A new core-based morphing algorithm for polygons

Martina Málková*†

Department of Computer Science and Engineering
University of West Bohemia
Pilsen / Czech Republic

Abstract

Morphing is a process of shape transformation between two objects. This paper focuses on morphing of simple polygons. In general, the key part of most morphing methods is to find correspondence between vertices of both objects. We present a new algorithm trying to avoid this step. Using an idea of intersection of two polygons (called core) the problem of morphing polygons is decomposed to several sub-problems of morphing polylines. We describe a solution of morphing problem for the case when the core consists of one polygon. The proposed solution of morphing two polylines does not bring results smooth enough for all polygons, but it has satisfying results for polygons of spiral type which are usually problematic for correspondence based approaches. The algorithm is designed in a way keeping the doors open for other methods of morphing two polylines.

Keywords: morphing, polygon, computer graphics

1 Introduction

Morphing can be understood as a process when one shape continuously transforms into another one. In this context, we can find morphing everywhere in the nature: plants and animals are growing, clouds are moving on the sky, rocks change their shape because of erosion etc. Also movement can be described as morphing, if we set key positions of the moving object as the morphed objects and the intermediate positions can be computed by morphing.

In its electronic form, morphing is used mainly for computer animation. But there are many other applications such as special effects in movies, design, image compression and data visualisation.

Papers concerned with morphing range from morphing images and polygons [7, 9] to morphing meshes [5] and volume data [8] in 3D.

Most morphing methods need to find correspondence between vertices of both objects. Effort to avoid this step resulted in [8], where a new algorithm based on intersection of two objects is introduced.

Most algorithms have problems with morphing polygons that are not star-shaped or monotonous, or with morphing of considerably different polygons. We concentrated on this problem trying to find an algorithm dealing with these configurations. Our effort ended up in an algorithm based on the idea of object intersection from [8]. While [8] concerns volumetric data, we concentrate on morphing polygons. Our approach was inspired by [8] but the technical solution is different, see Section 3 in detail. Our algorithm does not compute the correspondence between the two given polygons, so its output is dependent on their mutual position. This concept has advantages as well as disadvantages - the user is free to specify any mutual position of polygons and in this way to influence the results. On the other hand, the correspondence approach is able to find the best mutual position without user interaction.

Our approach divides the problem of morphing polygons to morphing polylines with the advantage that polylines start and end at the same point and do not intersect. Other advantages and disadvantages of the whole method depend on the actual process used to morph these polylines. Here we present a simple method, where points are moving towards their neighbours and then towards their neighbours' neighbours, etc.

Section 2 describes existing techniques for polygon morphing, their advantages and disadvantages. Our algorithm is outlined in Section 3. Section 4 provides a comparison between our algorithm and two of the methods described in Section 2. Possible improvements and future work are proposed in Section 5.

2 State of the art

As was already outlined in Section 1, most morphing techniques consist of two main steps. The first of them is finding correspondence between vertices of the source and the target polygon. The source and the target polygon usually do not have the same number of vertices, so the algorithms usually add new vertices to polygons at this step. The second step is to find trajectories between corresponding vertices. A simple choise is to use linear interpolation, but it is usually not very useful, because it does not perform well in computing rotational morphing. Figure 1 provides an example where linear interpolation is not a good choice.

*tina.malkova@centrum.cz

†This work was supported by Ministry of Education – project No. LC06008

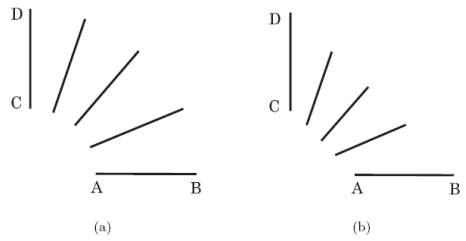

Figure 1: An expected morphing sequence (a) and a morphing sequence using linear interpolation (b), from [3]

First approach worth mentioning is Sederberg's algorithm [7]. The main idea there is that the polygon is constructed of some type of wire (the parameters of the wire can be changed by the user). To get the resulting polygon, we need to bend or stretch the wire. The goal is to minimize the amount of work needed to create the target polygon. This algorithm is suitable for morphing between similar polygons, where one of them is rotated or translated. For the case of a rotated polygon, it does not preserve its shape during the process, because it uses linear interpolation between the corresponding vertices, but it still has sufficient outputs. It has problems with highly dissimilar shapes, where intersections usually occur.

In [6] they use an interpolation of *intrinsic parametres* (e.g. edge lengths or internal angles) instead of interpolating vertices. This avoids edge collapsing and non-monotonic angle changes. This idea was further used for morphing of planar triangulations in [10] and [11].

Another optimization-based algorithm was presented in [12]. It uses a similarity function to obtain the vertex correspondence.

In [9] morphing using the star skeleton was outlined. The algorithm first decomposes the source and the target polygons into star-shaped polygons. Then constructs the skeletons of the decompositions. To get the actual shape, it first interpolates the skeletons and then reconstructs the shape according to those skeletons. This technique does not include finding correspondence between vertices, so it needs to be found by using some other method or manually specified. The decomposition into star-shaped polygons is useful because these polygons can be morphed without self-intersections.

Topology merging technique [4, 1] is used in 3D to obtain isomorphic meshes from two input meshes with different connectivity. This method can be easily adapted in 2D for polygons [3] (so called "2D merging algorithm"). Input polygons are mapped to the unit disc. Then both mappings are merged, the vertices of the first polygon are mapped on the second polygon and vice versa using inverse mapping. This results in polygons with the same number of vertices. A linear interpolation is used to obtain the resulting morphing transition. This technique is suitable for convex, star-shaped or slightly non-convex polygons. For highly non-convex polygons (spirals etc.) it produces self-intersections during the morphing transition.

[8] introduced an algorithm based on a volume intersection of two objects, called core. During the morphing process, the core is left untouched and the other object parts grow out of or disappear into it. Necessary changes to achieve this effect are computed according to the neighbourhood of voxels.

3 The proposed solution

Brief description of the proposed solution is as follows: First, the core is computed as the intersection of both input polygons. The parts of polygons which are not in the core will either grow up or disappear in the core. Then it is necessary to compute for each vertex the so-called topological distance. Next step is to compute trajectories for those particular vertices, which are not part of the core, according to their topological distance.

Main novelty of our method in comparison with [8] is a different approach for vertices trajectory computation, which is in [8] based on neighbourhood of voxels, not on topological distances.

Let us now describe our algorithm in detail. The core is an intersection of the source polygon and the target polygon. It can consist of several parts (Fig.2). In this text, we will suppose that the core has only one part. As our algorithm does not compute correspondence between the source polygon and the target polygon, its output is dependent on the mutual position of these polygons. Therefore it is not able to solve the case with no core (Fig.2a). The case of more than one intersection (Fig.2c) is more complicated and our algorithm is not able to solve it yet.

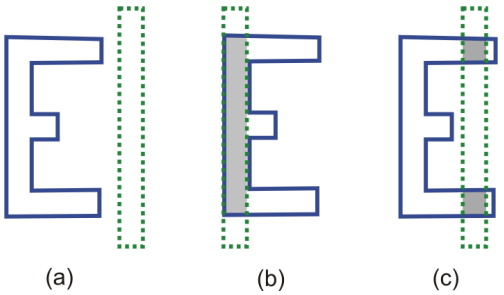

Figure 2: Different number of core parts
(Continuous line: source polygon, dotted: target polygon, grey: core)

The core is considered to be a morph base - part which does not change during the morphing process. The other parts of objects are either growing from the core or disappearing in it. If A is the source polygon, and B the target polygon, the parts growing from the core are gained by computing $B-A$ and the parts disappearing in the core are gained by computing $A-B$ and the core is computed as $A \cap B$. One way to compute $B-A$, $A-B$ and $A \cap B$ is to

use Weiler Atherton algorithm [2]. In our implementation, we use GPC library [13].

As growing from the core is an inverse process to disappearing, we will describe only disappearing. All the non-core parts of the polygon are treated in the same way, so the rest of this section focuses only on one part.

There are many possible solutions how to make the parts disappear in the core. But most of them use so called "topological distance" of vertices.

Topological distance of the vertex v_i is the number of vertices on the boundary between v_i and the nearest intersection vertex (vertex, that was not on the original polygons, but arised from the intersection of the polygons). Topological distance is an integer number.

Each part consists of vertices lying outside or inside the core. To distinguish between those two sequences of vertices, we will use so called "negative topological distance" for the vertices lying on the core. It has the same meaning as the positive one but it has a negative sign.

Figure 3 shows topological distances for one part of the polygon.

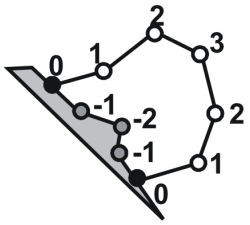

Figure 3: Topological distances of vertices
(White: vertices outside the core, black: intersection vertices, grey: vertices inside the core, light grey: part of the core)

According to the topological distance of vertex v_i, we can compute its "vertex path". Vertex path is a list of coordinates describing the key positions of the vertex during the whole morphing process. The list starts with the coordinates of vertex v_i and ends with its last position (which for the case of disappearing in the core lies on the core). Having only the list of key positions gives us a possibility to choose if the final path will be linear or if the points will be used to compute some kind of curved path.

Vertex path is computed only for such vertices in the part that have positive topological distance. There are many ways how to compute path for a vertex v_i using its topological distance. We propose the so-called perimeter algorithm for this work.

This algorithm is called perimeter, because the points travel along the perimeter – each point travels to its neighbour and then to its "neighbours' neighbour" and so on until they reach the point with "minimal" topological distance (signed distance is considered). The process is outlined in Figure 4. The vertices are numbered according to their topological distance, for our purpose it does not matter that there can be two vertices with the same number. For the polygon in Figure 3, the algorithm consists of four main steps: first, the vertices with the highest topological distance (in this case, there is only one such vertex: 3) travel to their neighbours. Because there is an odd number of vertices, the vertex 3 is duplicated, so there are two vertices (with the same starting position) that travel to vertices with the topological distance of 2. After the vertices 3 reach their neighbours, they both travel to their neighbours' neighbours - vertices with topological distance 1. And so on it goes until all the vertices reach the vertex with topological distance -1.

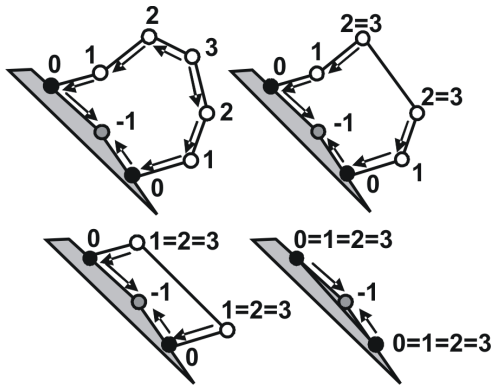

Figure 4: The perimeter algorithm

In Figure 4 we recall two important facts: first and more general is that only the vertices outside the core travel (that results from the fact that the core does not change during the morphing process). Second, if there is an odd number of vertices outside the core, the one with maximum topological distance is duplicated.

The vertex path is used for computation of a time plan.

Time plan represents the used treatment with the vertex path. For a given time $t \in \langle 0, 1 \rangle$, it computes the actual position of the point. Here we describe only the disappearence in the core, the growth from the core can be gained by reversing the time plan.

There are two main possibilities of representing how the points travel during the morphing process. The first one is that all the points are traveling for all the time. It means that the points do not travel with the same velocity, their velocity depends on their topological distance (points with bigger topological distance will have bigger velocity). We will call such variant "constant time".

The second one is that all the points do not travel all the time, only the point(s) with the highest topological distance travel during the whole process. The time amount for the other points depends on their topological distance. This variant will be called "constant velocity".

There are two possible outputs of this algorithm according to a used time plan variant. For the "constant velocity" plan, the result is as was shown in Figure 3: points with topological distance k wait until the points with topological distance $k' > k$ arrive to their position. For the "con-

stant time" plan, all the points are travelling during the whole process. This plan has slightly better results. The difference is shown in Figures 5, 6. In both Figures 5 and 6, there are only the first two steps of the process. Figure 5 shows the constant time variant. In Figure 5a, the vertex number 3 is duplicated and both it an its copy are in the first third of their journey - on the coordinates of vertices 2. But because this is the constant time variant, the vertices 2 travel at the same and they are in the first third of their journey as well (their position is marked by a small circle). Also vertices number 1 travel towards vertices number 0 and they are in the first third of their journey - here their journey consists only of one edge (from 1 to 0).

Figure 5b shows the second step of the process - all the vertices are in the second third of their journey - vertices number 3 are on the coordinates of vertices number 1, vertices number 2 are in the first third of the edge from 1 to 0 and vertices number 1 are in the second third of the same edge.

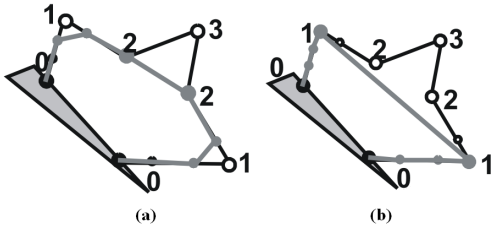

Figure 5: The perimeter algorithm with constant time (Grey: the current polygon and positions of the vertices)

Figure 6 shows the first two steps of constant velocity variant - this is exactly the algorithm shown above (along with the description of the perimeter algorithm).

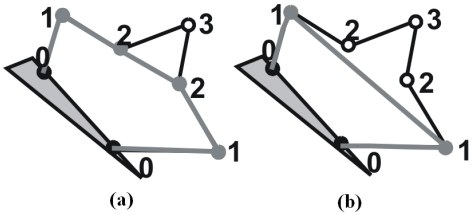

Figure 6: The perimeter algorithm with constant velocity (Grey: The current polygon and positions of the vertices)

Notice that the second step appears to be the same, but the points are on different positions.

4 Experiments

The results of our algorithm[1] were compared with the results of [7] and 2D merging algorithm [3].

[1]Implemented in Microsoft Visual Studio 2005, C#, .NET Framework 2.0. Using configuration Mobile AMD Sempron 3100+, 1800MHz, 512MB RAM

4.1 A rectangle & a spiral polygon

As an example of behaviour of our algorithm, let us present a rectangle and a spiral polygon.

Our algorithm usually gives satisfying results for polygons of such a type, but it depends on how the points are organized. There are two cases shown in Figure 7: in 7a, the polygon contains only points needed to preserve its shape. However in 7b, there is one extra point, which has a significant impact on the output. Let us clarify the reason: For each part of the polygon (either disappearing in the core or growing from the core), there is always one line not lying on the perimeter. This line is connecting the points with the highest topological distance. If the points with the same topological distance lie within the bends of the polygon, the line can never intersect the edges. But if there exists an extra point with only slight impact on the shape of the polygon, this point causes that points at the bends will not have the same topological distance and intersections may occur. Figure 7 shows how this influences the output – if there is one point more, the side containing this point is "one point slower" than the other one, and intersections may occur.

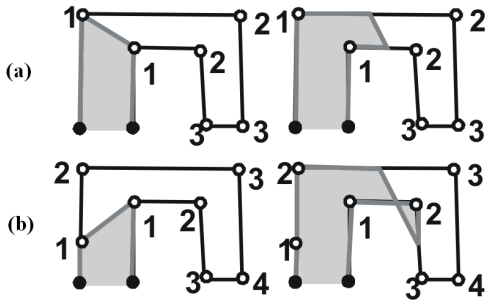

Figure 7: Self-intersection (Bottom: extra point added; grey: the current polygon)

In Figures 8 and 9, there are results of our algorithm used with the constant velocity plan. In Figure 9, one extra point is added, which results in local self-intersections of resulting polygon. In Figures 10 and 11, there are results of our algorithm used with constant time plan.

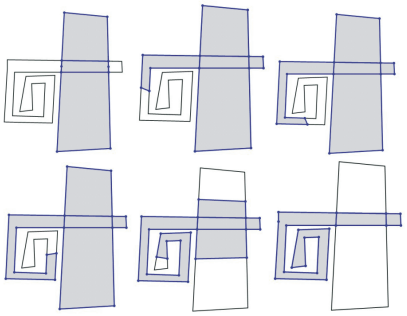

Figure 8: The perimeter algorithm with the const. velocity (Grey: the current polygon, black: source and target polygons)

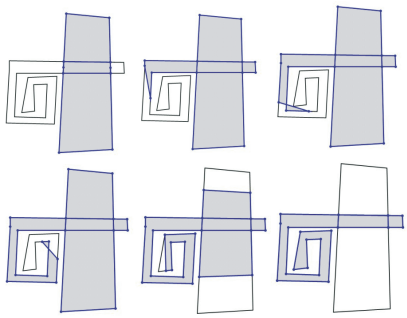

Figure 9: The perimeter algorithm with the const. velocity - an extra point added

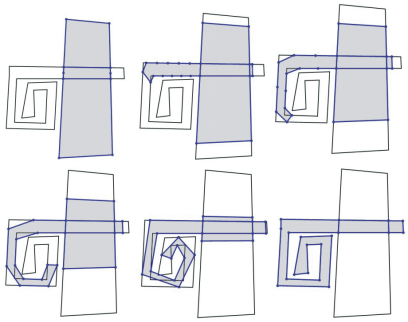

Figure 10: The perimeter algorithm with the const. time

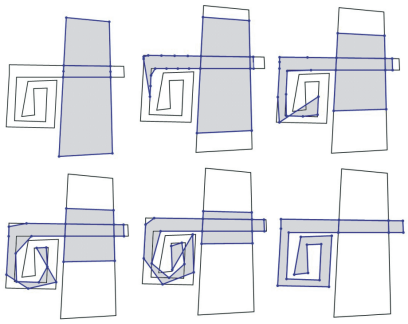

Figure 11: The perimeter algorithm with the const. time - an extra point added

The comparative methods give similar results for both described cases (with and without the extra point), so we show only the case with one extra point (to show they are not influenced) in Figures 12 and 13.

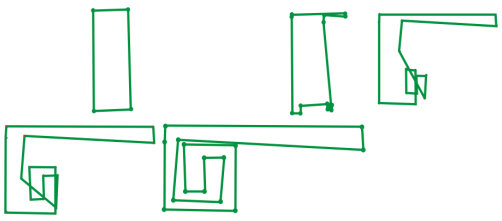

Figure 12: Sederberg's algorithm

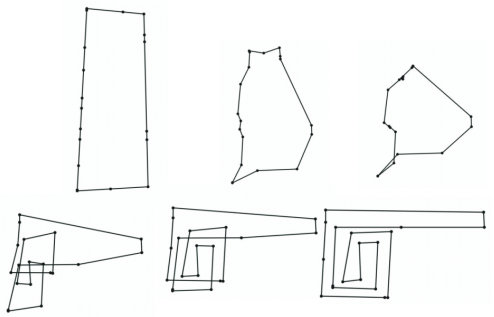

Figure 13: 2D merging algorithm

Both other methods experience self-intersection, result of our method depends on the distribution of the points on polygons. Points that do not influence the polygon shape can be deleted in a preprocessing part. To deal with points that do influence the polygon shape needs to make few changes in the algorithm and so belongs to our future work.

4.2 A cross & a lamp

Our algorithm is suitable not only for polygons of spiral shape, but also for other polygons where one would expect the growth-like process, such as thin non-convex polygons with only a small intersection part. One example of such pair of polygons is "a cross" and "a lamp" (shown in Figures 14-17). The part where the cross is disappearing in the core is still not satisfactory, but we want to improve it in the future by using some more sophisticated algorithm than the perimeter.

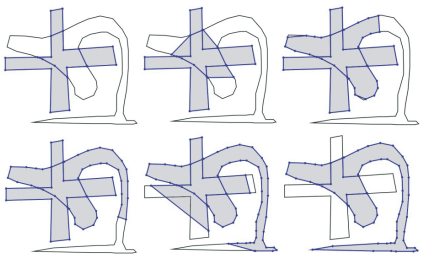

Figure 14: The perimeter algorithm with the const. velocity

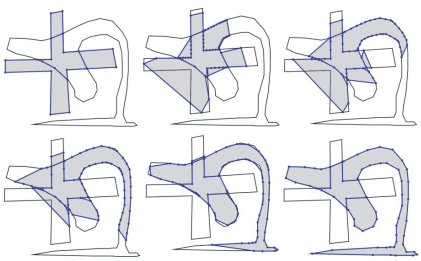

Figure 15: The perimeter algorithm with the const. time

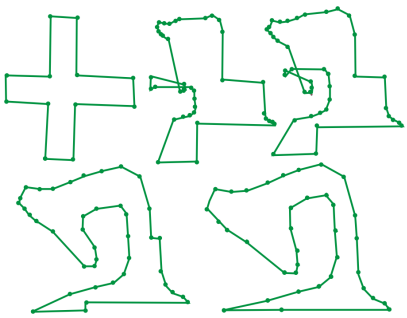

Figure 16: Sederberg's algorithm

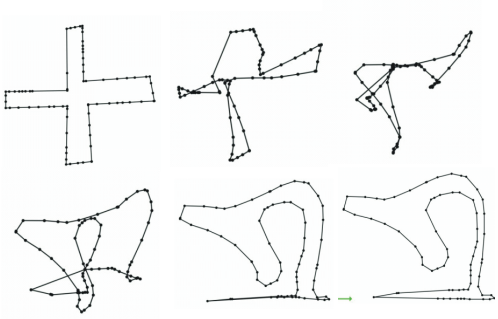

Figure 17: 2D merging algorithm

As we can see, both Sederberg's and 2D merging algorithms experience intersections, while the perimeter algorithm does not.

4.3 A flower & leaves

This example is to show that our algorithm is not very suitable for polygons with serrated edges. The problem is that the algorithm follows only topological distances, and it is not influenced by vertices' distance from the core. That is why we can observe a straight line going up and down within the bloom in Figure 18. Use of the constant time does not deal successfully with this configuration either (see Fig.19).

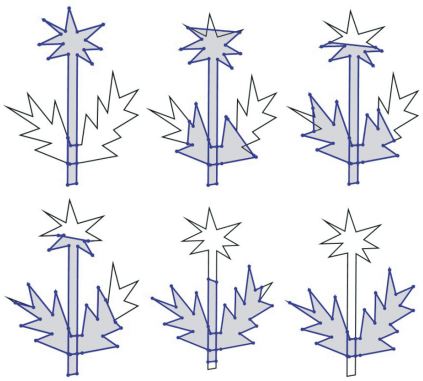

Figure 18: The perimeter algorithm with the const. velocity

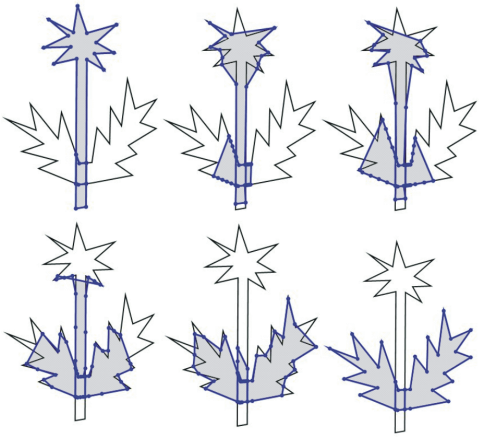

Figure 19: The perimeter algorithm with the const. time

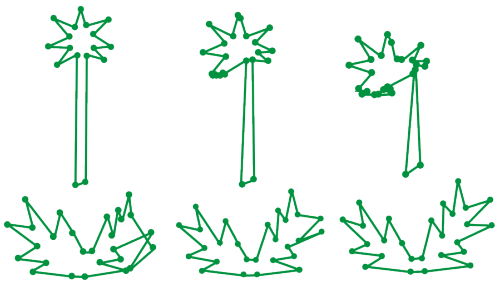

Figure 20: Sederberg's algorithm

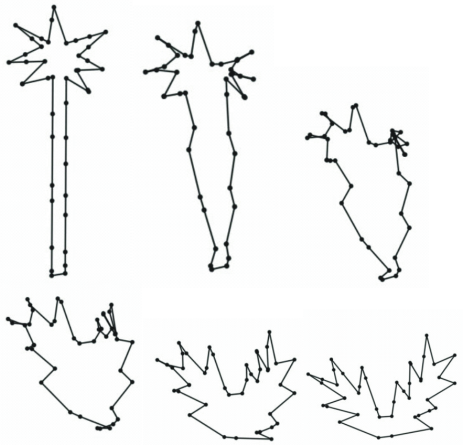

Figure 21: 2D merging algorithm

4.4 An arc & a translated arc

This example shows objects of a similar but translated shape. The Figures 22-25 show the difference between the behaviour of the two already mentioned concepts of morphing. The first one (represented by our algorithm) does not compute the correspondence and depends on the mutual position of the input polygons (Figures 22, 23). The second one does compute the correspondence and so utilizes similarity of the shape of polygons (Figures 24,25).

We cannot say which outputs are better, both approaches are interesting.

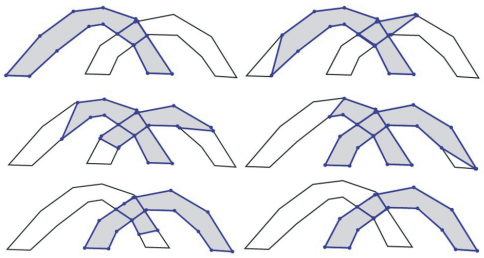

Figure 22: The perimeter algorithm with the const. velocity

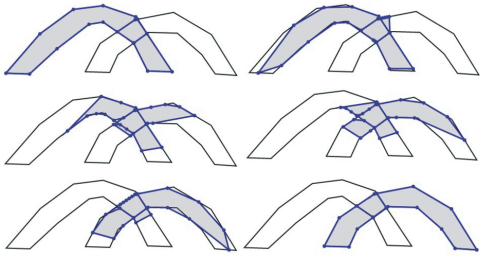

Figure 23: The perimeter algorithm with the const. time

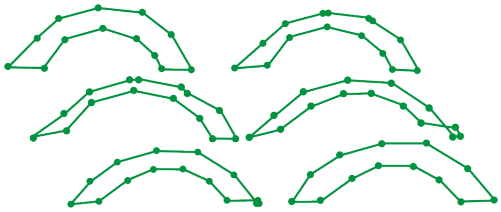

Figure 24: Sederberg's algorithm

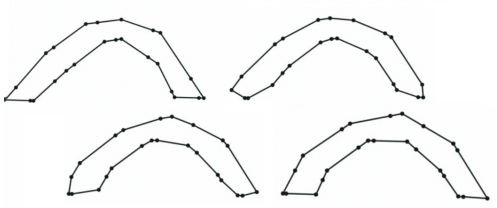

Figure 25: 2D merging algorithm

4.5 Influence of mutual position of polygons

To illustrate that completely different results can be obtained if the mutual position of polygons is changed, we introduce Fig.26 and Fig.27 (compare also to Fig.14). This behaviour can be understood as a disadvantage because our method does not compute the best mutual position, but also as an advantage because many different interesting effects can be achieved.

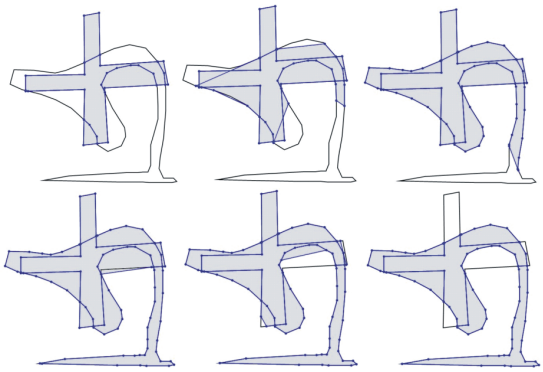

Figure 26: The perimeter algorithm with the const. velocity

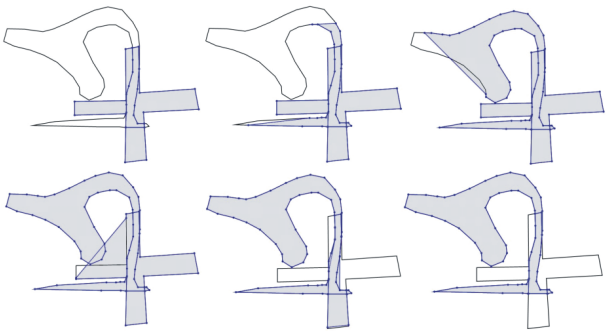

Figure 27: The perimeter algorithm with the const. velocity

5 Conclusion and Future work

We presented a method for polygons based on the idea of their intersection. It produces satisfactory results for polygons that are usually problematic for other methods, such as polygons of a spiral type or a source polygon with a large part far from the target polygon.

The presented method is only the first step in this direction of core-based morphing algorithms. There is a possibility to use some other more sophisticated methods instead of the perimeter algorithm, such as to project points of each part to the core, try other types of the vertex path – not round the perimeter, but through the inner part of the polygon etc. Another option to enhance our algorithm could be adding an information about total pathlength to each vertex. It would then move with a speed inversely proportional to this length. This could probably solve the problem with an extra point (Fig. 9,11).

Another future work area will be to enhance our algorithm to work when the core consists of more or less than one part. We also plan to extend our algorithm to 3D.

Acknowledgement

This work was done in cooperation with Jindřich Parus, to whom I would like to thank for his ideas that gave birth to the algorithm and for his help with the whole work. The same big thanks belong to Dr. I. Kolingerová from the University of West Bohemia, Pilsen, Czech Republic, for help with the paper preparation and advices during the work. I would also like to thank to Jindřich Parus and Pavel Celba for providing their programs that were used to compare our method with others.

References

[1] M. Alexa. Recent advances in mesh morphing. *Computer Graphics Forum*, 21(2):173–197, 2002.

[2] P. Comninos. *Mathematical and computer programming techniques for computer graphics*. Springer, 2006.

[3] J. Gomes, L. Darsa, B. Costa, and L. Velho. *Warping and morphing of graphical objects*. Morgan Kaufmann Publishers, Inc., 1999.

[4] J.R. Kent, W.E. Carlson, and R.E. Parent. Shape transformation for polyhedral objects. *Computer Graphics (SIGGRAPH'92 Proceedings)*, 26:25–34, 1996.

[5] J. Parus. Morphing of meshes. Technical Report No. DCSE/TR-2005-02, University of West Bohemia in Pilsen, April 2005.

[6] T. W. Sederberg, P. Gao, and G. Mu H. Wang. 2-d shape blending: An intrinsic solution to the vertex path problem. *Computer Graphics*, 27:15–18, 1993.

[7] T. W. Sederberg and E. Greenwood. A physically based approach to 2-d shape blending. *ACM SIGGRAPH*, 26:25–34, 1992.

[8] S. K. Semwal and K. Chandrashekhar. Cellular automata for 3d morphing of volume data. *WSCG Conference Proceedings*, pages 195–202, 2005.

[9] M. Shapira and A. Rappoport. Shape blending using the star-skeleton representation. *IEEE Computer Graphics and Applications*, 15:44–50, 1995.

[10] V. Surazhsky and C. Gotsman. Controllable morphing of compatible planar triangulations. *ACM Transactions on Graphics*, 20:203–231, 2001.

[11] V. Surazhsky and C. Gotsman. Intrinsic morphing of compatible triangulations. *International Journal of Shape Modeling*, 9:191–201, 2003.

[12] Y. Zhang. A fuzzy approach to digital image warping. *IEEE Computer Graphics and Applications*, 16(4):34–41, 1996.

[13] http://www.cs.man.ac.uk/ toby/alan/software.