UNIVERSITÄT OSNABRÜCK

Polygon Morphing

A Solution to the Vertex Correspondence Problem in 2D Polygon Morphing

Bachelor Thesis
of
Sven Albrecht

Referees
Prof. Dr. Oliver Vornberger
Prof. Dr. Joachim Hertzberg

Supervisor Dr. Ralf Kunze

Department of Mathematics/Computer Science
Institute of Computer Science

Universität Osnabrück

26th October 2006

Abstract

This thesis deals with an approach, based on human perception to solve the vertex correspondence problem in two dimensional polygon morphing. Correspondences are established between points of high curvature and their local neighborhood in both source and target shape. To compare local neighborhoods, several criteria are introduced which compose a cost function, assigning costs depending on the similarity of local neighborhoods in source and target. The optimal correspondence will be assumed as the correspondence with the least costs overall. Later on implementation details of these ideas will be shown in the Java programming language with the help of exemplary source code.

The ideas of a perceptually based solution for the vertex correspondence problem presented here are elaborations of an algorithm proposed by Liu et al. presented during PG'04 (11).

Acknowledgments

First, I would like to thank some people who helped me during researching and writing this thesis:

- Prof. Oliver Vornberger, for providing me with an interesting and challenging subject for my thesis and acting as first referee.
- Prof. Joachim Hertzberg, for being second referee and for all the seminars concerned with scientific writing in English.
- Dr. Ralf Kunze, for excellent supervision, helpful tips, constructive criticism and providing the pressure I needed in order to finish in time.
- Christof Söger, for listening to my problems and complaints; his continuous encouragement, helpful suggestions and proofreading of this thesis
- Jochen Sprickerhof, for additional help concerning the layout and proofreading
- Klaus Gresbrand for additional proofreading
- last, but not least my parents for financing my studies, and supporting in every decision I have taken so far.

Trademarks

Contents

1 Introduction

1.1 Motivation

Nowadays morphing techniques are widely used in animations, computer graphics, modeling, movie making and apparently also in industrial design (4). Though over the years progress towards automation has been made still a lot of user interaction is required to successfully generate a satisfactory morphing sequence. Especially determining correspondences between source and target image can be a time-consuming process, if both images are complex. The discussed approach focuses on morphing sequences with polygons and tries to reduce the amount of work for a user, by heuristically choosing suitable correspondences for each vertex in a polygon.

1.2 Structure of this thesis

The title of this thesis reads "A Solution to the Vertex Correspondence Problem in 2D Polygon Morphing". This introduction shall provide a short outlook what is meant with this title and what the reader might expect in the following sections.

The term *morphing* in computer graphics accumulates various techniques to transform one image into another image through a seamless transition. This is done by calculating in-between images, so that displaying these images in an orderly fashion creates a smooth animation from the original image (in the following called *source* or $\mathcal{S}$) to the second image (*target* or $\mathcal{T}$). How the in-betweens are created is dependent on the kind of images used and of course on the algorithm employed on the images. The discussed approach can be roughly divided in 5 steps:

1. Source $\mathcal{S}$ and target $\mathcal{T}$ have to be specified as two 2D polygons
2. Both $\mathcal{S}$ and $\mathcal{T}$ will undergo a preprocessing step called *feature point detection* to concentrate in the morphing process on the important parts of a shape
3. The *Vertex Correspondence Problem* has to be solved
4. The *Vertex Path Problem* has to be solved
5. The morph sequence generated by the previous steps has to be displayed

The first step will be elaborated in section 2, which deals mainly with definitions of different two dimensional shapes. Section 3 defines the term *feature point* and describes an algorithm suitable for the preprocessing of $\mathcal{S}$ and $\mathcal{T}$. Section 4 will explain in more detail about morphing of two dimensional shapes and the main problems which need to be solved. First, in 4.1 an introduction to the *Vertex Correspondence Problem* will be given and secondly, in 4.2 the *Vertex Path Problem* will be briefly sketched which is a quite complex problem of its own. A solution with the focus on human perception of the *Vertex Correspondence Problem* will be presented in theoretical detail in 5. The implementation of this ideas in the Java programming language will be discussed in section 6, where exemplary code will be provided as well as an overview using UML class diagrams. Subsequently an introduction in the resulting application will be provided in section 7, demonstrating the user interface and its usage. In section 8 some results created by the application are shown as well as the influence of several algorithmic parameters on a morphing sequence. Finally section 9 will contain an outlook concerning algorithm and application and section 10 will present a final discussion of the the previous sections.

The author wants to encourage readers who are already familiar with certain topics of different sections to skip these sections. If terms used in a section may require previously mentioned knowledge there will be generally a reference provided, so that the reader is able to consult the referenced section in case she needs additional information.

2 Shapes

Although the title of the thesis already restricts the problem to a special case of two dimensional shapes, namely polygons, the algorithmic principles and structures, discussed in section 3 and 5 can also be applied in the more general scenario of two dimensional shapes. Therefore this section will give a short introduction what in the following will be meant with the term *shape* and what *polygons* are in general and in the context of this thesis.

2.1 Two dimensional shapes

A two dimensional *shape* in the context of this thesis is defined as a curve in a two dimensional plane. The curve should be free of self-intersections and is either *non-closed* or *closed*. A shape is called *closed* if it has no visible start and end, otherwise it is *non-closed* (see Fig. 1). If the curve defining a shape does not cross itself the shape is *intersection-free*, else it will be referred to as *self-intersecting*. In general a closed shape can be *filled*, which means that the points inside the curve have a different color than the background, or a shape can be unfilled. For the process of morphing shapes without any self-intersections it is irrelevant if a shape is filled or not, so in the following shapes are assumed to be not filled. A two dimensional point belonging to a shape may have arbitrary color, as long as the curve is distinguishable from the background. Since the color information of points belonging to a shape will be unaccounted for the morphing process, it will be assumed that all points belonging to a shape have the same color.

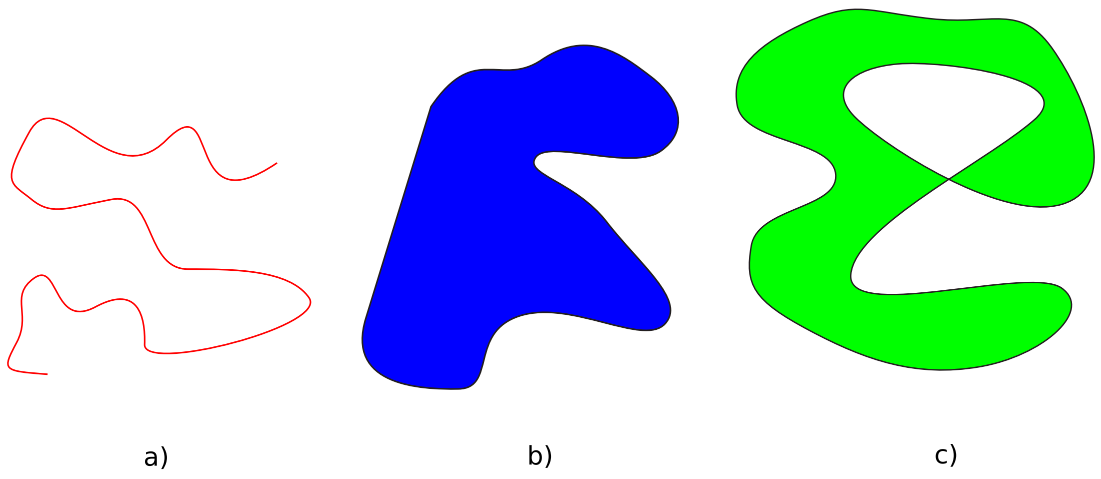


Figure 1: Examples for two dimensional shapes:
a) shows a non-closed intersection-free shape,
b) shows a closed intersection-free filled shape and
c) shows a closed self-intersecting shape

The general definition of a shape does not restrict the way how the shape is represented in forms of data structure in any way. It would be possible to represent a shape just as the set of points which belong to the shape. However such a representation is neither efficient in terms of storage nor suitable to handle for two dimensional graphical transformations like translation, rotation or scaling. Generally it can be assumed that a shape will be represented by an ordered finite collection of points P_i where the points P_{i-1} and P_{i+1} are referred to as predecessor and successor of P_i respectively. If the shape is closed and the collection contains m different points $P_0 = P_m$ holds. In the case of not closed shapes P_0 is called the start point of the shape and P_{m-1} is the end point. Between a point P_i and its successor P_{i+1} the form of the shape will be generally defined by a function, like a Bezier curve or a cubic spline for example.

2.2 Two dimensional polygons

A *polygon* is a special case of a shape as described in 2.1. In a polygon the functions describing the shape between two points are restricted to straight lines. The collection of points of a polygon is referred to as *vertices*, with a single Point P_i being called *vertex*. The line segments connecting two vertices are called either *edges* or *sides*.

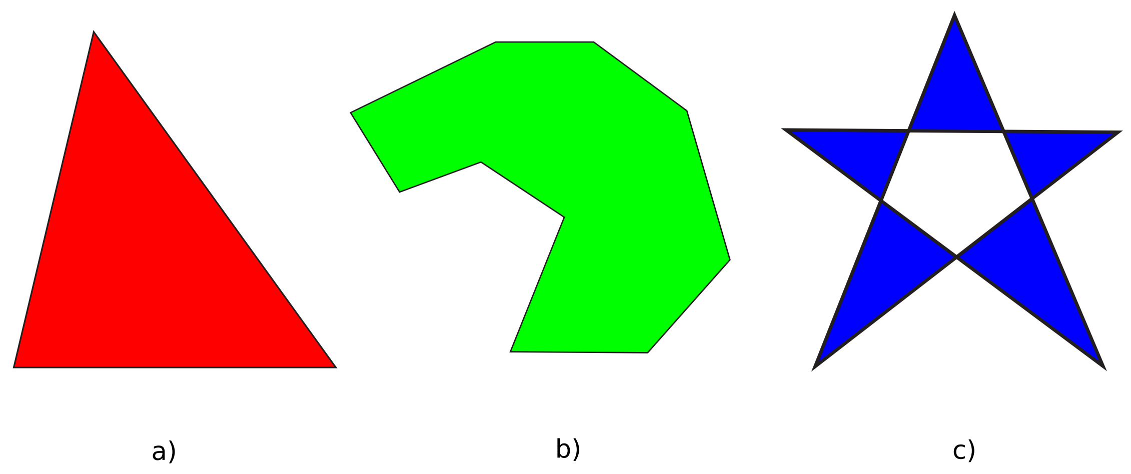


Figure 2: Examples for two dimensional polygons:
a) shows a simple convex polygon,
b) shows a simple concave decagon and
c) shows a complex pentagon

In some definitions a polygon is always closed and a non-closed "polygon" should be referred to as a *polygonal path* or sometimes a *polygonal-line*. A polygon with m vertices is also called *m-gon* where for $m \leq 20$ oftentimes latin names are used (like heptagon for example). If a polygon is free of self-intersections it is *simple*, otherwise it is called *complex* (see Fig. 2). The interior of a simple polygon, bounded by its edges and vertices, is known as the *polygonal region*, which is in some cases also identified with the term *polygon*. In the context of this thesis polygon applies solely to the vertices and the edges. A simple polygon whose interior is a convex set is called a *convex polygon*. Otherwise it is a *concave polygon*. The following two criteria are equivalent to convexity of a polygon:

- Every internal angle at a vertex is at most 180°.
- Every line segment (not only the edges) between two arbitrary vertices remains inside or on the boundary of the polygon.

In a concave polygon at least the internal angle at one vertex is greater than 180°. Such a vertex will be called a *concave vertex*, all other vertices are *convex vertices*. With the definition of a *feature point* (see 3.1) the distinction between *convex feature points* and *concave feature points* will be analogous.

Polygons are generally easier to handle in terms of graphical transformations than other shapes: In most cases it is sufficient to apply the transformation on every single vertex and then draw the edges between the modified vertices, while in the case of shapes possibly other parameters would have to be modified as well, depending on the underlying data structure representing the shape. The simplicity of a polygon compared to a shape has another advantage: The computations needed to display a polygon can be done very efficiently by employing the line algorithm of J. Bresenham based on (3).

3 Feature Points

3.1 Feature Point Definition

The term *feature point* describes, in the context of this thesis, a single point in a two dimensional plane. Besides values for x and y coordinates other properties will be associated with a feature point during the matching process, but those properties and their applications will be discussed later (5.3) in detail.

Every feature point belongs exactly to one shape and the shape in turn can be represented, with a certain loss of information, by all feature points appendant to it. Unlike points in a uniform sample of a shape, feature points are in general not uniformly distributed over a shape. Feature points are high curvature points in a shape, so they mark prominent areas. In shape perception of human observers such points play a dominant role (1). In most cases a shape can be represented better by a small number of feature points, positioned at high curvatures, than a equal number of sample points distributed uniformly over the shape. For this reason the approach described in section 5 utilizes feature points to describe a shape and its properties instead of mere samples. This ensures that areas which are closely observed by humans are represented with satisfactory detail, while areas without any striking curvatures will be represented by few points.

3.2 Detection of feature Points in planar curves

The problem of detecting points of high curvature in two dimensional shapes has been researched since the early 1970's. A recent and fast algorithm was developed by Chetverikov and Szabó (5) in 1999. In their research they compare their approach with other approaches by Rosenfeld and and Johnston (13), Rosenfeld and Weszka (14), Freeman and Davis (7) and Beus and Tio (2). The methods described in the following are based on the descriptions in (5). Information on the approaches (2; 7; 13; 14) can be either found in the according publications or in a short overview in (5).

Chetverikov and Szabó propose in their approach a two-pass algorithm. In a first pass potential candidates are detected and in a second pass the possible candidates are filtered to avoid multiple registration of what should actually be viewed as one high curvature point.

3.2.1 Detecting possible candidates for feature points

In the first step of the two pass algorithm the shape will be represented by an ordered sample of points P_i. The Euclidean distance between a sample point P_i and its adjacent point P_{i+1} should be small, but not necessarily equal for every pair of adjacent points. Such a sequence will be scanned for potential feature points. A potential point of high curvature is detected, if a triangle with specified size and specified opening angle at the point can fit inside the shape.

This is done by a set of three rules:

$$\begin{aligned} d_{min} &\leq \|P_i - P_i^+\| \leq d_{max} \\ d_{min} &\leq \|P_i - P_i^-\| \leq d_{max} \\ \alpha &\leq \alpha_{max}, \end{aligned} \tag{1}$$

where P_i is the candidate to be checked, P_i^+ is a point succeeding P_i in the order of the sequence of the point sample, P_i^- a point preceding P_i and $\alpha \in [-\pi, \pi]$ describes the opening angle of the triangle at P_i. $\|P_i - P_i^+\|$ measures the Euclidean distance between P_i and P_i^+; $\|P_i - P_i^-\|$ is defined analogous. Please observe Fig. 3a) for some visualization.

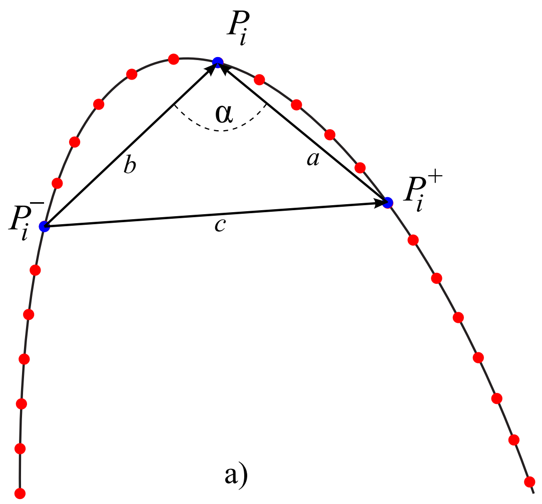


Figure 3: Detecting high curvature points
a) Determining if P_i is a candidate for a feature point. Sample points are colored red, P_i, P_i^+ and P_i^- are colored blue.
b) depicts the same scenario as **a)**, but at a subsequent time with different choices for P_i^+ and P_i^-. For comparison the old choices are visualized less colorful.

In Fig. 3 $\|P_i - P_i^-\|$ is labeled b and $\|P_i - P_i^+\|$ is labeled a. The opening angle α is computed as

$$\alpha = \arccos \frac{a^2 + b^2 - c^2}{2ab}$$

During the computation P_i^- and P_i^+ are moved outward (see Fig. 3b))from P_i until $\|P_i - P_i^-\| \leq d_{max}$ and $\|P_i - P_i^+\| \leq d_{max}$ respectively do not hold. Of all triangles which fulfill all conditions in (1) the least opening angle $\alpha(P_i)$ is selected and $\pi - |\alpha(P_i)|$ stored as the *sharpness* of P_i. If for all permitted points for P_i^- and P_i^+ no triangle satisfies the conditions in (1) P_i is no candidate for a feature point.

The triangles can also be used to determine if a feature point is considered concave or convex: For vectors $\vec{b} = (P_i - P_i^-) = (b_x, b_y, 0)^T \in \mathbb{R}^3$ and $\vec{c} = (P_i^+ - P_i^-) = (c_x, c_y, 0) \in$

$\mathbb{R}^3$ the vector product $\vec{b} \times \vec{c} = \|\vec{b}\|\|\vec{c}\|\vec{n} \sin\theta$ with $\vec{n}$ being a unit vector perpendicular to both $\vec{b}$ and $\vec{c}$ is simplified to $b_x c_y - b_y c_x = \|\vec{b}\|\ |\vec{c}\|\vec{n} \sin\theta$. That means a point where $b_x c_y - b_y c_x \geq 0$ will be considered convex, since that implies $\sin\theta \geq 0$. Otherwise the curvature at the feature point will be considered concave. Every point contained in the point sample of the shape has to go through this process.

3.2.2 Filtering of possible candidates

In the second pass multiple detected feature points for the same corner are filtered. If in the first pass a dense sample of the curve was used to detect potential feature points, it is highly likely that more than one potential candidate was chosen to represent a high curvature (Fig. 4). On the other hand if the curve was not densely sampled the danger to ignore otherwise valid feature points increases. For a visual example please observe Fig. 5.

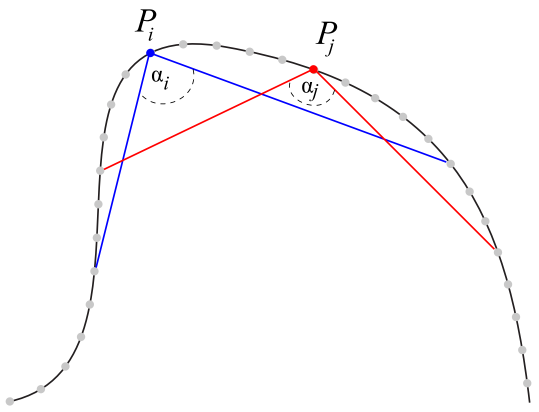


Figure 4: Choosing between feature point candidates
Sampling points are depicted in gray. Points P_i and P_j have been detected as candidates for feature points. The *sharpness* of both candidates will be tested to determine which candidate will be chosen to represent the feature. In the depicted scenario the blue colored P_i will be preferred to the red colored P_j, because of its greater sharpness.

If one corner is represented by multiple feature point candidates, it can be assumed that all those candidates are consecutive sample points. It seems suitable to choose the point from all candidates which has the strongest response to the corner. The *sharpness* $\pi - |\alpha(P_i)|$ can be utilized to determine which point P_i should be associated with the corner. A candidate P_i will be discarded if another candidate P_j exists in its neighborhood and the *sharpness* assigned to P_j is greater than the *sharpness* assigned to P_i. Put into a formula this means $\alpha(P_i) > \alpha(P_j)$ holds. A visualization is depicted in Fig. 4. The term "neighborhood" of a point P_i leaves a certain play in the implementation. For example the neighborhood for a point P_i could consist of $P_j \in \{P_j \mid \|P_i - P_j\| \leq d_{max}\}$.

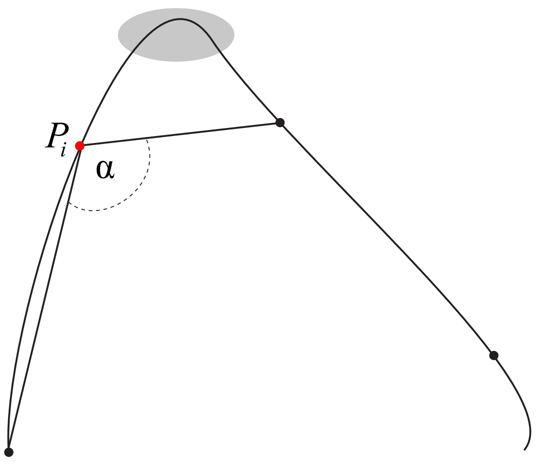


Figure 5: Suboptimal feature detection due to lose sampling
Sampling points are depicted a black points, the detected feature point is colored red. Instead of the red point a more dense sampling would have produced a preferable feature point in the gray area.

However choosing all points $P_j \in \{P_j \mid \|P_i - P_j\| \leq d_{min}\}$ or all consecutive points $P_j, i < j$ and $P_k, k < i$ which are valid candidates are equally meaningful definitions for a neighborhood.

Parameters and their meaning Though first and second pass of the algorithm are described in sections 3.2.1 and 3.2.2 there might still be some questions concerning the meaning and influence of the parameters d_{min}, d_{max} and α_{max}. The maximal angle α_{max} sets the upper limit for the opening angle at a potential feature point. For example if α_{max} is set to 130° the triangle fitting into a corner of the shape must have an opening angle of less than or equal to 130° in order to be a valid candidate. If a shape contains many areas of high curvature and it should be represented by a small number of feature points it is appropriate to set α_{max} to a relative small value. For shapes which do not feature many points of high curvature a larger value for α_{max} is advised, so that at least some feature points can be detected. The lower limit d_{min} restricts the detection of feature points to areas of a certain size. This way small sized areas of high curvature, that would not be noticed by an observer, will be ignored and the detection becomes more robust against the appearance of noise in the shape. The upper limit d_{max} is used to prevent detection of false feature points. Without this upper limit triangles with small opening angles could be constructed, which are not located inside the corresponding corner anymore. For example every sample point P_i in a closed shape would be detected as a feature point by using points on the opposite side of the shape as P_i^- and P_i^+ (see Fig. 6).

While the optimal choice of d_{min}, d_{max} and α_{max} of course depends on the kind of shape on which the algorithm should be applied, it turns out that the algorithm is

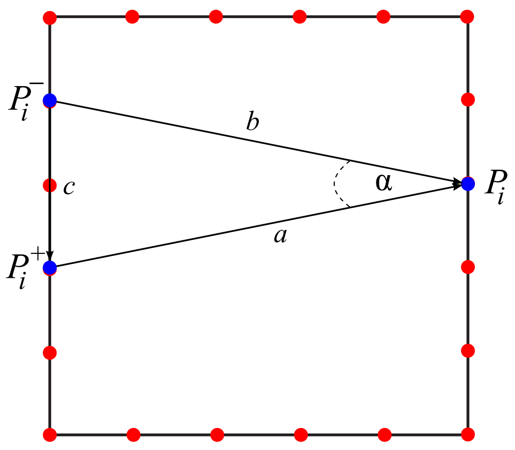


Figure 6: Wrongly detected feature point
due to missing restriction of d_{max}. Without a boundary for d_{max} every sampling point can become a feature point in a closed shape. Sampling points are colored red, the falsely detected feature point candidate P_i, P_i^+ and P_i1- are colored blue.

quite robust in relation to the parameters. In the experiments conducted in (5) the algorithm was tested on various shapes without any fine-tuning of the used parameters and delivered satisfying results. The default values used in (5) were $d_{min} = 7$, $d_{max} = 9$ and $\alpha_{max} = 150°$.

In the case of purely polygonal shapes the detection of feature points is of reduced difficulty. The only possible candidates for feature points can be found at the junction of two edges. Without any further algorithm one could simply assume that every vertex of a polygon should be a feature point. A more sophisticated approach would be to use two similar criteria to determine if a vertex is a feature point. Firstly only vertices with an opening angle smaller than α_{max} should be considered as a feature point. This way the detection of feature points at almost straight segments of a polygon, containing more than two vertices, can be avoided. Secondly a parameter similar to d_{min} should be used. Instead of comparing the combined length of both edges connecting a vertex with its adjacent vertices to an absolute value, another solution seems to be more feasible: Only vertices where the combined length of both edges in respect to the total length of all edges is larger than a threshold should be considered feature points. This ensures that the parameter is independent from the size of the total polygon and will be unaffected by operations like scaling of the polygon.

Additional remark Though the algorithm proposed in (5) is robust it has its limitations. For example one should keep in mind that it can only detect feature points in a shape if they exist. This remark might sound strange, but for example in a circle no feature points will be detected. The same would apply for a *n-gon* (polygon with n vertices) where for every internal angle $\alpha(P_i)$ the equation $\alpha(P_i) > \alpha_{max}$ holds. However this should not be treated as a flaw of the algorithm, but as a special feature of these shapes: A single-colored circle just has no point on which an observer would especially focus on and the same applies for shapes with extremely large internal angles.

4 Morphing of two dimensional shapes

As already mentioned in the introduction 1 morphing of two dimensional shapes can be divided into two subproblems which have to be solved. These problems are called the *Vertex Correspondence Problem* (VCP) and the *Vertex Path Problem* (VPP). Common literature concerned with shape or polygon morphing addresses mostly the VPP, while suggested solutions to the VCP tend to be a bit more sparse. However both problems are equally important and in the chronology of a morphing sequence the VCP is a requirement before the VPP can be solved. In the case of morphing of two dimensional shapes special attention is turned to the goal of creating morphs where all in-between shapes are free of self-intersections, because apart from some fanciful special cases a morphing sequence that contains self-intersections will be perceived as an “unnatural” transition from source to target. To sum up both problems in one short phrase, one might say that the VCP has to solve which point from $\mathcal{S}$ shall travel to which point of $\mathcal{T}$ and the VPP is concerned with the paths on which every point has to travel during the morphing process. In section 4.1 the VCP will be described in more detail, but without any suggested solution. Such a solution will be discussed in 5. The VPP will be briefly presented in section 4.2.

4.1 The Vertex Correspondence Problem

Before any animation can take place in the morphing of two dimensional shapes a *correspondence* between *source* and *target* has to be established. This correspondence assigns to every point $S_i = (x_{P_i}, y_{P_i})$ contained in $\mathcal{S}$ a corresponding point $T_j = (x_{T_j}, y_{T_j})$ in $\mathcal{T}$. Since a shape can easily contain thousands of points the magnitude of this problem has to be downsized, before it becomes computational processable. Therefore algorithms often focus not on every point contained in a shape, but the points defining the appearance of a shape, namely feature points (see section 3) or in the case of polygons oftentimes just all vertices. Points belonging to edges between two vertices P_i and P_j will be assigned to points between the two corresponding points $T_{C(i)}$ and $T_{C(j)}$, where $C(i)$ is a function acquiring a correspondence for a point. So basically if the VCP is solved, for every point contained in a shape the starting position in the first frame of the animation $P_i = (x_{P_i}, y_{P_i})$ is known and the end position in the last frame $T_{C(i)} = (x_{T_{C(i)}}, y_{T_{C(i)}})$.

Once such a correspondence is established the *Vertex Path Problem* (see section 4.2) arises, which deals with the traveling path for every point during the morph.

However one might wonder why the VCP is a noteworthy problem. If a human being is confronted with *source* and *target* in form of two polygons she often has a clear notion which vertices should correspond to each other. In many cases these correspondences are established intuitively in a split second, so for a human being the VCP seems to be quite trivial. Unfortunately computers do not perceive the world in the way a human being does (otherwise a lot of challenging problems like *symbol grounding* in AI and knowledge based robotics would have been solved probably years ago). In the following the VCP will be discussed for the case of two dimensional polygons, but the statements can be transfered to the case two dimensional shapes without much modification.

A valid solution to the VCP can be interpreted as a bijective mapping function from the vertices contained in $\mathcal{S}$ to the vertices in $\mathcal{T}$, in a way that for every vertex in $\mathcal{S}$ there is exactly one vertex in $\mathcal{T}$ which it corresponds to and vice versa. Obviously such a mapping is not always possible, without further modifications of the polygons involved, if for example $\mathcal{S}$ and $\mathcal{T}$ do not contain the same number of vertices. On the other hand only a small number of these mappings would qualify as a suitable solution to the VCP: A morphing sequence that satisfies an observer should be free of self-intersecting motions during the morphing sequence. Thus in many cases, though it might sound implausible at first, even if $\mathcal{S}$ and $\mathcal{T}$ have the same number of vertices a solution is not trivial. Assuming there was an algorithm that could efficiently compute the correspondence with the least traveling distance for all vertices, and additionally assuming linear animation paths, there are many simple cases where such a solution would lead to self-intersections. A simple example is depicted in Fig. 7 and the reader will surely be able to come up with more simple examples.

These demands increase the complexity of the VCP. To give the reader an idea of the complexity the author will try to depict it step by step:

If *source* $\mathcal{S}$ has m vertices and *target* $\mathcal{T}$ has n vertices there are in theory $n \cdot m$ correspondences, since one point in $\mathcal{S}$ could choose between n correspondences in $\mathcal{T}$. However in this consideration every vertex is treated independently of all other vertices, contained in the same polygon. Since a bijective mapping is required it has to be ensured that a vertex in $\mathcal{T}$ corresponding with one vertex in $\mathcal{S}$ can not be chosen by another vertex in $\mathcal{S}$ as its correspondence. That means already established correspondences have to be kept track of. In addition a policy is needed to decide if two vertices S_{i_1} and S_{i_2} of $\mathcal{S}$ are candidates for a correspondence with T_j of $\mathcal{T}$ which vertex will be preferred. An example for a very simple policy would be a "first come, first served" policy, but such a policy will certainly not deliver suitable results in the general case. Even if some sophisticated policy is employed and the result is a bijective mapping, the vertices are still treated independently from each other, just double correspondences have been prevented. A desired mapping is not only bijective, but has to fulfill certain ordering constraints. For example, if it is assumed that $\mathcal{S}$ and $\mathcal{T}$ have the same number n of vertices. If it is further assumed that all vertices with an index smaller than some special index $k-1$ correspond to each other and the same applies for all vertices greater than index $k+1$: $S_i \leftrightarrow T_i \; \forall (i < k-1) \vee (i > k+1)$, where $S_i \leftrightarrow T_j$ indicates a correspondence between S_i and T_j. If in addition now $S_{k-1} \leftrightarrow T_{k+1}$, $S_k \leftrightarrow T_k$ and $S_{k+1} \leftrightarrow T_{k-1}$ hold, the result may be a bijective mapping, but is not a solution for the VCP. The mapping is invalid, because it will be impossible to find animation paths (see 4.2) that will not self-intersect, as demanded in the beginning of section 4. To visualize the above described exemplary scenario please observe the scene depicted in Fig. 8.

Algorithms trying to solve the VCP either have to recognize such invalid mappings and refuse them or have to be designed in a way that they will only construct valid mappings. If an algorithm is able to find more than one valid mapping, again a policy is needed to choose which mapping should be preferred.

In the common case that $\mathcal{S}$ and $\mathcal{T}$ do not contain the same number of vertices additional criteria need to be introduced. These criteria could indicate for example if it

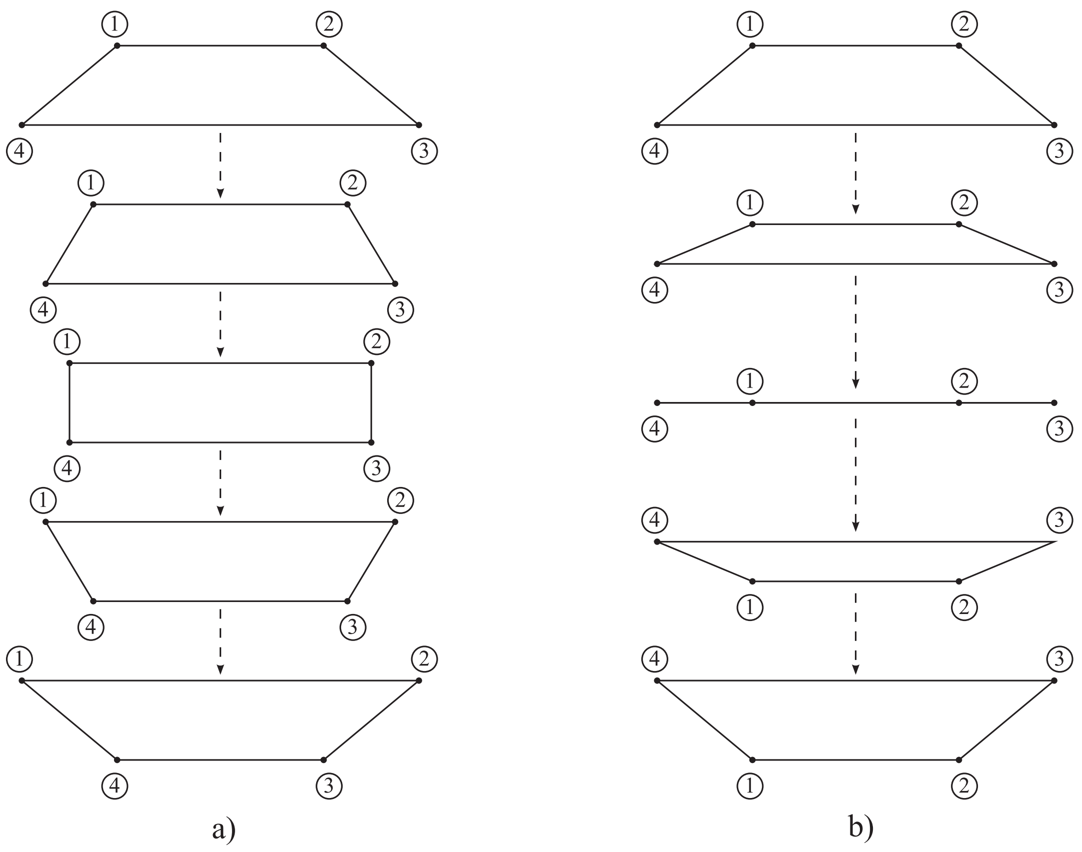


Figure 7: Intersections caused by minimal vertex movement
a) Source is depicted on top and target at the bottom. Between source and target several in-between polygons are depicted, linked by dashed arrows. Vertices are marked by black points and the numbers indicate the movement of the vertex during the morphing sequence.In this correspondence and the animation paths for each vertex self-intersections are prevented.
b) Labeling as described in **a)**. This morphing sequence is the result if only the least total movement of all vertices is taken as a criteria to choose correspondences.

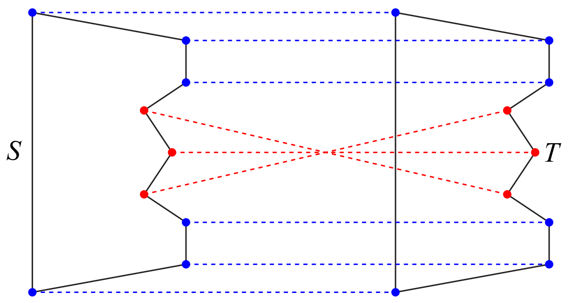


Figure 8: Intersecting correspondences
Polygon $\mathcal{S}$ depicted on the left, (identical) polygon $\mathcal{T}$ depicted on the right. Vertices are indicated with points, correspondences between vertices are depicted with dashed lines. Red lines and points indicate unavoidable self-intersections during the morphing, if the blue ones are set.

would be beneficial to completely ignore vertices or to add additional vertices into one of the polygons, so that in the end a bijective mapping is achieved.

All these requirements prevent the VCP to be solvable in polynomial runtime. Efficient implementations will, like in the domain of path finding, try to use heuristics to find solutions, instead of calculating the actual optimum. Of course the quality of the solutions depends strongly on the policies used to choose which correspondences and mappings are to be preferred. Such policies can try to approximate human perception, but considering the current state of the art will not always emulate successfully. This fact and the use of heuristics prevent that even sophisticated approaches will always deliver the result expected by a human observer. However this might be entirely impossible since not all humans would perceive a scene in the same way and might intuitively choose different correspondences.

A common technique to simplify the VCP is to let a human operator choose a few initial correspondences of the most important features and then try to find suitable correspondences for the remaining vertices between the manually established correspondences.

The goal of (11) and their approach described in section 5 is to get a robust heuristic, that will deliver satisfactory results in many scenarios without human interaction.

4.2 The Vertex Path Problem

If a complete correspondence between source and target is established either by a human operator or by an algorithmic solution of the VCP, the Vertex Path Problem has to be solved. If for a Point $P_{i_{source}} = (x_{i_{source}}, y_{i_{source}})$ of source its destination $P_{i_{target}} = (x_{i_{target}}, y_{i_{target}})$ is set, the question arises, how that point should travel from $P_{i_{source}}$ to $P_{i_{target}}$. Again this question may seem trivial, since one could simply use the straight line connection $P_{i_{source}}$ and $P_{i_{target}}$. Indeed this is already a simple and valid solution in some cases. However, if every point travels on a line, the points will travel in a common scenario with different speed, since the Euclidean distance between every pair of corresponding vertices will not be the same. This can lead to self-intersections in the morphing sequence, which are to be prevented if possible. There are also a lot of other scenarios in which self-intersections would occur if every vertex would simply travel on a straight line (see Fig. 9). The same exemplary scenario could be solved, if all vertices travel on other curves, for example Bezier curves, without self-intersections as is depicted in Fig. 10.

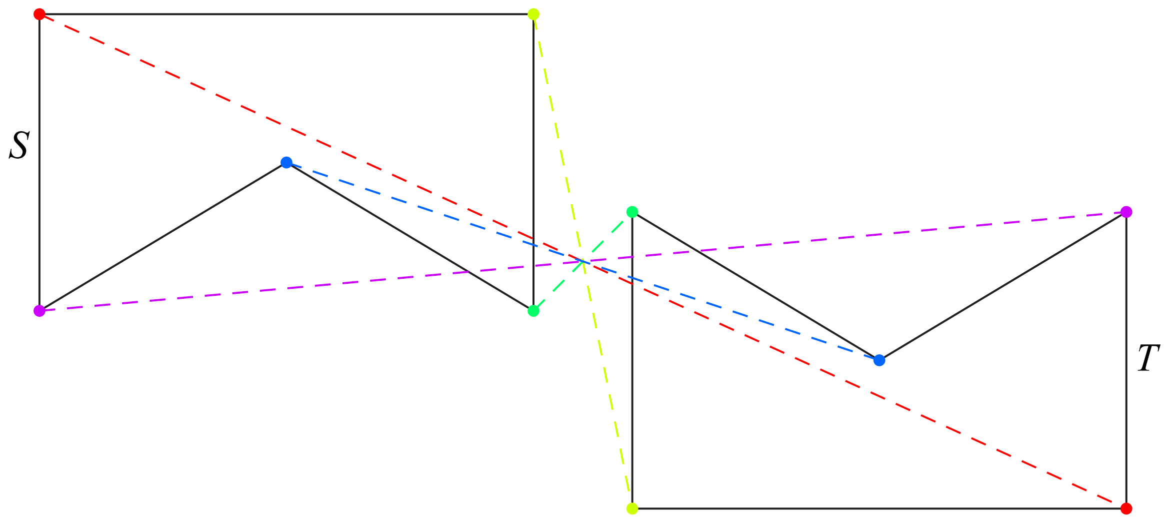


Figure 9: Self-intersections caused by linear animation paths
Correspondences between vertices are indicated by the same color, the animation paths are depicted as dashed colored lines between the corresponding vertices

To avoid self-intersections many different approaches have been introduced since the late 1980's. In 1992 Sederberg and Greenwood (16) introduced an approach they called "physically based" which tries to avoid self-intersections and handles both the VCP and the VPP. Based on the solution of the VCP presented in (16) Sederberg et al. suggested another approach to the VPP in (15). In 1995 Shapira and Rappoport (17) introduced another method using so called "star-skeletons" to avoid global self-intersections, given a complete correspondence. Though these approaches might not be the state of the art any more, they provide good introduction into the problem and additional background information. A more recent approach, also requiring a solution to the VCP, is suggested by Gotsman and Surazhsky in (8), which *guarantees* that polygons in the in-between sequence are also simple, if source and target are simple.

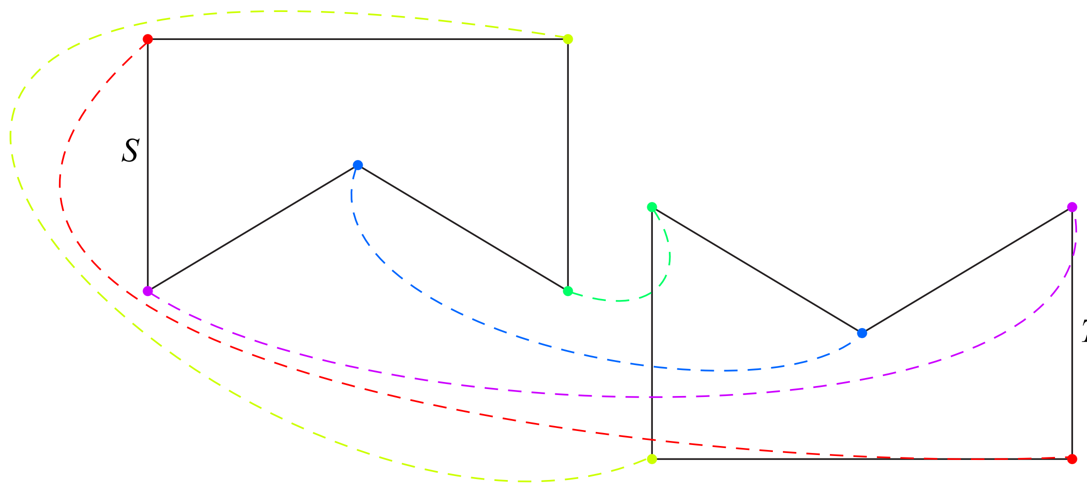


Figure 10: Avoided self-intersection in animation
Same scenario as in Fig. 9, but this time using (handmade) Bezier curves as animation paths avoids self-intersections, again correspondences and associated animation paths are distinguished by color

5 A human perception based approach

5.1 Overview

The following ideas and algorithms were developed and published by Ligang Liu, Guopu Wang, Bo Zhang, Baining Guo, and Heung-Yeung Shum in 2004 (11). In their approach the process of developing correspondences between vertices of source and target is focused mainly on the way humans perceive a polygon and how they notice changes in shapes. Though (11) is an excellent source to gain insight into their ideas some parts of their algorithms are sketched rather briefly. Their algorithms were implemented in C♯, but source code is not freely available, to the best of the author's knowledge.

This section will deal with the theoretical part of the algorithm while section 6 will provide details on the implementation of the approach presented here.

5.2 Algorithmic overview

Liu et al.(11) focus on a perceptually based approach in their ideas. This means that the algorithm will try to find common characteristics between source and target, which stand out to a human observer. Basically this means that vertices at pointed corners are observed more consciously than relatively flat corners. Also the size of a part of a shape plays an important role: If the distances from a vertex to its predecessor and successor in a polygon are relatively small in relation to the total length of the polygon, its absence may go almost unnoted for a human observer. However if the distance to predecessor and successor contribute greatly to the total length of the polygon the absence of such a vertex would be noticed immediately. In the general case of two dimensional shapes the appearance of the shape between two vertices should also be taken into consideration, when trying to create a correspondence. To visualize these ideas please note the following example: Consider a morphing sequence between the polygon in Fig.11a the polygon depicted in Fig. 11b. As you may recognize both polygons can be interpreted as humans, each wearing a hat and waving their right arm. For a human observer it will probably feel "natural" if during a morphing the heads of Fig. 11a and Fig. 11b will correspond to each other as well as the legs and arms of Fig. 11a and Fig. 11b from left to right should correspond to each other.

5.3 Similarity and Discard Costs

As mentioned in 4.1 criteria are needed in order to decide which correspondences between vertices can be considered suitable. Taking into account the desire for morphings which should fulfill the demands expressed in 5.2 the criteria must be able to distinguish between similar parts of two different shapes and parts that are unlike in appearance.

For this purpose a cost function seems to be reasonable, which assigns high costs to correspondences of local areas of shapes which are different in appearance and assigns low costs to areas which are likely to resemble each other. As you may notice the focus here is on "areas" and not on single feature points, implying that the local neighborhood

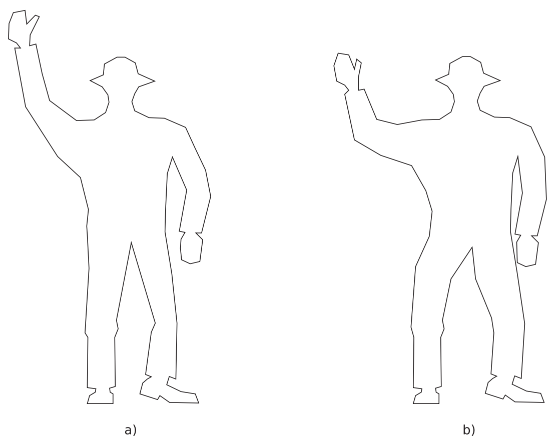


Figure 11: Exemplary source and target for a morph
a) depicts the source and **b)** the target shape. During a morphing sequence it would be desirable if noticeable parts of the shapes like heads, arms and legs correspond.

of a feature point will be taken into account as well. This will be done by the *Region of Support*.

5.3.1 Region of Support

In the following sections the term *Region of Support* will be abbreviated using simply *ROS*. The *ROS* is defined as the local neighborhood of a feature point P_i as follows:

$$ROS_h(P_i) = \{P_j | j = i - h, i - h + 1, \ldots, i + h\} \tag{2}$$

Where h is an integer which can be varied by the user. The influence of the parameter h will be explained later in this section.

Several studies on the extraction of features from point clouds (9; 12) show the use of the covariance of a local neighborhood of a point to estimate local surface properties of a shape. These methods can be utilized for feature points P_i and their appendant $ROS_h(P_i)$. To calculate the covariance for a point $P_i = (x_i, y_i)$, first the center of $ROS_h(P_i)$ is calculated, which will be denoted as $\bar{P}_i$.

$$\bar{P}_i = \frac{1}{2h+1} \sum_{j=i-h}^{i+h} P_j \tag{3}$$

With the help of $\bar{P}_i$ the covariance matrix of $ROS_h(P_i)$ is defined as

$$C(P_i) = \frac{1}{2h+1} \sum_{j=i-h}^{i+h} (P_j - \bar{P}_i)^T (P_j - \bar{P}_i) \tag{4}$$

The resulting 2×2 matrix together with its eigenvectors $\{e_0, e_1\}$ and the corresponding eigenvalues $\{\lambda_0, \lambda_1\}$ define the correlation ellipse which resembles the general form of $ROS_h(P_i)$, (see Fig. 12). As you can see one of the eigenvectors points closely into the tangent direction of the local region, while the other eigenvector points into the direction of the normal. Thus one eigenvector can be called the tangent eigenvector e_T with its corresponding eigenvalue λ_T and the other one will be referred to as the normal eigenvector e_N with the normal eigenvalue λ_N.

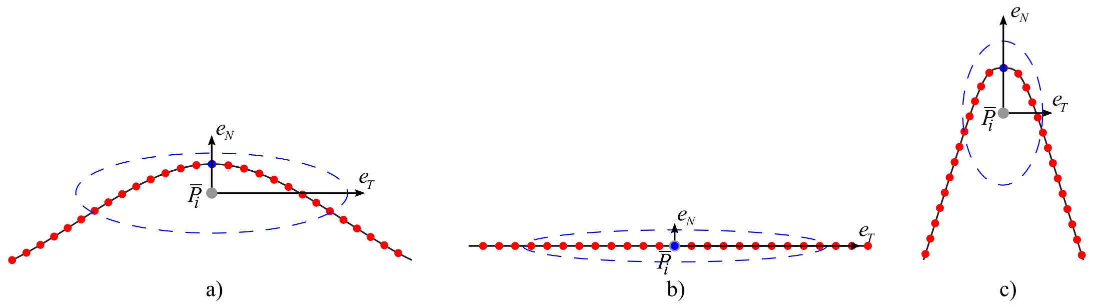


Figure 12: Eigenvectors, eigenvalues at different shapes
Points belonging to the region of support are colored red, the point P_i is colored blue and the center of ROS $\bar{P}_i$ is colored gray.
a) $\lambda_T > \lambda_N > 0$ **b)** $\lambda_T > \lambda_N = 0$ **c)** $\lambda_N > \lambda_T > 0$

To determine which eigenvector points into the direction of the normal, the bisector of the angle created by a feature point P_i and its nearest neighbors in $ROS_h(P_i)$ (the Points P_{i-1} andP_{i+1}) is calculated. Since the tangent meets the bisector at a right angle, the dot product of both eigenvectors and bisector is calculated. From linear Algebra it is known for two vectors u and v

$$\langle u, v \rangle = \|u\| \|v\| \cos\theta \Leftrightarrow \cos\theta = \frac{\langle u, v \rangle}{\|u\| \|v\|}, \quad u, v \neq 0$$

Thus the eigenvector with the larger result in the dot product with the bisector can be considered the normal eigenvector e_N while the eigenvector with the smaller result can be assumed to be the tangent eigenvector e_T.

Increasing the number of points in the local neighborhood (by increasing the parameter h) has a similar effect as employing a low-pass filter: The covariance (equation (4)) is defined as the sum of squared distances from the center $\bar{P}_i$ divided by the number of points contained in $ROS_h(P_i)$. Thus if the number of points contained in the local neighborhood increases the influence of a single point on the the resulting covariance matrix decreases. Eigenvalues and eigenvectors are directly dependent on the covariance matrix and thereby the ellipse resembling the shape of the local neighborhood. If it is desired to inhibit the effect of small outliers or noise in certain areas of a shape on the matching process generally a higher value for h is recommended. On the other hand one should try to avoid increasing the size of the local neighborhood to a size where many points belonging to regions beyond adjacent feature points are included, because this would reduce the significance of the appendant feature point and thus the covariance would not necessarily yield valid information on the geometrical properties of the local neighborhood of the feature point.

A compromise to accomplish both, low-pass filtering and preserving including only the local neighborhood of a feature point is a to create $ROS_h(P_i)$ by a semi uniform sampling. That means, the parameter h for the size of $ROS_h(P_i)$ is set to a fixed value and for each feature point P_i its predecessor S_{i-1} and successor S_{i+1} are determined. Please note that points S_{i-1} and S_{i+1} are labeled with "S" instead of "P" to indicated that they are feature points and not just mere vertices. Now the distance between S_{i-1} and P_i along the edges connecting both feature points has to be calculated. The distance information is now used to create a sample of h points distributed uniformly along the edges from S_{i-1} to P_i The first point included into this sample should be S_{i-1}. As center of $ROS_h(P_i)$ of course P_i itself is added and the missing h points in $ROS_h(P_i)$ are calculated between P_i and S_{i+1} using the same method as between S_{i-1} and P_i. The last point added into $ROS_h(P_i)$ should therefore be S_{i+1}. A visual example is given in Fig. 13 a). All points in $ROS_h(P_i)$ between S_{i-1} and P_i will form the so called *Region of left side* $ROL(P_i)$ which describes the appearance of the local neighborhood on the left side of the feature point and the points between P_i and S_{i+1} compose the *Region of right side* $ROR(P_i)$. Both $ROL(P_i)$ and $ROR(P_i)$ are needed in section 5.3.2 along with the $ROS_h(P_i)$.

An alternative solution for the creation of $ROS_h(P_i)$ would be to sample every edge with a constant number of vertices and include all sampling points according to the definition of $ROS_h(P_i)$ in equation (2). The different outcome to the former method is depicted in Fig. 13 b). The number of points which is used to sample one edge of a polygon in combination with parameter h restricts $ROS_h(P_i)$ to a predefined number of edges. Depending on the form of the polygon results of the two different methods to create $ROS_h(P_i)$ may greatly differ (as can be seen by comparison of Fig. 13 a) and 13 b). These difference will of course influence the properties of a feature point described in 5.3.2 and by this the outcome of the resulting correspondence. Which method for the creation of $ROS_h(P_i)$ delivers the better results for the solution of the Vertex Correspondence Problem depends on the form of both source and target.

The first method to calculate $ROS_h(P_i)$ ensures that only parts of the shape between the current feature point P_i and its predecessor S_{i-1} and successor S_{i+1} are included into the sample. The second variant can not ensure this policy. Depending on the ratio of points representing an edge and parameter h it can be chosen how many neighboring edges may will be included into the sample. If these edges are between feature point P_i and its predecessor and successor or may go beyond is not certain. Eigenvectors and eigenvalues are directly dependent on the covariance matrix (see equation (4)). In the first described method the parameter h has little influence on the determination of tangent and normal eigenvector. However in the second method if the number of points representing an edge is fixed, the parameter h has a strong influence on the determination

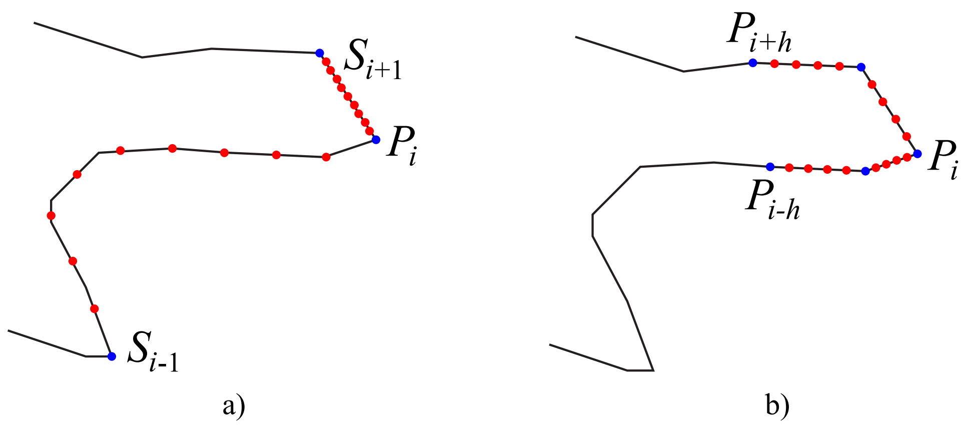


Figure 13: Different methods to calculate $ROS_h(P_i)$
a) Detected feature points are colored blue, sampling points are colored red. Parameter h is set to 10 in this example. As you can see the sampling points between S_{i-1} and P_i are farther apart than the ones between P_i and S_{i+1} due to the greater distance between S_{i-1} and P_i. In both $ROL(P_i)$ and $ROR(P_i)$ all sampling points are distributed uniformly. Parameter α_{max} for the detection of feature points was set to 130°.
b) Vertices are colored blue, sampling points are colored red. Parameter h is set to 10. Number of points representing an edge is set to 5. This means that the 2 neighboring edges on each side of feature point P_i are included in $ROS_h(P_i)$.

of eigenvectors and eigenvalues, since it controls how far $ROS_h(P_i)$ will spread.

5.3.2 Criteria to distinguish Feature Points

Employing the definitions from section 5.3.1 it is possible to assign several values to a feature point which describe the form of its local neighborhood. These values will be used to compare feature points from *source* and *target* and assign costs depending on the similarity of the local neighborhoods. In the following the left and right *Feature Elements* of a feature point shall be denoted as $ROL(P_i)$ respectively $ROR(P_i)$. If the selection of the size of $ROS_h(P_i)$ adhered the suggestion made at the end of section 5.3.1, $ROL(P_i)$ and $ROR(P_i)$ will include h points each.

Liu et al. (11) chose three main criteria to distinguish between the shape of a local neighborhood near a feature point

- Feature Variation:

  The feature variation of a feature point P_i is defined as

  $$\sigma(P_i) = \xi \frac{\lambda_N}{\lambda_N + \lambda_T} \tag{5}$$

  where λ_N is the normal eigenvalue, λ_T the tangent eigenvalue and $\xi = 1$ if P_i is considered a convex feature point and $\xi = -1$ if P_i is considered concave (for definition of convex / concave feature point, please refer to the corresponding paragraphs in sections 2.2 and 3.2.1). The feature variation yields information on the position of the points in $ROS_h(P_i)$ in respect to $\bar{P}_i$. If the shape in the local area around P_i is relatively flat, the neighboring points lie close to the tangent direction at P_i (see Fig. 12). The value of $\sigma(P_i)$ is within the closed interval $[-1, 1]$ with the absolute value approaching 1 if P_i is the tip of a sharp curvature and drawing near 0 if P_i is is surrounded by a flat local neighborhood.

- Feature Side Variation:

  Side feature variation is defined as

  $$\tau(P_i) = \frac{\sigma(ROL(P_i)) + \sigma(ROR(P_i))}{2} \tag{6}$$

  where $ROL(P_i)$ and $ROR(P_i)$ are defined as mentioned above. For each $ROL(P_i)$ and $ROR(P_i)$ a covariance is calculated (similar to the covariance of $ROS_h(P_i)$ in equation (4) which yields eigenvalues λ_N^L and λ_T^L for $ROL(P_i)$ and accordingly λ_N^R and λ_T^R for $ROR(P_i)$. Using these eigenvalues $\sigma(ROL(P_i))$ and $\sigma(ROR(P_i))$ are defined as $\sigma(ROL(P_i)) = \frac{\lambda_N^L}{\lambda_N^L + \lambda_T^L}$ and $\sigma(ROR(P_i)) = \frac{\lambda_N^R}{\lambda_N^R + \lambda_T^R}$. The feature side variation $\tau(P_i)$ is used to gain information about the appearance of the local neighborhood on the left and right side of P_i. Similar to the feature variation $\sigma(ROL(P_i))$ and $\sigma(ROR(P_i))$ adopt values in $[0, 1]$ (the absence of parameter ξ prevents negative negative values). If $\tau(P_i)$ is close to 0 the side neighbors are flat, while high values of $\tau(P_i)$ represent bended parts in ROL or ROR.

- Feature Size

  The size of a feature is measured by

  $$\rho(P_i) = \frac{\rho^L(P_i) + \rho^R(P_i)}{2} \tag{7}$$

  where $\rho^L(P_i)$ and $\rho^R(P_i)$ denote the length of $ROL(P_i)$ and $ROR(P_i)$ in relation to the total length of the shape. The value of $\rho(P_i)$ is an indicator for its importance in the whole shape: If $\rho(P_i)$ yields a small value for P_i, the local neighborhood of P_i is small in respect to the entire shape and most likely not to leave a dominant impression on an observer (although this might not be completely true if $\sigma(P_i)$ and $\tau(P_i)$ both yield high values). High values of $\rho(P_i)$ indicate that the local neighborhood of P_i takes up a significant part of the total shape and thus will likely leave a strong imprint on an observer.

It is noteworthy to mention that all three criteria, describing geometric properties of the local neighborhood of a feature point, are unaffected by rescaling, translation or rotations of the shape. The choice of feature points, the size h of $ROS_h(P_i)$ and how to draw sample points between adjacent feature points on the other hand can strongly affect feature variation, feature side variation and feature size.

5.3.3 Similarity Costs

Section 5.3.2 established three criteria to distinguish feature points belonging to the same shape from one another. Now these properties can be utilized to find a suitable correspondence between feature points in *source* and *target*. As mentioned in the introduction to the *Vertex Correspondence Problem* (see 4.1) it seems to be a good idea if areas which are similar in *source* and *target* are tried to be matched to each other. Regions in *source* and *target* which are alike in shape should have similar values for their feature variation, feature side variation and feature size. So it seems feasible to compare the geometric properties of feature points in *source* and *target* and assign costs to a pair of feature points indicating if they and their associated regions are similar or not. Let *source* be described by $\mathcal{S} = \{S_i \mid i = 0, 1, \cdots, m\}$ and *target* be $\mathcal{T} = \{T_j \mid j = 0, 1, \cdots, n\}$. If *source* and *target* are closed $S_0 = S_m$ and $T_0 = T_n$ respectively holds. The *Similarity Costs* for a pair of feature points S_i and T_j are defined as

$$SimCost(S_i, T_j) = \Psi(S_i, T_j) \sum_{q=\sigma,\tau,\rho} \omega_q \Delta_q(S_i, T_j) \tag{8}$$

where $\Psi(S_i, T_j)$ acts as a weight to determine the importance of this correspondence and is defined as $\Psi(S_i, T_j) = \max\{\rho(S_i), \rho(T_j)\}$ with ρ being defined as in equation (7). This is done to focus on matching large parts of *source* and *target* with similar parts, since large parts are generally watched more consciously during a morphing sequence by a human observer. The term Δ_q measures the costs assigned to the pair (S_i, T_j) for each of the three geometric quantities and is defined as

$$\Delta_\sigma(S_i, T_j) = |\sigma(S_i) - \sigma(T_j)|,$$

$$\Delta_\tau(S_i, T_j) = \frac{1}{2}(|\sigma(ROL(S_i)) - \sigma(ROL(T_j))| + |\sigma(ROR(S_i)) - \sigma(ROR(T_j))|),$$

$$\Delta_\rho(S_i, T_j) = \frac{1}{2}(|\rho^L(S_i) - \rho^L(T_j)| + |\rho^R(S_i) - \rho^R(T_j)|)$$

where σ, τ and ρ are defined as in equations (5), (6) and (7) respectively. Lastly ω_q describes weights for every Δ_q which have to fulfill $\omega_q \geq 0$ and $1 = \sum_{q=\sigma,\tau,\rho} \omega_q$. These weights allow for varying importance of the three criteria to measure similarity.

5.3.4 Discard Costs

In the process to find an good solution to the *Vertex Correspondence Problem* it can sometimes be helpful to omit feature points if there is no suitable match to be found.

Intuitively it becomes clear that a feature point might be easily discarded, if its local neighborhood is small and relatively flat, since it will probably not be observed as closely as large and bended local neighborhoods. In order to decide which feature points might be discarded, the three criteria of section 5.3.2 can be utilized as well. Since the absolute values of $\sigma(P_i)$, $\tau(P_i)$ and $\rho(P_i)$ are generally larger if the local neighborhood of P_i is highly noticeable the discard costs of a feature point S_i of *source* $\mathcal{S}$ are defined as follows:

$$DisCost(S_i) = \rho(S_i) \sum_{q=\sigma,\tau,\rho} \omega_q |q(S_i)|, \tag{9}$$

where $\sigma(S_i)$, $\tau(S_i)$ and $\rho(S_i)$ are defined as usual (see equations (5),(6) and (7)) and the weights ω_q are the same as in equation (8). Similar to equation (8) the size of the local neighborhood is used as a coefficient to evaluate the importance of the neighborhood in respect to the total shape. Naturally the discard costs for a feature point T_j of *target* $\mathcal{T}$ is analogous.

5.4 Minimization of the Correspondence Problem

The similarity cost function (equation (8)) can be utilized to measure similarity not only between two different feature points S_i and T_j, but to assign costs to a correspondence between two complete shapes $\mathcal{S}$ and $\mathcal{T}$. A correspondence in this case means a mapping between feature points of $\mathcal{S}$ with feature points of $\mathcal{T}$. A similarity cost function between $\mathcal{S}$ and $\mathcal{T}$ can be established if we consider a mapping J as $J : \{S_i\} \rightarrow \{T_j\}$:

$$SimCosts(\mathcal{S}, \mathcal{T}, J) = \sum_{i=0}^{m-1} SimCosts(S_i, T_{J(i)}),$$

still bearing in mind that $\mathcal{S}$ has m different *Feature Points*. Considering the behavior of the similarity function (equation (8)) an optimal solution for a correspondence can be considered as a mapping J which minimizes $SimCosts(\mathcal{S}, \mathcal{T}, J)$. Thus the following problem has to be solved:

$$\min_J \{SimCosts(\mathcal{S}, \mathcal{T}, J\}$$

If $\mathcal{S}$ contains m feature points and $\mathcal{T}$ contains n feature points there would be n^m possible mappings J between $\mathcal{S}$ and $\mathcal{T}$. An algorithm considering all possible mappings would therefore become very inefficient. Luckily a large number of these mappings may be disregarded, because most of these mappings do not take into account that it would be suitable for a morph, if $J : S_i \rightarrow T_j$ holds, S_{i+1} should be mapped to a feature point in $\mathcal{T}$ which is close to T_j. In this case “close” means that S_{i+1} should correspond to some *Feature Point* $T_k \in \{T_k \mid k = i-l, i-l+1, \ldots, i-1, i+1, \ldots, i+l\}$ for some integer l. How this restricted minimization problem can be solved efficiently using *Dynamic Programming* techniques will be shown in 5.4.2. In 5.4.1 a short introduction into *Dynamic Programming* will be presented. If you are already familiar with the concept of Dynamic Programming you might want to skip 5.4.1 and move straight to 5.4.2.

5.4.1 Dynamic Programming

The information presented here on *dynamic programming* is based on an article by Wagner (21). Since it is not necessary to cover more than the basic principles of dynamic programming, in order to understand the methods employed to solve the *Vertex Correspondence Problem* the material presented in this section will not be as extensive as Wagners remarks on this topic. If you are interested in more information on dynamic programming reading (21) as a more complete introduction is strongly recommended.

Basic Principles of Dynamic Programming It is a common technique in computer science to divide a large problem into smaller subproblems which can be solved easier, if it is possible to reconstruct a solution to the original problem by combining the solutions of the subproblems. However it is often not possible to split the original problem into a *small* number of easily solvable subproblems, but solutions to a *large* number of subproblems are often required. In some cases the subproblems themselves have to be divided into smaller subproblems again, so that the number of problems that need to be solved may increase exponential. Oftentimes problems with this peculiar nature are solved by recursive methods, because they present very natural and easily implementable solutions. In most cases these recursive solutions become very inefficient, because of many identical calls during the solution of the original problem. If a problem shows this behavior it is commonly referred to as an overlapping subproblem property. Dynamic programming techniques can be used to make solving such problems more efficiently. In many cases dynamic programming also benefits if the problems have an *optimal substructure*. Optimal substructure means that optimal solutions for subproblems can be used to construct an optimal solution for the original problem. An example for optimal substructure could be finding the shortest path from a vertex to a goal in an acyclic graph: In a first step the distances to all vertices adjacent to the goal will stored. Each of those distances can be considered as the optimal path connecting that vertex with the goal. If optimal paths from the start vertex to the vertices adjacent to the goal can be found, it is possible to construct from both optimal solutions of the subproblems (e.g. the optimal paths from start vertex to the adjacent vertices and the optimal paths from the adjacent vertices to the goal) several solutions to solve the whole problem. Of these solutions the one is optimal which has the shortest path. If a problem has optimal substructure a three-step schematic can be applied to solve problems belonging to this class:

1. Divide the problem into smaller subproblems
2. Solve these subproblems optimally either
    - if the problem is simple calculate optimal solution, or
    - using this three-step schematic
3. Use these optimal solutions to construct an optimal solution for the original problem

As you will notice, this schematic is recursively itself, which emphasizes that problems featuring the attributes mentioned above can be solved using recursive algorithms quite often.

Algorithms which extend the original function in some way during the computation time are clustered in the term *dynamic programming*. This can be done by ordering the problems and solving the problem from simpler to more complex subproblems, with the intention to solve each subproblem before it is needed in the computation of another subproblem. On the other hand it is often a nontrivial problem itself to find a suitable ordering of the different subproblems. In many cases a dynamic programming technique can be applied which avoids scheduling the subproblems. This technique is often referred to as *memoization* (not memorization, although that would also be an appropriate name) or *result catching*. The idea of memoization is to store all evaluations of subproblems which have been computed once and if the subproblem needs to be evaluated for a second and subsequent time just to return the stored value. Thus evaluation of once computed subproblems needs constant computational time, for returning the already computed value and the overall running time can be reduced significantly in many cases.

These sums up the basic principles of dynamic programming. In the following paragraph a simple example will be presented to illustrate the techniques of dynamic programming and memoization. Readers who feel already comfortable with the depicted concepts might want to skip the following paragraph.

Dynamic Programming example: Fibonacci numbers A common example to illustrate the principles of memoization are the *Fibonacci numbers*. Similar to the *Tower of Hanoi* problem which is often used to illustrate the principles of recursion, the Fibonacci numbers are concise enough for an introductory example and still provide all the elements to visualize the general ideas.

The Fibonacci numbers are a sequence of numbers constructed in the following fashion: $F_i = F_{i-1} + F_{i-2}$, most often with the initial assumptions $F_0 = 1$ and $F_1 = 1$. The computation of F_5, for example, would result in the following sequence of function calls:

$$\begin{aligned} F_5 &= F_4 + F_3 \\ &= (F_3 + F_2) + F_3 \\ &= ((F_2 + F_1) + F_2) + F_3 \\ &= \cdots \\ &= \underbrace{(((F_0 + F_1) + F_1) + (F_0 + F_1))}_{F_4} + \underbrace{((F_0 + F_1) + F_1)}_{F_3} \end{aligned} \tag{10}$$

as you can see, many subproblems have to be solve multiple times. For instance F_3 first has to be solved to get a solution for F_4 and has to be solved later again in order to deliver a result for F_5 in combination with F_4. The schematic order of function calls is depicted in Fig. 14, in more detail.

Memoization avoids these multiple computation of subproblems. In the small example if F_3 is solved the first time, memoization stores the solution $F_3 = 3$ and if F_3 is needed

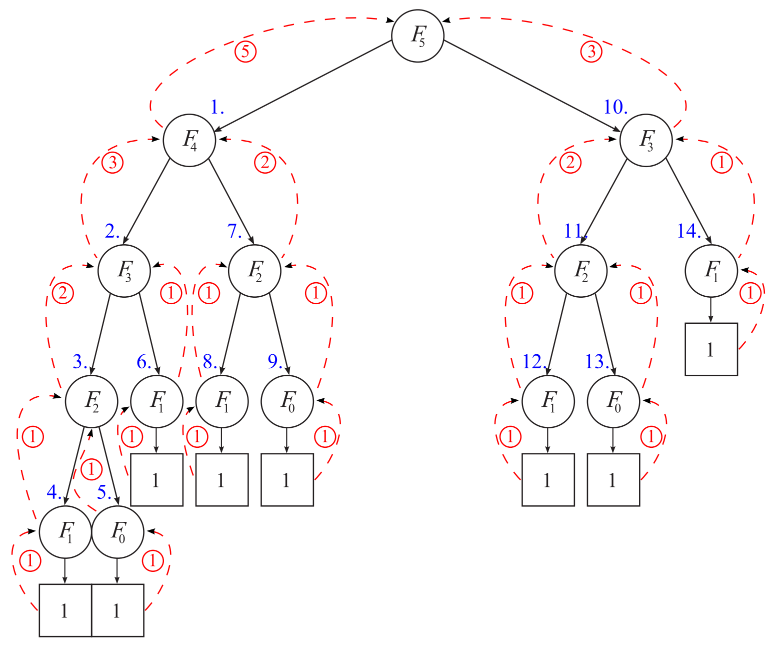


Figure 14: Schematic of recursive call of Fibonacci function
Function calls are depicted as circles, reaching the termination of a recursive call is indicated with a square. The blue numbers display the order of the function calls, while the red numbers display the return values.

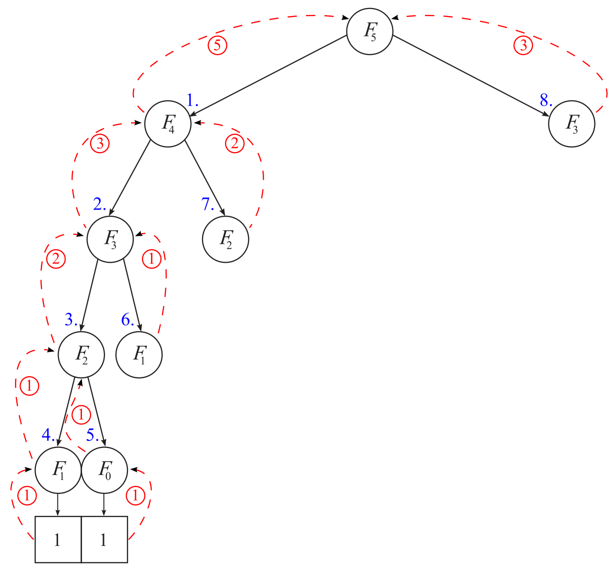


Figure 15: Schematic of Fibonacci function with memoization
Same colors and symbols as in Fig. 14, please observe the reduced number of function calls, if memoization is used. In larger examples even more function calls would be avoided.

for a second and subsequent times returns the stored value instead of calling F_2 and F_1 to solve F_3. For a visual comparison of the function call of F_5 with memoization please compare the schematic of Fig. 15 with Fig. 14

For the calculation of larger Fibonacci numbers or problems where subproblems require more complex computations, this simple technique enhances the performance considerably.

5.4.2 Solving the Minimization Problem

If *source* $\mathcal{S}$ includes m feature points and *target* $\mathcal{T}$ n respectively all possible correspondences between vertices can be depicted in an $m \times n$ rectangular graph where rows represent feature points S_i of $\mathcal{S}$ and columns represent feature points T_j of $\mathcal{T}$. A node in the graph at the intersection of row i and column j will be denoted as $node(i, j)$. A $node(i, j)$ indicates a correspondence between S_i and T_j and with a sequence of nodes starting at $node(0, 0)$ and ending at $node(m, n)$ a complete correspondence between $\mathcal{S}$ and $\mathcal{T}$ can be described. In the following such a sequence will be called a `path` Γ. Please note that the nodes a path Γ do not necessarily have to be adjacent and if both shapes are closed, conditions $S_0 = S_m$ and $T_0 = T_n$ hold. An exemplary dynamic programming graph containing a path Γ is depicted in Fig. 16.

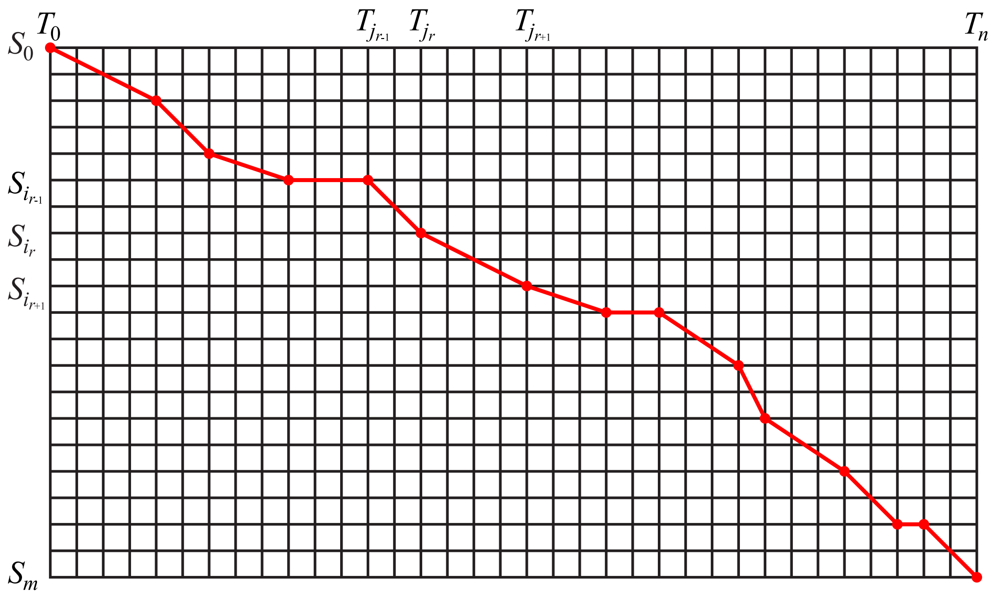


Figure 16: Example for a DP graph containing a complete path Γ

If the path contains $R+1$ nodes it will be noted as $\Gamma = ((i_0, j_0), (i_1, j_1), \cdots, (i_R, j_R))$, where to satisfy the requirements, mentioned above (i_0, j_0) refers to $node(0,0)$ and (i_R, j_R) refers to $node(m,n)$. A node $node(i_{r-1}, j_{r-1}), 1 \leq r \leq R$ will be referred to as the parent of $node(i_r, j_r)$. Sequences of consecutive feature points of $\mathcal{S}$ will be denoted as $\mathcal{S}(i \mid i+k)$, meaning $S_i, S_{i+1}, \cdots, S_{i+k}$. The same notation will be used for $\mathcal{T}$.

The costs for a complete path Γ, containing $R+1$ nodes of $\mathcal{S}$ and $\mathcal{T}$ are defined as:

$$Costs(\mathcal{S}, \mathcal{T}, \Gamma) = \sum_{r=1}^{R} \delta(\mathcal{S}(i_{r-1} \mid i_r), \mathcal{T}(j_{r-1} \mid j_r)), \tag{11}$$

where the term $\delta(\mathcal{S}(i_{r-1} \mid i_r), \mathcal{T}(j_{r-1} \mid j_r))$ assigns costs depending on the similarity of $\mathcal{S}(i_{r-1} \mid i_r)$ and $\mathcal{T}(j_{r-1} \mid j_r)$:

$$\begin{aligned} \delta(\mathcal{S}(i_{r-1} \mid i_r), \mathcal{T}(j_{r-1} \mid j_r)) &= DisCosts(\mathcal{S}(i_{r-1} \mid i_r)) \\ &+ DisCosts(\mathcal{T}(j_{r-1} \mid j_r)) \\ &+ \lambda \cdot SimCosts(S_{i_r}, T_{j_r}), \end{aligned} \tag{12}$$

where

$$DisCosts(\mathcal{S}(i_{r-1} \mid i_r)) = \sum_{k=i_{r-1}+1}^{i_r-1} DisCosts(S_k)$$

As you will notice, $DisCosts(\mathcal{S}(i_{r-1} \mid i_r))$ accumulates the costs for discarding all feature points between $S_{i_{r-1}}$ and S_{i_r}. Depending on the actual values of indices i_{r-1} and i_r $DisCosts(\mathcal{S}(i_{r-1} \mid i_r))$ might return 0 costs, for example if $i_{r-1} + 1 = i_r$ holds. $DisCosts(\mathcal{T}(j_{r-1} \mid j_r))$ are defined analogous. The coefficient λ is a weight which can be used to influence the importance of finding good matches relative to discarding feature points. If λ is set to a high value the influence of the discard costs on the whole costs are reduced, discarding feature points is encouraged. Very low values for λ increase the influence of discard costs in equation (12) thus preventing the discard of many consecutive points. The choice of λ should depend on the kind of shapes that will be morphed: If the shapes contain many feature points a large value for λ is recommended. Shapes with few feature points should be morphed with $\lambda \leq 1$ so that the majority of the existing feature points will be matched. The choice of $\lambda = 1$ seems to be a "neutral" value, which works in many cases quite good. Equation (11) yields the costs for a complete path, by adding up the costs for all nodes in Γ (e.g. all similarity costs $SimCost(S_k, T_l), \forall\, node(k,l) \in \Gamma$) and the costs for discarding all feature points which are not included in Γ.

Discarding points can be beneficial for the total costs of a path if there is no suitable correspondence for a feature point. If the other feature points correspond well with each other the discard costs of one feature point might be less than the additional similarity costs that may arise if all other feature points change their correspondence. A scenario where discarding a feature point is beneficial is depicted in Fig. 17

This way equation (11) allows to determine the costs associated with a certain path and thus determine which path to prefer (i.e. prefer the path with less assigned costs). For the construction of such a path dynamic programming techniques are exploited. As a

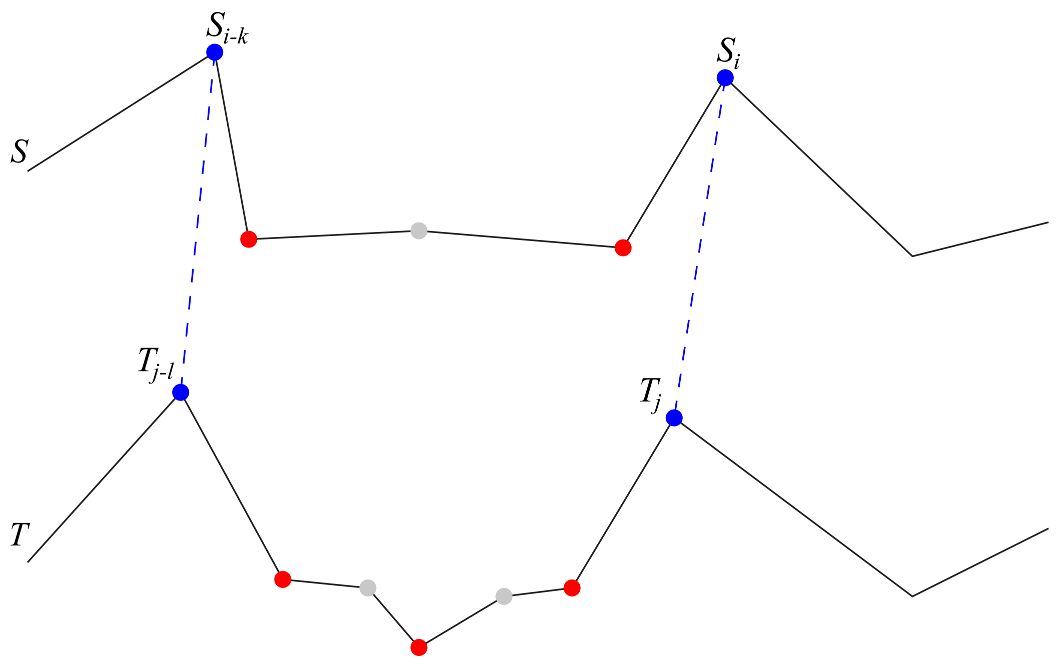


Figure 17: Skips during the creation of a complete path
Corresponding feature points are colored blue, discarded feature points are colored red, normal vertices are colored gray. If the correspondence between S_i and T_j is set, the best previous correspondence would be the one between S_{i-1} and T_{j-1}. The discarded feature point in polygon $\mathcal{T}$ has no suitable match in $\mathcal{S}$, since there is no concave point. So discarding will be beneficial instead of enforcing a correspondence with S_{i-1}.

basis a correspondence between the first feature points of $\mathcal{S}$ and $\mathcal{T}$ is stored by calculating $SimCost(S_0, T_0)$ and storing that value in $node(0, 0)$. With this initial correspondence it is possible to determine the optimal predecessor for $node(i, j)$ using the following equation:

$$node(i, j) = \min_{k,l}\{node(i - k, j - l) + \delta(\mathcal{S}(i - k \mid i), \mathcal{T}(j - l \mid j))\}, \tag{13}$$

with $k, l \geq 0$ and not $k = l = 0$. Equation (13) finds the best predecessor for $node(i, j)$ comparing the whole costs for the incomplete path starting at $node(0, 0)$ and ending at $node(i, j)$. The costs are the sum of the costs leading to all possible predecessors of $node(i, j)$ and the transition costs from that predecessor to $node(i, j)$ using equation (12). If the values of k and l have been calculated, they are stored in $node(i, j)$ as well as the cumulative costs for the incomplete path ending at $node(i, j)$. This way it is possible to backtrack the path from a node to its predecessor. Upper boundaries k_{max} l_{max} for parameters k and l are used to configure how many feature points are allowed to be omitted while trying to find the best predecessor. For instance if k_{max} is set to 1, then the algorithm is not allowed to omit any feature points of $\mathcal{S}$, if k_{max} is set to 2 a maximum of 1 feature point may be discarded while searching for the best predecessor. Equation (13) is a recursive formula, where for $node(i,j)$ all allowable nodes (determined by k_{max} and l_{max}) have to be solved as subproblems, which in turn will have several subproblems as well, to find their optimal predecessors. To enhance the algorithm the technique of *memoization* briefly sketched in section 5.4.1 is used: Instead of calculating the same subproblems again and again, the path costs and the optimal predecessor are stored in each node. To construct a complete path the algorithm starts at $node(0,0)$. After this initial step the optimal predecessors for each node in the two dimensional field of all possible nodes are calculated line by line, ending with $node(m,n)$. If $node(m,n)$ is reached the optimal path leading to this node can be reconstructed by back tracking all predecessors up to $node(0,0)$. This path is the complete correspondence with the least costs. Finding the best possible predecessor for a $node(i,j)$ is depicted in Fig. 18.

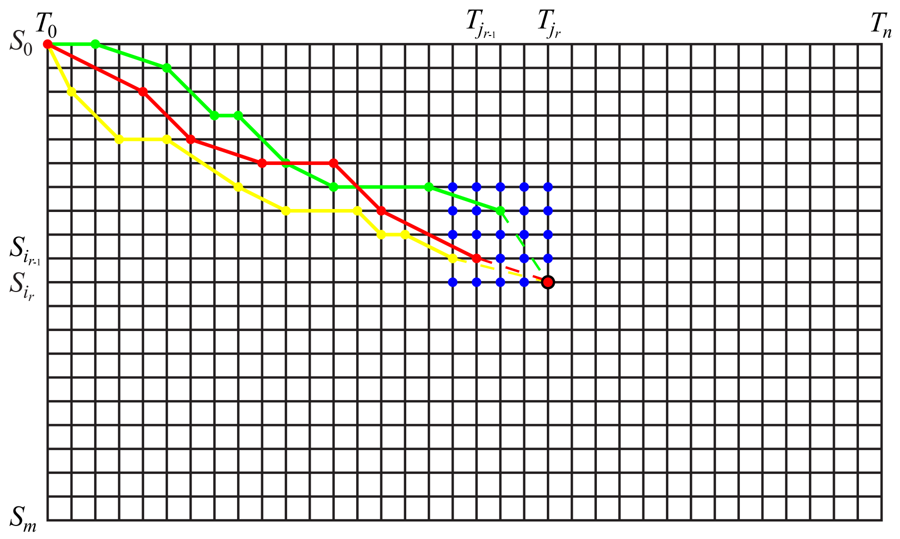


Figure 18: Choosing between allowed predecessors in the DP graph
The current $node(i,j)$ is depicted as the large red point with a black border. In this example parameters k and l for the allowed number of skips are set to 4. Blue points show allowed predecessors of $node(i,j)$ are marked by blue points. For three possible predecessors, colored red, green and yellow the incomplete path (see equation (13)) leading to those nodes is depicted as well. The red colored node is meant to be the optimal predecessor in the depicted scenario.

The above algorithm allows for one feature point of $\mathcal{S}$ to correspond with more than one feature point of $\mathcal{T}$ and vice versa. How to deal with such cases will be discussed later in this section. If no skips are allowed during the matching process (i.e. $l, k \leq 1$ in equation (13)) all in all the algorithm needs to check $3mn$ nodes: For each node (apart from $node(0,0), node(1,0)$ and $node(0,1)$) $node(i,j)$ has 3 possible predecessors which have to be checked ($node(i-1,j), node(i,j-1)$ and $node(i-1,j-1)$. Hence if no skips are allowed in path Γ the algorithm is in $O(mn)$, still assuming that $\mathcal{S}$ has m feature points and $\mathcal{T}$ has n. However if it is desired to omit feature points during the matching process, the runtime complexity increases to $O(k_{max}l_{max}mn)$. So the values for k_{max} and l_{max} should be considered carefully. In most cases it is reasonable to assume $k_{max}, l_{max} \ll m, n$ since matching single pairs of feature points with huge gaps between the pairs are often not desired. If equation (13) is simplified a bit to $k_{max} = l_{max} = C$ the runtime complexity becomes $O(C^2mn)$.

As you will have recognized the choice of the initial correspondence $node(0,0)$ to construct a complete path is somewhat random and does in many cases not guarantee the desired result of the correspondence with the least possible costs. For non-closed shapes this initial assumption is reasonable since the both ends of a shape should generally correspond with each other. In the case of closed shaped however the algorithm has to be repeated with all possible initial correspondences in order to guarantee that the complete correspondence with the least possible costs is actually found. Since there are mn possible initial correspondences the runtime complexity becomes $O(C^2m^2n^2)$.

Creating a 1:1 correspondence from a path The optimal path found through the algorithm depicted above still is not what is required to start the morphing process. In a complete correspondence is is possible to have one feature point correspond with multiple other feature points, but before any animation can start a 1:1 correspondence is highly favorable. In order to create a 1:1 correspondence all feature points with more than one correspondence must be examined. Of all multiple assigned correspondences to a feature point the one with the least similarity costs is kept, all other correspondences are to be ignored. After this step every feature point has at most one correspondence. Now the points which eventually lost a correspondence in the previous step need a new correspondence. This is done by either assigning another free feature point or by creating a new point that will correspond with the feature point. The first case is applicable, if for a *Feature Point* S_i there exists at least one unassigned feature point T_j between the correspondences $T_{j_{pred}}$ and $T_{j_{succ}}$ of the nearest feature points S_{i-a} and S_{i+b} of S_i which have correspondences. If no such free candidate T_j for a correspondence exists a new point will be inserted on the curve curve between $T_{j_{pred}}$ and $T_{j_{succ}}$. In the creation of this point it is advised to take the distance from S_i to S_{i-a} and S_{i+b} into account and create the new point featuring the same relative distances to $T_{j_{pred}}$ and $T_{j_{succ}}$.

One might wonder how to deal with discarded feature points. If they would just disappear between two frames in the morphing process it might possibly raise the attention of an observer. The costs for discarding a feature point can also be interpreted as the costs to match this feature point with another feature point where all criteria (see equations (5),(6) and (7)) yield 0. If discarding a feature point is interpreted this way, it allows to insert new points into a shape which as long as they do not affect the appearance of the shape and let them correspond with the discarded feature points. The construction of such points is similar to the case of feature points where no free candidate for a correspondence exists (see above). After all this is done, a 1:1 correspondence between all points of $\mathcal{S}$ and $\mathcal{T}$ is established and algorithms to deal with the *Vertex Path Problem* can be applied. Section 6 will show, how these algorithms were realized in the Java programming language.

6 Implementation in the Java programming language

There are two principal reasons why the implementation of the algorithm, described in section 5, was done using the Java programming language. First, programs written in the Java programming language are platform independent, which allows usage of the application presented in section 7 on every computer, on which a current version of the Java Runtime Environment is installed, without further modifications. Secondly the Java programming language enables running programs in a common web browser very comfortably via applets. This way the application can also be used via internet.

In the following a short overview of the main classes needed in the application (see section 7) will be given, partly with excerpts of the source code partly in the form of UML diagrams. Readers who are only interested in the usage of the application and not in the programming details might want to skip this section.

6.1 Overview of the class hierarchy

At this point the whole project currently consists of approximately 9300 lines of code, distributed over 40 classes and split into 7 packages. The packages are called `shapes`, `math`, `featureDetection`, `tools`, `morphing`, `controls` and `application`. In the following sections the general purpose of each package will be described and some of their important classes will be introduced.

6.1.1 Package `shapes`

Though the application deals with the morphing of polygons it is sensible to implement more than just a polygon class to represent a two dimensional polygon. The base class in package `shapes` is the abstract class `GraphicObject`, all other classes contained in the package are subclasses of `GraphicObject`. For a class diagram of `shapes` please refer to Fig. 19.

Two abstract methods in `GraphicObject` are essential for all other classes in package `shapes`. The `paint(Graphics)` method is needed to display a class derived from `GraphicObject` and will be called in the application every time a `GraphicObject` is manipulated and its display needs to be updated. Method `toSVG()` enables a subclass of `GraphicObject` to be exported to the SVG format (see (20)).

The direct subclass `Point` and its subclass `FeaturePoint` were implemented to model the characteristics of the approach described in section 5. Since the approach distinguishes between vertices and feature points it seemed sensible to do the same in the class hierarchy, especially since a feature point has several additional attributes like its *feature variation*, *feature size* etc. (see 5.3.2) which are not needed for a simple vertex in a polygon. For that reason the class `Polygon` may contain instances of `Points` and of `FeaturePoint` as its vertices. The `Point` class itself represents a point in a two dimensional plane with two coordinates (referred to as x and y). To ensure that every instance of a point can be displayed in the application, the values for x and y have a lower and upper boundary, which is set in the `Point` class. Important for the solution

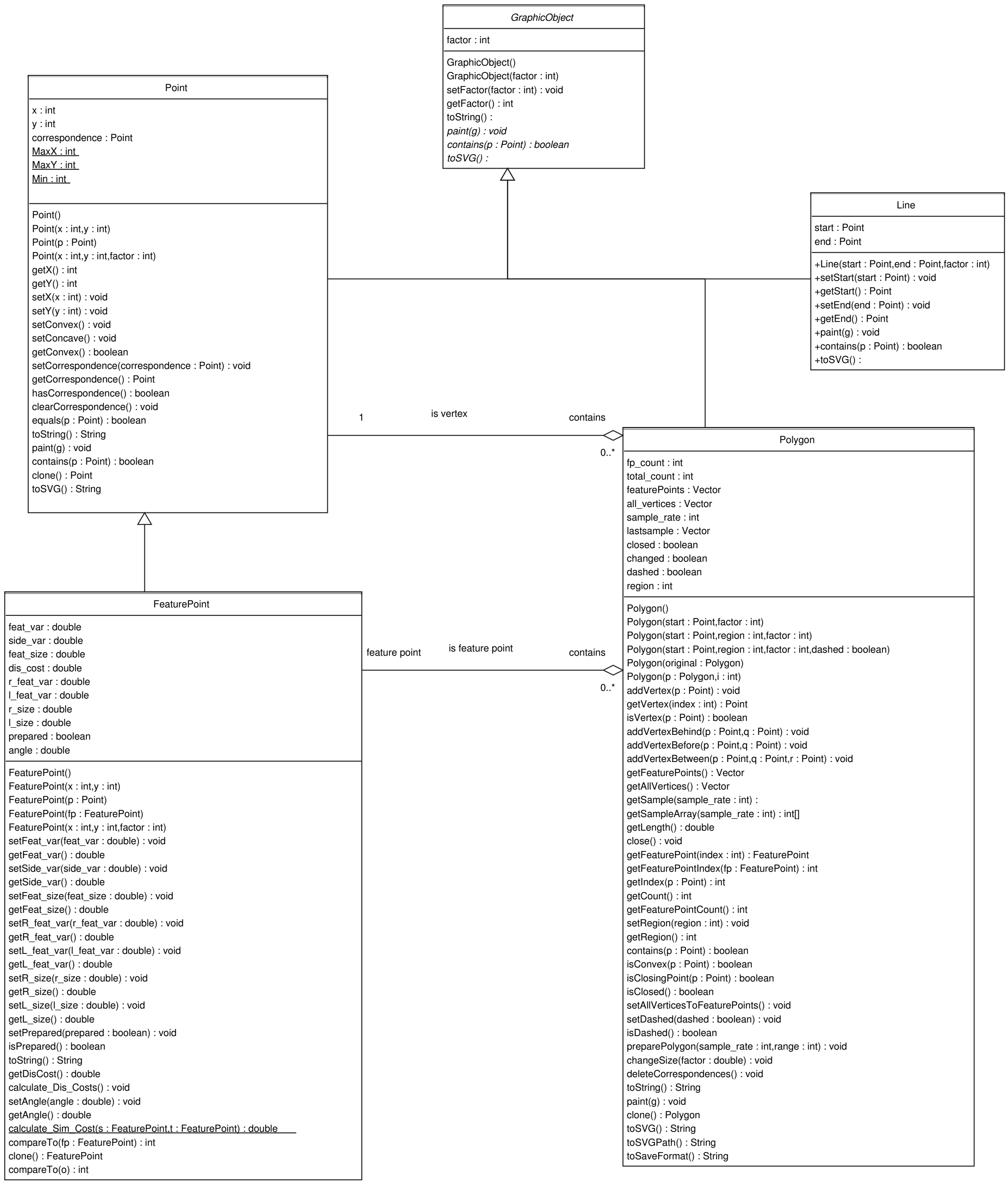

GraphicObject
factor : int
GraphicObject()
GraphicObject(factor : int)
setFactor(factor : int) : void
getFactor() : int
toString() :
paint(g) : void
contains(p : Point) : boolean
toSVG() :
Point
x : int
y : int
correspondence : Point
MaxX : int
MaxY : int
Min : int
Point()
Point(x : int,y : int)
Point(p : Point)
Point(x : int,y : int,factor : int)
getX() : int
getY() : int
setX(x : int) : void
setY(y : int) : void
setConvex() : void
setConcave() : void
getConvex() : boolean
setCorrespondence(correspondence : Point) : void
getCorrespondence() : Point
hasCorrespondence() : boolean
clearCorrespondence() : void
equals(p : Point) : boolean
toString() : String
paint(g) : void
contains(p : Point) : boolean
clone() : Point
toSVG() : String
Line
start : Point
end : Point
+Line(start : Point,end : Point,factor : int)
+setStart(start : Point) : void
+getStart() : Point
+setEnd(end : Point) : void
+getEnd() : Point
+paint(g) : void
+contains(p : Point) : boolean
+toSVG() :
1
is vertex
contains
0..*
Polygon
fp_count : int
total_count : int
featurePoints : Vector
all_vertices : Vector
sample_rate : int
lastsample : Vector
closed : boolean
changed : boolean
dashed : boolean
region : int
Polygon()
Polygon(start : Point,factor : int)
Polygon(start : Point,region : int,factor : int)
Polygon(start : Point,region : int,factor : int,dashed : boolean)
Polygon(original : Polygon)
Polygon(p : Polygon,i : int)
addVertex(p : Point) : void
getVertex(index : int) : Point
isVertex(p : Point) : boolean
addVertexBehind(p : Point,q : Point) : void
addVertexBefore(p : Point,q : Point) : void
addVertexBetween(p : Point,q : Point,r : Point) : void
getFeaturePoints() : Vector
getAllVertices() : Vector
getSample(sample_rate : int) :
getSampleArray(sample_rate : int) : int[]
getLength() : double
close() : void
getFeaturePoint(index : int) : FeaturePoint
getFeaturePointIndex(fp : FeaturePoint) : int
getIndex(p : Point) : int
getCount() : int
getFeaturePointCount() : int
setRegion(region : int) : void
getRegion() : int
contains(p : Point) : boolean
isConvex(p : Point) : boolean
isClosingPoint(p : Point) : boolean
isClosed() : boolean
setAllVerticesToFeaturePoints() : void
setDashed(dashed : boolean) : void
isDashed() : boolean
preparePolygon(sample_rate : int,range : int) : void
changeSize(factor : double) : void
deleteCorrespondences() : void
toString() : String
paint(g) : void
clone() : Polygon
toSVG() : String
toSVGPath() : String
toSaveFormat() : String
FeaturePoint
feat_var : double
side_var : double
feat_size : double
dis_cost : double
r_feat_var : double
l_feat_var : double
r_size : double
l_size : double
prepared : boolean
angle : double
FeaturePoint()
FeaturePoint(x : int,y : int)
FeaturePoint(p : Point)
FeaturePoint(fp : FeaturePoint)
FeaturePoint(x : int,y : int,factor : int)
setFeat_var(feat_var : double) : void
getFeat_var() : double
setSide_var(side_var : double) : void
getSide_var() : double
setFeat_size(feat_size : double) : void
getFeat_size() : double
setR_feat_var(r_feat_var : double) : void
getR_feat_var() : double
setL_feat_var(l_feat_var : double) : void
getL_feat_var() : double
setR_size(r_size : double) : void
getR_size() : double
setL_size(l_size : double) : void
getL_size() : double
setPrepared(prepared : boolean) : void
isPrepared() : boolean
toString() : String
getDisCost() : double
calculate_Dis_Costs() : void
setAngle(angle : double) : void
getAngle() : double
calculate_Sim_Cost(s : FeaturePoint,t : FeaturePoint) : double
compareTo(fp : FeaturePoint) : int
clone() : FeaturePoint
compareTo(o) : int
feature point
is feature point
contains
0..*

Figure 19: Class hierarchy of package `shape`

of the VCP is the data field `Point.correspondence`, where a corresponding point can be referenced.

6.1.2 Package `math`

The package `math` contains tools to calculate the more complicated mathematical properties of the algorithm, namely covariances, eigenvectors, eigenvalues and the bisector of the inner angle at a vertex. These tools are summarized in just two classes (see Fig. 20). First, class *covariance* covers the calculation of the center of a region of support and the covariance for a region of support (see 5.3.1, equation (3, 4)). In addition a method to calculate the dot product of two vectors (`dotProduct(double, double, double, double)`) and a method to calculate the bisector (`getBisector(Vector)`) are included. The method `covariance(Vector)` is optimized in terms of runtime and storage usage for the special case needed in the algorithm, namely symmetrical 2×2 matrices and will not deliver correct results for calculation of general covariance matrices. All methods try to avoid the usage of arrays or array like structures as much as possible to ensure fast calculations. Further information on speed optimization, when programming in the Java programming language can be found in (19).

In class `Eigenvalue` methods are collected to calculate eigenvectors and eigenvalues of covariance matrices. To calculate eigenvectors and eigenvalues of a covariance method `hqr2(double[][], double[][])` can be used which is an adaption to the Java programming language of a FORTRAN method contained in the EISPACK library (6). In the implemented variation the method can deal with symmetrical covariances with real entries. For unsymmetrical covariances or covariances containing complex entries the method will deliver wrong results. Based on this adaption and even more optimized for the restricted case, needed in the algorithm, the methods `hrq_tweaked(double[][])` and `eigenvalue(double, double, double)` calculate eigenvectors and eigenvalues respectively more efficiently.

Since all classes contained in package `math` just deliver results used in the calculation of the feature point properties (see 5.3.2) all methods in this package are in fact `static` methods. Splitting the methods in two different classes was not necessary, but integrates methods with similar properties and usage into a common class.

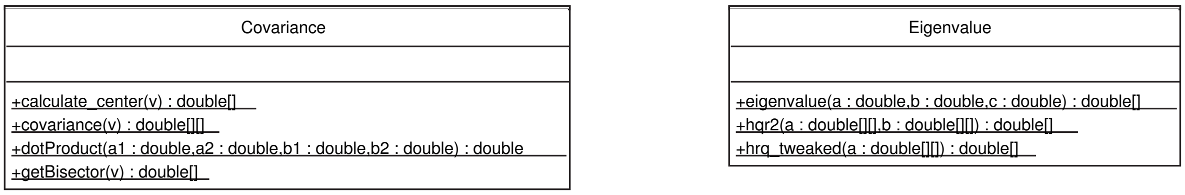

Covariance
+calculate_center(v) : double[]
+covariance(v) : double[][]
+dotProduct(a1 : double,a2 : double,b1 : double,b2 : double) : double
+getBisector(v) : double[]
Eigenvalue
+eigenvalue(a : double,b : double,c : double) : double[]
+hqr2(a : double[][],b : double[][]) : double[]
+hrq_tweaked(a : double[][]) : double[]

Figure 20: Class hierarchy of package `math`

6.1.3 Package `featurePointDetection`

In package `featurePointDetection` two classes are contained which are used to detect feature points in a polygon. A class diagram of package `featurePointDetection` is depicted in Fig. 21.

Class `FeaturePointDetector` is the main class in the package with class `QuickSort` being a utility class used by `FeaturePointDetector`. In class `FeaturePointDetector` mostly two methods are of interest: First, `featureDetection(Polygon)` which detects feature points in a polygon employing the methods described in 3.2. The method `filterMostProminent(Polygon)` can be used in experiments. Given an integer value l as an upper limit the method sorts all feature points according to their sharpness (see 3.2.1) and accepts only the l feature points with the greatest sharpness.

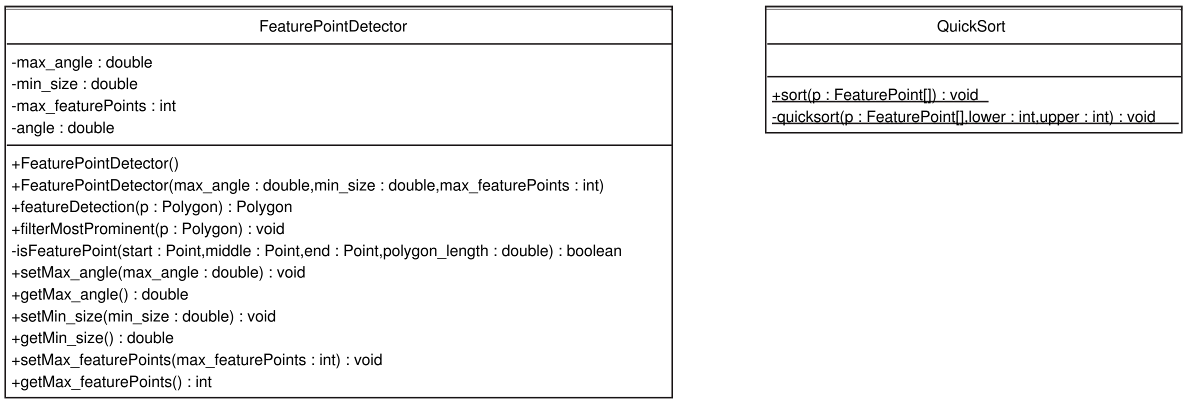

FeaturePointDetector
-max_angle : double
-min_size : double
-max_featurePoints : int
-angle : double
+FeaturePointDetector()
+FeaturePointDetector(max_angle : double,min_size : double,max_featurePoints : int)
+featureDetection(p : Polygon) : Polygon
+filterMostProminent(p : Polygon) : void
-isFeaturePoint(start : Point,middle : Point,end : Point,polygon_length : double) : boolean
+setMax_angle(max_angle : double) : void
+getMax_angle() : double
+setMin_size(min_size : double) : void
+getMin_size() : double
+setMax_featurePoints(max_featurePoints : int) : void
+getMax_featurePoints() : int
QuickSort
+sort(p : FeaturePoint[]) : void
-quicksort(p : FeaturePoint[],lower : int,upper : int) : void

Figure 21: Class hierarchy of package `featurePointDetection`

6.1.4 Package `tools`

Package `tools` contains all classes that are used as tools by other classes, but have no significant similarities that would justify another package. For a swift overview of all classes, please observe Fig. 22. Noteworthy is class `Path` which is used to store the path through a dynamic programming graph (see 5.4.2) and by that a complete correspondence between source and target. Instances of class `Node` represent path nodes of the dynamic programming graph (see 5.4.2). References of the two corresponding feature points are stored along with the optimal predecessor node and the costs of the incomplete path ending at the current node (see 5.4.2, equation (13)). In `Bresenham` the well-known line algorithm of Jack Bresenham (3) is adapted in the Java programming language. The class `Constants` contains no method, but is used to store all initial values for all parameters of the algorithm in one place.

6.1.5 Package `controls`

In package `controls` most classes implement the interface `ActionListener` defined in package `java.awt.event`. These classes are mainly used to handle user interaction

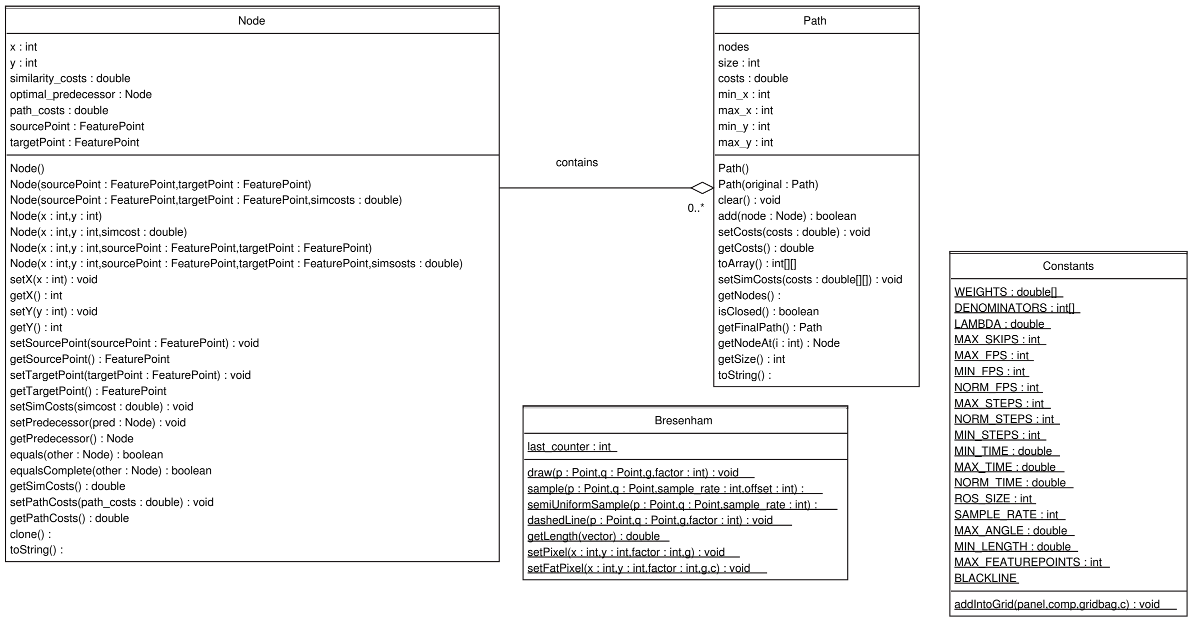

Node
x : int
y : int
similarity_costs : double
optimal_predecessor : Node
path_costs : double
sourcePoint : FeaturePoint
targetPoint : FeaturePoint
Node()
Node(sourcePoint : FeaturePoint,targetPoint : FeaturePoint)
Node(sourcePoint : FeaturePoint,targetPoint : FeaturePoint,simcosts : double)
Node(x : int,y : int)
Node(x : int,y : int,simcost : double)
Node(x : int,y : int,sourcePoint : FeaturePoint,targetPoint : FeaturePoint)
Node(x : int,y : int,sourcePoint : FeaturePoint,targetPoint : FeaturePoint,simsosts : double)
setX(x : int) : void
getX() : int
setY(y : int) : void
getY() : int
setSourcePoint(sourcePoint : FeaturePoint) : void
getSourcePoint() : FeaturePoint
setTargetPoint(targetPoint : FeaturePoint) : void
getTargetPoint() : FeaturePoint
setSimCosts(simcost : double) : void
setPredecessor(pred : Node) : void
getPredecessor() : Node
equals(other : Node) : boolean
equalsComplete(other : Node) : boolean
getSimCosts() : double
setPathCosts(path_costs : double) : void
getPathCosts() : double
clone() :
toString() :
contains
0..*
Path
nodes
size : int
costs : double
min_x : int
max_x : int
min_y : int
max_y : int
Path()
Path(original : Path)
clear() : void
add(node : Node) : boolean
setCosts(costs : double) : void
getCosts() : double
toArray() : int[][]
setSimCosts(costs : double[][]) : void
getNodes() :
isClosed() : boolean
getFinalPath() : Path
getNodeAt(i : int) : Node
getSize() : int
toString() :
Bresenham
last_counter : int
draw(p : Point,q : Point,g,factor : int) : void
sample(p : Point,q : Point,sample_rate : int,offset : int) :
semiUniformSample(p : Point,q : Point,sample_rate : int) :
dashedLine(p : Point,q : Point,g,factor : int) : void
getLength(vector) : double
setPixel(x : int,y : int,factor : int,g) : void
setFatPixel(x : int,y : int,factor : int,g,c) : void
Constants
WEIGHTS : double[]
DENOMINATORS : int[]
LAMBDA : double
MAX_SKIPS : int
MAX_FPS : int
MIN_FPS : int
NORM_FPS : int
MAX_STEPS : int
NORM_STEPS : int
MIN_STEPS : int
MIN_TIME : double
MAX_TIME : double
NORM_TIME : double
ROS_SIZE : int
SAMPLE_RATE : int
MAX_ANGLE : double
MIN_LENGTH : double
MAX_FEATUREPOINTS : int
BLACKLINE
addIntoGrid(panel,comp,gridbag,c) : void

Figure 22: Class hierarchy of package `tools`

with the application and update the other classes. Package `controls` also contains the interface `Resetable` which forces every class that implements the interface to include a method `reset()` that will be used to put a class back to the state it had right after calling its constructor. A class diagram of `controls` is depicted in Fig. 23.

6.1.6 Package `morph`

Package `morph` accumulates the classes which are mainly concerned with the creation of a morphing sequence. Class `MorphCalculator` contains two main methods. The first one is able to construct a path through the dynamic programming graph (see section 5.4.2). The second one is able, given such a path, to create two polygons which have the same outer appearance as source and target, but may contain additional vertices along the edges. These polygons have a 1:1 correspondence and act as first and last frame of the morphing sequence. In class `Animator` all in-between frames are computed given the first and the last frame of the morphing sequence. The animation paths used here are straight lines. The third and last class contained in this package is responsible for displaying the morphing sequence in the GUI and is a direct subclass of `java.lang.Thread`. Source code excerpts of the methods employed to create path through the dynamic programming graph can be found later on in 6.2.3.

6.1.7 Package `application`

In this package finally all classes are contained which are used to construct the GUI. Single elements of the GUI like the drawing panels and their corresponding sliders or the menu to set the parameters for the feature detection are capsuled in an own class. All

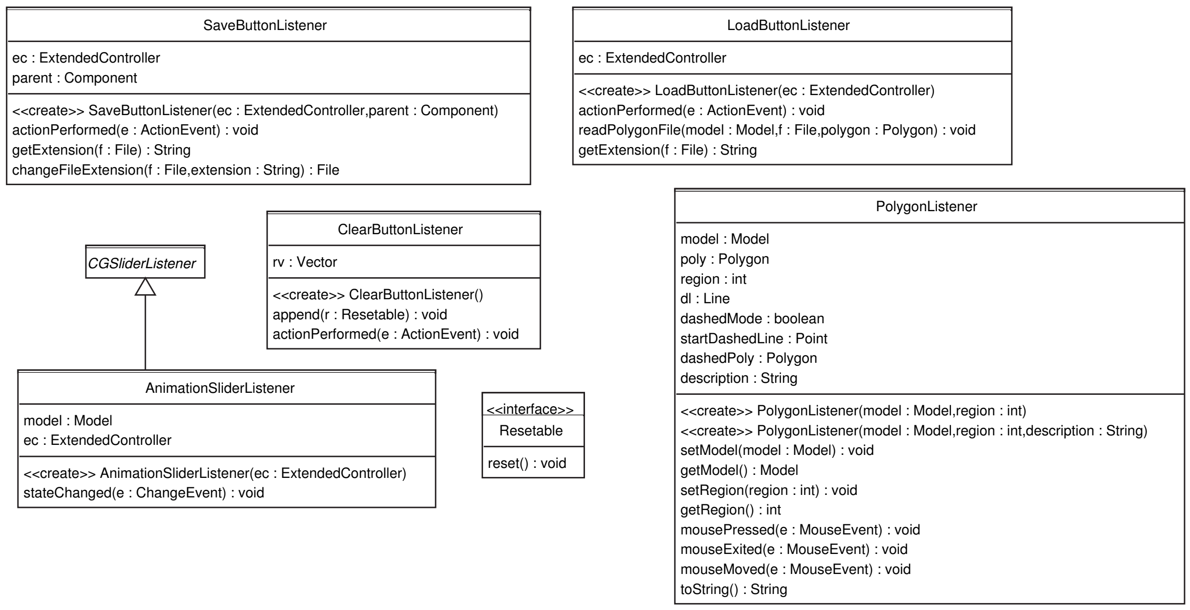

SaveButtonListener
ec : ExtendedController
parent : Component
<<create>> SaveButtonListener(ec : ExtendedController,parent : Component)
actionPerformed(e : ActionEvent) : void
getExtension(f : File) : String
changeFileExtension(f : File,extension : String) : File
LoadButtonListener
ec : ExtendedController
<<create>> LoadButtonListener(ec : ExtendedController)
actionPerformed(e : ActionEvent) : void
readPolygonFile(model : Model,f : File,polygon : Polygon) : void
getExtension(f : File) : String
CGSliderListener
ClearButtonListener
rv : Vector
<<create>> ClearButtonListener()
append(r : Resetable) : void
actionPerformed(e : ActionEvent) : void
PolygonListener
model : Model
poly : Polygon
region : int
dl : Line
dashedMode : boolean
startDashedLine : Point
dashedPoly : Polygon
description : String
<<create>> PolygonListener(model : Model,region : int)
<<create>> PolygonListener(model : Model,region : int,description : String)
setModel(model : Model) : void
getModel() : Model
setRegion(region : int) : void
getRegion() : int
mousePressed(e : MouseEvent) : void
mouseExited(e : MouseEvent) : void
mouseMoved(e : MouseEvent) : void
toString() : String
AnimationSliderListener
model : Model
ec : ExtendedController
<<create>> AnimationSliderListener(ec : ExtendedController)
stateChanged(e : ChangeEvent) : void
<<interface>>
Resetable
reset() : void

Figure 23: Class hierarchy of package `controls`

these classes are combined in class `ExtendedController` to create the GUI and enable all user interaction. Finally class `Start` contains just a method which creates an instance of `ExtendedController` to start the application.

6.2 Implementation of the algorithm in detail

In this section the implementation of the algorithm described in theory in section 5 will be discussed in more detail. Section 6.2.1 will show how the algorithm of Chetverikov and Szabó (5) was modified to process two dimensional polygons efficiently. In 6.2.2 some details on the calculation of the three properties for a feature point (Feature Variation, Feature Side Variation and Feature Size (see 5.3.2)) will be displayed. Details on the implementation of the dynamic programming path algorithm will be presented in 6.2.3. In all these sections only small excerpts of the source code will be shown, which seemed suitable to illustrate how to realize the basic ideas of the algorithm of Liu et al. If you are interested in more parts of the source code, please feel free to ask the author.

6.2.1 Detecting feature points

The feature point detection, discussed in section 3.2 was able to detect points of high curvature in general two dimensional shapes. As already proposed in the later parts of that section the algorithm can be simplified in the case of two dimensional polygons. In a polygon only vertices qualify as candidates for a feature point, therefore a sampling of the polygon is not needed for the detection of feature points. Also the construction of different triangles (see 3.2.1) is gratuitous, the edges connecting a vertex with its neighbors already fulfill this purpose. That leaves the computation of the inner angle for convex or the outer angle for concave vertices and as a second criteria the relative length of both edges at the vertex to the total length of the polygon. Every instance of class `FeaturePointDetector` has two data fields (`double max_angle` and `double min_size`) where the upper limit for the angle and the lower limit for the proportional length of a valid feature point are stored. In table 1 an excerpt of class `FeaturePointDetector` is displayed, that illustrates how the demands mentioned above have been realized.

In table 1 only the source code for the case of closed polygons is depicted, the case of an unclosed polyline is handled in the `else` block, starting in line 36, but was left out, because it does not differ significantly from the closed case. As you might have already noticed instead of modifying parameter `Polygon p` the method creates a new polygon (called `detected`) which will later contain the same number of vertices, but is able to distinguish if a vertex is just a normal vertex or a feature point. The for-loop from line 15 to line 26 checks every vertex, except the last vertex if it qualifies as a feature point. The checking is done by method `isFeaturePoint(Point, Point, Point, double)`, called in line 17. Method `isFeaturePoint(·)` is described in the following: If a point is recognized as a feature point a new feature point with the geometric properties of the point is created and inserted into polygon `detected` (line 17). If the point is rejected, it will be inserted as a normal point (line 22).

Method `isFeaturePoint` (see table 2) expects 4 arguments. The first arguments are of type `Point`, where `middle` should be a reference to the vertex which is currently checked for being a feature point and `start` and `end` are its predecessor and successor respectively. The `double` value called `polygon_length` should contain the total Euclidean distances of all edges of the current polygon. In lines 5 to 8 the vectors from vertex `middle` pointing to its neighbors are calculated. Line 10 calculates the dot product of these vectors. In lines 11 and 12 the Euclidean distances between `middle`, its predecessor and successor are calculated. Line 13 calculates the angle at vertex `middle`, according to $\cos\theta = \frac{\langle u,v\rangle}{\|u\|\|v\|}$. Line 15 checks if the calculated angle is smaller than the upper limit and line 17 determines if the part influenced by vertex `middle` is large enough to be significant for the total polygon. If both conditions are satisfied vertex `middle` qualifies for being a feature point and the method will return `true`, in all other cases the method will return `false`.

6.2.2 Calculation of feature properties

The calculation of the feature properties is quite straight forward. A 64-bit floating point value provided in the Java programming language by the simple data type `double` is sufficiently accurate to handle all calculations needed. Once all feature points of a polygon have been determined the properties for every point have to be calculated. The calculation of the covariance (see table 3) for a region of support of a feature point can be simplified due to the exclusive occurrence of symmetrical 2×2 matrices containing real values. Thus a matrix A can be represented by 3 `double` values (for position $a_{1,1}, a_{1,2}$ and $a_{2,2}$ since $a_{1,2} = a_{2,1}$ holds). Therefore during the calculation in lines 10 to 20 no access of array elements is necessary which helps to increase efficiency. Lines 12 and 13 compute the x and y values for $P_j - \bar{P}_i$ (see equation 4) and the peculiar nature of the

```
public class FeaturePointDetector{

  private double max_angle;
  private double min_size;
  ...
  public Polygon featureDetection(Polygon p) {
    Polygon detected = new Polygon();
    if (p.isClosed()) {
      int count = p.getCount();
      Point start, middle, end;
      start = p.getVertex(count-1);
      middle = p.getVertex(0);
      double polygon_length;
      polygon_length = p.getLength();
      for (int i = 1; i < count; i++) {
        end = p.getVertex(i);
        if (isFeaturePoint(start, middle, end, polygon_length)) {
          detected.addVertex(new FeaturePoint(middle));
          ((FeaturePoint)detected.getFeaturePoints().lastElement()).
             setAngle(angle);
        }
        else {
          detected.addVertex(new Point(middle));
        }
        start = middle;
        middle = end;
      }
      end = p.getVertex(0);
      if (isFeaturePoint(start, middle,end, polygon_length)) {
        detected.addVertex(new FeaturePoint(middle));
        ((FeaturePoint)detected.getFeaturePoints().lastElement()).
           setAngle(angle);
      }
      else
        detected.addVertex(new Point(middle));
      detected.close();
    }
    else{
      ...
    }
    return detected;
  }
}
```

Table 1: Excerpt of class `FeaturePointDetector`

```java
private boolean isFeaturePoint(Point start, Point middle, Point end,
    double polygon_length) {
  double start_middle_x, start_middle_y, end_middle_x, end_middle_y;
  double dotProd, length1, length2, relative_length;

  start_middle_x = start.getX() - middle.getX();
  start_middle_y = start.getY() - middle.getY();
  end_middle_x = end.getX() - middle.getX();
  end_middle_y = end.getY() - middle.getY();

  dotProd = Covariance.dotProduct(start_middle_x, start_middle_y,
      end_middle_x, end_middle_y);
  length1 = Math.sqrt(Math.pow(start_middle_x, 2) + Math.pow(
      start_middle_y, 2));
  length2 = Math.sqrt(Math.pow(end_middle_x, 2) + Math.pow(
      end_middle_y, 2));
  angle = Math.acos(dotProd / (length1 * length2));
  angle = Math.toDegrees(angle);
  if (angle <= max_angle) {
    relative_length = (length1 + length2) / polygon_length;
    if (relative_length >=  min_length) {
      return true;
    }
    else return false;
  }
  else return false;
}
```

Table 2: method `isFeaturePoint` of class `FeaturePointDetector`

needed covariances allows to calculate the entries with the add operations in lines 14 to 16 (a00, a01 and a11 represent the matrix elements $a_{1,1}, a_{1,2}$ and $a_{2,2}$, they are just named this way to correspond to the array indices of the return value). However also array usage could be avoided during the calculation of the covariance the return value is a two dimensional array of type double (line 25), because it becomes more convenient to use the covariance as an argument for other methods. The method to calculate the

```
   public static double[][] covariance(Vector v) {
     double[][] matrix = new double [2][2];
     double count = v.size();
     double[] center = calculate_center(v);
5    double x_c = center[0];
     double y_c = center[1];
     double x,y;
     double a00 = 0.0, a01 = 0.0, a11 = 0.0;
     Point p;
10   for (int i=0; i < count; i++) {
       p = (Point)v.elementAt(i);
       x = p.getX() - x_c;
       y = p.getY() - y_c;
       a00 += x*x;
15     a01 += x*y;
       a11 += y*y;
     }
     a00 /= count;
     a01 /= count;
20   a11 /= count;

     matrix[0][0] = a00;
     matrix[1][0] = matrix [0][1] = a01;
     matrix[1][1] = a11;
25   return matrix;
   }
```

Table 3: method Covariance.covariance

eigenvectors, which was adapted from the EISPACK library (6) and "fine-tuned" for 2×2 symmetrical matrices with real values is less suited for discussion of source code, because of its focus on efficiency, but reviewing the original EISPACK source code is encouraged. The calculation of eigenvectors is done in file hqr2.f.

To determine if a feature point P_i is convex or concave (needed in equation (5)) it is checked, if P_i is inside the polygon without the vertex P_i. If P_i is inside, the feature point is concave, otherwise it is convex. An efficient algorithm to check whether a given point is inside or outside of a two dimensional polygon is to proceed from the point along an axis (commonly the x-axis) the the boundary of the drawing plane and count how many time an edge is crossed. An odd number of crossings indicate that the point is inside of the polygon, while an even number means that the point lies on the outside. The earliest presentation of this algorithm can be found in (18), although this early version does not include special treatment for the cases if the line intersects one or more vertices. A good overview of the topic can be found in (10).

To determine which eigenvector is the tangent eigenvector and which is the normal eigenvectors (also needed in equation (5)) the perpendicular bisector at a feature point is calculated. Calculation of the bisector for a point is depicted in table 4. The parameter

```
   public static double[] getBisector(Vector v) {
     double[] bisector = new double[2];
     int mid = v.size() / 2;
     Point left = (Point)v.elementAt(mid - 1);
5    Point right = (Point)v.elementAt(mid + 1);
     Point center = (Point)v.elementAt(mid);

     double center_x, center_y, left_x, left_y, right_x, right_y;
     center_x = (double)center.getX();
10   center_y = (double)center.getY();
     left_x = (double)left.getX();
     left_y = (double)left.getY();
     right_x = (double)right.getX();
     right_y = (double)right.getY();
15   double dist1 = Math.sqrt(Math.pow(left_x - center_x, 2) + Math.
        pow(left_y - center_y, 2));
     double dist2 = Math.sqrt(Math.pow(center_x - right_x, 2) + Math.
        pow(center_y - right_y, 2));
     double dist_ratio = dist1 / (dist1 + dist2);

     double bisec_x = left_x + dist_ratio * (right_x - left_x);
20   double bisec_y = left_y + dist_ratio * (right_y - left_y);

     bisec_x = bisec_x - center_x;
     bisec_y = bisec_y - center_y;

25   bisector[0] = bisec_x;
     bisector[1] = bisec_y;
     return bisector;
   }
```

Table 4: Method `Covariance.getBisector`

`Vector v` contains the region of support of the current feature point with the feature point being the element in the middle of the vector. Points `left` and `right` (lines 4 and 5) are points on the edges connecting the feature point with its neighboring vertices. In lines 15 and 16 the Euclidean distance from `left` and `right` to the feature point (stored in `center`) are calculated and the ratio of one distance to the total distance is measured.

To get the bisector a point on the line connecting `left` and `right` is needed where the ratio of the distances has to be the same as to the feature point. This is done in lines 19 and 20. The vector from this point to the feature point is the desired bisector. Once the bisector is calculated the dot product can be used to determine which eigenvector is tangent eigenvector and which is normal eigenvector. The eigenvector where the dot product with the bisector yields 1 is the normal eigenvector and the other one where the dot product yields 0 is the tangent eigenvector. After these properties have been determined the further calculation of feature variation, feature side variation and feature size (equations (5), (6) and (7)) becomes quite simple.

6.2.3 Path creation

This section deals with the creation of a path representing a correspondence between two polygons using dynamic programming techniques (see 5.4). Recapitulating that all optimal paths for every possible initial correspondence have to be calculated it is highly recommended to minimize the computation during each path calculation. Thus it is sensible to compute the discard and similarity costs (see 5.3.3 and 5.3.4) in advance once and access the calculated results later multiple times. This is done in method `calculateAllDeltaCosts(Polygon, Polygon, int)` depicted in table 5. The para-

```
public static double[][][][] calculateAllDeltaCosts(Polygon source
    , Polygon target, int skips) {
  int dim1 = source.getFeaturePointCount();
  int dim2 = target.getFeaturePointCount();
  int source_index, target_index;
  double[][][][] delta_field = new double[dim1][dim2][skips+1][
      skips+1];
  for (int i = 0; i < dim1; i++) {
    for (int j = 0; j < dim2; j++) {
      for (int k = 0; k <= skips; k++) {
        source_index = (i - k + dim1) % dim1;
        for (int l = 0; l <= skips; l++) {
          target_index = (j - l + dim2) % dim2;
          delta_field[i][j][k][l] = deltaCosts(source, target, i,
              source_index, j, target_index);
        }
      }
    }
  }
  return delta_field;
}
```

Table 5: method `MorphCalculator.calculateAllDeltaCosts`

meter `skips` specifies how many feature points are allowed to be omitted while determining the best allowable predecessor for a node in the dynamic programming graph.

In line 2 and 3 the number of feature points for source and target are set and line 5 reserves the amount of data storage needed for all allowed delta costs given the number of feature points and the number of skips. The nested for-loops assign the according delta costs to every data field in the four dimensional `double` array `deltaCosts`. Costs for a correspondences of point $P_{\mathcal{S}i}$ of source and $P_{\mathcal{T}j}$ of target are stored at positions `deltaCosts[i][j][][]`.

The associated costs for a correspondence between S_i and T_j and assuming as the previous correspondence S_{i-k} and T_{j-l} if those points are insides the allowed boundaries are stored at `deltaCosts[i][j][k][l]`. Method `deltaCosts(Polygon, Polygon, int, int, int, int)`, displayed in table 6 calculates just these costs according to equation (12). Lines 3 and 4 add the costs for discarding the feature points between S_i and

```
public static double deltaCosts(Polygon source, Polygon target,
    int start_index_s, int end_index_s, int start_index_t, int
    end_index_t) {
  double costs = 0.0;
  costs += disCosts(source, start_index_s, end_index_s);
  costs += disCosts(target, start_index_t, end_index_t);
  costs += FeaturePoint.calculate_Sim_Cost(source.getFeaturePoint(
      end_index_s), target.getFeaturePoint(end_index_t));
  return costs;
}
```

Table 6: method `MorphCalculator.deltaCosts`

S_{i-k} and between T_j and T_{j-l} respectively. In line 5 the costs for the correspondence between S_{i-k} and T_{j-l} are added. After all costs have been calculated the path finding process can commence. This is done in method `calculatePath(Polygon, Polygon, int)` which is depicted in table 7. In section 5.4.2 the dynamic programming graph is depicted as an $m \times n$ rectangular graph if source has m and target has n feature points. A complete correspondence is a path starting at $node(0,0)$ and ending at $node(m,n)$ which represents the same correspondence as $node(0,0)$. The graph is represented in `calculatePath` with a two dimensional array of `Node`, allocated in line 8. Since the dynamic programming graph has to be built for every possible initial correspondence of feature points of source and target, method `initializeField(Node[][], int, int, Vector, Vector)` is able to initialize `field` with variable values depending on the initial correspondence so that independent from the initial correspondence always a path from $node(0,0)$ to $node(m,n)$ has to be constructed. To illustrate this more clearly, if the initial correspondence is set to S_i corresponds with T_j instead of S_0 and T_0 in $node(0,0)$ the correspondence of S_i and T_j will be stored along the associated similarity costs. In $node(k,l)$ the correspondence between $S_{(i+k) \pmod n}$ and $T_{(j+1) \pmod m}$ is stored, with their similarity costs and the added costs for the optimal path leading from $node(0,0)$ to this node. Finding the optimal path to a node is the task of method `setPredecessor(Node[][], int, int, double[][][][], int, int, int)`.

This method checks the path costs to every allowable predecessor of a $node(i, j)$ and picks the node with the least path costs as the optimal predecessor. In $node(i, j)$ these path costs are stored as well as the delta costs (see equation (12)) to get from the the optimal predecessor to $node(i, j)$. Storing these path costs in every node is the technique of *memoization*, introduced in section 5.4.1.

Allowable nodes for predecessors are restricted, either by the maximal number of skips or by the boundaries of the rectangular dynamic programming graph. For example if 2 points might be discarded in both, target and source $node(1, 4)$ would have the following allowable predecessors: $node(1, 3)$, $node(1, 2)$, $node(1, 1)$, $node(0, 3)$, $node(0, 2)$ and $node(0, 1)$. Though the number of skips would allow more predecessors, like $node(m - 1, 3)$ this is not allowed because of the boundaries of the dynamic programming graph since it would not fit to the initial correspondence in $node(0, 0)$. According to this restrictions in the dynamic programming graph first of all the first column and the first row are calculated (table 7 lines 16 to 18 and 19 to 21). After this step the other rows are calculated from row 1 to row m (lines 22 to 26). Once $node(m, n)$ is set, the dynamic programming graph is completed and the optimal path is constructed. It is stored in $node(m, n)$ and since every node knows its optimal predecessor can be tracked back to $node(0, 0)$. Once this is done, it has to be checked, if the optimal path with the set initial correspondence has less costs than all other paths with different initial correspondences calculated so far. This is done in line 28.

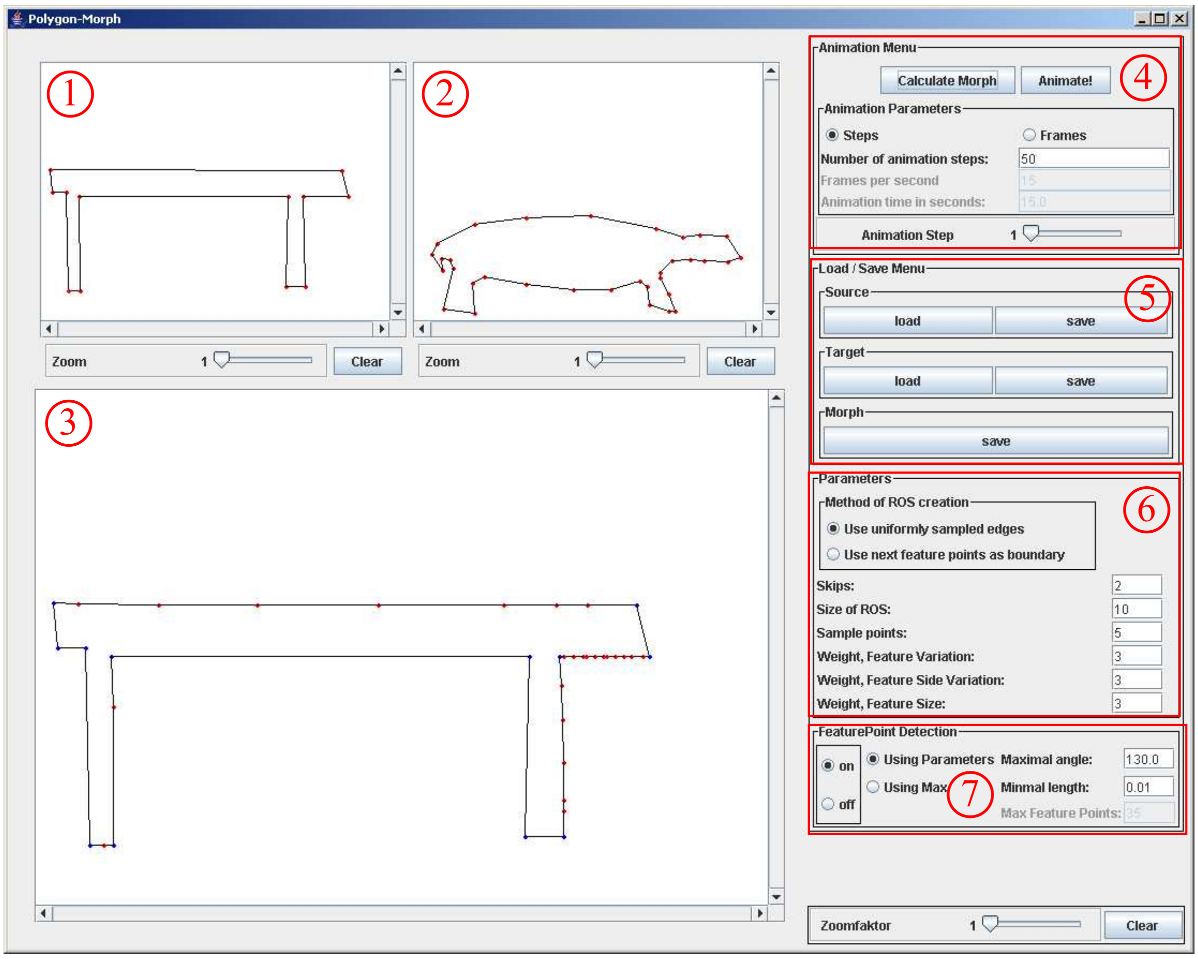
This excerpts close the section about source code detail and next section will give an overview over the capabilities of the GUI and how to experiment with it.

```java
  public static Path calculatePath(Polygon source, Polygon target, int skips) {
    double[][][][] deltaCosts = calculateAllDeltaCosts(source, target, skips);
    int dim1 = source.getFeaturePointCount();
    int dim2 = target.getFeaturePointCount();
    double min_path_costs = Double.MAX_VALUE;
    Path path = new Path();
    Node current_node;
    Node[][] field = new Node[dim1+1][dim2+1];
    int i,j;
    Vector source_fps = source.getFeaturePoints();
    Vector target_fps = target.getFeaturePoints();
    for (int k = 0; k < dim1; k++) {
      for (int l = 0; l < dim2; l++) {
        field = initializeField(field, k, l, source_fps, target_fps);
        field[0][0].setPathCosts(deltaCosts[k][l][0][0]);
        for (i = 1; i < dim1+1; i++) {
          setPredecessor(field, i, 0, deltaCosts, k, l, skips);
        }
        for (i = 1; i < dim2+1; i++) {
          setPredecessor(field, 0, i, deltaCosts, k, l, skips);
        }
        for (i = 1; i < dim1+1; i++) {
          for ( j = 1; j < dim2+1; j++) {
            setPredecessor(field, i, j, deltaCosts, k, l, skips);
          }
        }
        current_node = field[dim1][dim2];
        if (current_node.getPathCosts() < min_path_costs) {
          min_path_costs = current_node.getPathCosts();
          path = new Path();
          path.add(current_node);
          path.setCosts(current_node.getPathCosts());
          while (current_node.getPredecessor() != null) {
            current_node = current_node.getPredecessor();
            path.add(current_node);
          }
        }
      }
    }
    return path;
```

Table 7: method `MorphCalculator.calculatePath`

7 Application

This section is supposed to provide a short manual to the application that evolved during this thesis. The GUI is depicted in Fig. 24. In the following the use and meanings of the components of the GUI, marked by the red numbers and boxes will be given.



[● Steps] ○ Frames
Number of animation steps: 50
Frames per second 15
Animation time in seconds: 15.0
Animation Step 1

[● Use uniformly sampled edges]
○ Use next feature points as boundary
Skips: 2
Size of ROS: 10
Sample points: 5
Weight, Feature Variation: 3
Weight, Feature Side Variation: 3
Weight, Feature Size: 3

[● on] ○ off
[● Using Parameters] ○ Using Max
Maximal angle: 130.0
Minmal length: 0.01
Max Feature Points: 35

Zoom 1 · Zoom 1 · Zoomfaktor 1

Figure 24: Application

7.1 Source and target drawing pane

Marked by number 1 and 2 are the source and target drawing panes. In these panes the user can define source and target by mouse commands. To start a new polygon the mouse pointer has to be moved either inside the pane labeled 1 or labeled 2. Clicking the left mouse button creates a vertex of a polygon. A preview of the polygon is displayed in red dashed lines in the drawing pane. To finish a once started polygon it must be closed. To close a polygon simply move the mouse pointer near the fist created vertex and press the left mouse button again. If the polygon is successfully closed, the dashed lines will become solid. Please be aware that in the current state of the application it is not possible to delete once created vertices. To abort the creation of a polygon the mouse pointer has just to be moved outside of the drawing pane. The sliders below the drawing panes can be used to zoom into the drawing panes and the clear button can be used to reset the drawing panes. Saving and loading of polygons is also possible, please refer to section 7.4 for information on this topic.

7.2 Morphing sequence pane

The pane labeled with number 3 is used to display the morphing sequence. Therefore the it can not be accessed directly by a user, apart from zooming and clearing operations. The pane will not display anything unless source and target are created either by drawing (see 7.1) or by loading existing polygons (see 7.4). If source and target are set, the a morphing sequence will have to be calculated in the animation menu, marked by number 4 (7.3). After this is done, the morphing sequence will finally be displayed.

7.3 Animation menu

The animation menu consists of two buttons, a parameter menu and an additional slider. The first button, labeled “Calculate Morph”, is enabled after source and target are set in drawing panes 1 and 2. This button calculates a morphing sequence dependent on the parameters set in the animation menu and dependent on the choices of the other parameters (marked by 6 in Fig. 24 and described in 7.5 and the choices made in the feature point detection menu (marked by 7 in Fig. 24, described in 7.6. After the calculation is completed the slider and the button labeled “Animate!” will be enabled. As long as no morphing sequence is calculated, both slider and animation button will stay disabled. Once a morphing sequence is calculated clicking on the “Animate!” button will trigger the display of the sequence in the morphing sequence pane (Fig. 24, number 3). The length of the animation will depend on the settings of the parameters in the animation menu. The user can choose between a fixed number of in-between frames, which each frame being displayed for about 50 milliseconds, depending on the speed of the computer running the application. The second option is the set the duration of the animation to a fixed number of seconds and pick the desired rate of frames per seconds. Please not that this option is still in an experimental status - depending on the computational power the morphing sequence will roughly take the time specified in the text field, but it is far from being really accurate. The slider can be used to display single in-between steps of the morphing sequence. After the calculation of a morphing sequence is completed changing the sliders position will display the frame, according to the value of the slider. The current value of the slider is displayed on its left.

7.4 Load / save menu

The load/save menu (Fig. 24, number 5) allows the user to store polygons drawn in the source or target panes (Fig. 24, number 1 and 2), to store a complete morphing sequence displayed in the morphing sequence pane (Fig. 24, number 3) or to load polygons as source and target.

7.4.1 Saving options

A morphing sequence can be stored to the SVG file format, while source and target can be stored either as an SVG file or in the `.2dp` file format. The extension `.2dp` marks that a text file contains information of a polygon. An exemplary polygon in the `.2dp` format is displayed in table 8. Lines 1 and 2 work as header information, every valid polygon in `.2dp` format has to first include these lines. Line 3 has to be empty to separate the header from the actual polygon data. In line 4 it is determined if the polygon is closed or not closed. Another blank separates this information and the data of all vertices, contained in the polygon. Each following line represents one vertex. A vertex is described by first its x coordinate, then a comma as a separator and its y coordinate. The simple format of a `.2dp` file allows for creation of polygons in common

```
   2D-Polygon for Morphing
   listing all vertices as x and y coordinates now:

   is closed
 5
   37,63
   289,47
   301,210
   169,121
10 20,206
```

Table 8: Exemplary Polygon in `.2dp` format

text editor.

An SVG file representing a complete morphing sequence will contain the source polygon as a `<path>` element. The animation sequence will be displayed, using the `<animate>` element. Hence the morphing sequence will only be displayed correctly if the `animate` element is properly supported in the used SVG viewer.

The "Save" buttons for source and target are enabled as soon as a polygon is contained in the drawing panes (Fig. 24, number 1 and 2), the "Save" button for a morphing sequence is enabled, after the calculation of a morphing sequence. Clicking the buttons opens a new dialog to choose the location and name of the file which should be stored. For a polygon the default file format is `.2dp`.

7.4.2 Load options

Clicking on one of the "Load" buttons opens a new dialog to choose a `.2dp` which shall be loaded. The import of SVG files is not supported at the current state of the application. Since `.2dp` files only represent a single polygon and not a morphing sequence only contents for the drawing panes of source and target (Fig. 24, number 1 and 2) can be loaded.

7.5 Parameters

In this menu (Fig.24, number 6) the user has several options to change the different parameters used in the algorithm to solve the Vertex Correspondence Problem. First of all, the user can choose between two different methods to create the region of support for a feature point $ROS_h(P_i)$. The different methods are discussed in section 5.3.1. The first option labeled "Use uniformly sampled edges" samples every edge of the polygon with a constant number of point (specified below in the field labeled "Sample Points"). Depending on the size h of $ROS_h(P_i)$ the user can choose how many adjacent edges of a feature point shall be included into the region of support. Parameter h can be set in the text field labeled "Size of ROS". The second option for the region of support restricts the region of support to the parts of the polygon that are between feature point P_i and its adjacent feature points. In this method the number of points representing an edge has no influence. The text field labeled "Skips" determines the maximum of feature points that may be discarded during the creation of a path in the dynamic programming graph (see section 5.4.2). The last three parameters determine the weights for the three feature point properties (see 5.3.2) in equations (8) and (9).

7.6 Feature point detection

In this menu the parameters concerning the detection of feature points can be modified. The first option to choose is, whether there should be a feature point detection at all. This can be done by the radio buttons labeled "on" and "off" displayed in Fig. 24, number 7. If the feature detection is turned off, every vertex of a polygon is treated as a feature point. Please be aware that the number of feature points strongly affects the amount of computation needed to solve the Vertex Correspondence Problem, employing the approach, discussed in section 5. If the feature detection is turned on, feature points will be detected, using the techniques, suggested in 3.2 and 6.2.1. The user can modify the parameter α_{max} in the text field labeled "Maximal angle" and change the minimal size of the local neighborhood of a valid feature point, in respect to the total size of the polygon in the text field labeled "Minimal length". As an alternative option for experiments the user could also choose the feature points with the greatest sharpness. If this option is all vertices are ordered according to their sharpness. Dependent on the parameter in text field "Max Feature Points" the first n vertices in this ordering will be assumed to be feature points.

7.7 Differences in the Applet of the application

The applet of the application misses the load and save menu (see 7.4). It is replaced by a drop down menu to choose from some predefined polygons. All other components of the application work the same.

8 Results

This section will show some exemplary morphing sequences, that were created using the application, described in section 7. Unlike stated otherwise the weights $\omega_\sigma, \omega_\tau$ and $\omega_\rho$for the different feature properties (see equation (8) and (9)) are set to $\frac{1}{3}$ each. The size h for $ROS_h(P_i)$ was set to 20. If feature point detection was enabled, α_{max} was set to 130°and the minimal proportional size of a feature element was set to 0.01.

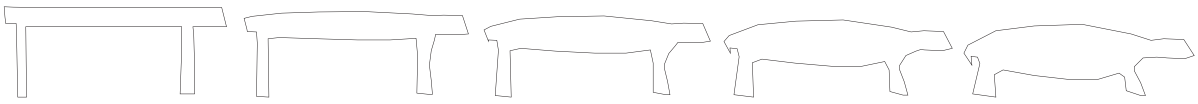


Figure 25: Morph sequence table to tortoise
Parameters set as described in the beginning of this section, feature detection was enabled.

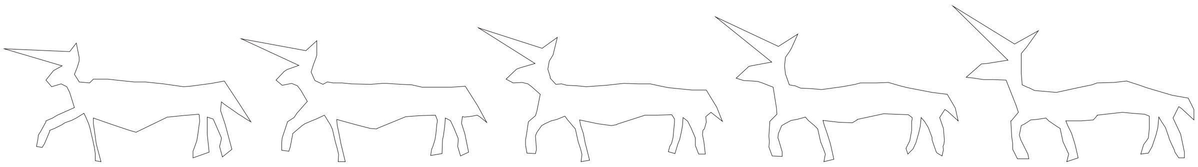


Figure 26: Morph sequence unicorns
Parameters set as described in the beginning of this section, feature detection was enabled.

In Fig. 25 and 26 the number of allowed skips of feature points was set to 2. In both examples the region of support for a feature point was bounded by its adjacent feature points.

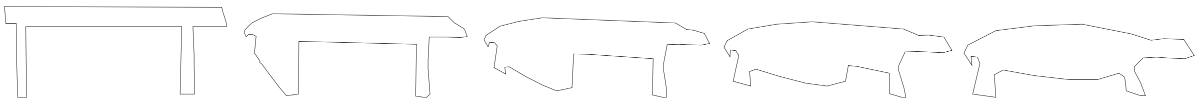


Figure 27: Morph sequence table to tortoise without feature detection

Turning on the feature detection does not only enhance the algorithm, but can also lead to better results in the morphing sequence. Please observe Fig. 27. In the depicted morphing sequence all parameters were set to the same values as in Fig. 25, but the feature detection was disabled. The resulting morphing sequence is significantly worse than the sequence displayed in Fig. 25.

Oftentimes the method how to calculate $ROS_h(P_i)$ (see section 5.3.1)also affects the outcome of the algorithm. A priori the author has not been able to determine so far which calculation method of $ROS_h(P_i)$ works best for given source and target. A set of examples is presented in Fig. 28. While in the first example depicted in Fig. 28 (labeled with **a)** and **b)**) the method of uniformly sampled edges, shown in **b)** delivers a preferable result, since less vertices are moved and self-intersection could be prevented.

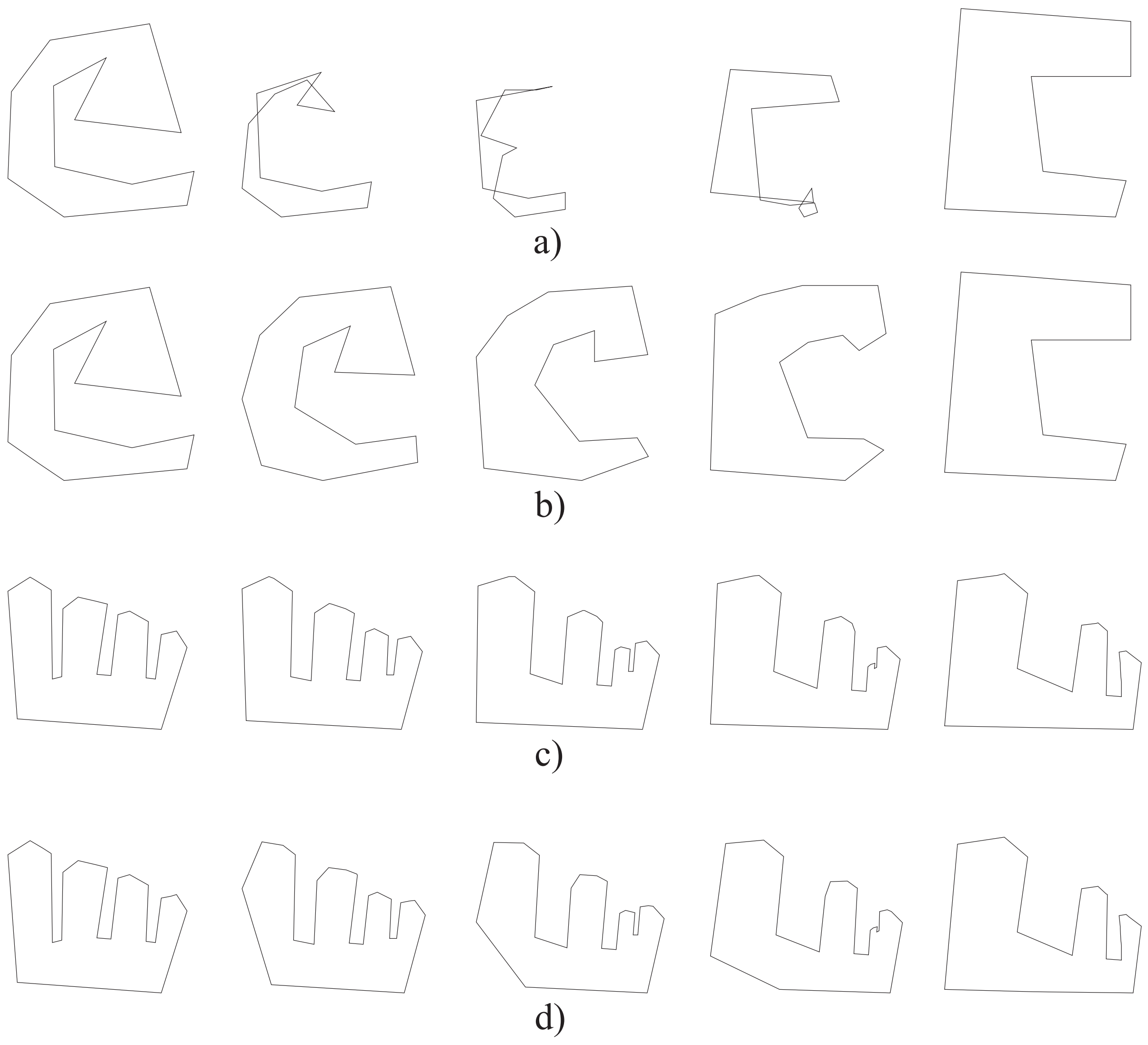


Figure 28: Influence of $ROS_h(P_i)$ on the resulting morphing sequences
Apart from the calculation of $ROS_h(P_i)$ all parameters were set to the same values. The size h of $ROS_h(P_i)$ was set to 20. The sequences displayed in **a)** and **c)** use the calculation method, where $ROS_h(P_i)$ is bounded by the adjacent feature points of P_i. In **b)** and **d)** each edge is represented by 5 sample points, which means that for each P_i $ROS_h(P_i)$ is composed of the two next edges to the left and to the right.

In the second example (**c)** and **d)**), on the other hand, the region of support bounded by feature points, displayed in **c)**, delivers a favorable result. Please observe the distortions on the lower left side in morphing sequence **d)**, although source and target look quite similar. In **c)** these distortions could be prevented.

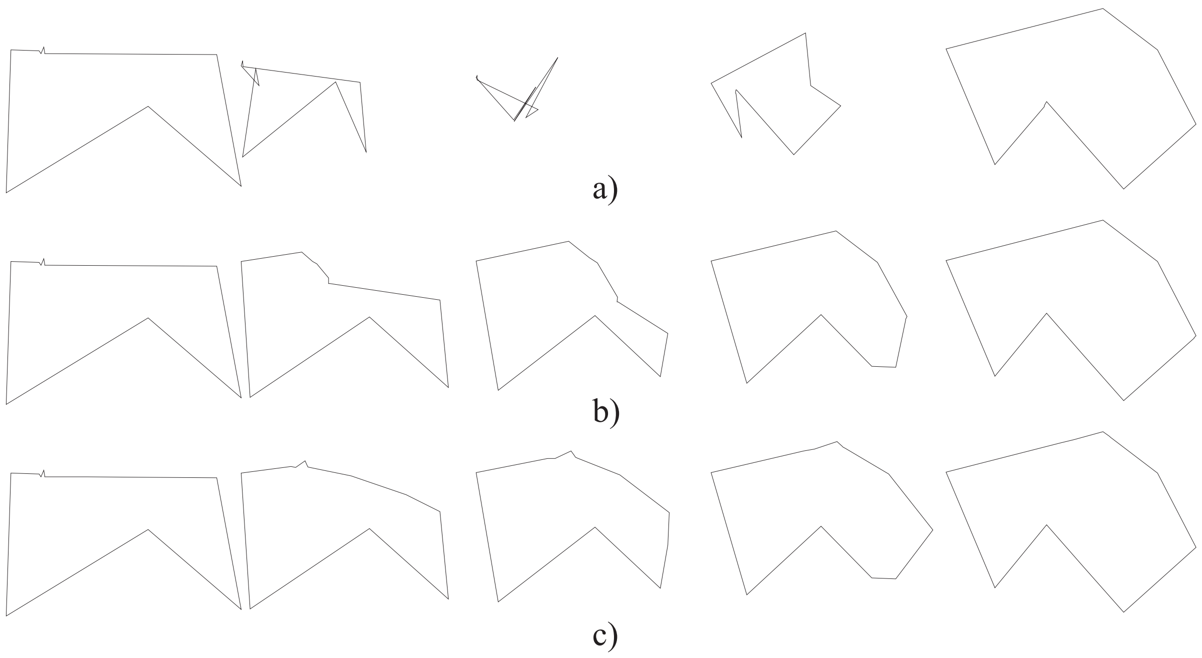


Figure 29: Influence of the maximal number of skips
In all sequences apart from the number of skips the same parameter values were used. **a)** allows for 0, **b)** for 1 and **c)** for 2 skips in the dynamic programming graph. Feature point detection was enabled.

The number of allowed skips during the calculation of the dynamic programming graph can also have a huge influence on the morphing sequence. In Fig. 29 this influence is shown. All sequences use, apart from the maximal allowed number of skips, the same parameter values. In **a)** the number of allowed skips is set to 0. This leads, in the depicted scenario, to a correspondence which involves large movements in every part of the polygon and leads, using straight animation paths, to significant self-intersections. The second sequence **b)** already avoids self intersections and the chosen correspondences seem to be more reasonable than in **a)**. The maximum of skips in this sequence was set to 1. Though **b)** is undoubtedly better than **a)** still a lot of distortions happen in the upper part of the displayed polygon from the left to the right side. In **c)** most of these "unnecessary" movement could be avoided, allowing for 2 skips in the dynamic programming graph. The resulting morphing sequence is preferable to those depicted in **a)** and **b)**. However the influence of the number of skips is not always as visible and beneficial as in the shown example Fig. 29.

9 Outlook

Though the application described in section 7 allows the user to test the algorithm to solve the vertex correspondence problem, introduced in section 5, and the influence of several parameters, it can still be upgraded in some respects.

The introduction of general two dimensional shapes instead of mere polygons would be a definite improvement, but also require a lot of additional programming work. New classes to represent general shapes are needed, which are not only efficient in the amount of storage used, but also the steps of the algorithm must be efficiently applicable. In many cases of two dimensional shapes not only feature points will have to be handled, but also weight-points for curves will have to be moved accordingly. Methods that are capable of processing polygons will have to be altered to cope with the different form of data, although hopefully the core of the methods will applicable for the extended case scenario.

In the actual animation one could try to evolve application from simply animation along straight lines to more sophisticated approaches, like the ones described in (8; 15; 16; 17) for example. However employing these approaches for the vertex path problem will mean that the export to SVG will also have to be revised completely, since the `<animate>` element, used in its current form can only handle straight lines as animation paths.

The application would also benefit greatly if the user could alter source and target in their panels by dragging and dropping vertices.

Section 3.2 described one method to detect high curvature point in two dimensional shapes and the employed altered approach for appliance on polygons. It could be interesting to test other algorithms to detect high curvature points. Not only to compare the results for feature detection, but mainly to see how the performance of the algorithm introduced in 5 changes depending on different sets of detected feature points. To a certain degree this is already possible by changing the parameters for the implemented feature detector, but other algorithms would broaden the possibilities to experiment.

In respect to the algorithm (5) one could think about some other criteria, which could be introduced to measure the similarity of the local neighborhood of a feature point. At the moment no feature property (see 5.3.2) yields information on the physical position (in terms of x and y coordinates) of a feature point. For instance one could take into account the location and distance in respect to the center of a shape. Feature points that are located in the same area in respect to the center of their associated shape would get less additional similarity costs. If two feature points lie in opposite directions of the center a cost penalty could be added to discourage a correspondence between such points. However such an idea must be thoroughly tested, if it will influence the overall performance of the algorithm in a beneficial way.

An analysis, if techniques, used in other fields of computer science, like pattern matching or object recognition could be applied in the context of solving the correspondence problem for morphing sequences could also yield interesting insights.

Future work in this area could include, besides extensions to the proposed solution, a preprocessing of source and target to determine what kind of algorithm might work best for these particular polygons to solve the Vertex Correspondence Problem.

10 Conclusions

This thesis offered a brief introduction to the problems involved with polygon morphing and the approach of Liu et al. (11) was discussed in detail. The appendant application allows for experimentations with user defined polygons and is able to demonstrate the influence of the different parameters involved in the algorithm.

The approach proposed in section 5 offers a fast solution to the Vertex Correspondence Problem, if it is combined with a feature detection as a preprocessing of the polygon. The feature detection is essential for the efficiency of the algorithm, because it greatly reduces the amount of computation in many cases.

However the approach does not work well for polygons, which do not have significant common parts in their appearance. Also regular figures, like a square for example, cause problems since all feature points have the same feature properties. To handle regular polygons either new feature properties have to be introduced or other solutions to the Vertex Correspondence Problem need to be employed. Furthermore the algorithm just takes local polygonal features into account and a combination with global features might yield better results.

One of the goals of the approach, to automatize the morphing process, if possible, is not satisfyingly achieved. Though the user has no direct influence on single correspondences, the choice of the optimal parameters for a morphing sequence turned out to be strongly dependent on source and target. This leads to user interaction which would ideally be not necessary.

List of Figures

List of Tables

References

[1] Asada, H ; Brady, M: The curvature primal sketch. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 8 (1986), Nr. 1, S. 2–14. – ISSN 0162–8828

[2] Beus, H. L. ; Tiu, S. S. H.: An improved corner detection algorithm based on chain-coded plane curves. In: *Pattern Recogn.* 20 (1987), Nr. 3, S. 291–296. http://dx.doi.org/http://dx.doi.org/10.1016/0031-3203(87)90004-5. – DOI http://dx.doi.org/10.1016/0031–3203(87)90004–5. – ISSN 0031–3203

[3] Bresenham, Jack E.: Algorithm for computer control of digital plotter. In: *IBM Syst.* 4 (1965), S. pp.25–30

[4] Chen, Shenchang E. ; Parent, Richard E.: Shape Averaging and it's Applications to Industrial Design. In: *IEEE Comput. Graph. Appl.* 9 (1989), Nr. 1, S. 47–54. http://dx.doi.org/http://dx.doi.org/10.1109/38.20333. – DOI http://dx.doi.org/10.1109/38.20333. – ISSN 0272–1716

[5] Chetverikov, Dmitry ; Szabó, Zsolt: A Simple and Efficient Algorithm for Detection of High Curvature Points in Planar Curves. In: *Proc. 23rd Workshop of the Austrian Pattern Recognition Group*, 1999, S. 175–184

[6] The University of Tennessee: *EISPACK library*. http://www.netlib.org

[7] Freeman, Herbert ; Davis, Larry S.: A Corner-Finding Algorithm for Chain-Coded Curves. In: *IEEE Trans. Computers* 26 (1977), Nr. 3, S. 297–303

[8] Gotsman, Craig ; Surazhsky, Vitaly: Guaranteed intersection-free polygon morphing. In: *Computers & Graphics* 25 (2001), Nr. 1, S. 67–75

[9] Gumhold, S. ; Wang, X. ; McLeod, R.: *Feature Extraction from Point Clouds*. citeseer.ist.psu.edu/gumhold01feature.html. Version: 2001

[10] Haines, Eric: Point in polygon strategies. In: *Graphics gems IV* (1994), S. 24–46. ISBN 0–12–336155–9

[11] Liu, Ligang ; Wang, Guopu ; Zhang, Bo ; Guo, Baining ; Shum, Heung-Yeung: Perceptually Based Approach for Planar Shape Morphing. In: *pg* 00 (2004), S. 111–120. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/PCCGA.2004.1348341. – DOI http://doi.ieeecomputersociety.org/10.1109/PCCGA.2004.1348341. – ISSN 1550–4085

[12] Pauly, M. ; Keiser, R. ; Gross, M.: *Multi-scale feature extraction on point-sampled surfaces*. citeseer.ist.psu.edu/pauly03multiscale.html. Version: 2003

[13] Rosenfeld, A. ; Johnston, E.: Angle Detection on Digital curves. In: *IEEE Trans. Computers* 22 (1973), Sept., S. 875 – 878

[14] Rosenfeld, Azriel ; Weszka, Joan S.: An Improved Method of Angle Detection on Digital Curves. In: *IEEE Trans. Computers* 24 (1975), Nr. 9, S. 940–941

[15] Sederberg, Thomas W. ; Gao, Peisheng ; Wang, Guojin ; Mu, Hong: 2-D shape blending: an intrinsic solution to the vertex path problem. In: *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 1993. – ISBN 0–89791–601–8, S. 15–18

[16] Sederberg, Thomas W. ; Greenwood, Eugene: A physically based approach to 2–D shape blending. In: *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 1992. – ISBN 0–89791–479–1, S. 25–34

[17] Shapira, Michal ; Rappoport, Ari: Shape Blending Using the Star-Skeleton Representation. In: *IEEE Comput. Graph. Appl.* 15 (1995), Nr. 2, S. 44–50. http://dx.doi.org/http://dx.doi.org/10.1109/38.365005. – DOI http://dx.doi.org/10.1109/38.365005. – ISSN 0272–1716

[18] Shimrat, M.: Algorithm 112: Position of point relative to polygon. In: *Commun. ACM* 5 (1962), Nr. 8, S. 434. http://dx.doi.org/http://doi.acm.org/10.1145/368637.368653. – DOI http://doi.acm.org/10.1145/368637.368653. – ISSN 0001–0782

[19] Shirazi, Jack: *Java: performance tuning*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2000. – ISBN 0–596–00015–4

[20] The World Wide Web Consortium (W3C): *Scalable Vector Graphics (SVG)*. http://www.w3.org/Graphics/SVG/

[21] Wagner, David B.: Dynamic Programming. In: *The Mathematica Journal* 5 (1995), Nr. 3, 42-51. citeseer.ist.psu.edu/268391.html

Legal statement

I hereby declare that this thesis was written single-handedly by me and that I did not use any other additional resources apart from those being stated.

Osnabrück, 26th October 2006 ....
(Signature)