of work to bend the angle formed by these two edges of polygon 0 into the angle formed by the corresponding edges of polygon 1.

2. The work at the graph vertex west of $[i-1, j]$, arrived at from the vertex directly northwest of $[i-1, j]$ (i.e. graph vertex $[i-2, j-1]$), plus the amount of work to bend the angle formed by these two edges of polygon 0 into the angle formed by the corresponding edges of polygon 1.

The second term, $W_{stretch}([i-1,j],[i,j])$, of equation 2.20 is the amount of work necessary to stretch the edge of polygon 0 defined by vertices P_{i-1}^0 and P_i^0 into the edge of polygon 1 defined by vertices P_j^1 and P_j^1 (i.e. the single vertex P_j^1). That is, it is the work involved in collapsing the edge of polygon 0 in question into a particular vertex of polygon 1.

Figs. 2.14a and 2.14b give the graph theory representation of equation 2.20, and Figs. 2.15a and 2.15b give a corresponding polygon representation.

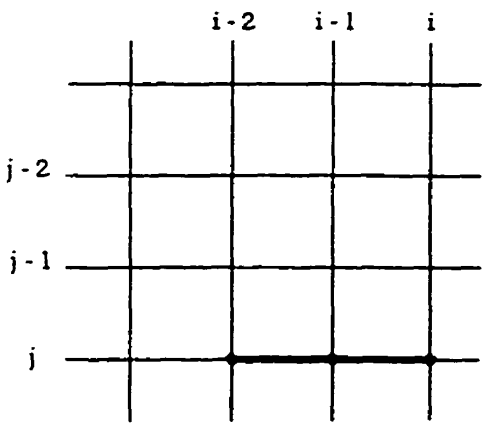


Fig. 2.14a

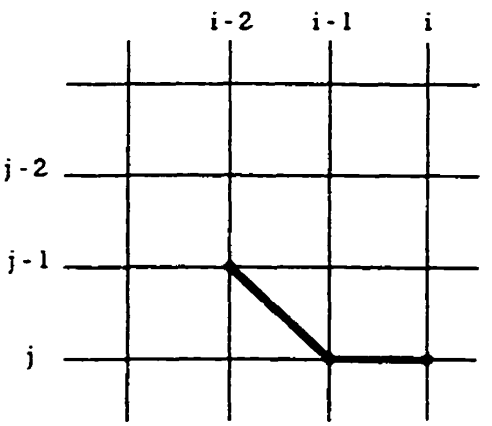


Fig. 2.14b

Fig. 2.14 – Work from the west vertex

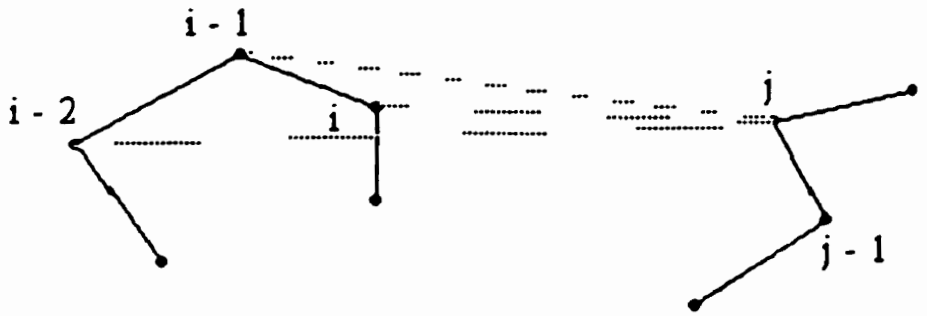


Fig. 2.15a

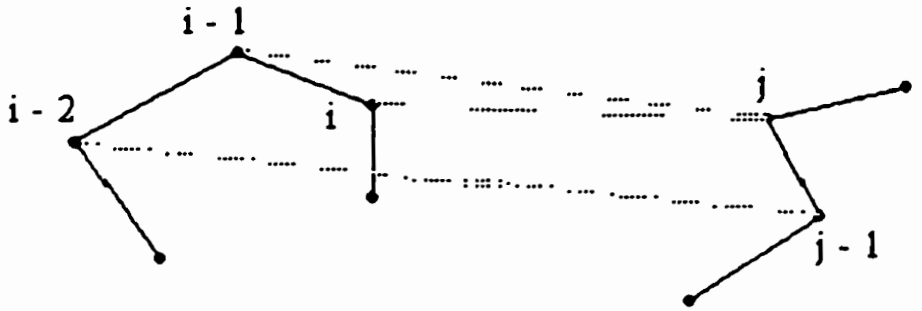


Fig. 2.15b

Fig. 2.15 – Polygon vertex correspondence for graph of Fig. 2.14

Once the work values have been calculated, the least work path through the graph must be found. This is done by backtracking, as follows:

1. Start with vertex (i,j), where $i = m$ and $j = n$.

2. Choose the smallest of the three work values for vertex (i,j) from equations (2.21), (2.22) and (2.23).

3. If the smallest is W_{west}, then let the next vertex in the backtrack list be the vertex west of (i,j), i.e. $(i-1,j)$. Similarly, if the smallest is $W_{northwest}$, the next vertex in the list will be $(i-1,j-1)$, and if the smallest is W_{north}, then the next vertex will be $(i,j-1)$.

4. Let this new vertex on the backtrack list be the new (i,j), and repeat from step 2 until $i = j = 0$.

In fact, our method does not guarantee the overall least work path, since backtracking to the previous vertex of minimum work is only a local minimization. However, our approximation to the least work path is quite satisfactory, as it produces results which are quite good. (See [7] for further details of the backtracking algorithm.)

2.3 Results

The equations of the Least Work Matching algorithm requires quite a bit of user input. A user must decide on the first vertex correspondence, set the seven constants associated with bending and stretching work, and pre-process the images to ensure an appropriate first vertex matching and a reasonable distribution of vertices around the polygons.

Consider the "m" and "n" polygons shown in Fig. 2.16. The vertices occur only at the obvious places (there are no "hidden" vertices along the interior of a straight edge). The "m" was blended into the "n" using the following parameters: $k_{bend} = 2$, $m_{bend} = 100$, $e_{bend} = 1$, $p_{bend} = 10000$, $k_{stretch} = 0.1$, $c_{stretch} = 0.1$, and $e_{stretch} = 2$, and starting vertices P_0^0 and P_0^1 are as shown in Fig. 2.16.

First, consider the blend if we use a match-by-order approach (in which the vertices are matched up based on the order in which they occur, with left-over vertices of one polygon simply mapping to the last vertex of the polygon with fewer vertices). The resulting blend is given in Fig. 2.17.

The result of using Least Work Matching on the polygons is shown in Fig. 2.18.

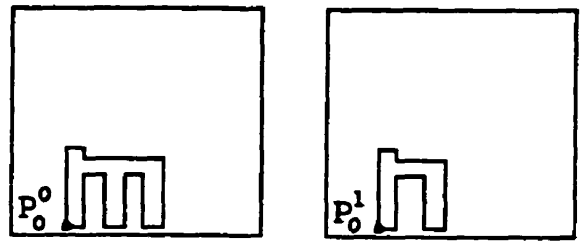


Fig. 2.16 – "m" and "n" polygons

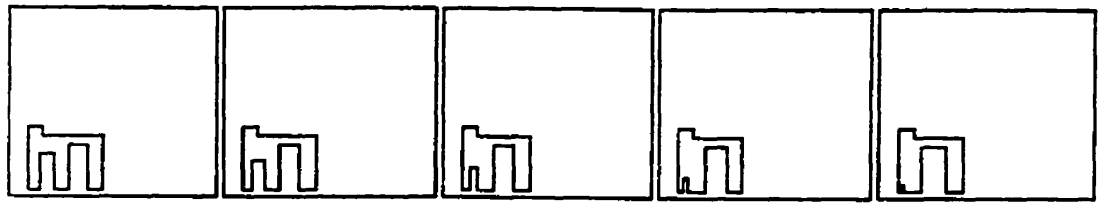


Fig. 2.17 – Match-by-order without pre-processing

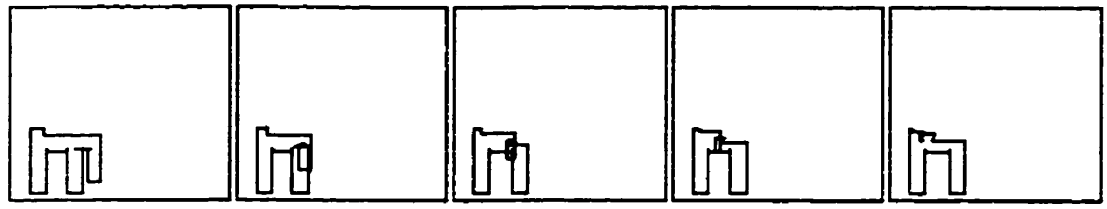


Fig. 2.18 – Least Work Matching without pre-processing

Clearly, Least Work Matching gives an even less appealing blend (with global self-intersection) than the blend in Fig. 2.17.

However, in the next blends, some pre-processing has been applied to the "n", in the form of adding two additional vertices, as shown in Fig. 2.19.

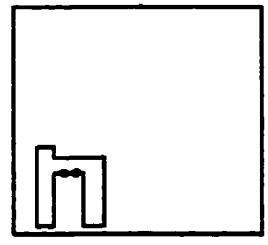


Fig. 2.19 – Additional vertices added to "n"

The blend which used match-by-order is given in Fig. 2.20, and the blend which used Least Work Matching is given in Fig. 2.21. (Both use the same parameters as the blends in Figs. 2.17 and 2.18). Least Work Matching yields a very elegant blend, unlike that of Fig. 2.20. Clearly, pre-processing can be a very important step in shape blending.

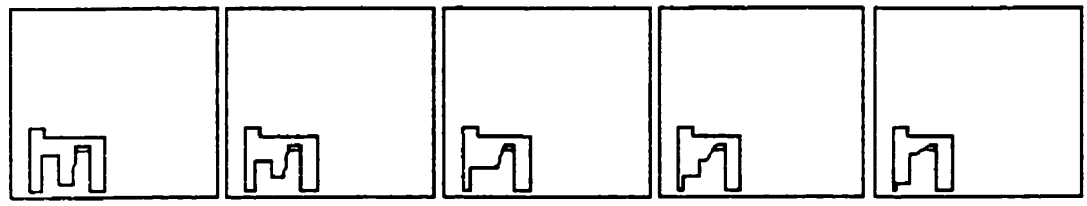


Fig. 2.20 – Match-by-order with pre-processing

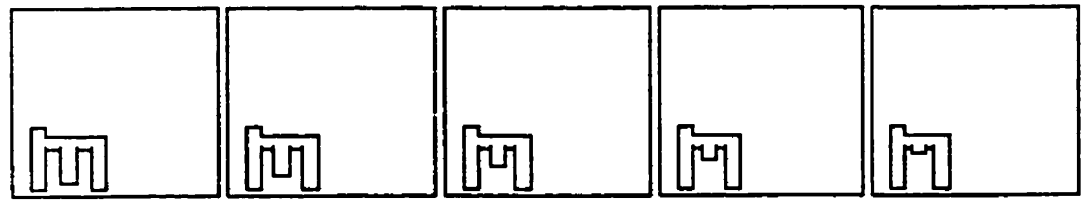


Fig. 2.21 – Least Work Matching with pre-processing

Ideally, one should be able to find a choice of parameters for which Least Work Matching would provide the sort of blend given in Fig. 2.20, but without pre-processing. I was unable to find such a parameter set. However, Fig. 2.23 gives a good blend of an "E" to an "F" (shown in Fig. 2.22), without any vertices added along the straight edges. The general idea of the "E" to "F" blend (the extra "limb" shrinking away) is the same as that of the "m" to "n" blend. The parameters used here are $k_{bend} = 0.5$, $m_{bend} = 1$, $e_{bend} = 1$, $p_{bend} = 10000$, $k_{stretch} = 0.1$, $c_{stretch} = 0.1$, and $e_{stretch} = 2$, with starting vertices P_0^0 and P_0^1 as shown in Fig. 2.22.

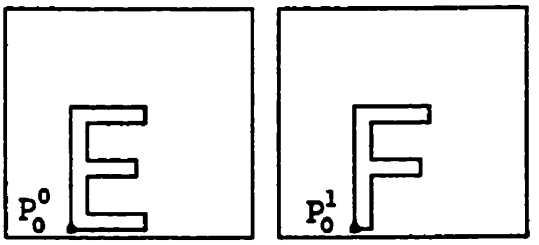


Fig. 2.22 – "E" and "F" polygons

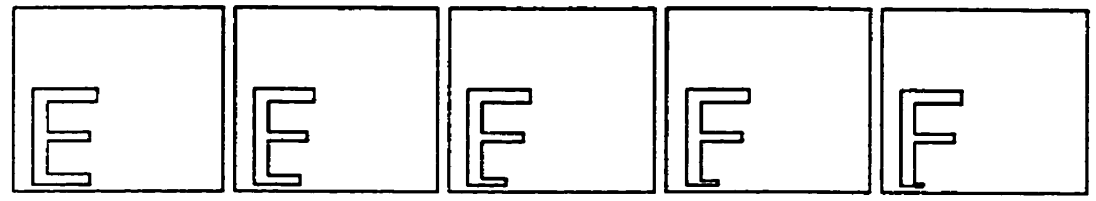


Fig. 2.23 – Least Work Matching without pre-processing

Clearly, choice of parameters is very important, and user intervention is necessary. A variety of good, but different, blends (as well as a variety of bad blends) can be achieved, depending on the choice of parameters. The choice of the first vertex correspondence is also extremely important; an example which demonstrates this is given in Chapter 4. Often only a human being can decide how much relative bending or stretching is desired for a particular blend, or which vertices should be chosen as starting points on the polygons.

Chapter 3: Intrinsic Interpolation

3.1 Introduction

An important aspect of 2D shape interpolation concerns the path along which each vertex must travel to arrive at its final destination. This is known as the vertex path problem.

One approach is to have each vertex follow a linear path. Although this method is simple to understand and to implement, it often leads to unappealing results. More often than not, in the physical world around us, points in motion do not follow a linear path. Linear interpolation causes all points in the first key image to follow straight line paths to their corresponding points in the second key image, creating unrealistic-looking approximations of motion. A classic example of the failure of the linear path is the withering limb, shown in Fig. 3.1 as a swinging pendulum. (A super-imposed version is given in Fig. 3.2. Here, it is much clearer that the pendulum is following a linear path). A pendulum

outstretched horizontally in one key frame and vertically in the other key frame will not retain its length in the in-between frames if linear interpolation is used. Clearly, more realistic vertex path methods must be found.

This "withering limb" problem is but one of many that can arise when performing a blend between two shapes. Some others include self-intersection, the loss of similar features in the in-between stages, and non-monotonically changing angles. These problems can produce in-between images which are visually displeasing and physically inaccurate.

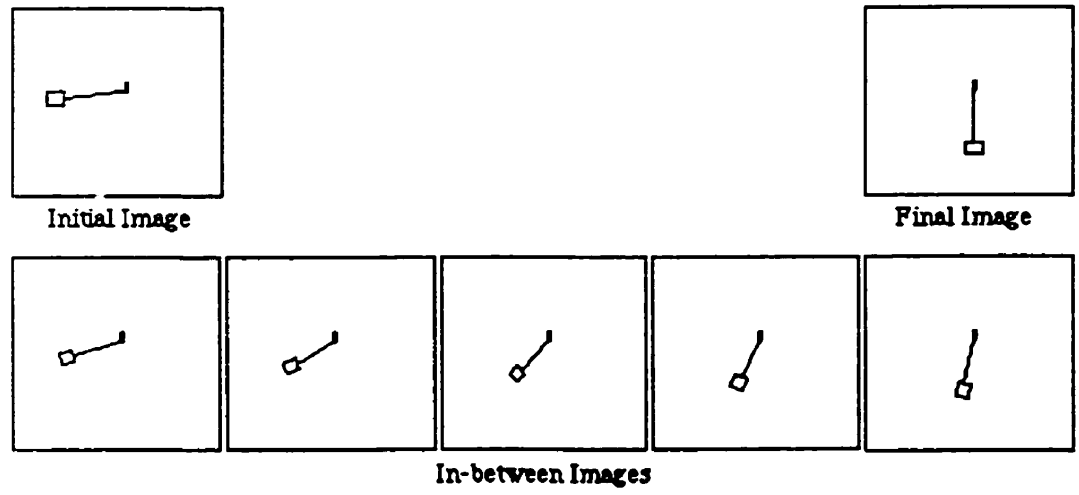


Fig. 3.1 – Withering limb

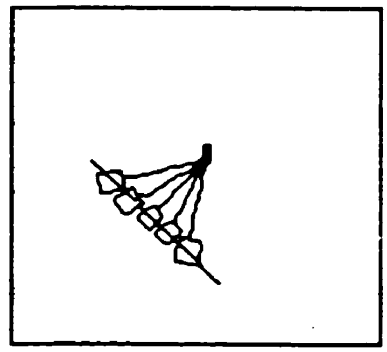


Fig. 3.2 – Super-imposed withering limb

An alternative solution to the vertex path problem, Intrinsic Interpolation, is given in [2]. In order to solve the vertex path problem, an appropriate vertex correspondence must first be found. The authors of [2] used the Least Work Matching solution to the vertex correspondence problem (given in [1], and discussed in Chapter 2 of this thesis).

The general idea behind Intrinsic Interpolation is as follows: each of the two key polygons is described intrinsically (that is, in terms of the edge lengths and the angles formed by each pair of adjacent edges), and interpolation between the values of these intrinsic features is performed to calculate the in-between polygons.

3.2 Development

Let the two key polygons be P^0, and P^1, each with $m+1$ vertices (0 through m). This assumption is valid, since, after Least Work Matching is performed, the two polygons can be considered to have the same number of vertices. (For example, if two different vertices of P^0 map to the same vertex of P^1, then that vertex of P^1 is considered to be two different but coincident vertices.) Let the lengths of the edges of P^0 and P^1 be denoted by L_i^0 and L_i^1 respectively, where

$$L_i^k = \left|P_{i+1}^k - P_i^k\right|, \text{ for } i = 0,\ldots,m. \tag{3.1}$$

Furthermore, we will define θ_i^k, (k = 0, 1) to be the angle formed by extending edge $\overline{P_{i-1}^k P_i^k}$ and calculating the directional angle between edge $\overline{P_i^k P_{i+1}^k}$ and this extension, as shown in Figs. 3.3a and 3.3b.

If the angle ψ_i^k, measured counter-clockwise from edge $\overline{P_{i-1}^k P_i^k}$ to edge $\overline{P_i^k P_{i+1}^k}$ is less than π radians, then define θ_i^k as

$$\theta_i^k = \pi - \psi_i^k, \tag{3.2a}$$

and otherwise, define θ_i^k as

$$\theta_i^k = -(\pi - \psi_i^k). \tag{3.2b}$$

As will soon be evident, the θ_i^k values are necessary to calculate the relative positions of the vertices of the in-between polygons.

Note that if the vertices coincide (i.e. have multiplicity greater than 1) they are handled in the same manner as described in Chapter 2 (see section 2.2.3.4).

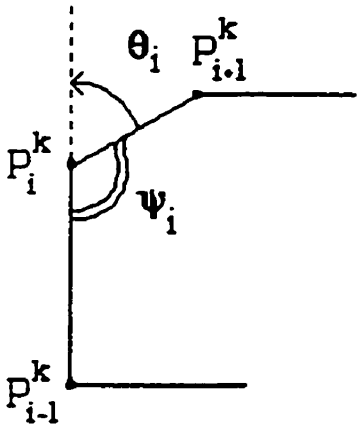


Fig. 3.3a

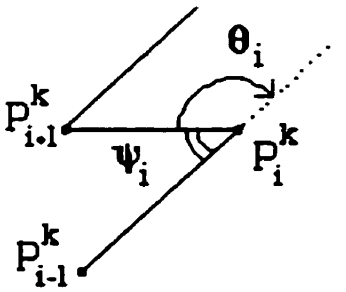


Fig. 3.3b

Fig. 3.3 – Definition of θ_i

Since we are using intrinsic definitions for the polygons, we do not have a description of the exact physical location of their vertices and edges. Hence, when we interpolate between the shapes, we must have an anchor point in each key shape whose interpolated position can specify the shape's translation throughout the blend, and a baseline, defined in relation to the anchor point, that specifies the shape's rotation during the blend. The anchor point is taken to be the first vertex of each polygon. These initial vertices must be chosen carefully; an inappropriate first-vertex correspondence can cause the polygon to follow a curious path through the blend. User-intervention may be required to ensure this.

The baseline is defined as a horizontal line through the anchor point (x_0, y_0) (see Fig. 3.4). The angle that edge $\overline{P_0^k P_1^k}$ makes with the angle line is denoted α_0^k.

Each edge $\overline{P_i^k P_{i+1}^k}$ makes some angle, α_i^k, with the horizontal (the baseline). These α_i^k values can be computed using the previous angle α_{i-1}^k in conjunction with θ_i^k (see Fig. 3.5):

$$\alpha_i^k = \alpha_{i-1}^k - \theta_i^k. \tag{3.3}$$

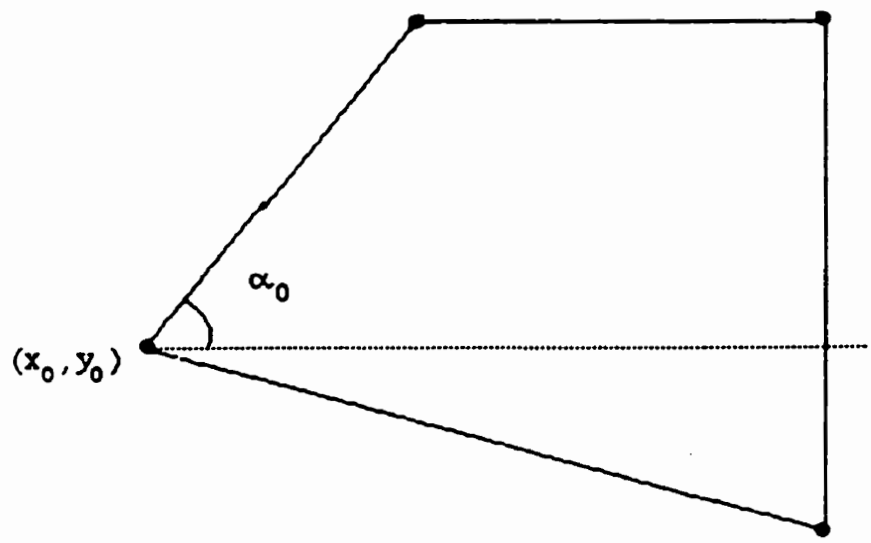


Fig. 3.4 – Anchor point (x_0, y_0) and anchor angle α_0

We need to know these values, α_i^k, in order to compute the position of each vertex. From the first vertex, P_i^k, of the ith edge, move a distance L_i^k at angle α_i^k to the second vertex, P_{i+1}^k, of the ith edge, which is the first vertex of the next edge.

Therefore, the x- and y-direction of the coordinates (x_{i+1}, y_{i+1}) of vertex P_{i+1}^k, relative to the coordinates (x_i, y_i) of the previous vertex P_i^k, can be calculated as $\sin\alpha_{i-1}$ and $\cos\alpha_{i-1}$, respectively.

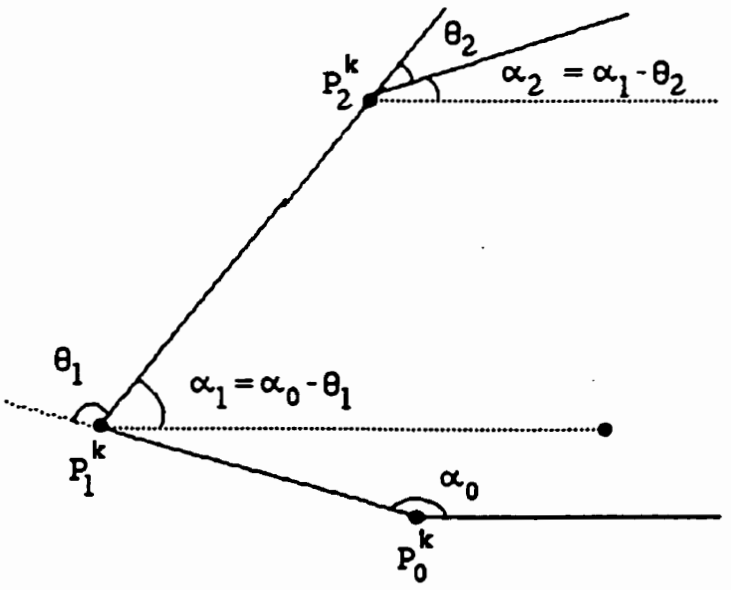


Fig. 3.5 – Definition of α_i^k

The amount to proceed in each of the x- and y- directions is simply found by multiplying $\sin\alpha_{i-1}$ and $\cos\alpha_{i-1}$ by the length of the edge L_{i-1} between P_{i-1} and P_i.

To determine the vertices, $P_0, P_1, ..., P_m$ of an intermediate polygon, the lengths of the edges and the angles formed by each pair of adjacent edges will be interpolated:

$$L_i = (1-t)L_i^0 + tL_i^1 \tag{3.4}$$

$$\alpha_i = (1-t)\alpha_i^0 + t\alpha_i^1 \tag{3.5}$$

$$\theta_i = (1-t)\theta_i^0 + t\theta_i^1 \tag{3.6}$$

for $i = 0,...,m$.

To obtain the vertex P_0 of the interpolated polygons, linear interpolation between P_0^0 and P_0^1 is used.

To calculate the position of coordinates (x_i, y_i) of vertex P_i of an intermediate polygon, the coordinates (x_{i-1}, y_{i-1}) of the previous vertex P_{i-1}, the interpolated edge length L_{i-1} of the edge between $\overline{P_{i-1}P_i}$, and the interpolated angle θ_{i-1} between this edge and the previous edge (equation 3.2) must all be known.

Therefore,

$$x_i = x_{i-1} + L_{i-1}\cos\alpha_{i-1}, \tag{3.7a}$$

and

$$y_i = y_{i-1} + L_{i-1}\sin\alpha_{i-1}. \tag{3.7b}$$

The Intrinsic Interpolation method offers a significant improvement over linear interpolation, as is shown in section 3.4. In fact, the images would be quite satisfactory if not for the fact that the in-between polygons do not typically close (for an example, see Figs. 3.8 and 3.9 of section 3.4). Therefore, the problem now becomes one of forcing the intermediate images to close.

3.3 Edge Tweaking

One solution to this problem is to slightly change the lengths of the edges of the intermediate polygons. In order to do this, we change the edge length interpolation equation (equation 3.4) to:

$$L_i = (1-t)L_i^0 + tL_i^1 + S_i \ , \quad i = 0,...,m, \tag{3.8}$$

where S_i is some small amount added to edge i. Now the trouble lies in determining S_i.

Since it is generally desirable to have the lengths of a given edge change gradually from the first key polygon to the second, the values of S_i should be fairly small relative to the difference in edge length from P^0 to P^1 (i.e. small relative to $\left|L_i^0 - L_i^1\right|$). That is, we want to fit lengths S_i into the polygon such that

the S_i are as small as possible, but yet proportional to the length of edge i, and such that the polygon will close. Thus, using least squares, we want to find such values of S_i such that

$$f(S_o, S_1, \ldots, S_m) = \sum_{i=0}^{m} \left(\frac{S_i}{|L_i^0 - L_i^1|} \right)^2 \tag{3.9}$$

is minimized.

In the event that L_i^0 and L_i^1 are the same length, the function $f(S_0, S_1, \ldots, S_m)$ would contain some elements in which division by zero would occur. Therefore, define

$$L_{small} = 0.0001 \times \left(\max_{i \in [0,m]} |L_i^0 - L_i^1| \right), \tag{3.10}$$

and then, to avoid division by zero,

$$L_i^{01} = \max\left\{ |L_i^0 - L_i^1|, L_{small} \right\}, \quad i = 0, \ldots, m. \tag{3.11}$$

Hence,

$$f(S_0, S_1, \ldots, S_m) = \sum_{i=0}^{m} \left(\frac{S_i}{L_i^{01}} \right)^2. \tag{3.12}$$

To ensure that the values of S_i will, in fact, cause the last vertex of the polygon to be equal to the first vertex of the polygon, the following constraints are imposed:

$$\varphi_1 = \sum_{i=0}^{m}[(1-t)L_i^0 + tL_i^1 + S_i]\cos\alpha_i = 0, \tag{3.13a}$$

and

$$\varphi_2 = \sum_{i=0}^{m}[(1-t)L_i^0 + tL_i^1 + S_i]\sin\alpha_i = 0. \tag{3.13b}$$

To find the values of S_i that satisfy f, φ_1, and φ_2 simultaneously, Lagrange multipliers are used.

Let

$$\Phi = f + \lambda_1\varphi_1 + \lambda_2\varphi_2, \tag{3.14}$$

where λ_1 and λ_2 are the multipliers, and Φ is a function of λ_1, λ_2, S_0, S_1,..., and S_m.

Differentiating Φ with respect to each S_i and setting each $\frac{\partial\Phi}{\partial S_i}$ equal to 0 yields $m+1$ equations of the form

$$\frac{\partial \Phi}{\partial S_i} = \frac{2S_i}{(L_i^{01})^2} + \lambda_1 \cos\alpha_i + \lambda_2 \sin\alpha_i = 0, \tag{3.15}$$

for $i = 0,...,m$, subject to $\varphi_1 = 0$ and $\varphi_2 = 0$.

Multiplying each $\frac{\partial \Phi}{\partial S_i}$ by $\left(L_i^{01}\right)^2$, and rearranging gives equations of the form

$$\lambda_1 \left(L_i^{01}\right)^2 \cos\alpha_i + \lambda_2 \left(L_i^{01}\right)^2 \sin\alpha_i = -2S_i. \tag{3.16}$$

We can create two new sets of equations by multiplying the set of equations 3.16 by $\sin\alpha_i$ and by multiplying the set of equations 3.16 by $\cos\alpha_i$. Doing so, and then summing each set of equations, gives

$$\lambda_1 \sum_{i=0}^{m} \left(L_i^{01}\right)^2 \cos^2\alpha_i + \lambda_2 \sum_{i=0}^{m} \left(L_i^{01}\right)^2 \cos\alpha_i \sin\alpha_i = -2\sum_{i=0}^{m} S_i \cos\alpha_i, \tag{3.17a}$$

and

$$\lambda_1 \sum_{i=0}^{m} \left(L_i^{01}\right)^2 \cos\alpha_i \sin\alpha_i + \lambda_2 \sum_{i=0}^{m} \left(L_i^{01}\right)^2 \sin^2\alpha_i = -2\sum_{i=0}^{m} S_i \sin\alpha_i. \tag{3.17b}$$

Rearranging the constraint equations $\varphi_1 = 0$ and $\varphi_2 = 0$ gives

$$\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\cos\alpha_i = -\sum_{i=0}^{m} S_i \cos\alpha_i\,, \tag{3.18a}$$

and

$$\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\sin\alpha_i = -\sum_{i=0}^{m} S_i \sin\alpha_i\,. \tag{3.18b}$$

Replacing the right-hand side of equations 3.17a and 3.17b with the left-hand side of equations 3.18a and 3.18b, respectively, yields two equations in two unknows, λ_1 and λ_2:

$$\lambda_1\sum_{i=0}^{m}\left(L_i^{01}\right)^2\cos^2\alpha_i + \lambda_2\sum_{i=0}^{m}\left(L_i^{01}\right)^2\cos\alpha_i\sin\alpha_i = 2\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\cos\alpha_i\,, \tag{3.19a}$$

and

$$\lambda_1\sum_{i=0}^{m}\left(L_i^{01}\right)^2\cos\alpha_i\sin\alpha_i + \lambda_2\sum_{i=0}^{m}\left(L_i^{01}\right)^2\sin^2\alpha_i = 2\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\sin\alpha_i\,. \tag{3.19b}$$

We can solve for λ_1 and λ_2 using Cramer's Rule:

$$\lambda_1 = \frac{\begin{vmatrix} 2\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\cos\alpha_i & \sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin\alpha_i\cos\alpha_i \\ 2\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\sin\alpha_i & \sum_{i-0}^{m}\left(L_i^{01}\right)^2 \sin^2\alpha_i \end{vmatrix}}{\begin{vmatrix} \sum_{i-0}^{m}\left(L_i^{01}\right)^2 \cos^2\alpha_i & \sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin\alpha_i\cos\alpha_i \\ \sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin\alpha_i\cos\alpha_i & \sum_{i-0}^{m}\left(L_i^{01}\right)^2 \sin^2\alpha_i \end{vmatrix}}, \tag{3.20a}$$

and

$$\lambda_2 = \frac{\begin{vmatrix} \sum_{i-0}^{m}\left(L_i^{01}\right)^2 \cos^2\alpha_i & 2\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\cos\alpha_i \\ \sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin\alpha_i\cos\alpha_i & 2\sum_{i=0}^{m}\left[(1-t)L_i^0 + tL_i^1\right]\sin\alpha_i \end{vmatrix}}{\begin{vmatrix} \sum_{i-0}^{m}\left(L_i^{01}\right)^2 \cos^2\alpha_i & \sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin\alpha_i\cos\alpha_i \\ \sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin\alpha_i\cos\alpha_i & \sum_{i-0}^{m}\left(L_i^{01}\right)^2 \sin^2\alpha_i \end{vmatrix}}, \tag{3.20b}$$

given that

$$\left(\sum_{i=0}^{m}\left(L_i^{01}\right)^2 \cos^2\alpha_i\right)\left(\sum_{i=0}^{m}\left(L_i^{01}\right)^2 \sin^2\alpha_i\right) \neq \left(\sum_{i=0}^{m}\left(L_i^{01}\right)^2 \cos\alpha_i\sin\alpha_i\right)^2.$$

Once λ_1 and λ_2 are found, the equations 3.16 can be used to solve for S_i:

$$S_i = -\frac{1}{2}\left(L_i^{01}\right)^2(\lambda_1 \cos\alpha_i + \lambda_2 \sin\alpha_i), \tag{3.21}$$

for $i = 0,...,m$.

Now that the values of S_i are known, equation 3.8 can be used to calculate the edge-lengths, L_i, and, as before, equations 3.7a and 3.7b can be used to calculate the vertices of the in-between polygons.

3.4 Results

Intrinsic Interpolation was applied to the pendulum of Fig. 3.1, with results given in Fig. 3.6. Fig. 3.7 shows an image with the five in-between frames super-imposed on one another. The pendulum follows a circular path, as we would expect of a real pendulum.

Edge tweaking works well. Fig. 3.9 gives an example of the polygons of Fig. 3.8, blended using intrinsic interpolation, in which the in-between polygons do not close. When edge tweaking is applied, the in-between polygons close nicely (Fig. 3.10).

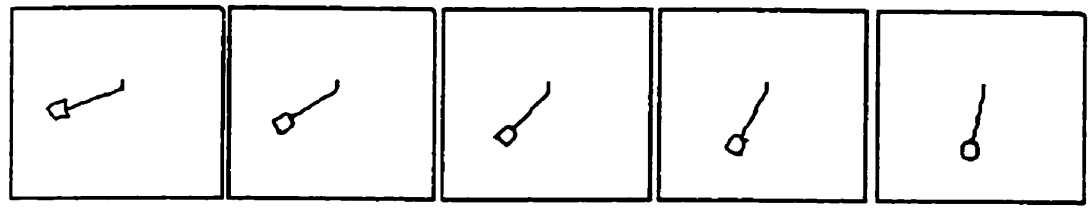


Fig. 3.6 – Intrinsic Interpolation applied to a pendulum

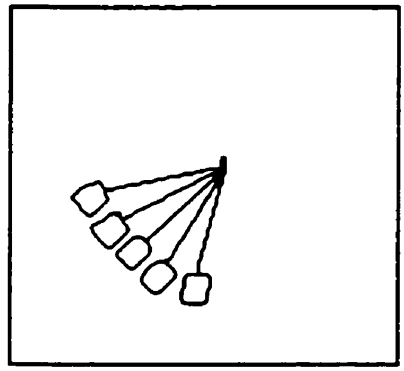


Fig. 3.7 – Super-imposed pendulum

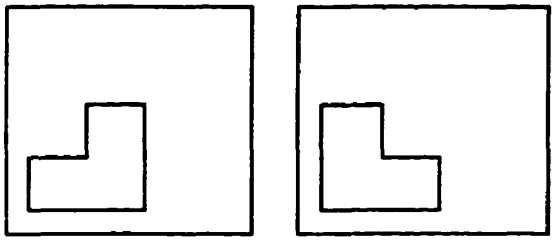


Fig. 3.8 – Polygons to be blended with Intrinsic Interpolation

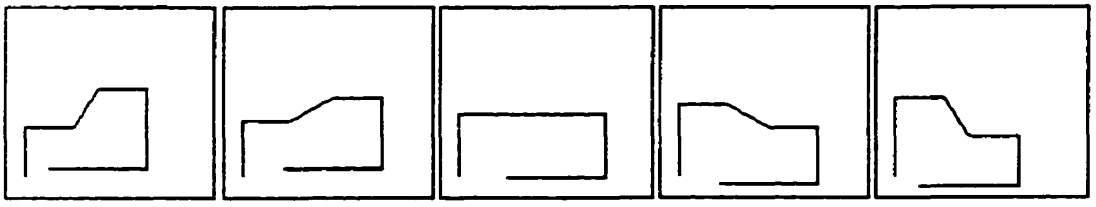


Fig. 3.9 – Intrinsic Interpolation without Edge Tweaking

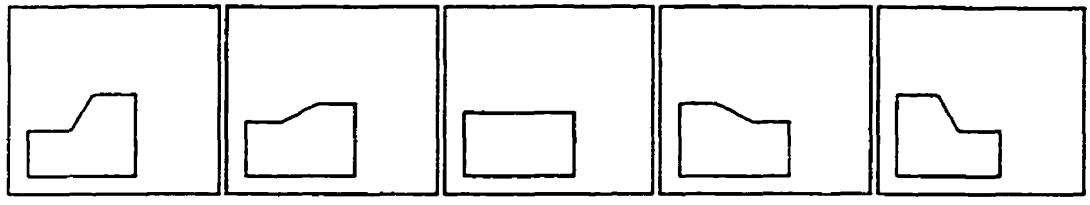


Fig. 3.10 – Intrinsic Interpolation with Edge Tweaking

The following two examples use the Least Work Matching and Intrinsic Interpolation with Edge Tweaking algorithms. Fig. 3.11 is a blend of the "m" and "n" polygons of Fig. 2.12 (without pre-processing), and Fig. 3.12 blends the "m" polygon of Fig. 2.12 and the "n" polygon of Fig. 2.15 (the "n" is pre-processed). The same parameters were used here as were used in the "m" to "n" blends of Chapter 2. Fig. 3.11 is a little odd, but Fig. 3.12 shrinks the extra "limb" even more elegantly than the blend of Fig. 2.17.

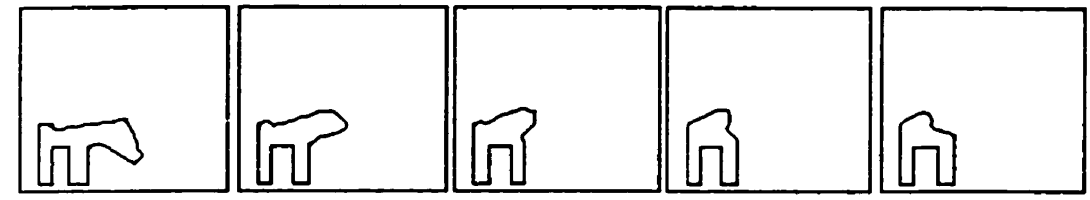


Fig. 3.11 – Intrinsic Interpolation with Edge Tweaking, no pre-processing

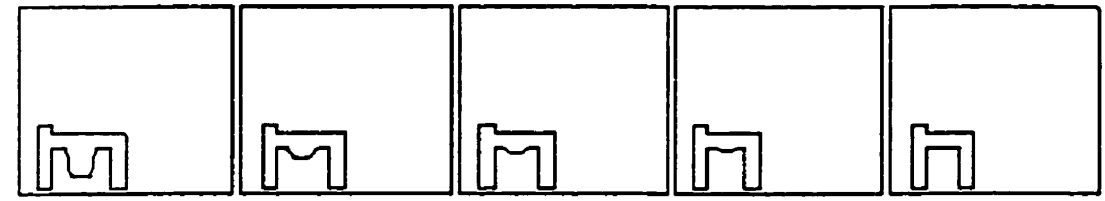


Fig. 3.12 – Intrinsic Interpolation with Edge Tweaking, with pre-processing

As an additional example, Intrinsic Interpolation with Edge Tweaking is applied to the "E" to "F" blend of Chapter 2 (Fig. 2.18), with good results (see Fig. 3.13). The limb disappears more quickly than the blend given in Fig. 2.19.

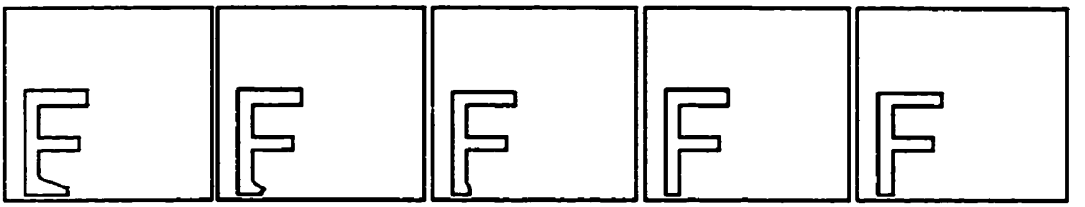


Fig. 3.13 – "E" to "F" using Intrinsic Interpolation with Edge Tweaking

Although in the examples here edge-tweaking produced good blends, the possibility may exist that the edge tweaking algorithm may produce some values $|S_i|$ that are too large to appear appealing in the blend.

Chapter 4: Curves

4.1 Introduction

So far, we have looked only at the blending of polygons. We will now turn our attention to adapting the previously discussed methods for use in the blending of curves.

Like the polygons described in Chapters 2 and 3, we will think of the curves as being made out of pieces of wire that can be bent or stretched, and we will attempt to bend and stretch the wires of the first curve into the shape of the second curve.

The curves used in the blending algorithm will be constructed from cubic Bézier curves. Cubic Bézier curves use polynomial curve segments which are guided by four control points $\mathbf{q}_0$, $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}_3$, and are defined by the equation

$$Q(t) = (1-t)^3\mathbf{q}_0 + 3t(1-t)^2\mathbf{q}_1 + 3t^2(1-t)\mathbf{q}_2 + t^3\mathbf{q}_3. \qquad (4.1)$$

The function associated with each control point is known as a blending function. The use of the word "blending" in the term "blending function" is unrelated to the blending of 2-dimensional shapes.

Bézier curves interpolate (pass through) the first and last control points (in the cubic case, $\mathbf{q}_0$ at $t = 0$, and $\mathbf{q}_3$ at $t = 1$), and have the property that the line through $\mathbf{q}_0$ and $\mathbf{q}_1$ is the tangent line to the curve at the point $\mathbf{q}_0$, and the line through $\mathbf{q}_2$ and $\mathbf{q}_3$ is tangent to the curve at the point $\mathbf{q}_3$.

The curves to be blended are defined as a list of the control points of the Bézier curves, whereby the last control point of one Bézier curve is the first control point of the next Bézier curve. The whole curve will therefore pass through the first control point, and every third control point thereafter.

In the discussion that follows, the phrase "curve segment" will refer to the portion of the Bézier curve defined by four control points, and the term "curve" will refer to the continuous curve formed by joining these segments. Several restrictions are placed on the curves for our purposes.

First of all, we restrict the curves to have no points of inflection in each segment. That is, the points of inflection must occur at the join points of the Bézier curves.

This restriction is introduced to aid in the calculation of bending work (section 4.2.2). Should a Bézier curve segment contain an inflection point, it can easily be found by solving

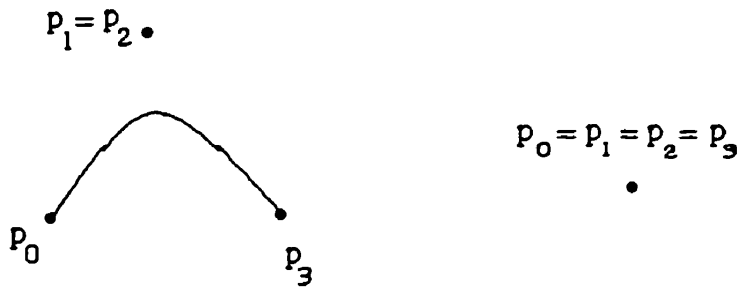
$$Q'(t) = 6(1-t)\mathbf{q}_0 - 6(2-3t)\mathbf{q}_1 + 6(1-3t)\mathbf{q}_2 + 6t\mathbf{q}_3 = 0$$

for t. The curve segment in question can then be subdivided into two Bézier curve segments at the inflection point.

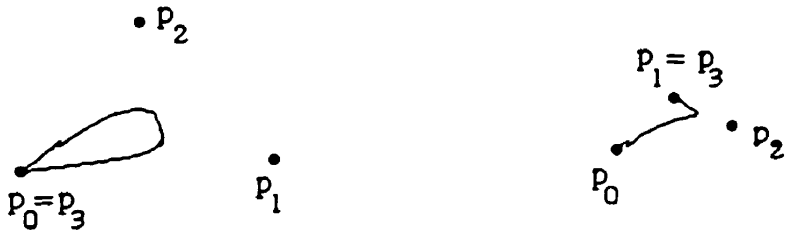
Furthermore, although we allow repeated control points, they must be adjacent to one another in the ordered list of control points. That is, the curve cannot cross back through itself, nor can the first and last control point of a segment be the same, unless the two interior control points are also the same as these first and last control points (see Fig. 4.1). Thus, we must also assume that there are at least two Bézier curve segments in our joined-together, closed curve.

Lastly, assume that the curvature of each segment is small enough and the length of the segment short enough that the angle formed by the intersection of the outward pointing normal lines at the endpoints of each segment are less than π radians (see Fig. 4.2).

For a more thorough treatment of Bézier curve, see [4], [5], et al.



4.1a – Allowed curves



4.1b – Disallowed curves

Fig. 4.1 – Bézier curves that are and are not allowed

Allowed Disallowed

Fig. 4.2 – Curve segments that are and are not allowed

4.2 Curve Blending via the Control Polygon

The simplest way to compute a blend between two curves is to look at the control polygon of the curve. (The control polygon is simply the polygon whose vertices are the control points of the strung-together Bézier curves.) The Least Work Matching algorithm discussed in Chapter 2 can be applied to the control polygon to find a control point correspondence, and then either linear interpolation or Intrinsic Interpolation may be applied. In-between Bézier curves are drawn based on these in-between control polygons.

A simple example in which control polygon blending works well is given below, by the two leaves to be blended (given in Fig. 4.3, and blended in Fig. 4.4). In

fact, this blend by the simple control polygon method produces an identical blend to one produced by the more complicated method of Least Work Curve Matching, discussed in section 4.3.

The parameters used for the control polygon method are $k_{bend} = 0.1$, $m_{bend} = 100$, $e_{bend} = 1$, $p_{bend} = 10000$, $k_{stretch} = 2$, $c_{stretch} = 0.1$, and $e_{stretch} = 2$. The parameters used for the Least Work Curve Matching method of section 4.3 are $C_{bend} = 1$, $C_{kink} = 0.1$, $E_{kink} = 1$, $K_{stretch} = 2$, $C_{stretch} = 0.1$, and $E_{stretch} = 2$. The starting vertices are as shown in the figure.

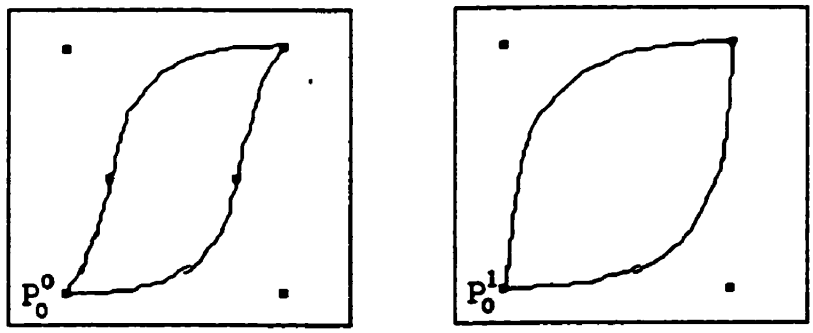


Fig. 4.3 – Two leaves to be blended

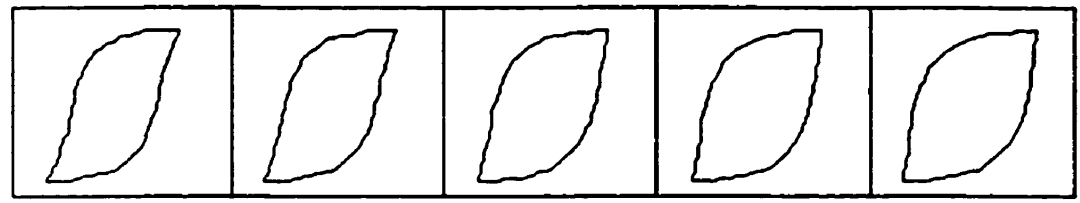


Fig. 4.4 – Blend using the control polygon

One obvious problem with the control polygon method is that entire Bézier curve segments of shape 0 may not be matched to entire Bézier curve segments of shape 1 (see Fig. 4.5). Inserting the additional control points required by this scenario will cause changes to the original curves before any blending even begins (see Fig. 4.6). The algorithm is oblivious to the changes it is causing in the curve, since it is dealing solely with the control polygon.

To deal with this problem, one could draw a pseudo-control polygon based only on the control points through which the curve passes (i.e. the first point of the strung-together curve, and every third point thereafter), and apply the Least Work Matching algorithm to this pared-down control polygon. However, the paring-down would provide only a very rough linear approximation to the curve, and would, in general, significantly reduce the accuracy of the work calculations.

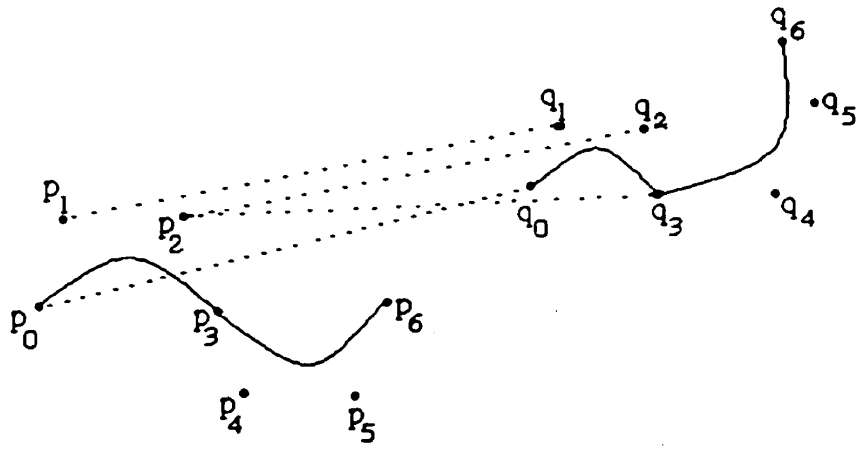


Fig. 4.5 – p_0, p_1, p_2, and p_2 may match to q_0, q_1, q_2, and q_3

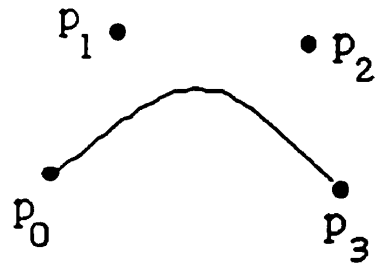


Fig. 4.6a – Original Bézier curve

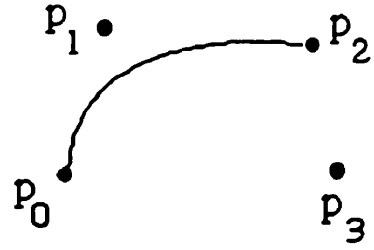


Fig. 4.6b – Bézier Curve when p_2 has multiplicity 2

Fig. 4.6 – Inserted control points can cause unwanted changes in the curve

A better idea would be to use only the control points through which the curve passes in the matching, but instead of approximating the curve segment by straight lines to calculate work, use the interior control points to determine the actual Bézier curve between the interpolated control points, and use these curves in the work calculations. This method is discussed in section 4.3.

4.3 Least Work Curve Matching

As with polygon blending, we must find a correspondence between the two key curves, and then determine the path along which the corresponding points of the curve will follow. The correspondence between the two key curves will be based on the interpolated (end) control points of each Bézier curve segment. That is, instead of matching vertices of the key polygons (as in Chapter 2), we will match the interpolated control points. The correspondence algorithm discussed here is similar in nature to the Least Work Matching algorithm for vertex correspondence of [1], discussed in Chapter 2 of this thesis.

Some quantity of work is required to transform one curve into another. The amount of work involved in blending a particular pair of curves will vary depending on the control point correspondence. Since the blend requiring the least amount of work is typically the most visually pleasing, we wish to find the control point correspondence that involves the least amount of work.

Before we can proceed with the control point correspondence, we first must describe the way in which work will be calculated.

4.3.1 Work

For two-dimensional shape blending, we concern ourselves with three sorts of work: stretching work, bending work, and kinking work.

4.3.1.1 Stretching Work

As in Chapter 2 (equation 2.8), the work required to stretch a wire of length L_0 into a wire of length L_1 is

$$W_{stretch} = k_{stretch} \frac{(L_1 - L_0)^{e_{stretch}}}{(1 - c_{stretch}) \min(L_0, L_1) + c_{stretch} \max(L_0, L_1)}, \tag{4.2}$$

where the length of the parametric Bézier curve segment is given by:

$$\int_0^1 \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\, dt.$$

(For a discussion of the equation and a description of each of the constants $k_{stretch}$, $e_{stretch}$ and $c_{stretch}$, see section 2.2.2.)

4.3.1.2 Bending Work

Bending work is the work required to elastically bend a curve segment. Bending work for a segment of the curve is based on the change of interior angles, ψ_i, formed by the intersection of the normal lines to the endpoints of the curve (see Fig. 4.7).

The computation of this quantity is straightforward since the control points of Bézier curves, by definition, create tangent lines to the endpoints. Knowing these tangent lines allows for easy computation of the normal lines. Since we assume that the degree of curvature of each curve segment is small, calculating ψ requires finding the point of intersection, $\mathbf{p}_{int}$, of these two normal lines, and then computing the angle $\angle[\mathbf{p}_0, \mathbf{p}_{int}, \mathbf{p}_3]$.

Bending moment is a measure of the resistance to bending of a wire. The bending moment applied to each end of the wire, M_{bend}, is defined by

$$M_{bend} = \frac{EI}{\rho},$$

where E is the modulus of elasticity of a material, I is the moment of inertia, and ρ is the radius of curvature.

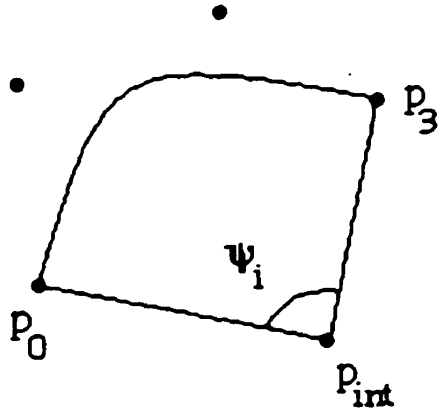


Fig. 4.7 – Calculating angles for bending work

Then the work required to bend a straight wire of length L into a circular arc of curvature κ, where $\kappa = \frac{1}{\rho}$, is

$$\begin{aligned} W_{bend} &= \int M_{bend} d\psi \\ &= M_{bend}\psi, \end{aligned} \tag{4.3}$$

where ψ is as given in Fig. 4.7.

To render this equation suitable for use, make the following substitutions:

$$\begin{aligned} W_{bend} &= M_{bend}\psi \\ &= \frac{EI\psi}{\rho} \\ &= (2EI)\frac{\psi^2}{2\rho\psi}. \end{aligned} \tag{4.4}$$

Since our wires have no physical properties, the user can choose E and I to suit her needs. Therefore, let $C_{bend} = 2EI$ be a user-defined constant.

Since we may be bending a curved wire into a curved wire (instead of bending a straight wire into a curved wire), we replace ψ in equation 4.3 by the difference between ψ_1 and ψ_2 (where ψ_1 and ψ_2 are the angles from key curves 1 and 2, respectively).

For ease of computation (and since we do not, for our purposes, require exact work value computations, but rather approximations of work values), we choose to approximate the curve by a circular arc when computing bending work. The length, L, of a circular arc is simply the product of the radius of curvature and the angle ψ. Therefore, the quantity $2\rho\psi$ is simply $2L$. Since the initial and final lengths of the wire may not be equal, and since we wish our work equation

to be representative of the arc lengths from both key curves, replace this quantity by $L_0 + L_1$.

Thus, the work equation for bending is

$$W_{bend} = C_{bend} \frac{(\psi_1 - \psi_0)^2}{L_0 + L_1}. \tag{4.5}$$

4.3.1.3 Kinking Work

If the moment of the wire exceeds the elastic limit, plastic bending (kinking) occurs in the wire.

We consider this sort of bending to occur only at the join points of the Bézier curve segments. If we define θ to be the angle between the two normal lines to a join point (see Fig. 4.8), we can view kinking as similar in nature to the bending at polygon vertices discussed in Chapter 2 of this thesis.

We therefore let kinking work be defined in a manner similar to the work of equation 2.9:

$$W_{kink} = K_{kink} |\theta_1 - \theta_2|^{E_{limit}}, \tag{4.6}$$

where θ_1 and θ_2 are the angles of key curves 1 and 2, respectively, K_{kink} is a user-defined kinking stiffness parameter, and E_{kink}, as usual, is an elasticity constant.

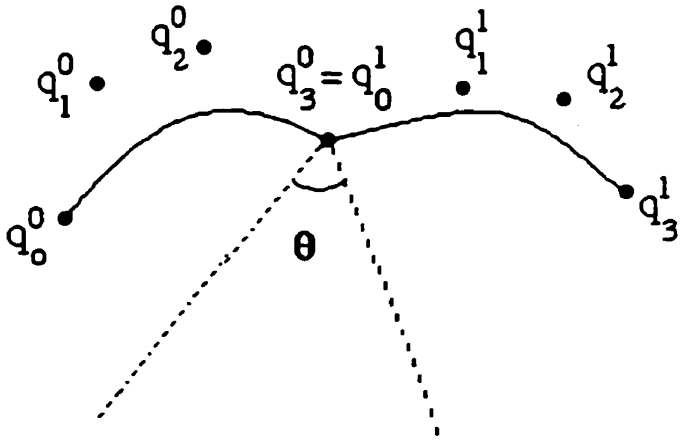


Fig. 4.8 – Calculating angles for kinking work

4.3.2 Changes to the Least Work Matching Algorithm

Here, the changes to the Least Work Matching algorithm of Chapter 2 are outlined.

As in Chapter 2, a rectangular grid is used to determine the Least Work control point correspondence. Here, instead of assigning every control point a column or row in the grid, we allow only the end control points of each Bézier curve to be represented in the grid. This is done to ensure that whole curve segments map to whole curve segments.

We denote the amount of work required to stretch (or shorten) the curve segment between end control points P_a^0 and P_b^0 (where, of course, two additional control points exist in between P_a^0 and P_b^0) of the whole curve P^0 into a curve segment between control points P_c^1 and P_d^1 of the whole curve P^1 (where control point P_a^0 corresponds to control point P_c^1, and control point P_b^0 corresponds to control point P_d^1) by

$$W_{stretch}([P_a^0, P_c^1],[P_b^0, P_d^1]).$$

Similarly, the amount of work required for kinking at a join point of Bézier curve segments, where P_a^0, P_b^0, and P_c^0 are the end control points of the two

adjoining segments of the whole curve P^0, and P_d^1, P_e^1, and P_f^1 are the end control points of the two adjoining segments of the whole curve P^1 (where P_a^0 corresponds to P_d^1, P_b^0 corresponds to P_e^1, and P_c^0 corresonds to P_f^1), is denoted by

$$W_{kink}([P_a^0, P_d^1],[P_b^0, P_e^1],[P_c^0, P_f^1]).$$

The amount of work required for bending a curve segment between end control points P_a^0 and P_b^0 of the whole curve P^0 into a curve segment between control points P_c^1 and P_d^1 of the whole curve P^1 (where control point P_a^0 corresponds to control point P_c^1, and control point P_b^0 corresponds to control point P_d^1) by

$$W_{bend}([P_a^0, P_c^1],[P_b^0, P_d^1]).$$

Like the Least Work Matching of Chapter 2, the algorithm here may insert additional control points. These control points may only be inserted at existing control points represented in the graph (that is, only at the curve segment's endpoints). In fact, when one control point is inserted, we must actually insert three points at that location; we are inserting an entire curve segment (which just so happens to be a point).

The same conditions for possible vertex correspondence apply to this graph as to the graph of Chapter 2.

The work equations for each grid vertex must consider not only stretching work (which is analogous to stretching work for polygon edges) and kinking work (which is analogous to bending work for polygon angles), but also curve bending work. Thus, analogous to equation 2.20, the equation for W_{west} becomes

$$\begin{aligned} W_{west} = \; & W_{stretch}([i-1,j],[i,j]) + W_{bend}([i-1,j],[i,j]) + \\ & \min\,[W_{west}(i-1,j) + W_{kink}([i-2,j],[i-1,j],[i,j]), \\ & W_{northwest}(i-1,j) + W_{kink}([i-2,j-1],[i-1,j],[i,j])], \end{aligned} \tag{4.7}$$

and the equations for $W_{north}(i,j)$ and $W_{northwest}(i,j)$ follow similarly.

Backtracking through the graph is exactly like that of Chapter 2, with regard to the control points that are represented in the graph. However, once we have completed the Least Work Matching list, we must insert the interior control points into the list for use in the interpolation.

Once the curves (and their control points) have been matched, linear interpolation can be used to calculate the in-between frames.

One of the most significant problems with using a Bézier curve representation when blending curves is the possibility that continuity will not be preserved throughout the blend. For example, in Fig. 4.9, the two Bézier curve segments of frame 1 are joined with C^1 continuity, and are matched with the two Bézier curve segments of frame 2, which also have C^1 continuity at their join point. However, throughout the blend, the continuity is decreased to C^0 at this join point (see Fig. 4.10), as the linear path followed by one of the control points causes a cusp in the in-between images.

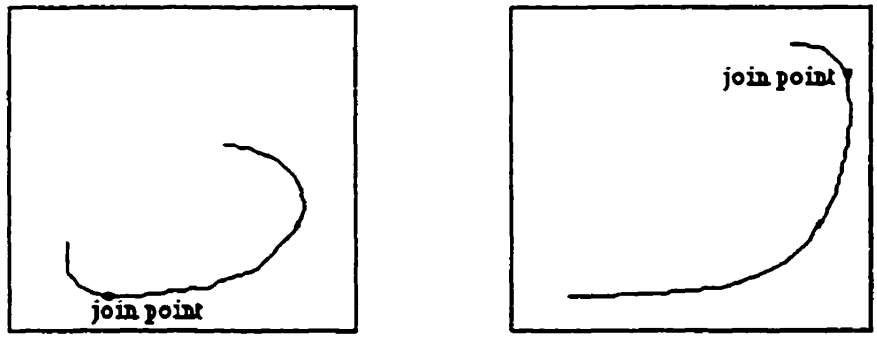


Fig. 4.9 – Two Bézier curves joined with C^1 continuity

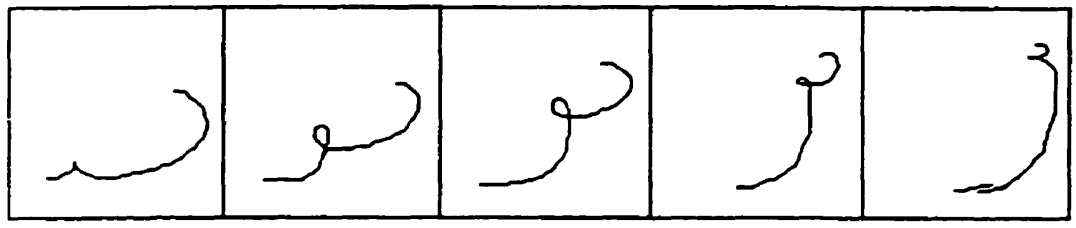


Fig. 4.10 – In-between images have reduced continuity

4.4 Results

Consider the "U" and the "J" of Fig. 4.11. Blending these letters using the Least Work Curve Matching algorithm coupled with linear interpolation, and using parameters $C_{bend} = 5$, $C_{kink} = 5$, $E_{kink} = 2$, $K_{stretch} = 0.1$, $C_{stretch} = 1$, and $E_{stretch} = 1$, gives a fairly good blend, as shown in Fig. 4.12. Each curve's starting control point for the blend of Fig. 4.12 is given by the dot on the images in Fig. 4.11. Contrary to our usual convention, the control points of these images have been labeled in counter-clockwise order.

To emphasize the importance of the starting control point correspondence, consider the blend given in Fig. 4.13. Here, the initial control point of the "U" is given in Fig. 4.11 by the square on the top left corner. The initial control point of the "J" is the same as in the previous blend.

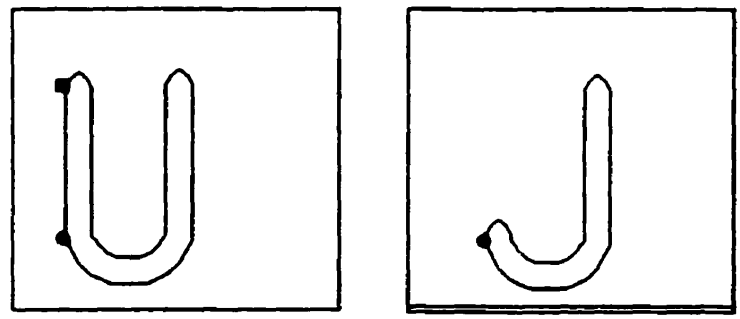


Fig. 4.11 – "U" and "J" to be blended

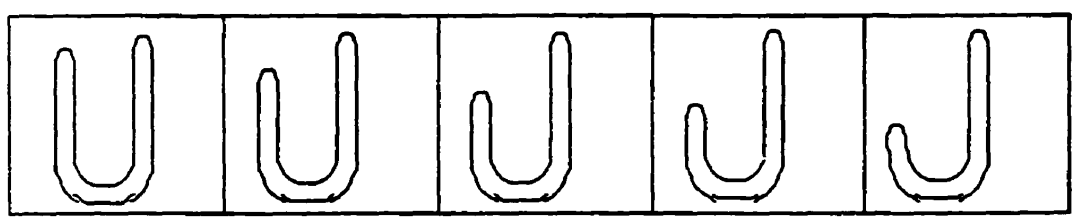


Fig. 4.12 – Least Work Curve Matching, initial vertex correspondence 1

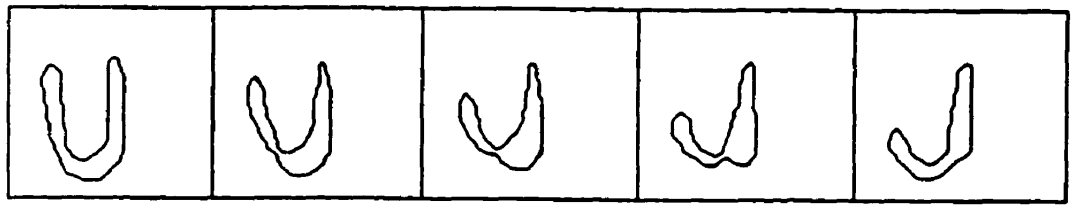


Fig. 4.13 – Least Work Curve Matching, initial vertex correspondence 2

Applying the Least Work Matching algorithm (of Chapter 2) to the control polygons of the letters with the first initial vertex correspondence, and using parameters $k_{bend} = 2$, $m_{bend} = 100$, $e_{bend} = 1$, $p_{bend} = 10000$, $k_{stretch} = 0.1$, $c_{stretch} = 0.1$, and $e_{stretch} = 1$, yields a reasonably good blend (Fig. 4.14), although not as smooth a transition as the blend given in Fig. 4.12. Using the same parameters on the control polygon blend with second initial vertex correspondence give less appealing results (Fig. 4.15).

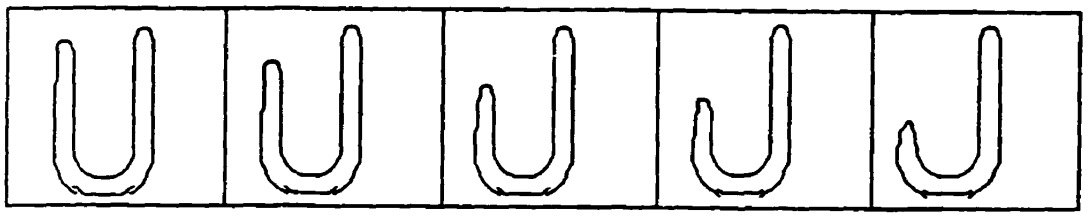


Fig. 4.14 – Control polygon blend, vertex correspondence 1

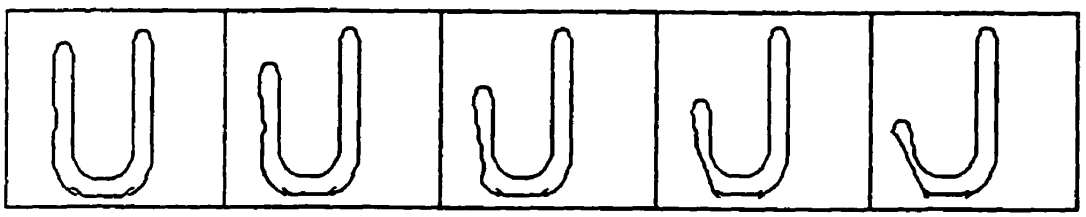


Fig. 4.15 – Control polygon blend, vertex correspondence 2

Chapter 5: Conclusion

5.1 Future Work

There are several problems associated with two-dimensional shape-blending that the methods discussed in this thesis do not address. Some of these have been discussed in the results sections of each chapter. Others are noted here.

To begin with, the Least Work Matching algorithm deals with local self-intersections (caused by angles going to zero) by imposing a penalty. However, there is nothing in the algorithm which tests for or penalizes global self-intersection.

As stated previously, the algorithm also requires a great deal of user-intervention. Firstly, the user is required to select starting points for the vertex correspondence. One way to avoid this would be to subject the polygon images

to some automated similar feature detection techniques of image processing. Secondly, the exponents $e_{stretch}$ and e_{bend} are set by the user before the program is run. Therefore, every edge or angle will use the same exponent, regardless of how much or how little a possible vertex correspondence will cause a particular edge or angle to stretch or bend. One suggestion is to use thresholding to allow the computer to select a value for $e_{stretch}$ and e_{bend} for each proposed edge and angle, based on the amount of stretching and bending that will occur for that edge or angle in a given situation. A user would, however, be required to set threshold levels.

The algorithms presented here deal only with images containing one polygon or closed curve. New methods would be required to deal with images containing several shapes (particularly if each image contained a different number of shapes), or with images of single shapes that contain one or more holes. Some problems associated with blending shapes containing holes include finding an appropriate matching of the inner shapes, and ensuring that all inner shapes remain completely inside the outer shape throughout the entire blend. If images contained different numbers of shapes, we would have to contend with problems such as deciding whether to split shapes apart to make new shapes or to create new shapes out of thin air, whether to join shapes together, or to make shapes vanish. Further, we would have to decide *which* of the shapes should be split, joined, deleted, or formed out of nothing.

An obvious improvement to the Least Work Curve Matching algorithm is to improve the manner in which kinking work is calculated, by taking into account

possible deviations from monotonicity and collapsing angles, as was done in Chapter 2 for the polygon bending work calculations.

Linear Interpolation was the only method used for the control point-path problem of curve blending. Intrinsic Interpolation could also be applied, either to the curves themselves, or to the control polygon.

A possible method to improve the continuity of the in-between frames of curve blending is to convert the Bézier curves to B-splines. B-splines allow greater continuity at join points. A method of converting Bézier curves to B-splines is outlined in [9].

5.2 Conclusion

This thesis has presented several techniques for blending 2-dimensional polygon. The Least Work Matching method of vertex or control point correspondence generally provides a good matching between the polygons (curves), provided that the vertices (control points) of each polygon (curve) are fairly evenly distributed, that the first vertex correspondence is appropriate, and that the bending and stretching (and kinking) parameters are chosen appropriately. Intrinsic Interpolation with Edge Tweaking clearly produces the most elegant results of the vertex path methods discussed in this work. Obviously, the most significant drawback of these methods is the amount of user

intervention that is necessary to produce a good blend. Despite the amount of input required, however, the methods are still quite satisfactory and, in many cases, have produced beautiful blends.

Appendix A: Implementation

A.1 Introduction

This appendix provides a discussion of the implementation of the algorithms outlined in this thesis.

A.2 Application

To assist with my study of shape blending, an application was created using Microsoft Visual Basic. Although Visual Basic is not the most efficient language to use, it did allow for a quick and easy user-interface. The user-interface for this application is shown in Fig. A.1. The application allows two key polygons or Bézier curves to be entered by a user. The polygons may be entered either by clicking points in the drawing windows, or by opening a file containing polygon

vertices. Bézier curves may only be entered by opening files of Bézier control points. To open a polygon, select either "Open Polygon in Key 1" or "Open Polygon in Key 2" from the File menu, and choose an appropriate text file. Files of Bézier control are opened similarly.

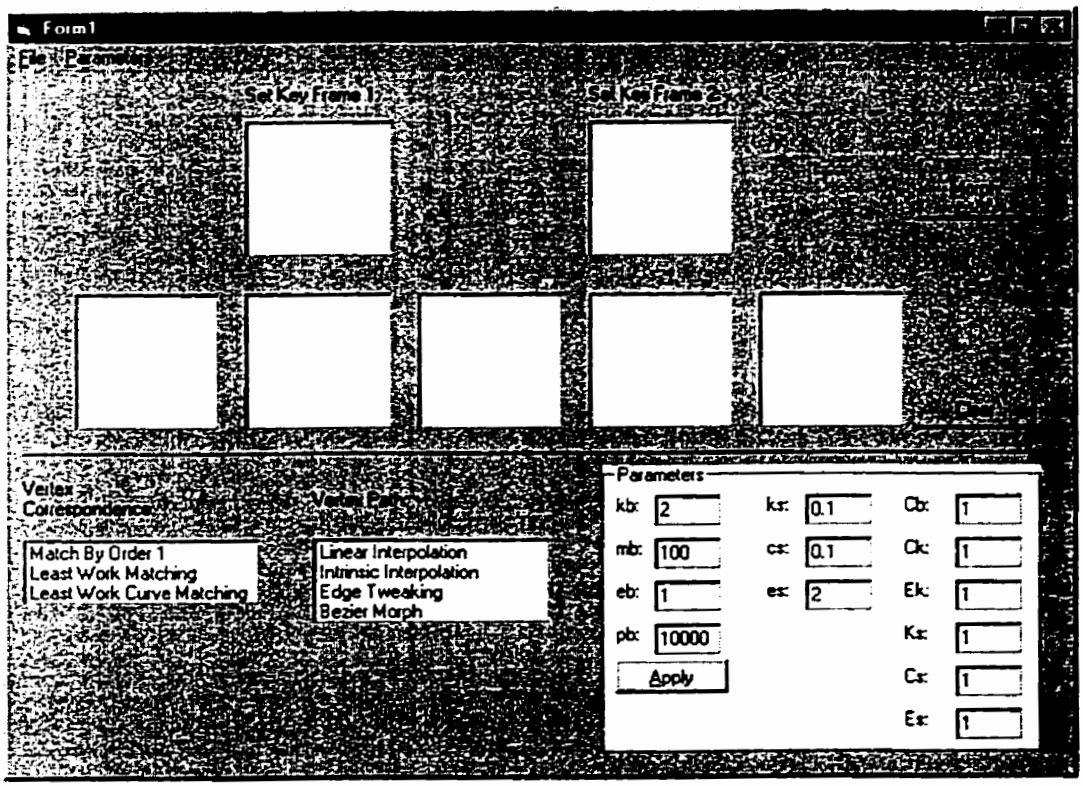


Fig. A.1 – User-interface of the application

Polygon vertices and Bézier control points are stored in text files as a list of real numbers, one per line, giving, alternately, the x- and y-coordinates of the points of the polygon, in a clockwise fashion.

Two list-boxes are given on the form, one containing a list of vertex correspondence methods, and the other containing a list of vertex path techniques. The user may select, from each list, the desired technique. Once the methods have been selected, the user can view the sequence of generated in-between frames by clicking the button labeled "Morph".

Note that the user is responsible for selecting appropriate methods, based on whether polygons or curves are to be blended. Further, it is the user's responsibility to ensure that both key frames contain polygons or that both key frames contain curves.

The number of in-between frames is hard-coded and may not be changed by the user at run-time.

The parameters for the work calculation of Least Work Matching and Least Work Curve Matching are given in a box in the lower right corner of the screen. The first two columns give the parameters for Least Work Matching. The first column contains parameters for bending work, and the second column gives parameters for stretching work. The parameters of the third column are for stretching, bending, and kinking work of the Least Work Curve Matching algorithm. The user can change these values as desired, but must click the

"Apply" button in order for the changes to take effect. One value of each parameter may be chosen, and this parameter is applied to the entire blend.

A.3 Discussion of the Implementations

This section gives a brief discussion of how some of the ideas in the thesis were implemented. Intrinsic Interpolation is a very straightforward implementation, so no comments are given. The Least Work Matching algorithm has several items to be noted (section A.2.1). Least Work Curve Matching is very similar to Least Work Matching, so no special mention is made of its implementation.

A.3.1 Least Work Matching

For each polygon, the vertices are read into an array of coordinates. For all possible graph vertices, W_{west}, W_{north}, and $W_{northwest}$ are calculated by determining the appropriate stretching and bending work calculations. My implementation takes the first vertex in a file of polygon vertices (or the first vertex clicked if the user is drawing her own polygon) as the first vertex correspondence.

Stretching work is a straight-forward calculation, requiring only edge length differences.

Bending work is somewhat more involved, and here we make a few notes on how bending work calculations were implemented.

If a triangle contains the origin, then together, the edges of the triangle must cross the positive x-axis, the negative x-axis, the positive y-axis, and the negative y-axis. The program tests each triangle edge for intersection with each of the axes, and keeps track of which axes have been intersected.

Monotonicity and deviation from monotonicity are calculated in a brute-force manner. Instead of solving equation 2.20 for $t \in (0,1)$, we take t in small increments and determine the angle that the vector from the origin to the point $Q(t)$ makes with the x-axis. We keep track of these angles to determine whether the angle changes monotonically. This list of angles also lets us determine how far from monotonicity the angle deviates (if it does), and in which direction. Furthermore, since we are calculating a list of angles, we take the opportunity to figure out if $Q(t)$ crosses the x-axis.

To backtrack, find the previous graph vertex that requires the least amount of work, and choose that one as the next vertex in the backtrack list.

A.4 Code

```
Option Explicit

Const MaxNum = 100
Const NumInBetweens = 5        ' Number of in-between frames
Const Epsilon = 0.001
Const PI = 3.1415926535

Dim Key1FirstClick As Boolean   ' Used when the user draws her own polygon by clicking
Dim Key2FirstClick As Boolean   ' points. Keepstrack of whether or not the mouse click
                                ' represents the first point of the polygon. Needed
                                ' for drawing (no Line_To used for the first click

Dim Key1NumPts As Integer       ' Number of vertices (or control points) in each
Dim Key2NumPts As Integer       ' key frame
Dim MaxCP1 As Integer           ' Number of control points in the B-Spline (converted
Dim MaxCP2 As Integer           ' from a Bezier)
Dim MaxKV1 As Integer           ' Number of knots in the converted B-Spline
Dim MaxKV2 As Integer
Dim Key1Pts(MaxNum) As Coords   ' Stores the coordinates of the vertices (control points)
Dim Key2Pts(MaxNum) As Coords   ' in the order they are read in (or clicked)
Dim Key1Knots(MaxNum + 4) As Double        ' Stores the knot vector for a spline
Dim Key2Knots(MaxNum + 4) As Double
Dim CP1(MaxNum) As Coords       ' Control Points of a B-Spline of Key Frame 1
 Dim CP2(MaxNum) As Coords      ' after the conversion from a Bezier curve.

Dim DistinctKnotList1(MaxNum) As Double ' a list of the distinct knots of B-Spline 1
Dim DistinctKnotList2(MaxNum) As Double '
Dim NumDistinctKnots1 As Integer ' the number of distinct knots of a b-spline
Dim NumDistinctKnots2 As Integer
Dim MorphPts(MaxNum) As Coords
Dim NumPts As Integer

Dim DrawPoly1 As Boolean        ' True if the user draws the polygon by clicking points.
Dim DrawPoly2 As Boolean        ' False if we read from a file. False if we are drawing
                                ' a curve. (N.B. This program does not allow the user
                                ' to draw a curve by clicking control points. All curves
                                ' must be read from a file.

Dim DrawBezier As Boolean       ' True if we are opening Bezier curves. False for polygons
Dim KeyDifference(MaxNum) As Coords

' Bending parameters
Dim kb As Double    ' bending stiffness
Dim mb As Double    ' penalizes non-monotonic angles
Dim eb As Integer   ' either 1 or 2
Dim pb As Integer   ' penalizes angles from going to 0

'Stretching parameters
Dim ks As Double        ' stretching stiffness constant
Dim cs As Double        ' controls penalty for edge collapsing to a point
Dim es As Integer       ' 1 or 2, depending on the stretchiness of the wire
```

```
' Curve bending parameters
Dim CurveCb As Double
Dim CurveCk As Double
Dim CurveEk As Double
Dim CurveKs As Double
Dim CurveCs As Double
Dim CurveEs As Double
'*************************************************************************
Private Sub cmdApply_Click()
   kb = CDbl(txtkb.Text)
   mb = CDbl(txtmb.Text)
   eb = Val(txteb.Text)
   pb = Val(txtpb.Text)

   ks = CDbl(txtks.Text)
   cs = CDbl(txtcs.Text)
   es = Val(txtes.Text)

   CurveKs = CDbl(txtCurveKs.Text)
   CurveCb = CDbl(txtCurveCb.Text)
   CurveEk = CDbl(txtCurveEk.Text)
   CurveCk = CDbl(txtCurveCk.Text)
   CurveCs = CDbl(txtCurveCs.Text)
   CurveEs = CDbl(txtCurveEs.Text)

End Sub
'*************************************************************************
Private Sub cmdClear_Click()
' Clears all drawing from the picture windows and re-initialized the data corresponding
'  to the pictures

   Dim I As Integer

   Key1FirstClick = True
   Key2FirstClick = True
   Key1NumPts = 0
   Key2NumPts = 0
   NumPts = 0
   DrawPoly1 = True
   DrawPoly2 = True
   DrawBezier = False
    picKey1.Cls
    picKey2.Cls
    picMorph1.Cls
    picMorph2.Cls
    picMorph3.Cls
    picMorph4.Cls
    picMorph5.Cls
End Sub
'*************************************************************************
```

```
Private Sub cmdMorph_Click()
' starts the selected vertex correspondence and vertex path methods
' Note: All polygons are required to be closed.  For each polygon, draw a line
'       from the last vertex to the first vertex.  Do not increment the number of
'        points, since the number of points remains the same. (We don't want
'       to count the first vertex twice). Add the coords of the first vertex to the
'       end of the vertex list (for simplicity in later code).
' Note: Only do this if we are drawing polygons, NOT if we are drawing
'      curves

   If DrawBezier = False Then
      ' Close the polygons. This is only  NECESSARY when the user is drawing
       '  her own polygonby clicking points. If a polygon file is opened, the
       ' "open" routine takes care of closing the polygon

      picKey2.Line (10 * Key2Pts(Key2NumPts - 1).X, _
              picKey2.Height - (10 * Key2Pts(Key2NumPts - 1).Y))- _
              (10 * Key2Pts(0).X, picKey2.Height - (10 * Key2Pts(0).Y))

      picKey1.Line (10 * Key1Pts(Key1NumPts - 1).X, _
              picKey1.Height - (10 * Key1Pts(Key1NumPts - 1).Y))- _
              (10 * Key1Pts(0).X, picKey1.Height - (10 * Key1Pts(0).Y))

      Key1Pts(Key1NumPts).X = Key1Pts(0).X
      Key1Pts(Key1NumPts).Y = Key1Pts(0).Y
      Key2Pts(Key2NumPts).X = Key2Pts(0).X
      Key2Pts(Key2NumPts).Y = Key2Pts(0).Y
   End If

    ' Choose a vertex correspondence method
    Select Case lstVertCorr.ListIndex
      Case 0
        MatchByOrder1
      Case 1
         LeastWorkMatching
      Case 2
         LeastWorkCurveMatching
   End Select
    ' Choose a vertex path method
    Select Case lstVertPath.ListIndex
      Case 0
         LinearInterpolation
      Case 1
          IntrinsicInterpolation
      Case 2
        EdgeTweaking
      Case 3
         LinearBezierMorph
      Case 4
          IntrinsicBezierMorph
    End Select

End Sub
```

```
'*************************************************************************
Private Sub Form_Load()
   Key1FirstClick = True
   Key2FirstClick = True
   Key1NumPts = 0
   Key2NumPts = 0
   DrawPoly1 = True
   DrawPoly2 = True
   DrawBezier = False
   ' Initialize bending parameters
   kb = 2    ' bending stiffness
    mb = 100    ' penalizes non-monotonic angles
   eb = 1    '
    pb = 10000    ' penalizes angles from going to 0
    ' Initialize stretching parameters
    ks = 0.1       ' stretching stiffness constant
    cs = 0.1       ' controls penalty for edge collapsing to a point
   es = 2     '
   ' Initialize curve parameters
   CurveCb = 1
   CurveCk = 1
   CurveEk = 1
   CurveKs = 0.1
   CurveCs = 1
   CurveEs = 1
End Sub

'*************************************************************************
Private Sub mnuExit_Click()
  End
End Sub

'*************************************************************************
Private Sub mnuOpenCurve1_Click()
' Opens a Bezier Curve in Frame 1
' Set CancelError is True
   CommonDialog1.CancelError = True
   On Error GoTo ErrHandler
   ' Set flags
   CommonDialog1.Flags = cdlOFNHideReadOnly
   ' Set filters
    CommonDialog1.Filter = "All Files (*.*)|*.*|Text Files" & _
     "(*.txt)|*.txt"
   ' Specify default filter
    CommonDialog1.FilterIndex = 2
    ' Display the Open dialog box
    CommonDialog1.ShowOpen
    ' Display name of selected file

    DrawPoly1 = False   'If we commit ourselves to opening a file in
                        ' the frame, we cannot then decide to draw
                        ' our own polygon (by clicking points).
   Dim Fnum As Integer
```

```
Dim Temp As String
Dim I As Integer
Dim J As Boolean
DrawBezier = True
I = 0
J = True
Fnum = FreeFile
 Open CommonDialog1.filename For Input As #Fnum

Do While Not EOF(Fnum)
   Line Input #Fnum, Temp
  If J = True Then
     Key1Pts(I).X = CDbl(Temp)
     J = False
  Else
     Key1Pts(I).Y = CDbl(Temp)
    J = True
    I = I + 1
    Key1NumPts = I
  End If
Loop

 Key1Pts(Key1NumPts).X = Key1Pts(0).X
 Key1Pts(Key1NumPts).Y = Key1Pts(0).Y
I = I + 1
Dim NumExtraPts As Integer
NumExtraPts = 0

 While (((Key1NumPts) Mod 3) <> 0)   ' we should have the right number of
    Key1Pts(I).X = Key1Pts(0).X      ' control points in the file, but just
   Key1Pts(I).Y = Key1Pts(0).Y      ' in case we don't, we do this
   NumExtraPts = NumExtraPts + 1
   Key1NumPts = Key1NumPts + 1
  I = I + 1
Wend

For I = 1 To (Key1NumPts - NumExtraPts) Step 1
    ' mark the control points
    picKey1.Circle (10 * Key1Pts(I - 1).X, picKey1.Height - 10 * Key1Pts(I - 1).Y), _
             1, RGB(0, 255, 0)
   'picKey1.Line (10 * Key1Pts(I - 1).X, _
         picKey1.Height - (10 * Key1Pts(I - 1).Y)) _
         -(10 * Key1Pts(I).X, picKey1.Height - (10 * Key1Pts(I).Y))
Next I

 Dim Pt0 As Coords
 Dim Pt1 As Coords
 Dim Pt2 As Coords
 Dim Pt3 As Coords
 Dim t1 As Integer
 Dim t As Double
 Dim NumCurves As Integer
 Dim TempX As Double
```

```
  Dim TempY As Double
  NumCurves = ((Key1NumPts) / 3)

  For I = 0 To (NumCurves - 1) Step 1
     Pt0 = Key1Pts(3 * I)
     Pt1 = Key1Pts(3 * I + 1)
     Pt2 = Key1Pts(3 * I + 2)
     Pt3 = Key1Pts(3 * I + 3)
     For t1 = 0 To 200 Step 1
        'calculate and plot the point of the bezier curve
       t = t1 / 200
       TempX = (1 - t) * (1 - t) * (1 - t) * Pt0.X + 3 * t * (1 - t) * (1 - t) * Pt1.X _
            + 3 * t * t * (1 - t) * Pt2.X + t * t * t * Pt3.X
       TempY = (1 - t) * (1 - t) * (1 - t) * Pt0.Y + 3 * t * (1 - t) * (1 - t) * Pt1.Y _
            + 3 * t * t * (1 - t) * Pt2.Y + t * t * t * Pt3.Y
        picKey1.Circle (10 * TempX, picKey1.Height - (10 * TempY)), 0.2
     Next t1
  Next I
  Exit Sub

ErrHandler:
   'User pressed the Cancel button
   Exit SubEnd Sub

'*************************************************************************
Private Sub mnuOpenCurve2_Click()
' Opens a Bezier Curve in Frame 2
' Set CancelError is True
   CommonDialog1.CancelError = True
   On Error GoTo ErrHandler
   ' Set flags
   CommonDialog1.Flags = cdlOFNHideReadOnly
   ' Set filters
   CommonDialog1.Filter = "All Files (*.*)|*.*|Text Files" & _
    "(*.txt)|*.txt"
   ' Specify default filter
   CommonDialog1.FilterIndex = 2
   ' Display the Open dialog box
   CommonDialog1.ShowOpen
   ' Display name of selected file

   DrawPoly2 = False   'If we commit ourselves to opening a file in
                       ' the frame, we cannot then decide to draw
                        ' our own polygon (by clicking points).
   Dim Fnum As Integer
   Dim Temp As String
   Dim I As Integer
   Dim J As Boolean
   I = 0
   J = True
   Fnum = FreeFile

   Open CommonDialog1.filename For Input As #Fnum
```

```
Do While Not EOF(Fnum)
  Line Input #Fnum, Temp
  If J = True Then
    Key2Pts(I).X = CDbl(Temp)
    J = False
  Else
    Key2Pts(I).Y = CDbl(Temp)
    J = True
    I = I + 1
    Key2NumPts = I
  End If
Loop

Key2Pts(Key2NumPts).X = Key2Pts(0).X
Key2Pts(Key2NumPts).Y = Key2Pts(0).Y
I = I + 1
Dim NumExtraPts As Integer
NumExtraPts = 0
While (((Key2NumPts) Mod 3) <> 0)
  Key2Pts(I).X = Key2Pts(0).X
  Key2Pts(I).Y = Key2Pts(0).Y
  NumExtraPts = NumExtraPts + 1
  Key2NumPts = Key2NumPts + 1
  I = I + 1
Wend

For I = 1 To (Key2NumPts - NumExtraPts) Step 1
  ' mark the control points
  picKey2.Circle (10 * Key2Pts(I - 1).X, picKey2.Height - 10 * Key2Pts(I - 1).Y), _
      1, RGB(0, 255, 0)
  'picKey2.Line (10 * Key2Pts(I - 1).X, _
      picKey2.Height - (10 * Key2Pts(I - 1).Y)) _
      -(10 * Key2Pts(I).X, picKey2.Height - (10 * Key2Pts(I).Y))
  ' note: uncomment the above line if you want the control polygon drawn
Next I

Dim Pt0 As Coords
Dim Pt1 As Coords
Dim Pt2 As Coords
Dim Pt3 As Coords
Dim tI As Integer
Dim t As Double
Dim NumCurves As Integer
Dim TempX As Double
Dim TempY As Double
NumCurves = ((Key2NumPts) / 3)
For I = 0 To (NumCurves - 1) Step 1
  Pt0 = Key2Pts(3 * I)
  Pt1 = Key2Pts(3 * I + 1)
  Pt2 = Key2Pts(3 * I + 2)
  Pt3 = Key2Pts(3 * I + 3)
```

```
    For t1 = 0 To 200 Step 1
        'calculate and plot the points of the bezier curve
       t = t1 / 200
       TempX = (1 - t) * (1 - t) * (1 - t) * Pt0.X + 3 * t * (1 - t) * (1 - t) * Pt1.X _
            + 3 * t * t * (1 - t) * Pt2.X + t * t * t * Pt3.X
       TempY = (1 - t) * (1 - t) * (1 - t) * Pt0.Y + 3 * t * (1 - t) * (1 - t) * Pt1.Y _
            + 3 * t * t * (1 - t) * Pt2.Y + t * t * t * Pt3.Y
        picKey2.Circle (10 * TempX, picKey2.Height - (10 * TempY)), 0.2
    Next t1
  Next I
  Exit Sub
ErrHandler:
  'User pressed the Cancel button
  Exit Sub
End Sub

'*********************************************************************
Private Sub mnuOpenKey1_Click()
' Lets the user choose a file of polygon vertices to be opened and drawn in Key frame 1

  CommonDialog1.CancelError = True    ' Set CancelError to True
  On Error GoTo ErrHandler
  CommonDialog1.Flags = cdlOFNHideReadOnly    ' Set flags
   CommonDialog1.Filter = "All Files (*.*)|*.*|Text Files" & _
  "(*.txt)|*.txt"                ' Set filters
  CommonDialog1.FilterIndex = 2       ' Specify default filter
  CommonDialog1.ShowOpen              ' Display the Open dialog box

  DrawPoly1 = False         ' Once we have opened a file in the frame, we cannot
                            ' draw our polygon
  DrawBezier = False        ' We are drawing a polygon, not a curve

  Dim Fnum As Integer
  Dim Temp As String
  Dim I As Integer
  Dim J As Boolean
  I = 0
  J = True
  Fnum = FreeFile
   Open CommonDialog1.filename For Input As #Fnum  ' Display name of selected file

  Do While Not EOF(Fnum)
    Line Input #Fnum, Temp
     ' Read in the points. The file contains point as one coordinate per line.
     ' i.e. x on one line, corresponding y on the next; next x on the next line, etc.
    If J = True Then
      Key1Pts(I).X = CDbl(Temp)
      J = False
    Else
      Key1Pts(I).Y = CDbl(Temp)
     J = True
     I = I + 1
     Key1NumPts = I
```

```
    End If
  Loop

  Key1Pts(Key1NumPts).X = Key1Pts(0).X      ' Repeat the first vertex as the last
  Key1Pts(Key1NumPts).Y = Key1Pts(0).Y      ' to force closure

  For I = 1 To Key1NumPts Step 1
    ' Draw the vertices and edges in the frame
    picKey1.Circle (10 * Key1Pts(I - 1).X, picKey1.Height - 10 * Key1Pts(I - 1).Y), _
            0.5, RGB(0, 0, 255)
    picKey1.Line (10 * Key1Pts(I - 1).X, _
          picKey1.Height - (10 * Key1Pts(I - 1).Y)) _
          -(10 * Key1Pts(I).X, picKey1.Height - (10 * Key1Pts(I).Y))
  Next I
  Exit Sub
ErrHandler:
  'User pressed the Cancel button
  Exit Sub
End Sub

'*************************************************************************=
Private Sub mnuOpenKey2_Click()
' Lets the user choose a file of polygon vertices to be opened and drawn in
' Key Frame 2
  CommonDialog1.CancelError = True      ' Set CancelError is True
  On Error GoTo ErrHandler
  CommonDialog1.Flags = cdlOFNHideReadOnly   ' Set flags
  CommonDialog1.Filter = "All Files (*.*)|*.*|Text Files" & _
  "(*.txt)|*.txt"                    ' Set filters
  CommonDialog1.FilterIndex = 2      ' Specify default filter
  CommonDialog1.ShowOpen     ' Display the Open dialog box
  DrawPoly2 = False          ' Once we have opened a file in the frame,
                    ' we cannot drawn our own polygon
  DrawBezier = False
  Dim Fnum As Integer
  Dim Temp As String
  Dim I As Integer
  Dim J As Boolean
  I = 0
  J = True
  Fnum = FreeFile
  Open CommonDialog1.filename For Input As #Fnum     ' Display name of selected file

  Do While Not EOF(Fnum)
    ' Read in the points. See mnuOpenKey1 for file description.
    Line Input #Fnum, Temp
    If J = True Then
      Key2Pts(I).X = CDbl(Temp)
      J = False
    Else
      Key2Pts(I).Y = CDbl(Temp)
      J = True
      I = I + 1
```

```
      Key2NumPts = I
    End If
  Loop

  Key2Pts(Key2NumPts).X = Key2Pts(0).X       ' Force polygon closure
  Key2Pts(Key2NumPts).Y = Key2Pts(0).Y

  For I = 1 To Key2NumPts Step 1
    ' Draw
     picKey2.Circle (10 * Key2Pts(I - 1).X, picKey2.Height - 10 * Key2Pts(I - 1).Y), _
              0.5, RGB(0, 0, 255)
     picKey2.Line (10 * Key2Pts(I - 1).X, _
           picKey2.Height - (10 * Key2Pts(I - 1).Y)) _
           -(10 * Key2Pts(I).X, picKey2.Height - (10 * Key2Pts(I).Y))
  Next I
 Exit Sub
ErrHandler:
   'User pressed the Cancel button
 Exit Sub
End Sub

'*********************************************************************
Private Sub mnuOpenSpline1_Click()
' Opens a Bezier Curve in Frame 1 and converts it to a B-Spline
' Set CancelError is True
   CommonDialog1.CancelError = True
   On Error GoTo ErrHandler
   ' Set flags
   CommonDialog1.Flags = cdlOFNHideReadOnly
   ' Set filters
    CommonDialog1.Filter = "All Files (*.*)|*.*|Text Files" & _
    "(*.txt)|*.txt"
   ' Specify default filter
    CommonDialog1.FilterIndex = 2
   ' Display the Open dialog box
    CommonDialog1.ShowOpen
   ' Display name of selected file

   DrawPoly1 = False   'If we commit ourselves to opening a file in the frame, we
                       ' cannot then decide to draw our own polygon by clicking pts
   Dim Fnum As Integer
   Dim Temp As String
   Dim I As Integer
   Dim J As Boolean
   DrawBezier = True
   I = 0
   J = True
   Fnum = FreeFile
    Open CommonDialog1.filename For Input As #Fnum
   Do While Not EOF(Fnum)
     Line Input #Fnum, Temp

     If J = True Then
```

```
      Key1Pts(I).X = CDbl(Temp)
      J = False
    Else
      Key1Pts(I).Y = CDbl(Temp)
      J = True
      I = I + 1
      Key1NumPts = I
    End If
  Loop

  Key1Pts(Key1NumPts).X = Key1Pts(0).X
  Key1Pts(Key1NumPts).Y = Key1Pts(0).Y
 I = I + 1
 Dim NumExtraPts As Integer
 NumExtraPts = 0

 While (((Key1NumPts) Mod 3) <> 0)  ' we should have the right number of
    Key1Pts(I).X = Key1Pts(0).X       ' control points in the file, but just
    Key1Pts(I).Y = Key1Pts(0).Y       ' in case we don't, we do this
    NumExtraPts = NumExtraPts + 1
    Key1NumPts = Key1NumPts + 1
   I = I + 1
 Wend

 ' Now convert the Bezier to a B-Spline
  ' initialize the control point list and knot vector
 For I = 0 To 3 Step 1
    CP1(I).X = Key1Pts(I).X
    CP1(I).Y = Key1Pts(I).Y
    Key1Knots(I) = 0
 Next I
 For I = 4 To 7 Step 1
    Key1Knots(I) = 1
 Next I
 MaxCP1 = 3
 MaxKV1 = 7

 ' Want to add the next Bezier curve control points
  ' to the list of control points
 Dim Slope1 As Double
 Dim Slope2 As Double
 Dim NewKnot As Double
  Dim Continuity As Integer
 Dim NumCurves As Integer
 Dim NextIndex As Integer
 NumCurves = ((Key1NumPts) / 3)
 ' This will be in some kind of loop
 NextIndex = MaxCP1 + 1
 While NextIndex < Key1NumPts
    If Abs(CP1(MaxCP1).X - CP1(MaxCP1 - 1).X) < Epsilon Then
      Slope1 = 32000
    Else
      Slope1 = (CP1(MaxCP1).Y - CP1(MaxCP1 - 1).Y) / _
```

```
        (CP1(MaxCP1).X - CP1(MaxCP1 - 1).X)
End If

If Abs(Key1Pts(NextIndex).X - CP1(MaxCP1).X) < Epsilon Then
  Slope2 = 32000
Else
  Slope2 = (Key1Pts(NextIndex).Y - CP1(MaxCP1).Y) / _
       (Key1Pts(NextIndex).X - CP1(MaxCP1).X)
End If

If (Abs(Slope1 - Slope2) > Epsilon) Then  ' curve have only C0 continuity
  NewKnot = Key1Knots(MaxKV1) + 1
  Continuity = 0
Else   ' the slope is the same, so curves have at least C1 continuity.
      ' Choose a knot value that reflects this continuity
  NewKnot = ((Key1Pts(NextIndex).X - CP1(MaxCP1).X) * _
        (Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 4)) / _
        (CP1(MaxCP1).X - CP1(MaxCP1 - 1).X)) + _
         Key1Knots(MaxKV1 - 3)
  Continuity = 1

  ' Now test to see if the curves are actually C2 continuous
  If Abs( _
      ( _
        (CP1(MaxCP1 - 2).X - Key1Pts(NextIndex + 1).X) * _
        (Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 5)) * _
        (Key1Knots(MaxKV1 - 3) - NewKnot) _
      ) _
    + _
      ( _
        (CP1(MaxCP1 - 1).X - CP2(MaxCP1 - 2).X) * _
        (NewKnot - Key1Knots(MaxKV1 - 5)) * _
        (Key1Knots(MaxKV1 - 3) - NewKnot) _
      ) _
    + _
      ( _
        (Key1Pts(NextIndex + 1).X - Key1Pts(NextIndex).X) * _
        (Key1Knots(MaxKV1 - 4) - NewKnot) * _
        (Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 5)) _
      ) _
      ) < Epsilon Then
    ' then we have C2 continuity
    Continuity = 2
  End If
' Now check for C3 continuity
  Dim PAlpha As Coords
  Dim PBeta As Coords
  Dim PGamma As Coords
  If (Abs(Key1Knots(MaxKV1 - 5) - Key1Knots(MaxKV1 - 6)) > Epsilon) Then
    PAlpha.X = ((Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 4)) * _
            Key1Pts(NextIndex + 1).X + _
          ((NewKnot - Key1Knots(MaxKV1 - 3)) * Key1Pts(NextIndex).X)) / _
          (NewKnot - Key1Knots(MaxKV1 - 4))
```

```
        PAlpha.Y = ((Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 4)) * Key1Pts(NextIndex +
1).Y + _
                ((NewKnot - Key1Knots(MaxKV1 - 3)) *Key1Pts(NextIndex).Y)) / _
               (NewKnot - Key1Knots(MaxKV1 - 4))

        PBeta.X = ((NewKnot - Key1Knots(MaxKV1 - 6)) * CP1(MaxCP1 - 2).X + _
                ((Key1Knots(MaxKV1 - 5) - NewKnot) *CP1(MaxCP1 - 3).X)) / _
                (Key1Knots(MaxKV1 - 5) - Key1Knots(MaxKV1 - 6))

        PBeta.Y = ((NewKnot - Key1Knots(MaxKV1 - 6)) *CP1(MaxCP1 - 2).Y + _
                ((Key1Knots(MaxKV1 - 5) - NewKnot) * CP1(MaxCP1 - 3).Y)) / _
                (Key1Knots(MaxKV1 - 5) - Key1Knots(MaxKV1 - 6))

        PGamma.X = ((Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 4)) *Key1Pts(NextIndex +
2).X + _
              ((Key1Knots(MaxKV1 - 4) - NewKnot) *Key1Pts(NextIndex + 1).X)) / _
             (Key1Knots(MaxKV1 - 3) - NewKnot)
        PGamma.Y = ((Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 4)) * Key1Pts(NextIndex +
2).Y + _
              ((Key1Knots(MaxKV1 - 4) - NewKnot) * Key1Pts(NextIndex + 1).Y)) / _
             (Key1Knots(MaxKV1 - 3) - NewKnot)

        If Abs(PAlpha.X - ((((NewKnot - Key1Knots(MaxKV1 - 3)) * PBeta.X) + _
                ((Key1Knots(MaxKV1 - 3) - Key1Knots(MaxKV1 - 5)) * PGamma.X)) / _
                (NewKnot - Key1Knots(MaxKV1 - 5)))) < Epsilon Then

          Continuity = 3
        End If
      End If
    End If
      ' Append the new knots and control points, depending on the
      ' continuity between the two curves
    Select Case Continuity
    Case 0
      Key1Knots(MaxKV1) = NewKnot
      Key1Knots(MaxKV1 + 1) = NewKnot
      Key1Knots(MaxKV1 + 2) = NewKnot
      Key1Knots(MaxKV1 + 3) = NewKnot
      MaxKV1 = MaxKV1 + 3
      CP1(MaxCP1 + 1).X = Key1Pts(MaxCP1 + 1).X
      CP1(MaxCP1 + 1).Y = Key1Pts(MaxCP1 + 1).Y
      CP1(MaxCP1 + 2).X = Key1Pts(MaxCP1 + 2).X
      CP1(MaxCP1 + 2).Y = Key1Pts(MaxCP1 + 2).Y
      CP1(MaxCP1 + 3).X = Key1Pts(MaxCP1 + 3).X
      CP1(MaxCP1 + 3).Y = Key1Pts(MaxCP1 + 3).Y
      MaxCP1 = MaxCP1 + 3
    Case 1
      Key1Knots(MaxKV1 - 1) = NewKnot
      Key1Knots(MaxKV1) = NewKnot
      Key1Knots(MaxKV1 + 1) = NewKnot
      Key1Knots(MaxKV1 + 2) = NewKnot
      MaxKV1 = MaxKV1 + 2
      CP1(MaxCP1).X = Key1Pts(MaxCP1 + 1).X
```

```
    CP1(MaxCP1).Y = Key1Pts(MaxCP1 + 1).Y
    CP1(MaxCP1 + 1).X = Key1Pts(MaxCP1 + 2).X
    CP1(MaxCP1 + 1).Y = Key1Pts(MaxCP1 + 2).Y
    CP1(MaxCP1 + 2).X = Key1Pts(MaxCP1 + 3).X
    CP1(MaxCP1 + 2).Y = Key1Pts(MaxCP1 + 3).Y
    MaxCP1 = MaxCP1 + 2
  Case 2
    Key1Knots(MaxKV1 - 2) = NewKnot
    Key1Knots(MaxKV1 - 1) = NewKnot
    Key1Knots(MaxKV1) = NewKnot
    Key1Knots(MaxKV1 + 1) = NewKnot
    MaxKV1 = MaxKV1 + 1
    CP1(MaxCP1 - 1).X = PAlpha.X
    CP1(MaxCP1 - 1).Y = PAlpha.Y
    CP1(MaxCP1).X = Key1Pts(MaxCP1 + 2).X
    CP1(MaxCP1).Y = Key1Pts(MaxCP1 + 2).Y
    CP1(MaxCP1 + 1).X = Key1Pts(MaxCP1 + 3).X
    CP1(MaxCP1 + 1).Y = Key1Pts(MaxCP1 + 3).Y
    MaxCP1 = MaxCP1 + 1
  Case 3
    Key1Knots(MaxKV1 - 3) = NewKnot
    Key1Knots(MaxKV1 - 2) = NewKnot
    Key1Knots(MaxKV1 - 1) = NewKnot
    Key1Knots(MaxKV1) = NewKnot
    MaxKV1 = MaxKV1
    CP1(MaxCP1 - 2).X = PBeta.X
    CP1(MaxCP1 - 2).Y = PBeta.Y
    CP1(MaxCP1 - 1).X = PGamma.X
    CP1(MaxCP1 - 1).Y = PGamma.Y
    CP1(MaxCP1).X = Key1Pts(MaxCP1 + 3).X
    CP1(MaxCP1).Y = Key1Pts(MaxCP1 + 3).Y
    MaxCP1 = MaxCP1
  End Select
  NextIndex = NextIndex + 3
Wend   For I = 1 To (MaxCP1 + 1) Step 1
   ' mark the control points
   picKey1.Circle (10 * CP1(I - 1).X, picKey1.Height - 10 * CP1(I - 1).Y), _
          1, RGB(0, 255, 0)
Next I
Dim t1 As Integer
Dim t As Double
Dim TempX As Double
Dim TempY As Double
' Draw the curve
I = 3
While I <= MaxCP1
   For t1 = (200 * Key1Knots(I)) To (200 * Key1Knots(I + 1)) Step 1
     If (Key1Knots(I) <> Key1Knots(I + 1)) Then
      'calculate and plot the points of the b-spline
       t = t1 / (200 * (Key1Knots(I + 1) - Key1Knots(I)))
      Dim Term1 As Double
      Dim Term2 As Double
      Dim Term3 As Double
```

```
Dim Term4 As Double
Term1 = (Key1Knots(I + 1) - t) * (Key1Knots(I + 1) - t) * (Key1Knots(I + 1) - t) _
    * CP1(I - 3).X / _
    ((Key1Knots(I + 1) - Key1Knots(I - 2)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I)))

Term2 = (t - Key1Knots(I - 2)) * (Key1Knots(I + 1) - t) * (Key1Knots(I + 1) - t) _
     * CP1(I - 2).X / _
      ((Key1Knots(I + 1) - Key1Knots(I - 2)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I))) + _
     (Key1Knots(I + 2) - t) * (t - Key1Knots(I - 1)) * (Key1Knots(I + 1) - t) _
    * CP1(I - 2).X / _
     ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I))) + _
     (Key1Knots(I + 2) - t) * (t - Key1Knots(I)) * (Key1Knots(I + 2) - t) _
    * CP1(I - 2).X / _
     ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 2) - Key1Knots(I)) _
     * (Key1Knots(I + 1) - Key1Knots(I)))

Term3 = (t - Key1Knots(I - 1)) * (t - Key1Knots(I - 1)) * (Key1Knots(I + 1) - t) _
    * CP1(I - 1).X / _
     ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I))) + _
     (t - Key1Knots(I)) * (t - Key1Knots(I - 1)) * (Key1Knots(I + 2) - t) _
    * CP1(I - 1).X / _
     ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 2) - Key1Knots(I)) _
     * (Key1Knots(I + 1) - Key1Knots(I))) + _
     (Key1Knots(I + 3) - t) * (t - Key1Knots(I)) * (t - Key1Knots(I)) _
    * CP1(I - 1).X / _
     ((Key1Knots(I + 3) - Key1Knots(I)) * (Key1Knots(I + 2) - Key1Knots(I)) _
     * (Key1Knots(I + 1) - Key1Knots(I)))

Term4 = (t - Key1Knots(I)) * (t - Key1Knots(I)) * (t - Key1Knots(I)) * _
    CP1(I).X / _
     ((Key1Knots(I + 3) - Key1Knots(I)) * (Key1Knots(I + 2) - Key1Knots(I)) _
     * (Key1Knots(I + 1) - Key1Knots(I)))

TempX = Term1 + Term2 + Term3 + Term4

Term1 = (Key1Knots(I + 1) - t) * (Key1Knots(I + 1) - t) * (Key1Knots(I + 1) - t) _
    * CP1(I - 3).Y / _
     ((Key1Knots(I + 1) - Key1Knots(I - 2)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I)))

Term2 = (t - Key1Knots(I - 2)) * (Key1Knots(I + 1) - t) * (Key1Knots(I + 1) - t) _
    * CP1(I - 2).Y / _
     ((Key1Knots(I + 1) - Key1Knots(I - 2)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I))) + _
     (Key1Knots(I + 2) - t) * (t - Key1Knots(I - 1)) * (Key1Knots(I + 1) - t) _
    * CP1(I - 2).Y / _
     ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
     * (Key1Knots(I + 1) - Key1Knots(I))) + _
     (Key1Knots(I + 2) - t) * (t - Key1Knots(I)) * (Key1Knots(I + 2) - t) _
```

```
            * CP1(I - 2).Y / _
             ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 2) - Key1Knots(I)) _
             * (Key1Knots(I + 1) - Key1Knots(I)))

          Term3 = (t - Key1Knots(I - 1)) * (t - Key1Knots(I - 1)) * (Key1Knots(I + 1) - t) _
              * CP1(I - 1).Y / _
              ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 1) - Key1Knots(I - 1)) _
              * (Key1Knots(I + 1) - Key1Knots(I))) + _
              (t - Key1Knots(I)) * (t - Key1Knots(I - 1)) * (Key1Knots(I + 2) - t) _
              * CP1(I - 1).Y / _
              ((Key1Knots(I + 2) - Key1Knots(I - 1)) * (Key1Knots(I + 2) - Key1Knots(I)) _
              * (Key1Knots(I + 1) - Key1Knots(I))) + _
              (Key1Knots(I + 3) - t) * (t - Key1Knots(I)) * (t - Key1Knots(I)) _
              * CP1(I - 1).Y / _
              ((Key1Knots(I + 3) - Key1Knots(I)) * (Key1Knots(I + 2) - Key1Knots(I)) _
              * (Key1Knots(I + 1) - Key1Knots(I)))

          Term4 = (t - Key1Knots(I)) * (t - Key1Knots(I)) * (t - Key1Knots(I)) * _
              CP1(I).Y / _
              ((Key1Knots(I + 3) - Key1Knots(I)) * (Key1Knots(I + 2) - Key1Knots(I)) _
              * (Key1Knots(I + 1) - Key1Knots(I)))

          TempY = Term1 + Term2 + Term3 + Term4

            picKey1.Circle (10 * TempX, picKey1.Height - (10 * TempY)), 0.2
        End If
        Next t1
        I = I + 1
      Wend
    Exit Sub
ErrHandler:
    'User pressed the Cancel button
    Exit Sub
End Sub

'*************************************************************************
Private Sub mnuOpenSpline2_Click()
' Opens a Bezier Curve in Frame 2 and converts it to a B-Spline
' Set CancelError is True
    CommonDialog1.CancelError = True
    On Error GoTo ErrHandler
    ' Set flags
    CommonDialog1.Flags = cdlOFNHideReadOnly
    ' Set filters
    CommonDialog1.Filter = "All Files (*.*)|*.*|Text Files" & _
     "(*.txt)|*.txt"
    ' Specify default filter
    CommonDialog1.FilterIndex = 2
    ' Display the Open dialog box
    CommonDialog1.ShowOpen
    ' Display name of selected file

    DrawPoly2 = False   'If we commit ourselves to opening a file in the frame, we can't
```

```
                    ' then decide to draw our own polygon (by clicking points).
Dim Fnum As Integer
Dim Temp As String
Dim I As Integer
Dim J As Boolean
DrawBezier = True
I = 0
J = True
Fnum = FreeFile
 Open CommonDialog1.filename For Input As #Fnum
Do While Not EOF(Fnum)
   Line Input #Fnum, Temp
   If J = True Then
      Key2Pts(I).X = CDbl(Temp)
      J = False
   Else
      Key2Pts(I).Y = CDbl(Temp)
     J = True
     I = I + 1
     Key2NumPts = I
   End If
Loop

 Key2Pts(Key2NumPts).X = Key2Pts(0).X
 Key2Pts(Key2NumPts).Y = Key2Pts(0).Y
I = I + 1
Dim NumExtraPts As Integer
NumExtraPts = 0
While (((Key2NumPts) Mod 3) <> 0)   ' we should have the right number of
   Key2Pts(I).X = Key2Pts(0).X       ' control points in the file, but just
   Key2Pts(I).Y = Key2Pts(0).Y       ' in case we don't, we do this
   NumExtraPts = NumExtraPts + 1
   Key2NumPts = Key2NumPts + 1
  I = I + 1
Wend
' Now convert the Bezier to a B-Spline
 ' initialize the control point list and knot vector
For I = 0 To 3 Step 1
   CP2(I).X = Key2Pts(I).X
   CP2(I).Y = Key2Pts(I).Y
   Key2Knots(I) = 0
Next I

For I = 4 To 7 Step 1
   Key2Knots(I) = 1
Next I
MaxCP2 = 3
MaxKV2 = 7
 ' Want to add the next Bezier curve control points
 ' to the list of control points
 Dim Slope1 As Double
 Dim Slope2 As Double
 Dim NewKnot As Double
```

```
Dim Continuity As Integer
Dim NumCurves As Integer
Dim NextIndex As Integer
NumCurves = ((Key2NumPts) / 3)
' This will be in some kind of loop
NextIndex = MaxCP2 + 1

While NextIndex < Key2NumPts
   If Abs(CP2(MaxCP2).X - CP2(MaxCP2 - 1).X) < Epsilon Then
      Slope1 = 32000
   Else
      Slope1 = (CP2(MaxCP2).Y - CP2(MaxCP2 - 1).Y) / _
           (CP2(MaxCP2).X - CP2(MaxCP2 - 1).X)
   End If

   If Abs(Key2Pts(NextIndex).X - CP2(MaxCP2).X) < Epsilon Then
      Slope2 = 32000
   Else
      Slope2 = (Key2Pts(NextIndex).Y - CP2(MaxCP2).Y) / _
           (Key2Pts(NextIndex).X - CP2(MaxCP2).X)
   End If

   If (Abs(Slope1 - Slope2) > Epsilon) Then  ' curve have only C0 continuity
      NewKnot = Key2Knots(MaxKV2) + 1
      Continuity = 0
   Else   ' the slope is the same, so curves have at least C1 continuity.
         ' Choose a knot value that reflects this continuity
      NewKnot = ((Key2Pts(NextIndex).X - CP2(MaxCP2).X) * _
           (Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 4)) / _
           (CP2(MaxCP2).X - CP2(MaxCP2 - 1).X)) + _
           Key2Knots(MaxKV2 - 3)
      Continuity = 1

      ' Now test to see if the curves are actually C2 continuous
      If Abs( _
          ( _
            (CP2(MaxCP2 - 2).X - Key2Pts(NextIndex + 1).X) * _
            (Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 5)) * _
            (Key2Knots(MaxKV2 - 3) - NewKnot) _
          ) _
        + _
          ( _
            (CP2(MaxCP2 - 1).X - CP2(MaxCP2 - 2).X) * _
            (NewKnot - Key2Knots(MaxKV2 - 5)) * _
            (Key2Knots(MaxKV2 - 3) - NewKnot) _
          ) _
        + _
          ( _
            (Key2Pts(NextIndex + 1).X - Key2Pts(NextIndex).X) * _
            (Key2Knots(MaxKV2 - 4) - NewKnot) * _
            (Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 5)) _
          ) _
          ) < Epsilon Then
```

```
            ' then we have C2 continuity
            Continuity = 2
        End If

    ' Now check for C3 continuity
        Dim PAlpha As Coords
        Dim PBeta As Coords
        Dim PGamma As Coords

        PAlpha.X = ((Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 4)) * Key2Pts(NextIndex +
1).X + _
                ((NewKnot - Key2Knots(MaxKV2 - 3)) * Key2Pts(NextIndex).X)) / _
                (NewKnot - Key2Knots(MaxKV2 - 4))
        PAlpha.Y = ((Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 4)) * Key2Pts(NextIndex +
1).Y + _
                ((NewKnot - Key2Knots(MaxKV2 - 3)) * Key2Pts(NextIndex).Y)) / _
                (NewKnot - Key2Knots(MaxKV2 - 4))

        PBeta.X = ((NewKnot - Key2Knots(MaxKV2 - 6)) * CP2(MaxCP2 - 2).X + _
                ((Key2Knots(MaxKV2 - 5) - NewKnot) * CP2(MaxCP2 - 3).X)) / _
                (Key2Knots(MaxKV2 - 5) - Key2Knots(MaxKV2 - 6))
        PBeta.Y = ((NewKnot - Key2Knots(MaxKV2 - 6)) * CP2(MaxCP2 - 2).Y + _
                ((Key2Knots(MaxKV2 - 5) - NewKnot) * CP2(MaxCP2 - 3).Y)) / _
                (Key2Knots(MaxKV2 - 5) - Key2Knots(MaxKV2 - 6))

        PGamma.X = ((Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 4)) * Key2Pts(NextIndex +
2).X + _
                ((Key2Knots(MaxKV2 - 4) - NewKnot) * Key2Pts(NextIndex + 1).X)) / _
                (Key2Knots(MaxKV2 - 3) - NewKnot)
        PGamma.Y = ((Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 4)) * Key2Pts(NextIndex +
2).Y + _
                ((Key2Knots(MaxKV2 - 4) - NewKnot) * Key2Pts(NextIndex + 1).Y)) / _
                (Key2Knots(MaxKV2 - 3) - NewKnot)

        If Abs(PAlpha.X - ((((NewKnot - Key2Knots(MaxKV2 - 3)) * PBeta.X) + _
                    ((Key2Knots(MaxKV2 - 3) - Key2Knots(MaxKV2 - 5)) * PGamma.X)) / _
                    (NewKnot - Key2Knots(MaxKV2 - 5)))) < Epsilon Then
            Continuity = 3
        End If
    End If
        ' Append the new knots and control points, depending on the
        ' continuity between the two curves
    Select Case Continuity
    Case 0
        Key2Knots(MaxKV2) = NewKnot
        Key2Knots(MaxKV2 + 1) = NewKnot
        Key2Knots(MaxKV2 + 2) = NewKnot
        Key2Knots(MaxKV2 + 3) = NewKnot
        MaxKV2 = MaxKV2 + 3
        CP2(MaxCP2 + 1).X = Key2Pts(MaxCP2 + 1).X
        CP2(MaxCP2 + 1).Y = Key2Pts(MaxCP2 + 1).Y
        CP2(MaxCP2 + 2).X = Key2Pts(MaxCP2 + 2).X
```

```
      CP2(MaxCP2 + 2).Y = Key2Pts(MaxCP2 + 2).Y
      CP2(MaxCP2 + 3).X = Key2Pts(MaxCP2 + 3).X
      CP2(MaxCP2 + 3).Y = Key2Pts(MaxCP2 + 3).Y
      MaxCP2 = MaxCP2 + 3
    Case 1
      Key2Knots(MaxKV2 - 1) = NewKnot
      Key2Knots(MaxKV2) = NewKnot
      Key2Knots(MaxKV2 + 1) = NewKnot
      Key2Knots(MaxKV2 + 2) = NewKnot
      MaxKV2 = MaxKV2 + 2
      CP2(MaxCP2).X = Key2Pts(MaxCP2 + 1).X
      CP2(MaxCP2).Y = Key2Pts(MaxCP2 + 1).Y
      CP2(MaxCP2 + 1).X = Key2Pts(MaxCP2 + 2).X
      CP2(MaxCP2 + 1).Y = Key2Pts(MaxCP2 + 2).Y
      CP2(MaxCP2 + 2).X = Key2Pts(MaxCP2 + 3).X
      CP2(MaxCP2 + 2).Y = Key2Pts(MaxCP2 + 3).Y
      MaxCP2 = MaxCP2 + 2
    Case 2
      Key2Knots(MaxKV2 - 2) = NewKnot
      Key2Knots(MaxKV2 - 1) = NewKnot
      Key2Knots(MaxKV2) = NewKnot
      Key2Knots(MaxKV2 + 1) = NewKnot
      MaxKV2 = MaxKV2 + 1
      CP2(MaxCP2 - 1).X = PAlpha.X
      CP2(MaxCP2 - 1).Y = PAlpha.Y
      CP2(MaxCP2).X = Key2Pts(MaxCP2 + 2).X
      CP2(MaxCP2).Y = Key2Pts(MaxCP2 + 2).Y
      CP2(MaxCP2 + 1).X = Key2Pts(MaxCP2 + 3).X
      CP2(MaxCP2 + 1).Y = Key2Pts(MaxCP2 + 3).Y
      MaxCP2 = MaxCP2 + 1
    Case 3
      Key2Knots(MaxKV2 - 3) = NewKnot
      Key2Knots(MaxKV2 - 2) = NewKnot
      Key2Knots(MaxKV2 - 1) = NewKnot
      Key2Knots(MaxKV2) = NewKnot
      MaxKV2 = MaxKV2
      CP2(MaxCP2 - 2).X = PBeta.X
      CP2(MaxCP2 - 2).Y = PBeta.Y
      CP2(MaxCP2 - 1).X = PGamma.X
      CP2(MaxCP2 - 1).Y = PGamma.Y
      CP2(MaxCP2).X = Key2Pts(MaxCP2 + 3).X
      CP2(MaxCP2).Y = Key2Pts(MaxCP2 + 3).Y
      MaxCP2 = MaxCP2
    End Select
    NextIndex = NextIndex + 3
  Wend

  For I = 1 To (MaxCP2 + 1) Step 1
     ' mark the control points
     picKey2.Circle (10 * CP2(I - 1).X, picKey2.Height - 10 * CP2(I - 1).Y), _
              1, RGB(0, 255, 0)
  Next I
```

```
Dim tI As Integer
Dim t As Double
Dim TempX As Double
Dim TempY As Double

' Draw the curve
I = 3
While I <= MaxCP2
   For tI = (200 * Key2Knots(I)) To (200 * Key2Knots(I + 1)) Step 1
     If (Key2Knots(I) <> Key2Knots(I + 1)) Then
      'calculate and plot the points of the b-spline
       t = tI / (200 * (Key2Knots(I + 1) - Key2Knots(I)))
      Dim Term1 As Double
      Dim Term2 As Double
      Dim Term3 As Double
      Dim Term4 As Double
      Term1 = (Key2Knots(I + 1) - t) * (Key2Knots(I + 1) - t) * (Key2Knots(I + 1) - t) _
          * CP2(I - 3).X / _
          ((Key2Knots(I + 1) - Key2Knots(I - 2)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
           * (Key2Knots(I + 1) - Key2Knots(I)))

      Term2 = (t - Key2Knots(I - 2)) * (Key2Knots(I + 1) - t) * (Key2Knots(I + 1) - t) _
           * CP2(I - 2).X / _
            ((Key2Knots(I + 1) - Key2Knots(I - 2)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
           * (Key2Knots(I + 1) - Key2Knots(I))) + _
           (Key2Knots(I + 2) - t) * (t - Key2Knots(I - 1)) * (Key2Knots(I + 1) - t) _
          * CP2(I - 2).X / _
           ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
           * (Key2Knots(I + 1) - Key2Knots(I))) + _
           (Key2Knots(I + 2) - t) * (t - Key2Knots(I)) * (Key2Knots(I + 2) - t) _
          * CP2(I - 2).X / _
           ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 2) - Key2Knots(I)) _
           * (Key2Knots(I + 1) - Key2Knots(I)))

      Term3 = (t - Key2Knots(I - 1)) * (t - Key2Knots(I - 1)) * (Key2Knots(I + 1) - t) _
          * CP2(I - 1).X / _
           ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
           * (Key2Knots(I + 1) - Key2Knots(I))) + _
           (t - Key2Knots(I)) * (t - Key2Knots(I - 1)) * (Key2Knots(I + 2) - t) _
          * CP2(I - 1).X / _
           ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 2) - Key2Knots(I)) _
           * (Key2Knots(I + 1) - Key2Knots(I))) + _
           (Key2Knots(I + 3) - t) * (t - Key2Knots(I)) * (t - Key2Knots(I)) _
          * CP2(I - 1).X / _
           ((Key2Knots(I + 3) - Key2Knots(I)) * (Key2Knots(I + 2) - Key2Knots(I)) _
           * (Key2Knots(I + 1) - Key2Knots(I)))

      Term4 = (t - Key2Knots(I)) * (t - Key2Knots(I)) * (t - Key2Knots(I)) * _
          CP2(I).X / _
           ((Key2Knots(I + 3) - Key2Knots(I)) * (Key2Knots(I + 2) - Key2Knots(I)) _
           * (Key2Knots(I + 1) - Key2Knots(I)))
```

```
        TempX = Term1 + Term2 + Term3 + Term4

        Term1 = (Key2Knots(I + 1) - t) * (Key2Knots(I + 1) - t) * (Key2Knots(I + 1) - t) _
            * CP2(I - 3).Y / _
             ((Key2Knots(I + 1) - Key2Knots(I - 2)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
             * (Key2Knots(I + 1) - Key2Knots(I)))

        Term2 = (t - Key2Knots(I - 2)) * (Key2Knots(I + 1) - t) * (Key2Knots(I + 1) - t) _
            * CP2(I - 2).Y / _
             ((Key2Knots(I + 1) - Key2Knots(I - 2)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
             * (Key2Knots(I + 1) - Key2Knots(I))) + _
             (Key2Knots(I + 2) - t) * (t - Key2Knots(I - 1)) * (Key2Knots(I + 1) - t) _
            * CP2(I - 2).Y / _
             ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
             * (Key2Knots(I + 1) - Key2Knots(I))) + _
             (Key2Knots(I + 2) - t) * (t - Key2Knots(I)) * (Key2Knots(I + 2) - t) _
            * CP2(I - 2).Y / _
             ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 2) - Key2Knots(I)) _
             * (Key2Knots(I + 1) - Key2Knots(I)))

        Term3 = (t - Key2Knots(I - 1)) * (t - Key2Knots(I - 1)) * (Key2Knots(I + 1) - t) _
            * CP2(I - 1).Y / _
             ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 1) - Key2Knots(I - 1)) _
             * (Key2Knots(I + 1) - Key2Knots(I))) + _
             (t - Key2Knots(I)) * (t - Key2Knots(I - 1)) * (Key2Knots(I + 2) - t) _
            * CP2(I - 1).Y / _
             ((Key2Knots(I + 2) - Key2Knots(I - 1)) * (Key2Knots(I + 2) - Key2Knots(I)) _
             * (Key2Knots(I + 1) - Key2Knots(I))) + _
             (Key2Knots(I + 3) - t) * (t - Key2Knots(I)) * (t - Key2Knots(I)) _
            * CP2(I - 1).Y / _
             ((Key2Knots(I + 3) - Key2Knots(I)) * (Key2Knots(I + 2) - Key2Knots(I)) _
             * (Key2Knots(I + 1) - Key2Knots(I)))

        Term4 = (t - Key2Knots(I)) * (t - Key2Knots(I)) * (t - Key2Knots(I)) * _
            CP2(I).Y / _
             ((Key2Knots(I + 3) - Key2Knots(I)) * (Key2Knots(I + 2) - Key2Knots(I)) _
             * (Key2Knots(I + 1) - Key2Knots(I)))

        TempY = Term1 + Term2 + Term3 + Term4
         picKey2.Circle (10 * TempX, picKey2.Height - (10 * TempY)), 0.2
      End If
      Next t1
      I = I + 1
    Wend
  Exit Sub
ErrHandler:
  'User pressed the Cancel button
  Exit SubEnd Sub


'*********************************************************************
```

```
Private Sub mnuSet1_Click()
' Parameter Test Set 1
    ' Set the values in the text boxes
    txtkb.Text = "2"    ' bending stiffness
    txtmb.Text = "100"    ' penalizes non-monotonic angles
   txteb.Text = "1"     '
    txtpb.Text = "10000"    ' penalizes angles from going to 0
    txtks.Text = "0.1"       ' stretching stiffness constant
    txtcs.Text = "0.1"       ' controls penalty for edge collapsing to a point
   txtes.Text = "2"       '
    ' Get the values from the text boxes
    kb = CDbl(txtkb.Text)
    mb = CDbl(txtmb.Text)
    eb = Val(txteb.Text)
    pb = Val(txtpb.Text)
    ks = CDbl(txtks.Text)
    cs = CDbl(txtcs.Text)
    es = Val(txtes.Text)
End Sub

'*********************************************************************
Private Sub mnuSet2_Click()
' Parameter Test Set 2    ' Set the values in the text boxes
    txtkb.Text = "0.1"    ' bending stiffness
    txtmb.Text = "1"    ' penalizes non-monotonic angles
   txteb.Text = "1"     '
    txtpb.Text = "10000"    ' penalizes angles from going to 0
    txtks.Text = "1"       ' stretching stiffness constant
    txtcs.Text = "0.1"       ' controls penalty for edge collapsing to a point
   txtes.Text = "1"       '
    ' Get the value from the text boxes
    kb = CDbl(txtkb.Text)
    mb = CDbl(txtmb.Text)
    eb = Val(txteb.Text)
    pb = Val(txtpb.Text)
    ks = CDbl(txtks.Text)
    cs = CDbl(txtcs.Text)
    es = Val(txtes.Text)
End Sub

'*********************************************************************
Private Sub mnuSet3_Click()
' Parameter Test Set 3    ' Set the values in the text boxes
    txtkb.Text = "0.1"    ' bending stiffness
    txtmb.Text = "1"    ' penalizes non-monotonic angles
   txteb.Text = "1"     '
    txtpb.Text = "10000"    ' penalizes angles from going to 0
    txtks.Text = "0.1"       ' stretching stiffness constant
    txtcs.Text = "0.1"       ' controls penalty for edge collapsing to a point
   txtes.Text = "2"       '
    ' Get the values from the text boxes
    kb = CDbl(txtkb.Text)
    mb = CDbl(txtmb.Text)
```

```
    eb = Val(txteb.Text)
    pb = Val(txtpb.Text)
    ks = CDbl(txtks.Text)
    cs = CDbl(txtcs.Text)
    es = Val(txtes.Text)
End Sub

'*************************************************************************
Private Sub picKey1_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
' Draws a line from the previous clicked point to the current clicked point.
   If DrawPoly1 = True Then       ' Only allowed to draw a polygon if we haven't already
                                  ' opened a polygon in the frame
      Y = picKey1.Height - Y      ' Switch the coordinate system so that
                                  ' y increases up
      If Key1FirstClick = True Then
         Key1FirstClick = False
         Key1Pts(Key1NumPts).X = X / 10     ' Decrease the values to avoid overflow error.
         Key1Pts(Key1NumPts).Y = Y / 10     ' They will be increased back when we draw (in
                                            ' the vertex path routines).
         Key1NumPts = Key1NumPts + 1
          picKey1.Circle (X, picKey1.Height - Y), 1, RGB(0, 0, 255)
      Else
         ' When the point is drawn, must switch the coordinate
         ' system back so that the point is drawn in the correct place
         picKey1.Line (10 * Key1Pts(Key1NumPts - 1).X, _
               picKey1.Height - (10 * Key1Pts(Key1NumPts - 1).Y)) _
              -(X, picKey1.Height - Y)
         Key1Pts(Key1NumPts).X = X / 10
         Key1Pts(Key1NumPts).Y = Y / 10
         Key1NumPts = Key1NumPts + 1
          picKey1.Circle (X, picKey1.Height - Y), 1, RGB(0, 0, 255)
      End If
   End If
End Sub

'*************************************************************************
Private Sub picKey2_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
  If DrawPoly2 = True Then     ' Only allowed to draw a polygon if we haven't already
                               ' opened a polygon in the frame
    Y = picKey2.Height - Y     ' Switch the coordinate system so that
                               ' y increases up
   If Key2FirstClick = True Then
      Key2FirstClick = False
      Key2Pts(Key2NumPts).X = X / 10     ' Decrease the values to avoid overflow. Increase
      Key2Pts(Key2NumPts).Y = Y / 10     ' them back when we draw (in vertex path
                                 ' routines).
     Key2NumPts = Key2NumPts + 1
      picKey2.Circle (X, picKey2.Height - Y), 1, RGB(0, 0, 255)
   Else
      ' When the point is drawn, must switch the coordinate
      ' system back so that the point is drawn in the correct place
      picKey2.Line (10 * Key2Pts(Key2NumPts - 1).X, _
            picKey2.Height - (10 * Key2Pts(Key2NumPts - 1).Y))- _
```

```
            (X, picKey2.Height - Y)
      Key2Pts(Key2NumPts).X = X / 10
      Key2Pts(Key2NumPts).Y = Y / 10
      Key2NumPts = Key2NumPts + 1
       picKey2.Circle (X, picKey2.Height - Y), 1, RGB(0, 0, 255)
   End If
  End If
End Sub

'*********************************************************************
Public Sub MatchByOrder1()
' A vertex correspondence method. Matches the vertices based on the order in which they
' are clicked If one polygon has more vertices than the other, the additional vertices in the
' polygon with more vertices are all mapped to the final vertex of the polygon with fewer
' vertices.
' N.B. Definitely not the spiffiest vertex correspondence plan, but certainly one
'      of the simplest. I just coded this for testing purposes.
   Dim I As Integer
   NumPts = Key2NumPts
   If Key1NumPts > Key2NumPts Then   ' if polygon 1 has more vertices than polygon
                          ' 2, then map the extra points of polygon 1
                          ' to the last vertex of polygon 2
      NumPts = Key1NumPts
      For I = Key2NumPts To (Key1NumPts) Step 1
         Key2Pts(I).X = Key2Pts(Key2NumPts).X
         Key2Pts(I).Y = Key2Pts(Key2NumPts).Y
      Next I
   End If
   If Key2NumPts > Key1NumPts Then
         NumPts = Key2NumPts
         For I = Key1NumPts To (Key2NumPts) Step 1
            Key1Pts(I).X = Key1Pts(Key1NumPts).X
            Key1Pts(I).Y = Key1Pts(Key1NumPts).Y
         Next I
   End If
End Sub

'*********************************************************************
Public Sub LinearInterpolation()
' A Vertex Path Method. Takes a pair of corresponding vertices and uses linear
' interpolation to calculate the path travelled by a vertex  as it morphs from one polygon
'  into the other.

   Dim I As Integer
   Dim J As Integer
   Dim Draw As Boolean
   Dim TempPic As PictureBox
   Draw = False    ' we only draw the in-between frames. We don't want
            ' to redraw the key frames.
   ' Calculate the step size to increment each of the x- and y- coords
   ' for each successive in-between image
   For I = 0 To (NumPts) Step 1
      KeyDifference(I).X = (Key2Pts(I).X - Key1Pts(I).X) / (NumInBetweens + 1)
```

```
    KeyDifference(I).Y = (Key2Pts(I).Y - Key1Pts(I).Y) / (NumInBetweens + 1)
  Next I

  For I = 0 To (NumInBetweens + 1) Step 1
    ' Determine the vertices for in-between frame I
    For J = 0 To (NumPts) Step 1
      MorphPts(J).X = Key1Pts(J).X + ((KeyDifference(J).X) * I)
      MorphPts(J).Y = Key1Pts(J).Y + ((KeyDifference(J).Y) * I)
    Next J

    ' Draw the lines in the appropriate picture box
    ' Note that the coord system is switched back for drawing
    Select Case I
      Case 1
        Set TempPic = picMorph1
        Draw = True
      Case 2
        Set TempPic = picMorph2
        Draw = True
      Case 3
        Set TempPic = picMorph3
        Draw = True
      Case 4
        Set TempPic = picMorph4
        Draw = True
      Case 5
        Set TempPic = picMorph5
        Draw = True
    End Select
    If Draw Then
      For J = 1 To (NumPts) Step 1
        TempPic.Line (10 * MorphPts(J).X, _
              TempPic.Height - (10 * MorphPts(J).Y)) _
              -(10 * MorphPts(J - 1).X, _
              TempPic.Height - (10 * MorphPts(J - 1).Y))
      Next J
    End If
    Draw = False
  Next I
End Sub

'*************************************************************************
Public Sub LeastWorkMatching()
' Determines the vertex correspondence between the two key frames that will result in
'  the least amount of work to morph from one image to the other.
' This method considers the polygon edges to be made of bendable, stretchable wire,
'  and determines the work need to stretch and bend the wire edges of polygon 1 into
'  polygon 2.
' Uses a graph theory solution to determine the least work "path" and then does a
' back track through this "graph" to find the least work matching.


  ' BackTrackList keep track of the graph vertices (I,J) that correspond to one
```

```
' another on the least work path. BackTrackList is defined as type "Coords",
' but is not really made of polygon vertex coordinates. Rather, the (X,Y)
' coordinates are actually the (I,J) vertices of the least work graph.
Dim BackTrackList() As Coords
If Key2NumPts > Key1NumPts Then
  NumPts = Key2NumPts
Else
  NumPts = Key1NumPts
End If
ReDim BackTrackList(Key1NumPts + Key2NumPts) As Coords

' WBack keeps track of the amount of work required to get to
' graph vertex (I,J) from the graph vertex (I-1,J)
Dim WBack() As Integer
ReDim WBack(Key1NumPts, Key2NumPts) As Integer

' WUp keeps track of the amount of work required to get
' to graph vertex (I,J) from the graph vertex (I,J-1)
Dim WUp() As Integer
ReDim WUp(Key1NumPts, Key2NumPts) As Integer

' WDiag keeps track of the amount of work required to get
' to graph vertex (I,J) from the graph vertex (I-1,J-1)
Dim WDiag() As Integer
ReDim WDiag(Key1NumPts, Key2NumPts) As Integer

Dim I As Integer
Dim J As Integer
' The polygon files should be stored carefully, since the program
' automatically matches the first vertices to each other.
WBack(0, 0) = 0
WUp(0, 0) = 0
WDiag(0, 0) = 0

For I = 0 To (Key1NumPts) Step 1
  For J = 0 To (Key2NumPts) Step 1
  'Note: If I=0 and J<>0 then we can only calculate WUp
    If I = 0 And J = 1 Then
      WUp(I, J) = Stretch(Key1Pts(I), Key2Pts(J - 1), _
                  Key1Pts(I), Key2Pts(J)) + _
            Minimum(WUp(I, J - 1) + Bend(Key1Pts(I), _
                Key2Pts(Key2NumPts - 1), _
                Key1Pts(I), Key2Pts(J - 1), _
                Key1Pts(I), Key2Pts(J)), _
_
                WDiag(I, J - 1) + _
                  Bend(Key1Pts(Key1NumPts - 1), _
                  Key2Pts(Key2NumPts - 1), _
                  Key1Pts(I), Key2Pts(J - 1), _
                  Key1Pts(I), Key2Pts(J)))

      WBack(I, J) = 15000    ' Initialize WBack and WDiag to some
      WDiag(I, J) = 15000    ' large number so that we don't think
```

```
                    ' that a vertex matching that doesn't
                    ' exist is actually the least work matching
    End If

    If I = 0 And J > 1 Then
        WUp(I, J) = Stretch(Key1Pts(I), Key2Pts(J - 1), _
                Key1Pts(I), Key2Pts(J)) + _
            Minimum(WUp(I, J - 1) + Bend(Key1Pts(I), _
                Key2Pts(J - 2), _
                Key1Pts(I), Key2Pts(J - 1), _
                Key1Pts(I), Key2Pts(J)), _
_
                WDiag(I, J - 1) + _
                Bend(Key1Pts(Key1NumPts - 1), _
                Key2Pts(J - 2), _
                Key1Pts(I), Key2Pts(J - 1), _
                Key1Pts(I), Key2Pts(J)))
        WBack(I, J) = 15000    ' Init WBack and WDiag to some large
        WDiag(I, J) = 15000    ' number so we don't think that a non-
                    ' existent vertex matching is the least
                    ' work matching
    End If

'   Also, if I<>0 and J=0 then we can only calculate WBack
    If I = 1 And J = 0 Then
        WBack(I, J) = Stretch(Key1Pts(I - 1), Key2Pts(J), _
                Key1Pts(I), Key2Pts(J)) + _
            Minimum(WBack(I - 1, J) + _
                Bend(Key1Pts(Key1NumPts - 1), _
                Key2Pts(J), Key1Pts(I - 1), _
                Key2Pts(J), Key1Pts(I), _
                Key2Pts(J)), _
_
                WDiag(I - 1, J) + _
                Bend(Key1Pts(Key1NumPts - 1), _
                Key2Pts(Key2NumPts - 1), _
                Key1Pts(I - 1), Key2Pts(J), _
                Key1Pts(I), Key2Pts(J)))

        WUp(I, J) = 15000
        WDiag(I, J) = 15000
    End If
    If I > 1 And J = 0 Then
        WBack(I, J) = Stretch(Key1Pts(I - 1), Key2Pts(J), _
                Key1Pts(I), Key2Pts(J)) + _
            Minimum(WBack(I - 1, J) + _
                Bend(Key1Pts(I - 2), _
                Key2Pts(J), Key1Pts(I - 1), _
                Key2Pts(J), Key1Pts(I), _
                Key2Pts(J)), _
_
                WDiag(I - 1, J) + _
                Bend(Key1Pts(I - 2), _
```

```
                    Key2Pts(Key2NumPts - 1), _
                    Key1Pts(I - 1), Key2Pts(J), _
                    Key1Pts(I), Key2Pts(J)))
        WUp(I, J) = 15000
        WDiag(I, J) = 15000
    End If
    If I = 1 And J = 1 Then
        WBack(I, J) = Stretch(Key1Pts(I - 1), Key2Pts(J), _
                Key1Pts(I), Key2Pts(J)) + _
_
            Minimum( _
              WBack(I - 1, J) + _
                Bend(Key1Pts(Key1NumPts - 1), _
                 Key2Pts(J), Key1Pts(I - 1), _
                 Key2Pts(J), Key1Pts(I), _
                 Key2Pts(J)), _
_
              WDiag(I - 1, J) + _
                Bend(Key1Pts(Key1NumPts - 1), _
                 Key2Pts(J - 1), Key1Pts(I - 1), _
                 Key2Pts(J), Key1Pts(I), _
                 Key2Pts(J)) _
                 ) ' end of Minimum parameters

        WUp(I, J) = Stretch(Key1Pts(I), Key2Pts(J - 1), _
                Key1Pts(I), Key2Pts(J)) + _
_
            Minimum( _
              WUp(I, J - 1) + _
              Bend(Key1Pts(I), _
                 Key2Pts(Key2NumPts - 1), _
                 Key1Pts(I), Key2Pts(J - 1), _
                 Key1Pts(I), Key2Pts(J)), _
_
              WDiag(I, J - 1) + _
              Bend(Key1Pts(I - 1), _
                 Key2Pts(Key2NumPts - 1), _
                 Key1Pts(I), Key2Pts(J - 1), _
                 Key1Pts(I), Key2Pts(J)) _
                ) ' end of Minimum Parameters

        WDiag(I, J) = Stretch(Key1Pts(I - 1), Key2Pts(J - 1), _
                Key1Pts(I), Key2Pts(J)) + _
_
            MinOf3( _
              WUp(I - 1, J - 1) + _
              Bend(Key1Pts(I - 1), _
                 Key2Pts(Key2NumPts - 1), _
                 Key1Pts(I - 1), Key2Pts(J - 1), _
                 Key1Pts(I), Key2Pts(J)), _
_
              WDiag(I - 1, J - 1) + _
              Bend(Key1Pts(Key1NumPts - 1), _
```

```
                    Key2Pts(Key2NumPts - 1), _
                    Key1Pts(I - 1), Key2Pts(J - 1), _
                    Key1Pts(I), Key2Pts(J)), _
-
                WBack(I - 1, J - 1) + _
                Bend(Key1Pts(Key1NumPts - 1), _
                    Key2Pts(J - 1), Key1Pts(I - 1), _
                    Key2Pts(J - 1), Key1Pts(I), _
                   Key2Pts(J)) _
                 )  'end of MinOf3 Paremeters
    End If
    If I > 1 And J > 1 Then
        WBack(I, J) = Stretch(Key1Pts(I - 1), Key2Pts(J), _
                    Key1Pts(I), Key2Pts(J)) + _
-
                Minimum( _
                  WBack(I - 1, J) + _
                    Bend(Key1Pts(I - 2), _
                     Key2Pts(J), Key1Pts(I - 1), _
                     Key2Pts(J), Key1Pts(I), _
                     Key2Pts(J)), _
-
                  WDiag(I - 1, J) + _
                    Bend(Key1Pts(I - 2), _
                      Key2Pts(J - 1), Key1Pts(I - 1), _
                      Key2Pts(J), Key1Pts(I), _
                     Key2Pts(J)) _
                     ) 'end of Minimum parameters
        WUp(I, J) = Stretch(Key1Pts(I), Key2Pts(J - 1), _
                    Key1Pts(I), Key2Pts(J)) + _
-
                Minimum( _
                  WUp(I, J - 1) + _
                  Bend(Key1Pts(I), _
                     Key2Pts(J - 2), _
                     Key1Pts(I), Key2Pts(J - 1), _
                     Key1Pts(I), Key2Pts(J)), _
-
                  WDiag(I, J - 1) + _
                  Bend(Key1Pts(I - 1), _
                     Key2Pts(J - 2), _
                      Key1Pts(I), Key2Pts(J - 1), _
                      Key1Pts(I), Key2Pts(J)) _
                    ) 'end of Minimum Parameters
        WDiag(I, J) = Stretch(Key1Pts(I - 1), Key2Pts(J - 1), _
                    Key1Pts(I), Key2Pts(J)) + _
-
                MinOf3( _
                  WUp(I - 1, J - 1) + _
                  Bend(Key1Pts(I - 1), _
                     Key2Pts(J - 2), _
                      Key1Pts(I - 1), Key2Pts(J - 1), _
                      Key1Pts(I), Key2Pts(J)), _
```

```
                    WDiag(I - 1, J - 1) + _
                    Bend(Key1Pts(I - 2), _
                        Key2Pts(J - 2), _
                        Key1Pts(I - 1), Key2Pts(J - 1), _
                        Key1Pts(I), Key2Pts(J)), _

                    WBack(I - 1, J - 1) + _
                    Bend(Key1Pts(I - 2), _
                        Key2Pts(J - 1), Key1Pts(I - 1), _
                        Key2Pts(J - 1), Key1Pts(I), _
                        Key2Pts(J)) _
                    )   ' end of MinOf3 Paremeters
        End If
    Next J
Next I

' now backtrack to find the path.
' The first point of BackTrackList is the final graph vertex
' i.e. [Key1NumPts-1, Key2NumPts-1] (note that the very last vertices
' (the duplicate first points that close up the polygons) are automatically
' matched to each other
BackTrackList(0).X = Key1NumPts - 1
BackTrackList(0).Y = Key2NumPts - 1
Dim TempX As Integer
Dim TempY As Integer
Dim CurrKey1Pt As Integer
Dim CurrKey2Pt As Integer
Dim NumBackTrackPts As Integer
NumBackTrackPts = 1
CurrKey1Pt = BackTrackList(0).X
CurrKey2Pt = BackTrackList(0).Y
I = 1
' Find the previous graph vertex that requires the least amount of work, and
' choose that one as the next vertex in the backtrack list.
Do While (CurrKey1Pt >= 0) And (CurrKey2Pt >= 0)
    TempX = BackTrackList(I - 1).X
    TempY = BackTrackList(I - 1).Y
    If WBack(TempX, TempY) <= WUp(TempX, TempY) And _
        WBack(TempX, TempY) <= WDiag(TempX, TempY) Then
            CurrKey1Pt = BackTrackList(I - 1).X - 1
            CurrKey2Pt = BackTrackList(I - 1).Y
    End If
    If WUp(TempX, TempY) < WBack(TempX, TempY) And _
        WUp(TempX, TempY) <= WDiag(TempX, TempY) Then
            CurrKey1Pt = BackTrackList(I - 1).X
            CurrKey2Pt = BackTrackList(I - 1).Y - 1
    End If
    If WDiag(TempX, TempY) < WBack(TempX, TempY) And _
        WDiag(TempX, TempY) < WUp(TempX, TempY) Then
            CurrKey1Pt = BackTrackList(I - 1).X - 1
            CurrKey2Pt = BackTrackList(I - 1).Y - 1
    End If
```

```
      BackTrackList(I).X = CurrKey1Pt
      BackTrackList(I).Y = CurrKey2Pt
      NumBackTrackPts = NumBackTrackPts + 1
      I = I + 1
      If CurrKey1Pt = 0 And CurrKey2Pt = 0 Then
        Exit Do
      End If
    Loop
    Dim TempKey1Pts() As Coords
    ReDim TempKey1Pts(Key1NumPts + Key2NumPts) As Coords
    Dim TempKey2Pts() As Coords
    ReDim TempKey2Pts(Key1NumPts + Key2NumPts) As Coords
    ' copy the points in reverse order into a new list
    For I = 0 To (NumBackTrackPts - 1) Step 1
      TempKey1Pts(NumBackTrackPts - 1 - I).X = Key1Pts(BackTrackList(I).X).X
      TempKey1Pts(NumBackTrackPts - 1 - I).Y = Key1Pts(BackTrackList(I).X).Y
      TempKey2Pts(NumBackTrackPts - 1 - I).X = Key2Pts(BackTrackList(I).Y).X
      TempKey2Pts(NumBackTrackPts - 1 - I).Y = Key2Pts(BackTrackList(I).Y).Y
    Next I
    ' And reassign these points to the old list of vertices.
    ' Now, Key1Pts(k) corresponds to Key2Pts(k).
    ' Note that Key1Pts and Key2Pts now contain the same number of
    ' vertices
    For I = 0 To (NumBackTrackPts - 1) Step 1
      Key1Pts(I).X = TempKey1Pts(I).X
      Key1Pts(I).Y = TempKey1Pts(I).Y
      Key2Pts(I).X = TempKey2Pts(I).X
      Key2Pts(I).Y = TempKey2Pts(I).Y
    Next I

    ' Make sure the polygon is closed
    Key1Pts(NumBackTrackPts).X = Key1Pts(0).X
    Key1Pts(NumBackTrackPts).Y = Key1Pts(0).Y
    Key2Pts(NumBackTrackPts).X = Key2Pts(0).X
    Key2Pts(NumBackTrackPts).Y = Key2Pts(0).Y
    ' Need a new NumPts, since now many vertices may have been
    ' duplicated, so we now may have more vertices than before
    NumPts = NumBackTrackPts
End Sub

'*************************************************************************Private
Function Length(P0 As Coords, P1 As Coords) As Double
' Accepts two 2D points as input.
' Calculates and returns the Euclidean distance between the two points
    Length = Sqr(((P1.X - P0.X) *.(P1.X - P0.X)) + ((P1.Y - P0.Y) * (P1.Y - P0.Y)))
End Function

'*************************************************************************Private
Function Bend(I0 As Coords, J0 As Coords, I1 As Coords, J1 As Coords, I2 As Coords, J2 As
Coords)
' Accepts six 2D points. Calculates and returns the amount of bending work required to
' convert the line segments I0-I1-I2 to the line segments J0-J1-J2
```

```
Dim F0 As Coords        ' Vector from I1 to I2
Dim F1 As Coords        ' Vector from J1 to J2
Dim B0 As Coords        ' Vector from I0 to I1
Dim B1 As Coords        ' Vector from J0 to J1
Dim X0 As Double        ' determined from the above vectors,
Dim X1 As Double        ' and used as coordinates of the control
Dim X2 As Double        ' points of a Bezier curve of degree 2.
Dim Y0 As Double
Dim Y1 As Double
Dim Y2 As Double
Dim Q0 As Coords        ' Control points of a Bezier curve of
Dim Q1 As Coords        ' degree 2.
Dim Q2 As Coords
Dim D0 As Double
Dim D1 As Double
Dim D2 As Double
Dim DeltaTheta As Double        ' Change in angle from polygon 1 to
                        ' polygon 2, in radians
Dim DeltaThetaStar As Double    ' Deviation from monotonicity, in radians
Dim Origin As Coords
Dim PosXAxis As Coords
Dim NegXAxis As Coords
Dim PosYAxis As Coords
Dim NegYAxis As Coords
Dim Alpha As Double             ' the angle of deviation (if any) of Q2
Dim Beta As Double              ' the angle of deviation (if any) of Q0

Origin.X = 0
Origin.Y = 0
PosXAxis.X = 1
PosXAxis.Y = 0
NegXAxis.X = -1
NegXAxis.Y = 0
PosYAxis.X = 0
PosYAxis.Y = 1
NegYAxis.X = 0
NegYAxis.Y = -1

F0.X = I2.X - I1.X
F0.Y = I2.Y - I1.Y
F1.X = J2.X - J1.X
F1.Y = J2.Y - J1.Y
B0.X = I0.X - I1.X
B0.Y = I0.Y - I1.Y
B1.X = J0.X - J1.X
B1.Y = J0.Y - J1.Y
X0 = Dot2D(F0, B0)
X1 = (Dot2D(F1, B0) + Dot2D(F0, B1)) / 2
X2 = Dot2D(F1, B1)
Y0 = Cross2D(F0, B0)
Y1 = (Cross2D(F1, B0) + Cross2D(F0, B1)) / 2
Y2 = Cross2D(F1, B1)
```

```
Q0.X = X0
Q0.Y = Y0
Q1.X = X1
Q1.Y = Y1
Q2.X = X2
Q2.Y = Y2
D0 = Cross2D(Q0, Q1)
D1 = Cross2D(Q0, Q2) / 2
D2 = Cross2D(Q1, Q2)

If (D1 * D1 - D0 * D2) < 0 And TriangleContainsOrigin(Q0, Q1, Q2) Then
   DeltaTheta = 2 * PI - Abs(AngleFromXAxis(Q0) - AngleFromXAxis(Q2))
Else
   DeltaTheta = Abs(AngleFromXAxis(Q0) - AngleFromXAxis(Q2))
End If

Alpha = 0      ' If the angle changes non-monotonically, we must
Beta = 0       'determine how far away we are from non-monotonicity.
          ' Alpha represents the angle of deviation (if any) of Q2 and Beta
          ' represent the angle of deviation (if  any) of Q0.  Alpha + Beta
          ' gives the total amount o fdeviation, and is called DeltaThetaStar

Dim t1 As Integer
Dim t As Double
Dim QtX As Double
Dim QtY As Double

' Below, we find the amount of deviation (if any) from the monotonicity.
' Also, Theta goes to zero if and only if Q(t) crosses the positive x-axis, so we
' take the opportunity to figure this out at the same time.

Dim ThetaGoesToZero As Boolean
Dim OneSide As Boolean
Dim OtherSide As Boolean
Dim ListOfAngles(100) As Double    ' in rads
Dim Qt(100) As Coords
Dim TCross As Integer
ThetaGoesToZero = False
OneSide = False
OtherSide = False
For t1 = 0 To 100 Step 1
   t = t1 / 100
   QtX = Q0.X * (1 - t) * (1 - t) + Q1.X * (1 - t) * 2 * t + Q2.X * t * t
   QtY = Q0.Y * (1 - t) * (1 - t) + Q1.Y * (1 - t) * 2 * t + Q2.Y * t * t
   Qt(t1).X = QtX
   Qt(t1).Y = QtY
   ListOfAngles(t1) = AngleFromXAxis(Qt(t1))       If t1 > 0 Then
      If Qt(t1).X > 0 And Qt(t1 - 1).X > 0 Then
         If SignOf(Qt(t1).Y) <> SignOf(Qt(t1 - 1).Y) Then
            TCross = t1
         End If
      End If
   End If
```

```
  If QtX > 0 And QtY < 0 Then
    OneSide = True
  End If
  If QtX > 0 And QtY >= 0 Then
    OtherSide = True
  End If
Next t1

If OneSide = True And OtherSide = True Then
  ThetaGoesToZero = True
End If

Dim TMinAngle As Double
Dim TMaxAngle As Double
TMinAngle = 0
TMaxAngle = 0
If ThetaGoesToZero Then
   If ListOfAngles(100) > ListOfAngles(0) Then
     For t1 = 0 To TCross Step 1
       If ListOfAngles(t1) > ListOfAngles(TMaxAngle) Then
         TMaxAngle = t1
       End If
     Next t1
     For t1 = TCross To 100 Step 1
       If ListOfAngles(t1) < ListOfAngles(TMinAngle) Then
         TMinAngle = t1
       End If
     Next t1
     Alpha = ListOfAngles(TMaxAngle) - ListOfAngles(0)
     Beta = ListOfAngles(100) - ListOfAngles(TMinAngle)
   Else   ' if ListOfAngles(100) < ListOfAngles(0)
     For t1 = 0 To TCross Step 1
       If ListOfAngles(t1) < ListOfAngles(TMinAngle) Then
         TMinAngle = t1
       End If
     Next t1
     For t1 = TCross To 100 Step 1
       If ListOfAngles(t1) > ListOfAngles(TMaxAngle) Then
         TMaxAngle = t1
       End If
     Next t1
     Alpha = ListOfAngles(TMaxAngle) - ListOfAngles(100)
     Beta = ListOfAngles(0) - ListOfAngles(TMinAngle)
  End If
Else
   For t1 = 0 To 100 Step 1
     If ListOfAngles(t1) > ListOfAngles(TMaxAngle) Then
       TMaxAngle = t1
     End If
     If ListOfAngles(t1) < ListOfAngles(TMinAngle) Then
       TMinAngle = t1
     End If
   Next t1
```

```
    If ListOfAngles(100) > ListOfAngles(0) Then
      Alpha = ListOfAngles(TMaxAngle) - ListOfAngles(100)
      Beta = ListOfAngles(0) - ListOfAngles(TMinAngle)
    Else
      Alpha = ListOfAngles(100) - ListOfAngles(TMinAngle)
      Beta = ListOfAngles(TMaxAngle) - ListOfAngles(0)
    End If
  End If
  DeltaThetaStar = Alpha + Beta   If ThetaGoesToZero = False Then
    If Abs(DeltaTheta + mb * DeltaThetaStar) < Epsilon Then
      Bend = 0
    Else
      Bend = kb * Exp(eb * Log(DeltaTheta + mb * DeltaThetaStar))
    End If
  Else
    If Abs(DeltaTheta + mb * DeltaThetaStar) < Epsilon Then
      Bend = pb
    Else
      Bend = (kb * (Exp(eb * (Log(DeltaTheta + mb * DeltaThetaStar))))) + pb
    End If
  End If
End Function

'*********************************************************************
Private Function Stretch(I0 As Coords, J0 As Coords, I1 As Coords, J1 As Coords)
' Accepts four 2D points. Calculates the stretching work used in morphing
'  the line segment I0-I1 to line segment J0-J1.

  Dim L0 As Double    ' length of segment from vertex I1 to vertex I0
              ' in the first frame
  Dim L1 As Double    ' length of segment from vertex J1 to vertex J0
              ' in the second frame
  L0 = Length(I1, I0)
  L1 = Length(J1, J0)
  If Abs((1 - cs) * Minimum(L0, L1) + cs * Maximum(L0, L1)) < Epsilon Then
    Stretch = 15000
  Else
    If Abs(L1 - L0) < Epsilon Then
      Stretch = 0
    Else
      Stretch = (ks * Exp(es * (Log((Abs(L1 - L0)))))) / ((1 - cs) * Minimum(L0, L1) _
          + cs * Maximum(L0, L1))
    End If
  End If
End Function

'*********************************************************************
Private Function Dot2D(A As Coords, B As Coords) As Double
' Accepts two 2D points and returns their dot product
  Dot2D = A.X * B.X + A.Y * B.Y
End Function

'*********************************************************************
```

```
Private Function Cross2D(A As Coords, B As Coords)
' Accepts two 2D points and returns the determinant
  Cross2D = A.X * B.Y - A.Y * B.X
End Function

'*********************************************************************
Private Function TriangleContainsOrigin(Q0 As Coords, Q1 As Coords, Q2 As Coords)
' Accepts three points to be vertices of a triangle. Returns True if this triangle contains
' the origin and False otherwise.

  Dim AX1 As Boolean
  Dim AX2 As Boolean
  Dim AX3 As Boolean
  Dim AX4 As Boolean
  Dim B As Integer
  Dim XInt As Integer
  ' segment 1 of the triangle
  If Abs(Q1.X - Q0.X) < Epsilon Then
    B = 0
  Else
    B = Q1.Y - Q1.X * ((Q1.Y - Q0.Y) / (Q1.X - Q0.X))
  End If
  If Abs(Q1.Y - Q0.Y) < Epsilon Then
    XInt = 0
  Else
    XInt = Q1.X - Q1.Y * ((Q1.X - Q0.X) / (Q1.Y - Q0.Y))
  End If
  If B > 0 And (SignOf(Q1.X) <> SignOf(Q0.X)) Then
    AX1 = True
  End If
  If B < 0 And (SignOf(Q1.X) <> SignOf(Q0.X)) Then
    AX3 = True
  End If
  If XInt > 0 And (SignOf(Q1.Y) <> SignOf(Q0.Y)) Then
    AX2 = True
  End If
  If XInt < 0 And (SignOf(Q1.Y) <> SignOf(Q0.Y)) Then
    AX4 = True
  End If
  ' segment 2
  If Abs(Q2.X - Q1.X) < Epsilon Then
    B = 0
  Else
    B = Q2.Y - Q2.X * ((Q2.Y - Q1.Y) / (Q2.X - Q1.X))
  End If
  If Abs(Q2.Y - Q1.Y) < Epsilon Then
    XInt = 0
  Else
    XInt = Q2.X - Q2.Y *((Q2.X - Q1.X) / (Q2.Y - Q1.Y))
  End If
  If B > 0 And (SignOf(Q2.X) <> SignOf(Q1.X)) Then
    AX1 = True
  End If
```

```
  If B < 0 And (SignOf(Q2.X) <> SignOf(Q1.X)) Then
    AX3 = True
  End If
  If XInt > 0 And (SignOf(Q2.Y) <> SignOf(Q1.Y)) Then
    AX2 = True
  End If
  If XInt < 0 And (SignOf(Q2.Y) <> SignOf(Q1.Y)) Then
    AX4 = True
  End If
  ' segment 3
  If Abs(Q0.X - Q2.X) < Epsilon Then
    B = 0
  Else
    B = Q0.Y - Q0.X * ((Q0.Y - Q2.Y) / (Q0.X - Q2.X))
  End If
  If Abs(Q0.Y - Q2.Y) < Epsilon Then
    XInt = 0
  Else
    XInt = Q0.X - Q0.Y * ((Q0.X - Q2.X) / (Q0.Y - Q2.Y))
  End If
  If B > 0 And (SignOf(Q0.X) <> SignOf(Q2.X)) Then
    AX1 = True
  End If
  If B < 0 And (SignOf(Q0.X) <> SignOf(Q2.X)) Then
    AX3 = True
  End If
  If XInt > 0 And (SignOf(Q0.Y) <> SignOf(Q2.Y)) Then
    AX2 = True
  End If
  If XInt < 0 And (SignOf(Q0.Y) <> SignOf(Q2.Y)) Then
    AX4 = True
  End If
  If AX1 = True And AX2 = True And AX3 = True And AX4 = True Then
     TriangleContainsOrigin = True
  Else
     TriangleContainsOrigin = False
  End If
End Function

'*************************************************************************
Public Function SignOf(X As Double)
' Accepts a number and returns true if the number is positive and false if the number is negative.
  If X >= 0 Then
    SignOf = True
  Else
    SignOf = False
  End If
End Function

'*************************************************************************


Private Function AngleFromXAxis(Q As Coords)
```

```
' Calculates the positive angle in rads from the positive x-axis to the point Q
  ' If Q = Origin or Q lies on the positive x-axis, then define the angle to be PI
  If (Q.X >= 0 And Q.Y = 0) Then
    AngleFromXAxis = PI
  Else
    Dim C As Double
    Dim A As Double
    Dim B As Double
    Dim Origin As Coords
    Dim PosXAxis As Coords
    Dim QuadQ As Integer
    Origin.X = 0
    Origin.Y = 0
    PosXAxis.X = 1
    PosXAxis.Y = 0
    C = Length(Origin, Q)
    A = Length(Q, PosXAxis)
    B = 1
    QuadQ = Quadrant(Q)
    If Abs(2 * C * B) < Epsilon Then
      AngleFromXAxis = 0
    Else
      If QuadQ = 1 Or QuadQ = 2 Then
        AngleFromXAxis = ArcCos((B * B + C * C - A * A) / (2 * C * B))
      Else 'if QuadQ = 3 or Quad! = 4
        AngleFromXAxis = 2 * PI - ArcCos((B * B + C * C - A * A) / (2 * C * B))
      End If
    End If
  End If
End Function

'*************************************************************************
Public Function ArcCos(X As Double)
' Takes a number and returns the ArcCos of that number.
  If Abs(X) < 1 + Epsilon And Abs(X) > 1 - Epsilon Then
    ArcCos = 0
  Else
    ArcCos = Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1)
  End If
End Function

'*************************************************************************
Public Function Minimum(A As Double, B As Double)
' Takes two numbers (double) and returns the minimum of the two.
  If A > B Then
    Minimum = B
  Else
    Minimum = A
  End If
End Function

'*************************************************************************
```

```
Private Function Quadrant(Q As Coords)
' Takes a point (x,y) and returns the quadrant in which the point lies.
  If Q.X >= 0 And Q.Y >= 0 Then
     Quadrant = 1
  End If
  If Q.X >= 0 And Q.Y < 0 Then
     Quadrant = 4
  End If
  If Q.X < 0 And Q.Y >= 0 Then
     Quadrant = 2
  End If
  If Q.X < 0 And Q.Y < 0 Then
     Quadrant = 3
  End If
End Function

'*********************************************************************
Public Function MinOf3(A As Double, B As Double, C As Double)
' Takes three numbers (double) and returns the minimum of the three
  If A <= B And A <= C Then
     MinOf3 = A
  End If
  If B < A And B <= C Then
     MinOf3 = B
  End If
  If C < A And C < B Then
     MinOf3 = C
  End If
End Function

'*********************************************************************
Public Sub EdgeLengthInterpolation()
' Does the same thing as linear interpolation
  Dim I As Integer
  Dim t As Integer
  Dim E1(100) As Coords    ' the x and y coords to get from the
  Dim E2(100) As Coords    ' previous point to the next point
  Dim E(100) As Coords
  Dim TempPic As PictureBox
  Dim Draw As Boolean
  Draw = False
  For I = 0 To (NumPts) Step 1
     E1(I).X = Key1Pts(I + 1).X - Key1Pts(I).X
     E1(I).Y = Key1Pts(I + 1).Y - Key1Pts(I).Y
     E2(I).X = Key2Pts(I + 1).X - Key2Pts(I).X
     E2(I).Y = Key2Pts(I + 1).Y - Key2Pts(I).Y
  Next I
  Dim t1 As Double
  For t = 0 To (NumInBetweens + 1) Step 1
     t1 = t / (NumInBetweens + 1)
     For I = 0 To (NumPts) Step 1
```

```
        E(I).X = (1 - t1) * E1(I).X + t1 * E2(I).X
        E(I).Y = (1 - t1) * E1(I).Y + t1 * E2(I).Y
      Next I
      For I = 0 To (NumPts) Step 1
        If I = 0 Then
          MorphPts(I).X = Key1Pts(0).X
          MorphPts(I).Y = Key1Pts(0).Y
        Else
          MorphPts(I).X = MorphPts(I - 1).X + E(I - 1).X
          MorphPts(I).Y = MorphPts(I - 1).Y + E(I - 1).Y
        End If
      Next I

      Select Case t
        Case 1
          Set TempPic = picMorph1
          Draw = True
        Case 2
          Set TempPic = picMorph2
          Draw = True
        Case 3
          Set TempPic = picMorph3
          Draw = True
        Case 4
          Set TempPic = picMorph4
          Draw = True
        Case 5
          Set TempPic = picMorph5
          Draw = True
      End Select

      If Draw Then
        For I = 1 To (NumPts) Step 1
          TempPic.Line (10 * MorphPts(I).X, _
                        TempPic.Height - (10 * MorphPts(I).Y)) _
                       -(10 * MorphPts(I - 1).X, _
                        TempPic.Height - (10 * MorphPts(I - 1).Y))
        Next I
      End If
      Draw = False
    Next t
End Sub

'*********************************************************************
Public Sub IntrinsicInterpolation()
  Dim I As Integer
  Dim t1 As Integer
  Dim t As Double
  Dim Theta1(100) As Double   ' Angles between edges of polygon 1
  Dim Theta2(100) As Double   ' Angles between edges of polygon 2
  Dim L1(100) As Double   ' Length of the edges of polygon 1
  Dim L2(100) As Double   ' Length of the edges of polygon 2
```

```
Dim Alpha1 As Double   ' Alpha for polygon 1
Dim Alpha2 As Double   ' Alpha for polygon 2
Dim Alpha(100) As Double    ' This is the Alpha for the in-between frames
Dim Theta(100) As Double   ' Theta for the in-between frames
Dim L(100) As Double  ' Edge Lengths for the in-between frames
Dim v1 As Coords
Dim v2 As Coords
Dim vCrossProd As Double
Dim TempPic As PictureBox
Dim Draw As Boolean
Draw = False
' Determine the angle Theta between an extended edge and the next edge.
' We find the cross product to see if the edges form a convex or concave
' part of the polygon (This affects the way in which theta is calculated
Dim Done As Boolean
Dim TempInt As Integer
For I = 1 To (NumPts - 1) Step 1
  If ((Key1Pts(I).X = Key1Pts(I + 1).X) And _
    (Key1Pts(I).Y = Key1Pts(I + 1).Y)) And ((Key1Pts(I).X <> Key1Pts(I - 1).X) _
      Or (Key1Pts(I).Y <> Key1Pts(I - 1).Y)) Then
    Done = False
    TempInt = I + 2
    While Not Done
      If (Key1Pts(TempInt).X <> Key1Pts(I).X) Or (Key1Pts(TempInt).Y <> Key1Pts(I).Y) Then
        Done = True
      Else
        TempInt = TempInt + 1
      End If
    Wend

    v1.X = Key1Pts(TempInt).X - Key1Pts(I).X
    v1.Y = Key1Pts(TempInt).Y - Key1Pts(I).Y
    v2.X = Key1Pts(I).X - Key1Pts(I - 1).X
    v2.Y = Key1Pts(I).Y - Key1Pts(I - 1).Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      Theta1(I) = PI - 0.5 * Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(TempInt)) - (PI / 2)
    Else
      If vCrossProd < -Epsilon Then
        Theta1(I) = -(PI - 0.5 * Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(TempInt)) - (PI / 2))
      Else
        Theta1(I) = 0   ' this should never occur
      End If
    End If
  Else
    If (Key1Pts(I - 1).X = Key1Pts(I).X) And (Key1Pts(I - 1).Y = Key1Pts(I).Y) _
      And (Key1Pts(I).X = Key1Pts(I + 1).X) And (Key1Pts(I).Y = Key1Pts(I + 1).Y) _
    Then
      Theta1(I) = 0
    Else
      If (Key1Pts(I - 1).X = Key1Pts(I).X) And (Key1Pts(I - 1).Y = Key1Pts(I).Y) _
      And ((Key1Pts(I).X <> Key1Pts(I + 1).X) Or (Key1Pts(I).Y <> Key1Pts(I + 1).Y)) Then
        Done = False
```

```
        TempInt = I - 1
        While Not Done
          If (Key1Pts(TempInt).X <> Key1Pts(I - 1).X Or _
            Key1Pts(TempInt).Y <> Key1Pts(I - 1).Y) Then
            Done = True
          Else
            TempInt = TempInt - 1
            If TempInt = 0 Then
              Done = True
            End If
          End If
        Wend
        Theta1(I) = Theta1(TempInt + 1)
      Else ' all points are distinct
        v1.X = Key1Pts(I + 1).X - Key1Pts(I).X
        v1.Y = Key1Pts(I + 1).Y - Key1Pts(I).Y
        v2.X = Key1Pts(I).X - Key1Pts(I - 1).X
        v2.Y = Key1Pts(I).Y - Key1Pts(I - 1).Y
        vCrossProd = Cross2D(v1, v2)
        If vCrossProd > Epsilon Then
          Theta1(I) = PI - Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(I + 1))
        Else
          If vCrossProd < -Epsilon Then
            Theta1(I) = -(PI - Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(I + 1)))
          Else
            Theta1(I) = 0
          End If
        End If
      End If
    End If
  End If
End If
If (Key2Pts(I).X = Key2Pts(I + 1).X) And (Key2Pts(I).Y = Key2Pts(I + 1).Y) _
  And ((Key2Pts(I).X <> Key2Pts(I - 1).X) Or (Key2Pts(I).Y <> Key2Pts(I - 1).Y)) Then
  Done = False
  TempInt = I + 2
  While Not Done
    If (Key2Pts(TempInt).X <> Key2Pts(I).X) Or (Key2Pts(TempInt).Y <> Key2Pts(I).Y) Then
      Done = True
    Else
      TempInt = TempInt + 1
    End If
  Wend
  v1.X = Key2Pts(TempInt).X - Key2Pts(I).X
  v1.Y = Key2Pts(TempInt).Y - Key2Pts(I).Y
  v2.X = Key2Pts(I).X - Key2Pts(I - 1).X
  v2.Y = Key2Pts(I).Y - Key2Pts(I - 1).Y
  vCrossProd = Cross2D(v1, v2)
  If vCrossProd > Epsilon Then
    Theta2(I) = PI - 0.5 * Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(TempInt)) - (PI / 2)
  Else
    If vCrossProd < -Epsilon Then

      Theta2(I) = -(PI - 0.5 * Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(TempInt)) - (PI / 2))
```

```
        Else
            Theta2(I) = 0   ' this occurs when pts lie along a straight line
        End If
      End If
    Else
      If (Key2Pts(I - 1).X = Key2Pts(I).X) And (Key2Pts(I - 1).Y = Key2Pts(I).Y) _
        And (Key2Pts(I).X = Key2Pts(I + 1).X) And (Key2Pts(I).Y = Key2Pts(I + 1).Y) Then
        Theta2(I) = 0
      Else
        If (Key2Pts(I - 1).X = Key2Pts(I).X) And (Key2Pts(I - 1).Y = Key2Pts(I).Y) _
          And ((Key2Pts(I).X <> Key2Pts(I + 1).X) Or (Key2Pts(I).Y <> Key2Pts(I + 1).Y)) _
        Then
          Done = False
          TempInt = I - 1
          While Not Done
            If (Key2Pts(TempInt).X <> Key2Pts(I - 1).X Or _
              Key2Pts(TempInt).Y <> Key2Pts(I - 1).Y) Then
              Done = True
            Else
              TempInt = TempInt - 1
              If TempInt = 0 Then
                Done = True
              End If
            End If
          Wend
          Theta2(I) = Theta2(TempInt + 1)
        Else ' all points are distinct
          v1.X = Key2Pts(I + 1).X - Key2Pts(I).X
          v1.Y = Key2Pts(I + 1).Y - Key2Pts(I).Y
          v2.X = Key2Pts(I).X - Key2Pts(I - 1).X
          v2.Y = Key2Pts(I).Y - Key2Pts(I - 1).Y
          vCrossProd = Cross2D(v1, v2)
          If vCrossProd > Epsilon Then
            Theta2(I) = PI - Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(I + 1))
          Else
            If vCrossProd < -Epsilon Then
              Theta2(I) = -(PI - Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(I + 1)))
            Else
              Theta2(I) = 0
            End If
          End If
        End If
      End If
    End If
  End If
Next I
For I = 0 To (NumPts - 1) Step 1
    ' Find the lengths of all edges of the polygon
  L1(I) = Length(Key1Pts(I + 1), Key1Pts(I))
  L2(I) = Length(Key2Pts(I + 1), Key2Pts(I))
Next I
Dim AxisPt As Coords

' Calculate the angle between the horizontal line through the anchor point
```

```
' and the first edge of the polygon
AxisPt.X = Key1Pts(0).X + 1
AxisPt.Y = Key1Pts(0).Y

' find the next distince vertex following the initial vertex
Done = False
TempInt = 1
While Not Done
   If (Key1Pts(0).X = Key1Pts(TempInt).X) And (Key1Pts(0).Y = Key1Pts(TempInt).Y) Then
     TempInt = TempInt + 1
  Else
     Done = True
  End If
Wend
 Alpha1 = Angle(Key1Pts(TempInt), Key1Pts(0), AxisPt)
AxisPt.X = Key2Pts(0).X + 1
AxisPt.Y = Key2Pts(0).Y
Done = False
TempInt = 1
While Not Done
   If (Key2Pts(0).X = Key2Pts(TempInt).X) And (Key2Pts(0).Y = Key2Pts(TempInt).Y) Then
     TempInt = TempInt + 1
  Else
     Done = True
  End If
Wend
 Alpha2 = Angle(Key2Pts(TempInt), Key2Pts(0), AxisPt)
For t1 = 1 To (NumInBetweens) Step 1
   t = t1 / (NumInBetweens + 1)
   For I = 0 To (NumPts - 1) Step 1
      L(I) = (1 - t) * L1(I) + t * L2(I)
      If I <> (NumPts - 1) Then
         Theta(I + 1) = (1 - t) * Theta1(I + 1) + t * Theta2(I + 1)
      End If
   Next I
    Alpha(0) = (1 - t) * Alpha1 + t * Alpha2    ' "anchor" angle

    ' anchor point gets linearly interpolated
    MorphPts(0).X = (1 - t) * Key1Pts(0).X + t * Key2Pts(0).X
    MorphPts(0).Y = (1 - t) * Key1Pts(0).Y + t * Key2Pts(0).Y
    MorphPts(1).X = Cos(Alpha(0)) * L(0) + MorphPts(0).X
    MorphPts(1).Y = Sin(Alpha(0)) * L(0) + MorphPts(0).Y
   Alpha(1) = Alpha(0) - Theta(1)
   For I = 2 To (NumPts) Step 1
      Alpha(I) = Alpha(I - 1) - Theta(I)
      MorphPts(I).X = MorphPts(I - 1).X + _
                  Cos(Alpha(I - 1)) * L(I - 1)
      MorphPts(I).Y = MorphPts(I - 1).Y + _
                  Sin(Alpha(I - 1)) * L(I - 1)
   Next I


   Select Case t1
```

```
        Case 1
           Set TempPic = picMorph1
          Draw = True
        Case 2
           Set TempPic = picMorph2
          Draw = True
        Case 3
           Set TempPic = picMorph3
          Draw = True
        Case 4
           Set TempPic = picMorph4
          Draw = True
        Case 5
           Set TempPic = picMorph5
          Draw = True
      End Select

      If Draw Then
        For I = 1 To NumPts Step 1
         TempPic.Line (10 * MorphPts(I).X, _
                  TempPic.Height - (10 * MorphPts(I).Y)) _
                  -(10 * MorphPts(I - 1).X, _
                   TempPic.Height - (10 * MorphPts(I - 1).Y))
        Next I
      End If
      Draw = False
    Next tl
End Sub

'*********************************************************************
Private Function Angle(Q0 As Coords, Q1 As Coords, Q2 As Coords)
' Takes three points, Q0, Q1 and Q2, and Calculates the angle at Q1.
  If (Q0.X = Q1.X And Q0.Y = Q1.Y) Or (Q1.X = Q2.X And Q1.Y = Q2.Y) _
    Or (Q0.X = Q2.X And Q0.Y = Q2.Y) Then
    Angle = PI
  Else
    Dim C As Double
    Dim A As Double
    Dim B As Double
    C = Length(Q1, Q2)
    A = Length(Q2, Q0)
    B = Length(Q0, Q1)
    If Abs(2 * C * B) < Epsilon Then
      Angle = 0
    Else
      Angle = ArcCos((B * B + C * C - A * A) / (2 * C * B))
    End If
    If Angle = 0 Then
      Angle = PI
    End If
  End If
End Function
```

```
'*************************************************************************
Public Function Maximum(A As Double, B As Double)
' Takes two numbers (double) and returns the maximum of the two.
  If A > B Then
    Maximum = A
  Else
    Maximum = B
  End If
End Function

'*************************************************************************
Public Sub EdgeTweaking()
  Dim I As Integer
  Dim tI As Integer
  Dim t As Double
  Dim Theta1(100) As Double  ' Angles between edges of polygon 1
  Dim Theta2(100) As Double  ' Angles between edges of polygon 2
  Dim L1(100) As Double  ' Length of the edges of polygon 1
  Dim L2(100) As Double  ' Length of the edges of polygon 2
  Dim Alpha1 As Double  ' Alpha for polygon 1
  Dim Alpha2 As Double  ' Alpha for polygon 2
  Dim Alpha(100) As Double  ' This is the Alpha for the in-between frames
  Dim Theta(100) As Double  ' Theta for the in-between frames
  Dim L(100) As Double  ' Edge Lengths for the in-between frames
  Dim v1 As Coords
  Dim v2 As Coords
  Dim vCrossProd As Double
  Dim S(100) As Double  ' the tweaking amounts
  Dim L12(100) As Double
  Dim LSmall As Double
  Dim E As Double
  Dim f As Double
  Dim G As Double
  Dim U As Double
  Dim V As Double
  Dim Lambda1 As Double
  Dim Lambda2 As Double
  Dim TempPic As PictureBox
  Dim Draw As Boolean
  Draw = False
  Dim Done As Boolean
  Dim TempInt As Integer
  ' Determine the angle Theta between an extended edge and the next edge.
  ' We find the cross product to see if the edges form a convex or concave
  ' part of the polygon (This affects the way in which theta is calculated

  For I = 1 To (NumPts - 1) Step 1
    If ((Key1Pts(I).X = Key1Pts(I + 1).X) And _
      (Key1Pts(I).Y = Key1Pts(I + 1).Y)) And ((Key1Pts(I).X <> Key1Pts(I - 1).X) _
        Or (Key1Pts(I).Y <> Key1Pts(I - 1).Y)) Then
      Done = False
      TempInt = I + 2
      While Not Done
```

```
      If (Key1Pts(TempInt).X <> Key1Pts(I).X) Or (Key1Pts(TempInt).Y <> Key1Pts(I).Y) Then
        Done = True
      Else
        TempInt = TempInt + 1
      End If
    Wend

    v1.X = Key1Pts(TempInt).X - Key1Pts(I).X
    v1.Y = Key1Pts(TempInt).Y - Key1Pts(I).Y
    v2.X = Key1Pts(I).X - Key1Pts(I - 1).X
    v2.Y = Key1Pts(I).Y - Key1Pts(I - 1).Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      Theta1(I) = PI - 0.5 * Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(TempInt)) - (PI / 2)
    Else
      If vCrossProd < -Epsilon Then
        Theta1(I) = -(PI - 0.5 * Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(TempInt)) - (PI / 2))
      Else
        Theta1(I) = 0
      End If
    End If
  Else
    If (Key1Pts(I - 1).X = Key1Pts(I).X) And (Key1Pts(I - 1).Y = Key1Pts(I).Y) _
     And (Key1Pts(I).X = Key1Pts(I + 1).X) And (Key1Pts(I).Y = Key1Pts(I + 1).Y) _
    Then
      Theta1(I) = 0
    Else
      If (Key1Pts(I - 1).X = Key1Pts(I).X) And (Key1Pts(I - 1).Y = Key1Pts(I).Y) _
      And ((Key1Pts(I).X <> Key1Pts(I + 1).X) Or (Key1Pts(I).Y <> Key1Pts(I + 1).Y)) Then
        Done = False
        TempInt = I - 1
        While Not Done
          If (Key1Pts(TempInt).X <> Key1Pts(I - 1).X Or _
            Key1Pts(TempInt).Y <> Key1Pts(I - 1).Y) Then
            Done = True
          Else
            TempInt = TempInt - 1
            If TempInt = 0 Then
              Done = True
            End If
          End If
        Wend
        Theta1(I) = Theta1(TempInt + 1)

      Else ' all points are distinct
        v1.X = Key1Pts(I + 1).X - Key1Pts(I).X
        v1.Y = Key1Pts(I + 1).Y - Key1Pts(I).Y
        v2.X = Key1Pts(I).X - Key1Pts(I - 1).X
        v2.Y = Key1Pts(I).Y - Key1Pts(I - 1).Y
        vCrossProd = Cross2D(v1, v2)
        If vCrossProd > Epsilon Then
          Theta1(I) = PI - Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(I + 1))
        Else
```

```
          If vCrossProd < -Epsilon Then
            Theta1(I) = -(PI - Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(I + 1)))
          Else
            Theta1(I) = 0  'NOTE USED TO BE 0
          End If
        End If
      End If
    End If
  End If

  If (Key2Pts(I).X = Key2Pts(I + 1).X) And (Key2Pts(I).Y = Key2Pts(I + 1).Y) _
    And ((Key2Pts(I).X <> Key2Pts(I - 1).X) Or (Key2Pts(I).Y <> Key2Pts(I - 1).Y)) Then
    Done = False
    TempInt = I + 2
    While Not Done
      If (Key2Pts(TempInt).X <> Key2Pts(I).X) Or (Key2Pts(TempInt).Y <> Key2Pts(I).Y) Then
        Done = True
      Else
        TempInt = TempInt + 1
      End If
    Wend
    v1.X = Key2Pts(TempInt).X - Key2Pts(I).X
    v1.Y = Key2Pts(TempInt).Y - Key2Pts(I).Y
    v2.X = Key2Pts(I).X - Key2Pts(I - 1).X
    v2.Y = Key2Pts(I).Y - Key2Pts(I - 1).Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      Theta2(I) = PI - 0.5 * Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(TempInt)) - (PI / 2)
    Else
      If vCrossProd < -Epsilon Then
        Theta2(I) = -(PI - 0.5 * Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(TempInt)) - (PI / 2))
      Else
        Theta2(I) = 0  '
      End If
    End If
  End If
Else
  If (Key2Pts(I - 1).X = Key2Pts(I).X) And (Key2Pts(I - 1).Y = Key2Pts(I).Y) _
    And (Key2Pts(I).X = Key2Pts(I + 1).X) And (Key2Pts(I).Y = Key2Pts(I + 1).Y) Then
    Theta2(I) = 0
  Else
    If (Key2Pts(I - 1).X = Key2Pts(I).X) And (Key2Pts(I - 1).Y = Key2Pts(I).Y) _
      And ((Key2Pts(I).X <> Key2Pts(I + 1).X) Or (Key2Pts(I).Y <> Key2Pts(I + 1).Y)) _
    Then
      Done = False
      TempInt = I - 1
      While Not Done
        If (Key2Pts(TempInt).X <> Key2Pts(I - 1).X Or _
          Key2Pts(TempInt).Y <> Key2Pts(I - 1).Y) Then
          Done = True
        Else
          TempInt = TempInt - 1
          If TempInt = 0 Then
            Done = True
```

```
            End If
          End If
        Wend
        Theta2(I) = Theta2(TempInt + 1)
      Else ' all points are distinct
        v1.X = Key2Pts(I + 1).X - Key2Pts(I).X
        v1.Y = Key2Pts(I + 1).Y - Key2Pts(I).Y
        v2.X = Key2Pts(I).X - Key2Pts(I - 1).X
        v2.Y = Key2Pts(I).Y - Key2Pts(I - 1).Y
        vCrossProd = Cross2D(v1, v2)
        If vCrossProd > Epsilon Then
          Theta2(I) = PI - Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(I + 1))
        Else
          If vCrossProd < -Epsilon Then
            Theta2(I) = -(PI - Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(I + 1)))
          Else
            Theta2(I) = 0
          End If
        End If
      End If
    End If
  End If
Next I
For I = 0 To (NumPts - 1) Step 1
  ' Find the lengths of all edges of the polygon
  L1(I) = Length(Key1Pts(I + 1), Key1Pts(I))
  L2(I) = Length(Key2Pts(I + 1), Key2Pts(I))
Next I
Dim AxisPt As Coords
' Calculate the angle between the horizontal line through the anchor point
' and the first edge of the polygon
AxisPt.X = Key1Pts(0).X + 1
AxisPt.Y = Key1Pts(0).Y
TempInt = 1
Done = False
While Not Done
  If (Key1Pts(0).X = Key1Pts(TempInt).X) And (Key1Pts(0).Y = Key1Pts(TempInt).Y) Then
    TempInt = TempInt + 1
  Else
    Done = True
  End If
Wend
Alpha1 = Angle(Key1Pts(TempInt), Key1Pts(0), AxisPt)
AxisPt.X = Key2Pts(0).X + 1
AxisPt.Y = Key2Pts(0).Y
TempInt = 1
Done = False
While Not Done
  If (Key2Pts(0).X = Key2Pts(TempInt).X) And (Key2Pts(0).Y = Key2Pts(TempInt).Y) Then
    TempInt = TempInt + 1
  Else
    Done = True
  End If
```

```
Wend
 Alpha2 = Angle(Key2Pts(TempInt), Key2Pts(0), AxisPt)
' Here insert tweaking stuff
Dim MaxEdgeLengthDiff As Double
Dim TempLength As Double
MaxEdgeLengthDiff = 0.1
For I = 0 To (NumPts - 1) Step 1
   TempLength = Abs(L1(I) - L2(I))
   If TempLength > MaxEdgeLengthDiff Then
      MaxEdgeLengthDiff = TempLength
   End If
Next I
LSmall = 0.0001 * MaxEdgeLengthDiff
For I = 0 To (NumPts - 1) Step 1
   L12(I) = Maximum(Abs(L1(I) - L2(I)), LSmall)
Next I
For tI = 1 To (NumInBetweens) Step 1
   t = tI / (NumInBetweens + 1)
   Alpha(0) = (1 - t) * Alpha1 + t * Alpha2
   For I = 0 To (NumPts - 1) Step 1
      Theta(I + 1) = (1 - t) * Theta1(I + 1) + t * Theta2(I + 1)
      If I > 0 Then
         Alpha(I) = Alpha(I - 1) - Theta(I)
      End If
   Next I

   E = 0
   f = 0
   G = 0
   For I = 0 To (NumPts - 1) Step 1
      E = E + L12(I) * L12(I) * Cos(Alpha(I)) * Cos(Alpha(I))
      f = f + L12(I) * L12(I) * Sin(Alpha(I)) * Cos(Alpha(I))
      G = G + L12(I) * L12(I) * Sin(Alpha(I)) * Sin(Alpha(I))
   Next I
   U = 0
   V = 0
   For I = 0 To (NumPts - 1) Step 1
      U = U + (((1 - t) * L1(I) + t * L2(I)) * Cos(Alpha(I)))
      V = V + (((1 - t) * L1(I) + t * L2(I)) * Sin(Alpha(I)))
   Next I

   U = U * 2
   V = V * 2
   Lambda1 = (U * G - f * V) / (E * G - f * f)
   Lambda2 = (E * V - U * f) / (E * G - f * f)

   For I = 0 To (NumPts) Step 1
      S(I) = -0.5 * L12(I) * L12(I) * (Lambda1 * Cos(Alpha(I)) + Lambda2 * Sin(Alpha(I)))
      L(I) = (1 - t) * L1(I) + t * L2(I) + S(I)
   Next I

    MorphPts(0).X = (1 - t) * Key1Pts(0).X + t * Key2Pts(0).X
    MorphPts(0).Y = (1 - t) * Key1Pts(0).Y + t * Key2Pts(0).Y
```

```
    MorphPts(1).X = Cos(Alpha(0)) * L(0) + MorphPts(0).X
    MorphPts(1).Y = Sin(Alpha(0)) * L(0) + MorphPts(0).Y

   For I = 2 To (NumPts) Step 1
      MorphPts(I).X = MorphPts(I - 1).X + _
                  Cos(Alpha(I - 1)) *L(I - 1)
      MorphPts(I).Y = MorphPts(I - 1).Y + _
                  Sin(Alpha(I - 1)) *L(I - 1)
   Next I
   Select Case t1
     Case 1
        Set TempPic = picMorph1
       Draw = True
     Case 2
        Set TempPic = picMorph2
       Draw = True
     Case 3
        Set TempPic = picMorph3
       Draw = True
     Case 4
        Set TempPic = picMorph4
       Draw = True
     Case 5
        Set TempPic = picMorph5
       Draw = True
   End Select
   If Draw Then
      For I = 1 To NumPts Step 1
          TempPic.Line (10 *MorphPts(I).X, _
                    TempPic.Height - (10 *MorphPts(I).Y)) _
                   -(10 * MorphPts(I - 1).X, _
                    TempPic.Height - (10 *MorphPts(I - 1).Y))
      Next I
   End If
   Draw = False
  Next t1
End Sub

'*************************************************************************
Public Sub LinearBezierMorph()
' A Vertex Path Method. Uses linear interpolation to calculate the path travelled by a vertex
' as it morphs from one polygon into the other.

  Dim I As Integer
  Dim J As Integer
  Dim Draw As Boolean
  Dim TempPic As PictureBox
  Draw = False
  ' Calculate the step size to increment each of the x- and y- coords
  ' for each successive in-between image

  While ((NumPts - 4) Mod 3) <> 0
```

```
    Key1Pts(NumPts) = Key1Pts(0)
    Key2Pts(NumPts) = Key2Pts(0)
    NumPts = NumPts + 1
Wend

For I = 0 To (NumPts) Step 1
    KeyDifference(I).X = (Key2Pts(I).X - Key1Pts(I).X) / (NumInBetweens + 1)
    KeyDifference(I).Y = (Key2Pts(I).Y - Key1Pts(I).Y) / (NumInBetweens + 1)
Next I

For I = 0 To (NumInBetweens + 1) Step 1
    ' Calculate the in-between points
    For J = 0 To (NumPts) Step 1
        MorphPts(J).X = Key1Pts(J).X + ((KeyDifference(J).X) * I)
        MorphPts(J).Y = Key1Pts(J).Y + ((KeyDifference(J).Y) * I)
    Next J
    ' Draw the lines in the appropriate picture box
    ' Note that the coord system is switched back for drawing
    Select Case I
      Case 1
          Set TempPic = picMorph1
        Draw = True
      Case 2
          Set TempPic = picMorph2
        Draw = True
      Case 3
          Set TempPic = picMorph3
        Draw = True
      Case 4
          Set TempPic = picMorph4
        Draw = True
      Case 5
          Set TempPic = picMorph5
        Draw = True
    End Select

    Dim Pt0 As Coords
    Dim Pt1 As Coords
    Dim Pt2 As Coords
    Dim Pt3 As Coords
    Dim t1 As Integer
    Dim t As Double
    Dim NumCurves As Integer
    Dim TempX As Double
    Dim TempY As Double
    Dim II As Integer
    NumCurves = ((NumPts - 4) / 3)

    If Draw Then
        For J = 1 To (NumPts) Step 1
            ' mark the control points
            TempPic.Circle (10 * MorphPts(J - 1).X, TempPic.Height - 10 * MorphPts(J - 1).Y), _
              1, RGB(0, 255, 0)
```

```
            ' TempPic.Line (10 * MorphPts(J).X, _
                TempPic.Height - (10 * MorphPts(J).Y)) _
                -(10 * MorphPts(J - 1).X, _
                TempPic.Height - (10 * MorphPts(J - 1).Y))
        Next J
        For II = 0 To NumCurves Step 1
          Pt0.X = MorphPts(3 * II).X
          Pt0.Y = MorphPts(3 * II).Y
          Pt1.X = MorphPts(3 * II + 1).X
          Pt1.Y = MorphPts(3 * II + 1).Y
          Pt2.X = MorphPts(3 * II + 2).X
          Pt2.Y = MorphPts(3 * II + 2).Y
          Pt3.X = MorphPts(3 * II + 3).X
          Pt3.Y = MorphPts(3 * II + 3).Y
          For t1 = 0 To 200 Step 1
           'calculate and plot the point of the bezier curve
            t = t1 / 200
            TempX = (1 - t) * (1 - t) * (1 - t) * Pt0.X + 3 * t * (1 - t) * (1 - t) * Pt1.X _
              + 3 * t * t * (1 - t) * Pt2.X + t * t * t * Pt3.X
            TempY = (1 - t) * (1 - t) * (1 - t) * Pt0.Y + 3 * t * (1 - t) * (1 - t) * Pt1.Y _
              + 3 * t * t * (1 - t) * Pt2.Y + t * t * t * Pt3.Y
            TempPic.Circle (10 * TempX, TempPic.Height - (10 * TempY)), 0.2
          Next t1
        Next II
      End If
      Draw = False
    Next I
End Sub

'*********************************************************************
Public Sub LeastWorkCurveMatching()
' Determines the control point correspondence between the two Bezier curves
' that will result in the least amount of work to morph from one curve to the other.

    Dim BackTrackList() As Coords
    If Key2NumPts > Key1NumPts Then
      NumPts = Key2NumPts
    Else
      NumPts = Key1NumPts
    End If

    ReDim BackTrackList(Key1NumPts + Key2NumPts) As Coords
    Dim WBack() As Integer
    ReDim WBack(Key1NumPts, Key2NumPts) As Integer
    Dim WUp() As Integer
    ReDim WUp(Key1NumPts, Key2NumPts) As Integer
    Dim WDiag() As Integer
    ReDim WDiag(Key1NumPts, Key2NumPts) As Integer
    Dim I As Integer
    Dim J As Integer


    WBack(0, 0) = 0
```

```
WUp(0, 0) = 0
WDiag(0, 0) = 0

For I = 0 To (Key1NumPts) Step 3
  For J = 0 To (Key2NumPts) Step 3
  'Note: If I=0 and J<>0 then we can only calculate WUp
    If I = 0 And J = 3 Then
      WUp(I, J) = StretchCurve(I, J - 3, I, J) + _
            BendCurve(I, J - 3, I, J) + _
          Minimum(WUp(I, J - 3) + KinkCurve(I, Key2NumPts - 3, _
            I, J - 3, I, J), _
_
            WDiag(I, J - 3) + _
              KinkCurve(Key1NumPts - 3, Key2NumPts - 3, _
              I, J - 3, I, J))

      WBack(I, J) = 15000
      WDiag(I, J) = 15000
    End If
    If I = 0 And J > 3 Then
      WUp(I, J) = StretchCurve(I, J - 3, I, J) + _
            BendCurve(I, J - 3, I, J) + _
          Minimum(WUp(I, J - 3) + KinkCurve(I, J - 6, _
            I, J - 3, I, J), _
_
            WDiag(I, J - 3) + _
              KinkCurve(Key1NumPts - 3, J - 6, _
              I, J - 3, I, J))
      WBack(I, J) = 15000
      WDiag(I, J) = 15000
    End If
  Also, if I<>0 and J=0 then we can only calculate WBack
    If I = 3 And J = 0 Then
      WBack(I, J) = StretchCurve(I - 3, J, I, J) + _
            BendCurve(I - 3, J, I, J) + _
          Minimum(WBack(I - 3, J) + _
            KinkCurve(Key1NumPts - 3, J, I - 3, _
              J, I, J), _
_
              WDiag(I - 3, J) + _
                KinkCurve(Key1NumPts - 3, Key2NumPts - 3, _
                I - 3, J, I, J))
      WUp(I, J) = 15000
      WDiag(I, J) = 15000
    End If
    If I > 3 And J = 0 Then
      WBack(I, J) = StretchCurve(I - 3, J, I, J) + _
            BendCurve(I - 3, J, I, J) + _
          Minimum(WBack(I - 3, J) + _
              KinkCurve(I - 6, J, I - 3, J, I, J), _
_
              WDiag(I - 3, J) + _
                KinkCurve(I - 6, Key2NumPts - 3, _
```

```
                    I - 3, J, I, J))
    WUp(I, J) = 15000
    WDiag(I, J) = 15000
End If
If I = 3 And J = 3 Then
    WBack(I, J) = StretchCurve(I - 3, J, I, J) + _
            BendCurve(I - 3, J, I, J) + _
-
            Minimum( _
              WBack(I - 3, J) + _
                 KinkCurve(Key1NumPts - 3, J, I - 3, J, I, J), _
-
              WDiag(I - 3, J) + _
                 KinkCurve(Key1NumPts - 3, J - 3, I - 3, J, I, J) _
                 ) ' end of Minimum parameters
    WUp(I, J) = StretchCurve(I, J - 3, I, J) + _
           BendCurve(I, J - 3, I, J) + _
-
           Minimum( _
             WUp(I, J - 3) + _
              KinkCurve(I, Key2NumPts - 3, I, J - 3, I, J), _
-
              WDiag(I, J - 3) + _
              KinkCurve(I - 3, Key2NumPts - 3, I, J - 3, I, J) _
               ) ' end of Minimum Parameters
    WDiag(I, J) = StretchCurve(I - 3, J - 3, I, J) + _
                 BendCurve(I - 3, J - 3, I, J) + _
-
          MinOf3( _
             WUp(I - 3, J - 3) + _
              KinkCurve(I - 3, Key2NumPts - 3, I - 3, J - 3, I, J), _
-
              WDiag(I - 3, J - 3) + _
              KinkCurve(Key1NumPts - 3, Key2NumPts - 3, _
                I - 3, J - 3, I, J), _
-
              WBack(I - 3, J - 3) + _
              KinkCurve(Key1NumPts - 3, J - 3, I - 3, J - 3, I, J) _
               )   ' end of MinOf3 Paremeters
End If
If I > 3 And J > 3 Then
    WBack(I, J) = StretchCurve(I - 3, J, I, J) + _
                 BendCurve(I - 3, J, I, J) + _
-
            Minimum( _
              WBack(I - 3, J) + _
                 KinkCurve(I - 6, J, I - 3, J, I, J), _
-
              WDiag(I - 3, J) + _
                 KinkCurve(I - 6, J - 3, I - 3, J, I, J) _
                 ) ' end of Minimum parameters

    WUp(I, J) = StretchCurve(I, J - 3, I, J) + _
```

```
                BendCurve(I, J - 3, I, J) + _
-
                Minimum( _
                  WUp(I, J - 3) + _
                   KinkCurve(I, J - 6, I, J - 3, I, J), _
-
                  WDiag(I, J - 3) + _
                   KinkCurve(I - 3, J - 6, I, J - 3, I, J) _
                    ) ' end of Minimum Parameters

        WDiag(I, J) = StretchCurve(I - 3, J - 3, I, J) + _
                BendCurve(I - 3, J - 3, I, J) + _
-
                MinOf3( _
                  WUp(I - 3, J - 3) + _
                   KinkCurve(I - 3, J - 6, I - 3, J - 3, I, J), _
-
                  WDiag(I - 3, J - 3) + _
                   KinkCurve(I - 6, J - 6, I - 3, J - 3, I, J), _
-
                  WBack(I - 3, J - 3) + _
                   KinkCurve(I - 6, J - 3, I - 3, J - 3, I, J) _
                    )  ' end of MinOf3 Paremeters
    End If
  Next J
Next I

' now backtrack to find the path.
BackTrackList(0).X = Key1NumPts
BackTrackList(0).Y = Key2NumPts
Dim TempX As Integer
Dim TempY As Integer
Dim CurrKey1Pt As Integer
Dim CurrKey2Pt As Integer
Dim NumBackTrackPts As Integer
NumBackTrackPts = 1
CurrKey1Pt = BackTrackList(0).X
CurrKey2Pt = BackTrackList(0).Y
I = 1

Do While (CurrKey1Pt >= 0) And (CurrKey2Pt >= 0)
  TempX = BackTrackList(I - 1).X
  TempY = BackTrackList(I - 1).Y
  If WBack(TempX, TempY) <= WUp(TempX, TempY) And _
    WBack(TempX, TempY) <= WDiag(TempX, TempY) Then
      CurrKey1Pt = BackTrackList(I - 1).X - 3
      CurrKey2Pt = BackTrackList(I - 1).Y
  Else
    If WUp(TempX, TempY) < WBack(TempX, TempY) And _
      WUp(TempX, TempY) <= WDiag(TempX, TempY) Then
        CurrKey1Pt = BackTrackList(I - 1).X
        CurrKey2Pt = BackTrackList(I - 1).Y - 3
```

```
        Else
          If WDiag(TempX, TempY) < WBack(TempX, TempY) And _
            WDiag(TempX, TempY) < WUp(TempX, TempY) Then
              CurrKey1Pt = BackTrackList(I - 1).X - 3
              CurrKey2Pt = BackTrackList(I - 1).Y - 3
          End If
        End If
      End If
      BackTrackList(I).X = CurrKey1Pt
      BackTrackList(I).Y = CurrKey2Pt
      NumBackTrackPts = NumBackTrackPts + 1
      I = I + 1
      If CurrKey1Pt = 0 And CurrKey2Pt = 0 Then
        Exit Do
      End If
    Loop
    Dim TempKey1InterpPts() As Coords
    ReDim TempKey1InterpPts(Key1NumPts + Key2NumPts) As Coords
    Dim TempKey2InterpPts() As Coords
    ReDim TempKey2InterpPts(Key1NumPts + Key2NumPts) As Coords
    For I = 0 To (NumBackTrackPts - 1) Step 1
      TempKey1InterpPts(NumBackTrackPts - 1 - I).X = Key1Pts(BackTrackList(I).X).X
      TempKey1InterpPts(NumBackTrackPts - 1 - I).Y = Key1Pts(BackTrackList(I).X).Y
      TempKey2InterpPts(NumBackTrackPts - 1 - I).X = Key2Pts(BackTrackList(I).Y).X
      TempKey2InterpPts(NumBackTrackPts - 1 - I).Y = Key2Pts(BackTrackList(I).Y).Y
    Next I

    Dim OldListMarker As Integer
    Dim NewListMarker As Integer
    Dim InterpListMarker As Integer
    Dim NewList1(MaxNum) As Coords
    Dim NewList2(MaxNum) As Coords
    OldListMarker = 0
    NewListMarker = 0
    InterpListMarker = 0
    Dim Done As Boolean
    Done = False
    While Not Done
      If (TempKey1InterpPts(InterpListMarker).X = TempKey1InterpPts(InterpListMarker + 1).X) _
        And (TempKey1InterpPts(InterpListMarker).Y = TempKey1InterpPts(InterpListMarker + 1).Y)
Then
        For I = 1 To 3 Step 1
          NewList1(NewListMarker).X = TempKey1InterpPts(InterpListMarker).X
          NewList1(NewListMarker).Y = TempKey1InterpPts(InterpListMarker).Y
          NewListMarker = NewListMarker + 1
        Next I
        InterpListMarker = InterpListMarker + 1
      Else
        ' if the interp points are not the same, record the next
        ' ones in the old list
        For I = 1 To 3 Step 1
          NewList1(NewListMarker).X = Key1Pts(OldListMarker).X
          NewList1(NewListMarker).Y = Key1Pts(OldListMarker).Y
```

```
            NewListMarker = NewListMarker + 1
            OldListMarker = OldListMarker + 1
          Next I
          InterpListMarker = InterpListMarker + 1
        End If
        If InterpListMarker = NumBackTrackPts Then
          Done = True
        End If
      Wend

      OldListMarker = 0
      NewListMarker = 0
      InterpListMarker = 0
      Done = False
      While Not Done
        If (TempKey2InterpPts(InterpListMarker).X = TempKey2InterpPts(InterpListMarker + 1).X) _
        And (TempKey2InterpPts(InterpListMarker).Y = TempKey2InterpPts(InterpListMarker + 1).Y)
Then
          For I = 1 To 3 Step 1
            NewList2(NewListMarker).X = TempKey2InterpPts(InterpListMarker).X
            NewList2(NewListMarker).Y = TempKey2InterpPts(InterpListMarker).Y
            NewListMarker = NewListMarker + 1
          Next I
          InterpListMarker = InterpListMarker + 1
        Else
          ' if the interp points are not the same, record the next
          ' ones in the old list
          For I = 1 To 3 Step 1
            NewList2(NewListMarker).X = Key2Pts(OldListMarker).X
            NewList2(NewListMarker).Y = Key2Pts(OldListMarker).Y
            NewListMarker = NewListMarker + 1
            OldListMarker = OldListMarker + 1
          Next I
          InterpListMarker = InterpListMarker + 1
        End If
        If InterpListMarker = NumBackTrackPts Then
          Done = True
        End If
      Wend
      For I = 0 To (((NumBackTrackPts - 1) * 3) - 1) Step 1
        Key1Pts(I).X = NewList1(I).X
        Key1Pts(I).Y = NewList1(I).Y
        Key2Pts(I).X = NewList2(I).X
        Key2Pts(I).Y = NewList2(I).Y
      Next I
      Key1Pts((NumBackTrackPts - 1) * 3).X = Key1Pts(0).X
      Key1Pts((NumBackTrackPts - 1) * 3).Y = Key1Pts(0).Y
      Key2Pts((NumBackTrackPts - 1) * 3).X = Key2Pts(0).X
      Key2Pts((NumBackTrackPts - 1) * 3).Y = Key2Pts(0).Y
      NumPts = (NumBackTrackPts - 1) * 3

End Sub
```

```
'*********************************************************************
Public Function CurveLength(I0 As Integer, I1 As Integer, Num As Integer)
  Dim P0 As Coords
  Dim P1 As Coords
  Dim P2 As Coords
  Dim P3 As Coords
  Dim f(11) As Double
  Dim t As Integer
  Dim N As Integer
  Dim t1 As Double
  Dim dxdt As Double
  Dim dydt As Double
  Dim CL As Double
  Dim h As Double
  N = 10
  If Num = 1 Then
    P0.X = Key1Pts(I0).X
    P0.Y = Key1Pts(I0).Y
    If (I0 = I1) Then
      P1.X = Key1Pts(I0).X
      P1.Y = Key1Pts(I0).Y
      P2.X = Key1Pts(I0).X
      P2.Y = Key1Pts(I0).Y
      P3.X = Key1Pts(I0).X
      P3.Y = Key1Pts(I0).Y
    Else
      P1.X = Key1Pts(I0 + 1).X
      P1.Y = Key1Pts(I0 + 1).Y
      P2.X = Key1Pts(I0 + 2).X
      P2.Y = Key1Pts(I0 + 2).Y
      P3.X = Key1Pts(I0 + 3).X
      P3.Y = Key1Pts(I0 + 3).Y
    End If
  Else
    P0.X = Key2Pts(I0).X
    P0.Y = Key2Pts(I0).Y
    If (I0 = I1) Then
      P1.X = Key2Pts(I0).X
      P1.Y = Key2Pts(I0).Y
      P2.X = Key2Pts(I0).X
      P2.Y = Key2Pts(I0).Y
      P3.X = Key2Pts(I0).X
      P3.Y = Key2Pts(I0).Y
    Else
      P1.X = Key2Pts(I0 + 1).X
      P1.Y = Key2Pts(I0 + 1).Y
      P2.X = Key2Pts(I0 + 2).X
      P2.Y = Key2Pts(I0 + 2).Y
      P3.X = Key2Pts(I0 + 3).X
      P3.Y = Key2Pts(I0 + 3).Y
    End If
  End If
```

```
    ' use the trapezoid rule, with n=10, h=0.1 to integrate to find curve length
    ' t always goes from 0 to 1

    If (P0.X = P3.X) And (P0.Y = P3.Y) Then
      CurveLength = 0
    Else
      For t = 0 To 10 Step 1
        t1 = t / N
        dxdt = CoeffA(t1) * P0.X + CoeffB(t1) * P1.X + CoeffC(t1) * P2.X + CoeffD(t1) * P3.X
        dydt = CoeffA(t1) * P0.Y + CoeffB(t1) * P1.Y + CoeffC(t1) * P2.Y + CoeffD(t1) * P3.Y
        f(t) = Sqr(dxdt * dxdt + dydt * dydt)
      Next t
      h = 1 / N
      CL = (f(0) + f(10)) / 2
      For t = 1 To 0 Step 1
        CL = CL + f(t)
      Next t
      CurveLength = CL * h
    End If
End Function

'*************************************************************************
Public Function CoeffA(t As Double)
    CoeffA = -3 * (1 - t) * (1 - t)
End Function

'*************************************************************************
Public Function CoeffB(t As Double)
    CoeffB = 3 * (1 - t) * (1 - 3 * t)
End Function

'*************************************************************************
Public Function CoeffC(t As Double)
    CoeffC = 3 * t * (2 - 3 * t)
End Function

'*************************************************************************
Public Function CoeffD(t As Double)
    CoeffD = 3 * t * t
End Function

'*************************************************************************
Private Function BendCurve(I0 As Integer, J0 As Integer, I1 As Integer, J1 As Integer)
' again, pass the index of the array and calculate all other points from that
    Dim P0 As Coords
    Dim P1 As Coords
    Dim P2 As Coords
    Dim P3 As Coords
    Dim m1 As Double
    Dim m2 As Double
    Dim Pt As Coords
    Dim xIntersection As Double
    Dim yIntersection As Double
```

```
Dim Phi1 As Double
Dim Phi2 As Double
Dim L1 As Double
Dim L2 As Double
If I0 = I1 Then
   P0.X = Key1Pts(I0).X
   P0.Y = Key1Pts(I0).Y
   P1.X = Key1Pts(I0).X
   P1.Y = Key1Pts(I0).Y
   P2.X = Key1Pts(I0).X
   P2.Y = Key1Pts(I0).Y
   P3.X = Key1Pts(I0).X
   P3.Y = Key1Pts(I0).Y
Else
   P0.X = Key1Pts(I0).X
   P0.Y = Key1Pts(I0).Y
   P1.X = Key1Pts(I0 + 1).X
   P1.Y = Key1Pts(I0 + 1).Y
   P2.X = Key1Pts(I0 + 2).X
   P2.Y = Key1Pts(I0 + 2).Y
   P3.X = Key1Pts(I0 + 3).X
   P3.Y = Key1Pts(I0 + 3).Y
End If

' calculate the slope of the normal lines at p0 and p3
If Abs(P0.Y - P1.Y) < Epsilon Then
   m1 = 15000
Else
   m1 = (P0.X - P1.X) / (P0.Y - P1.Y)
End If
If Abs(P3.Y - P2.Y) < Epsilon Then
   m2 = 15000
Else
   m2 = (P2.X - P3.X) / (P3.Y - P2.Y)
End If
If Abs(m1 - m2) < Epsilon Then
   Phi1 = 0
Else
   xIntersection = (P3.Y - P0.Y + m1 * P0.X - m2 * P3.X) / (m1 - m2)
   yIntersection = (xIntersection - P0.X) * m1 + P0.Y
   Pt.X = xIntersection
   Pt.Y = yIntersection
   If Length(P2, P1) < Length(P3, P0) Then
      Phi1 = Angle(P0, Pt, P3)
   Else
      Phi1 = 2 * PI - Angle(P0, Pt, P3)
   End If
End If
If J0 = J1 Then
   P0.X = Key2Pts(J0).X
   P0.Y = Key2Pts(J0).Y
   P1.X = Key2Pts(J0).X
   P1.Y = Key2Pts(J0).Y
```

```
    P2.X = Key2Pts(J0).X
    P2.Y = Key2Pts(J0).Y
    P3.X = Key2Pts(J0).X
    P3.Y = Key2Pts(J0).Y
  Else
    P0.X = Key2Pts(J0).X
    P0.Y = Key2Pts(J0).Y
    P1.X = Key2Pts(J0 + 1).X
    P1.Y = Key2Pts(J0 + 1).Y
    P2.X = Key2Pts(J0 + 2).X
    P2.Y = Key2Pts(J0 + 2).Y
    P3.X = Key2Pts(J0 + 3).X
    P3.Y = Key2Pts(J0 + 3).Y
  End If
  ' calculate the slope of the normal lines at p0 and p3
  If Abs(P0.Y - P1.Y) < Epsilon Then
    m1 = 15000
  Else
    m1 = (P0.X - P1.X) / (P0.Y - P1.Y)
  End If
  If Abs(P3.Y - P2.Y) < Epsilon Then
    m2 = 15000
  Else
    m2 = (P2.X - P3.X) / (P3.Y - P2.Y)
  End If
  If Abs(m1 - m2) < Epsilon Then
    Phi2 = 0
  Else
    xIntersection = (P3.Y - P0.Y + m1 * P0.X - m2 * P3.X) / (m1 - m2)
    yIntersection = (xIntersection - P0.X) * m1 + P0.Y
    Pt.X = xIntersection
    Pt.Y = yIntersection
    If Length(P2, P1) < Length(P3, P0) Then
      Phi2 = Angle(P0, Pt, P3)
    Else
      Phi2 = 2 * PI - Angle(P0, Pt, P3)
    End If
  End If
  L1 = CurveLength(I0, I1, 1)
  L2 = CurveLength(J0, J1, 2)
  If (L1 + L2) < Epsilon Then
    BendCurve = 15000
  Else
    BendCurve = CurveCb * (Phi2 - Phi1) * (Phi2 - Phi1) / (L1 + L2)
  End If
End Function

'*********************************************************************
Private Function KinkCurve(I0 As Integer, J0 As Integer, I1 As Integer, J1 As Integer, _
                I2 As Integer, J2 As Integer)
' accept the indices of the join points, as well as the indices of the end control
' points of the curves that meet at the join point. I1 (J1) is the join point.
```

```
' We assume that the curve segments have only a small degree of curvature
  Dim P2 As Coords
  Dim P3 As Coords
  Dim P4 As Coords
  Dim m1 As Double
  Dim m2 As Double
  Dim Pt As Coords
  Dim xIntersection As Double
  Dim yIntersection As Double
  Dim Phi1 As Double
  Dim Phi2 As Double
  Dim L1 As Double
  Dim L2 As Double
  Dim Done As Boolean
  Dim TempInt As Integer
  Dim v1 As Coords
  Dim v2 As Coords
  Dim vCrossProd As Double
  Dim DivAndAdd As Boolean
  P3.X = Key1Pts(I1).X
  P3.Y = Key1Pts(I1).Y
  If (I0 = I1) Then
    P2.X = Key1Pts(I1).X
    P2.Y = Key1Pts(I1).Y
  Else
    If I1 = 0 Then
      P2.X = Key1Pts(Key1NumPts - 1).X
      P2.Y = Key1Pts(Key1NumPts - 1).Y
    Else
      P2.X = Key1Pts(I1 - 1).X
      P2.Y = Key1Pts(I1 - 1).Y
    End If
  End If
  If I1 = I2 Then
    P4.X = Key1Pts(I1).X
    P4.Y = Key1Pts(I1).Y
  Else
    P4.X = Key1Pts(I1 + 1).X
    P4.Y = Key1Pts(I1 + 1).Y
  End If
  DivAndAdd = False
  ' if P2=P3 but P3<>P4
  If ((P2.X = P3.X) And (P2.Y = P3.Y)) And ((P3.X <> P4.X) Or (P3.Y <> P4.Y)) Then
    If I1 = 0 Then
      TempInt = Key1NumPts
    Else
      TempInt = I1 - 1
    End If
    Done = False
    While Not Done
      If (Key1Pts(TempInt).X <> P2.X) Or (Key1Pts(TempInt).Y <> P2.Y) Then
        Done = True
      Else
```

```
        TempInt = TempInt - 1
        If TempInt < 0 Then
          TempInt = Key1NumPts
        End If
      End If
    Wend
    P2.X = Key1Pts(TempInt).X
    P2.Y = Key1Pts(TempInt).Y
    DivAndAdd = True
  Else ' if P3=P4 but P2<>P3
    If ((P3.X = P4.X) And (P3.Y = P4.Y)) And ((P2.X <> P3.X) Or (P2.Y <> P3.Y)) Then
      If I1 = Key1NumPts Then
        TempInt = 1
      Else
        TempInt = I1 + 1
      End If
      Done = False
      While Not Done
        If (Key1Pts(TempInt).X <> P4.X) Or (Key1Pts(TempInt).Y <> P4.Y) Then
          Done = True
        Else
          TempInt = TempInt + 1
          If TempInt > Key1NumPts Then
            TempInt = 0
          End If
        End If
      Wend
      P4.X = Key1Pts(TempInt).X
      P4.Y = Key1Pts(TempInt).Y
      DivAndAdd = True
    End If
  End If
  If (P2.X = P3.X) And (P2.Y = P3.Y) And (P3.X = P4.X) And (P3.Y = P4.Y) Then
    Phi1 = PI
  Else
    v1.X = P4.X - P3.X
    v1.Y = P4.Y - P3.Y
    v2.X = P3.X - P2.X
    v2.Y = P3.Y - P2.Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      If DivAndAdd = False Then
        Phi1 = Angle(P2, P3, P4)
      Else
        Phi1 = 0.5 * Angle(P2, P3, P4) + (PI / 2)
      End If
    Else
      If vCrossProd < -Epsilon Then
        If DivAndAdd = False Then
          Phi1 = PI - Angle(P2, P3, P4)
        Else
          Phi1 = PI - (0.5 * Angle(P2, P3, P4) + (PI / 2))
        End If
```

```
      Else ' the points are collinear
         Phi1 = PI
      End If
    End If
  End If
  P3.X = Key2Pts(J0).X
  P3.Y = Key2Pts(J0).Y
  If J1 = J0 Then
    P2.X = Key2Pts(J1).X
    P2.Y = Key2Pts(J1).Y
  Else
     If J1 = 0 Then   ' keep in mind that key2pts(k3y2numpts-1) may be the same as the init pt
       P2.X = Key2Pts(Key2NumPts - 1).X
       P2.Y = Key2Pts(Key2NumPts - 1).Y
    Else
       P2.X = Key2Pts(J1 - 1).X
       P2.Y = Key2Pts(J1 - 1).Y
    End If
  End If
  If J1 = J2 Then
    P4.X = Key2Pts(J1).X
    P4.Y = Key2Pts(J1).Y
  Else
    P4.X = Key2Pts(J1 + 1).X
    P4.Y = Key2Pts(J1 + 1).Y
  End If
  DivAndAdd = False
  ' if P2=P3 but P3<>P4
  If ((P2.X = P3.X) And (P2.Y = P3.Y)) And ((P3.X <> P4.X) Or (P3.Y <> P4.Y)) Then
    If J1 = 0 Then
      TempInt = Key2NumPts
    Else
      TempInt = J1 - 1
    End If
    Done = False
    While Not Done
      If (Key2Pts(TempInt).X <> P2.X) Or (Key2Pts(TempInt).Y <> P2.Y) Then
        Done = True
      Else
        TempInt = TempInt - 1
        If TempInt < 0 Then
          TempInt = Key2NumPts
        End If
      End If
    Wend
    P2.X = Key2Pts(TempInt).X
    P2.Y = Key2Pts(TempInt).Y
    DivAndAdd = True
  Else
    If ((P3.X = P4.X) And (P3.Y = P4.Y)) And ((P2.X <> P3.X) Or (P2.Y <> P3.Y)) Then
      If J1 = Key2NumPts Then
        TempInt = 1
      Else
```

```
        TempInt = J1 + 1
      End If
      Done = False
      While Not Done
        If (Key2Pts(TempInt).X <> P4.X) Or (Key2Pts(TempInt).Y <> P4.Y) Then
          Done = True
        Else
          TempInt = TempInt + 1
          If TempInt > Key2NumPts Then
            TempInt = 0
          End If
        End If
      Wend
      P4.X = Key2Pts(TempInt).X
      P4.Y = Key2Pts(TempInt).Y
      DivAndAdd = True
    End If
  End If
  If (P2.X = P3.X) And (P2.Y - P3.Y) And (P3.X = P4.X) And (P3.Y = P4.Y) Then
    ' all points are equal
    Phi2 = PI
  Else
    v1.X = P4.X - P3.X
    v1.Y = P4.Y - P3.Y
    v2.X = P3.X - P2.X
    v2.Y = P3.Y - P2.Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      If DivAndAdd = False Then
        Phi2 = Angle(P2, P3, P4)
      Else
        Phi2 = 0.5 * Angle(P2, P3, P4) + (PI / 2)
      End If
    Else
      If vCrossProd < -Epsilon Then
        If DivAndAdd = False Then
          Phi2 = PI - Angle(P2, P3, P4)
        Else
          Phi2 = PI - (0.5 * Angle(P2, P3, P4) + (PI / 2))
        End If
      Else    'the points are collinear
        Phi2 = PI
      End If
    End If
  End If
  If Abs(Phi2 - Phi1) < Epsilon Then
    KinkCurve = 0
  Else
    KinkCurve = CurveCk * Exp(CurveEk * Log(Abs(Phi2 - Phi1)))
  End If
End Function

'*****************************************************************************
```

```
Public Function StretchCurve(I0 As Integer, J0 As Integer, I1 As Integer, J1 As Integer)
' Accepts the index of the starting point of each curve. Calculates the stretching work
' used in morphing the curve segment starting at Key1Pts(I0) and ending at Key1Pts(I1)
' to curve segment starting at Key2Pts(J0) and ending at Key2Pts(J1).

   Dim L0 As Double   ' length of segment from vertex I1 to vertex I0 in the 1st frame
   Dim L1 As Double   ' length of segment from vertex J1 to vertex J0 in the second frame

   L0 = CurveLength(I0, I1, 1)
   L1 = CurveLength(J0, J1, 2)
   If Abs((1 - CurveCs) * Minimum(L0, L1) + CurveCs * Maximum(L0, L1)) < Epsilon Then
      StretchCurve = 15000
   Else
      If Abs(L1 - L0) < Epsilon Then
         StretchCurve = 0
      Else
         StretchCurve = (CurveKs * Exp(CurveEs * Log(Abs(L1 - L0)))) / ((1 - CurveCs) * Minimum(L0,
L1) _
               + CurveCs * Maximum(L0, L1))
      End If
   End If
End Function

'*********************************************************************
Public Sub IntrinsicBezierMorph()
   Dim I As Integer
   Dim t1 As Integer
   Dim t As Double
   Dim Theta1(100) As Double   ' Angles between edges of polygon 1
   Dim Theta2(100) As Double   ' Angles between edges of polygon 2
   Dim L1(100) As Double   ' Length of the edges of polygon 1
   Dim L2(100) As Double   ' Length of the edges of polygon 2
   Dim Alpha1 As Double   ' Alpha for polygon 1
   Dim Alpha2 As Double   ' Alpha for polygon 2
   Dim Alpha(100) As Double   ' This is the Alpha for the in-between frames
   Dim Theta(100) As Double   ' Theta for the in-between frames
   Dim L(100) As Double   ' Edge Lengths for the in-between frames
   Dim v1 As Coords
   Dim v2 As Coords
   Dim vCrossProd As Double
   Dim S(100) As Double   ' the tweaking amounts
   Dim L12(100) As Double
   Dim LSmall As Double
   Dim E As Double
   Dim f As Double
   Dim G As Double
   Dim U As Double
   Dim V As Double
   Dim Lambda1 As Double
   Dim Lambda2 As Double
   Dim TempPic As PictureBox
   Dim Draw As Boolean
```

```
Draw = False
Dim Done As Boolean
Dim TempInt As Integer

' Determine the angle Theta between an extended edge and the next edge.
' We find the cross product to see if the edges form a convex or concave
' part of the polygon (This affects the way in which theta is calculated
For I = 1 To (NumPts - 1) Step 1
  If ((Key1Pts(I).X = Key1Pts(I + 1).X) And _
    (Key1Pts(I).Y = Key1Pts(I + 1).Y)) And ((Key1Pts(I).X <> Key1Pts(I - 1).X) _
      Or (Key1Pts(I).Y <> Key1Pts(I - 1).Y)) Then
    Done = False
    TempInt = I + 2
    While Not Done
      If (Key1Pts(TempInt).X <> Key1Pts(I).X) Or (Key1Pts(TempInt).Y <> Key1Pts(I).Y) Then
        Done = True
      Else
        TempInt = TempInt + 1
      End If
    Wend

    v1.X = Key1Pts(TempInt).X - Key1Pts(I).X
    v1.Y = Key1Pts(TempInt).Y - Key1Pts(I).Y
    v2.X = Key1Pts(I).X - Key1Pts(I - 1).X
    v2.Y = Key1Pts(I).Y - Key1Pts(I - 1).Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      Theta1(I) = PI - 0.5 * Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(TempInt)) - (PI / 2)
    Else
      If vCrossProd < -Epsilon Then
        Theta1(I) = -(PI - 0.5 * Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(TempInt)) - (PI / 2))
      Else
        Theta1(I) = 0  '
      End If
    End If
  Else
    If (Key1Pts(I - 1).X = Key1Pts(I).X) And (Key1Pts(I - 1).Y = Key1Pts(I).Y) _
      And (Key1Pts(I).X = Key1Pts(I + 1).X) And (Key1Pts(I).Y = Key1Pts(I + 1).Y) _
    Then
      Theta1(I) = 0
    Else
      If (Key1Pts(I - 1).X = Key1Pts(I).X) And (Key1Pts(I - 1).Y = Key1Pts(I).Y) _
      And ((Key1Pts(I).X <> Key1Pts(I + 1).X) Or (Key1Pts(I).Y <> Key1Pts(I + 1).Y)) Then
        Done = False
        TempInt = I - 1
        While Not Done
          If (Key1Pts(TempInt).X <> Key1Pts(I - 1).X Or _
            Key1Pts(TempInt).Y <> Key1Pts(I - 1).Y) Then
            Done = True
          Else
            TempInt = TempInt - 1
            If TempInt = 0 Then
              Done = True
```

```
            End If
          End If
        Wend
        Theta1(I) = Theta1(TempInt + 1)
      Else ' all points are distinct
        v1.X = Key1Pts(I + 1).X - Key1Pts(I).X
        v1.Y = Key1Pts(I + 1).Y - Key1Pts(I).Y
        v2.X = Key1Pts(I).X - Key1Pts(I - 1).X
        v2.Y = Key1Pts(I).Y - Key1Pts(I - 1).Y
        vCrossProd = Cross2D(v1, v2)
        If vCrossProd > Epsilon Then
          Theta1(I) = PI - Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(I + 1))
        Else
          If vCrossProd < -Epsilon Then
            Theta1(I) = -(PI - Angle(Key1Pts(I - 1), Key1Pts(I), Key1Pts(I + 1)))
          Else
            Theta1(I) = 0  'NOTE USED TO BE 0
          End If
        End If
      End If
    End If
  End If

  If (Key2Pts(I).X = Key2Pts(I + 1).X) And (Key2Pts(I).Y = Key2Pts(I + 1).Y) _
    And ((Key2Pts(I).X <> Key2Pts(I - 1).X) Or (Key2Pts(I).Y <> Key2Pts(I - 1).Y)) Then
    Done = False
    TempInt = I + 2
    While Not Done
      If (Key2Pts(TempInt).X <> Key2Pts(I).X) Or (Key2Pts(TempInt).Y <> Key2Pts(I).Y) Then
        Done = True
      Else
        TempInt = TempInt + 1
      End If
    Wend
    v1.X = Key2Pts(TempInt).X - Key2Pts(I).X
    v1.Y = Key2Pts(TempInt).Y - Key2Pts(I).Y
    v2.X = Key2Pts(I).X - Key2Pts(I - 1).X
    v2.Y = Key2Pts(I).Y - Key2Pts(I - 1).Y
    vCrossProd = Cross2D(v1, v2)
    If vCrossProd > Epsilon Then
      Theta2(I) = PI - 0.5 * Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(TempInt)) - (PI / 2)
    Else
      If vCrossProd < -Epsilon Then
        Theta2(I) = -(PI - 0.5 * Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(TempInt)) - (PI / 2))
      Else
        Theta2(I) = 0  '
      End If
    End If
  Else
    If (Key2Pts(I - 1).X = Key2Pts(I).X) And (Key2Pts(I - 1).Y = Key2Pts(I).Y) _
      And (Key2Pts(I).X = Key2Pts(I + 1).X) And (Key2Pts(I).Y = Key2Pts(I + 1).Y) Then
      Theta2(I) = 0
    Else
```

```
        If (Key2Pts(I - 1).X = Key2Pts(I).X) And (Key2Pts(I - 1).Y = Key2Pts(I).Y) _
          And ((Key2Pts(I).X <> Key2Pts(I + 1).X) Or (Key2Pts(I).Y <> Key2Pts(I + 1).Y)) _
       Then
          Done = False
          TempInt = I - 1
          While Not Done
             If (Key2Pts(TempInt).X <> Key2Pts(I - 1).X Or _
               Key2Pts(TempInt).Y <> Key2Pts(I - 1).Y) Then
              Done = True
            Else
              TempInt = TempInt - 1
              If TempInt = 0 Then
                Done = True
              End If
            End If
          Wend
          Theta2(I) = Theta2(TempInt + 1)
        Else ' all points are distinct
          v1.X = Key2Pts(I + 1).X - Key2Pts(I).X
          v1.Y = Key2Pts(I + 1).Y - Key2Pts(I).Y
          v2.X = Key2Pts(I).X - Key2Pts(I - 1).X
          v2.Y = Key2Pts(I).Y - Key2Pts(I - 1).Y
          vCrossProd = Cross2D(v1, v2)
          If vCrossProd > Epsilon Then
             Theta2(I) = PI - Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(I + 1))
          Else
             If vCrossProd < -Epsilon Then
               Theta2(I) = -(PI - Angle(Key2Pts(I - 1), Key2Pts(I), Key2Pts(I + 1)))
             Else
               Theta2(I) = 0
             End If
          End If
        End If
      End If
    End If
  End If
Next I
For I = 0 To (NumPts - 1) Step 1
   ' Find the lengths of all edges of the polygon
  L1(I) = Length(Key1Pts(I + 1), Key1Pts(I))
  L2(I) = Length(Key2Pts(I + 1), Key2Pts(I))
Next I
Dim AxisPt As Coords
 ' Calculate the angle between the horizontal line through the anchor point
' and the first edge of the polygon
AxisPt.X = Key1Pts(0).X + 1
AxisPt.Y = Key1Pts(0).Y
TempInt = 1
Done = False
While Not Done
   If (Key1Pts(0).X = Key1Pts(TempInt).X) And (Key1Pts(0).Y = Key1Pts(TempInt).Y) Then
     TempInt = TempInt + 1
  Else
    Done = True
```

```
  End If
Wend

 Alpha1 = Angle(Key1Pts(TempInt), Key1Pts(0), AxisPt)
 AxisPt.X = Key2Pts(0).X + 1
 AxisPt.Y = Key2Pts(0).Y
TempInt = 1
Done = False
While Not Done
   If (Key2Pts(0).X = Key2Pts(TempInt).X) And (Key2Pts(0).Y = Key2Pts(TempInt).Y) Then
     TempInt = TempInt + 1
  Else
     Done = True
  End If
Wend
 Alpha2 = Angle(Key2Pts(TempInt), Key2Pts(0), AxisPt)
' Here insert tweaking stuff
Dim MaxEdgeLengthDiff As Double
Dim TempLength As Double
MaxEdgeLengthDiff = 0.1
For I = 0 To (NumPts - 1) Step 1
   TempLength = Abs(L1(I) - L2(I))
   If TempLength > MaxEdgeLengthDiff Then
      MaxEdgeLengthDiff = TempLength
  End If
Next I
LSmall = 0.0001 * MaxEdgeLengthDiff
For I = 0 To (NumPts - 1) Step 1
   L12(I) = Maximum(Abs(L1(I) - L2(I)), LSmall)
Next I
For tI = 1 To (NumInBetweens) Step 1
   t = tI / (NumInBetweens + 1)
   Alpha(0) = (1 - t) * Alpha1 + t * Alpha2
   For I = 0 To (NumPts - 1) Step 1
      Theta(I + 1) = (1 - t) * Theta1(I + 1) + t * Theta2(I + 1)
      If I > 0 Then
         Alpha(I) = Alpha(I - 1) - Theta(I)
     End If
   Next I
   E = 0
   f = 0
   G = 0
   For I = 0 To (NumPts - 1) Step 1
      E = E + L12(I) * L12(I) * Cos(Alpha(I)) * Cos(Alpha(I))
      f = f + L12(I) * L12(I) * Sin(Alpha(I)) * Cos(Alpha(I))
      G = G + L12(I) * L12(I) * Sin(Alpha(I)) * Sin(Alpha(I))
   Next I

   U = 0
   V = 0
   For I = 0 To (NumPts - 1) Step 1

      U = U + (((1 - t) * L1(I) + t * L2(I)) * Cos(Alpha(I)))
```

```
    V = V + (((1 - t) * L1(I) + t * L2(I)) * Sin(Alpha(I)))
Next I

U = U * 2
V = V * 2
Lambda1 = (U * G - f * V) / (E * G - f * f)
Lambda2 = (E * V - U * f) / (E * G - f * f)

For I = 0 To (NumPts) Step 1
    S(I) = -0.5 * L12(I) * L12(I) * (Lambda1 * Cos(Alpha(I)) + Lambda2 * Sin(Alpha(I)))
    L(I) = (1 - t) * L1(I) + t * L2(I) + S(I)
Next I

 MorphPts(0).X = (1 - t) * Key1Pts(0).X + t * Key2Pts(0).X
 MorphPts(0).Y = (1 - t) * Key1Pts(0).Y + t * Key2Pts(0).Y
 MorphPts(1).X = Cos(Alpha(0)) * L(0) + MorphPts(0).X
 MorphPts(1).Y = Sin(Alpha(0)) * L(0) + MorphPts(0).Y

For I = 2 To (NumPts) Step 1
    MorphPts(I).X = MorphPts(I - 1).X + _
                Cos(Alpha(I - 1)) * L(I - 1)
    MorphPts(I).Y = MorphPts(I - 1).Y + _
                Sin(Alpha(I - 1)) * L(I - 1)
Next I
Select Case t1
  Case 1
      Set TempPic = picMorph1
     Draw = True
  Case 2
      Set TempPic = picMorph2
     Draw = True
  Case 3
      Set TempPic = picMorph3
     Draw = True
  Case 4
      Set TempPic = picMorph4
     Draw = True
  Case 5
      Set TempPic = picMorph5
     Draw = True
End Select
Dim Pt0 As Coords
Dim Pt1 As Coords
Dim Pt2 As Coords
Dim Pt3 As Coords
Dim t2 As Integer
Dim Tempt As Double
Dim NumCurves As Integer
Dim TempX As Double
Dim TempY As Double
Dim II As Integer

NumCurves = ((NumPts - 4) / 3)
```

```
    If Draw Then
      For II = 0 To NumCurves Step 1
        Pt0.X = MorphPts(3 *¶I).X
        Pt0.Y = MorphPts(3 * II).Y
        Pt1.X = MorphPts(3 * II + 1).X
        Pt1.Y = MorphPts(3 * II + 1).Y
        Pt2.X = MorphPts(3 *¶I + 2).X
        Pt2.Y = MorphPts(3 *¶I + 2).Y
        Pt3.X = MorphPts(3 *¶I + 3).X
        Pt3.Y = MorphPts(3 *¶I + 3).Y
        For t2 = 0 To 200 Step 1
        'calculate and plot the point of the bezier curve
          Tempt = t2 / 200
          TempX = (1 - Tempt) *¶(1 - Tempt) *¶(1 - Tempt) *¶Pt0.X + _
              3 *¶Tempt *¶(1 - Tempt) *¶(1 - Tempt) * Pt1.X _
              + 3 *¶Tempt *¶Tempt *¶(1 - Tempt) *¶Pt2.X + _
              Tempt *¶Tempt *¶Tempt * Pt3.X
          TempY = (1 - Tempt) * (1 - Tempt) *¶(1 - Tempt) * Pt0.Y + _
              3 * Tempt *¶(1 - Tempt) *¶(1 - Tempt) *¶Pt1.Y _
              + 3 *¶Tempt *¶Tempt *¶(1 - Tempt) *¶Pt2.Y + _
              Tempt *¶Tempt *¶Tempt * Pt3.Y
           TempPic.Circle (10 *¶TempX, TempPic.Height - (10 * TempY)), 0.2
        Next t2
      Next II
    End If
    Draw = False
  Next t1
End Sub
```

References

[1] Thomas W. Sederberg and Eugene Greenwood. A physically based approach to 2D shape blending. *Computer Graphics (Proc. SIGGRAPH)*, **26**(2):25-34, 1992.

[2] Thomas W. Sederberg, Peisheng Gao, Guojin Wang, and Hong Mu. 2D shape blending: an intrinsic solution to the vertex path problem. *Computer Graphics (Proc. SIGGRAPH)*, **27**:15-18, 1993.

[3] Shmuel Cohen, Gershon Elber and Reuven Bar-Yehuda. Matching of freeform curves. *Computer Aided Design*, **29**(5):369-378, 1997.

[4] James Foley, Andries van Dam, Steven Feiner and John Hughes. Computer Graphics - Principles and Practice, 2nd ed. Addison Wesley Publishing Company, Reading, Massachusetts. 1987.

[5] Francis S. Hill, Jr. Computer Graphics. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1990.

[6] Josef Hoschek and Dieter Lasser. Fundamentals of Computer Aided Geometric Design. A K Peters, Wellesley, Massachusetts. 1993.

[7] H. Fuchs, A.M. Kedem, and S.P. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM,* **20**(10):693-702, 1977.

[8] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. *Computer Graphics (Proc. SIGGRAPH),* **26**(2):35- 42, 1992.

[9] Thomas W. Sederberg and Eugene Greenwood. Shape Blending of 2-D Piecewise Curves. *Mathematical Methods for Curves and Surfaces,* 497-506, 1995.

[10] Morris G. Cox. Numerical methods for the interpolation and approximation of data by spline functions. Ph.D. thesis, Department of Mathematics, City University, St. John Street, London. 1975.

[11] Hans C. Ohanian. Physics, 2nd ed. W.W. Norton and Company, New York. 1989.

[12] Stephen H. Crandall, Norman C. Dahl, and Thomas J. Lardner. An Introduction to the Mechanics of Solids, 2nd ed. McGraw-Hill Book Company, New York. 1978.

[13] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. An Introduction to Splines for use in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[14] Robert C. Beach. An Introduction to the Curves and Surfaces of Computer-Aided Design. Van Nostrand Reinhold, 115 Fifth Avenue, New York, New York, 10003, 1991.

[15] Kenneth R. Castleman. Digital Image Processing. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 07632, 1996.

[16] James R. Kent, Wayne E. Carlson, and Richard E. Parent. Shape transformation for polyhedral objects. *Computer Graphics (Proc. SIGGRAPH)*, **26**(2):47-54,1992.

[17] Pascal Volino, Nadia Magnenat Thalmann, Shen Jianhua, and Daniel Thalmann. An evolving system for simulating clothes on virtual actors, *IEEE Computer Graphics and Applications*, September, 42-50, 1996.

[18] John F. Hughes. Scheduled Fourier volume morphing. *Computer Graphics (Proc. SIGGRAPH)*, **26**(2):43-46, 1992.