University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Diploma Thesis

Morphing of geometrical objects in boundary representation

Plzeň, 2008

Martina Málková

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky
Akademický rok: 2007/2008

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Martina MÁLKOVÁ**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačová grafika a výpočetní systémy**
Název tématu:	**Morfování geometrických povrchových modelů**

Zásady pro vypracování:

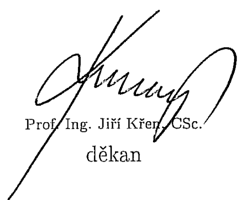
1. Prostudujte nejnovější metody morfování vzniklé od vaší bakalářské práce a získané poznatky zahrňte do další práce.
2. Navrhněte a implementujte 3D algoritmus morfování založený na jádře.
3. Použijte programové vybavení vlastní i vytvořené ing.Parusem pro modelování nějakého konkrétního efektu.
4. Vlastnosti jednotlivých užitých metod a získané výsledky zhodnoťte.

Rozsah grafických prací:	**dle potřeby**
Rozsah pracovní zprávy:	**min. 40 stran původního textu**
Forma zpracování diplomové práce:	**tištěná**

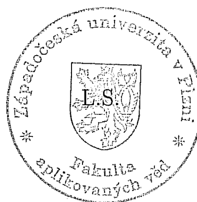
Seznam odborné literatury:

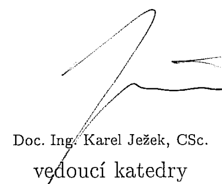
dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Doc. Dr. Ing. Ivana Kolingerová** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**31. srpna 2007**
Termín odevzdání diplomové práce:	**20. května 2008**



Prof. Ing. Jiří Křen, CSc.
děkan





Doc. Ing. Karel Ježek, CSc.
vedoucí katedry

V Plzni dne 11. září 2007

Acknowledgements

At the first place I would like to thank my supervisor doc. Dr. Ing. Ivana Kolingerová for her professional and psychical support and bottomless patience. My great thanks also belong to my family for its support and understanding during my whole studies. And the last but not least I would like to thank my friends, who have never hesitated to help me when I needed.

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Pilsen, 5.5.2008

. . . .
Martina Málková

Abstract

There are many effects that can be achieved by a shape transformation (morphing), and all of them cannot be covered by one algorithm. Most morphing algorithms concentrate on aligning and preserving similar features of the two input objects, which results in worse behaviour for the features that do not exist in the other object or cannot be aligned. Our algorithm concentrates on a totally different effect, effect of growing. The approach is based on the intersection of two objects. During the morphing process, a part of the former object is disappearing in the intersection, while a part of the latter object is growing from it. The described solution does not need user interaction, however, different effects can be achieved by a change of the mutual position of the objects.

Contents

1 Introduction **3**

2 Related terms **4**

2.1 2D Voronoi diagram 4

2.2 Medial axis, medial surface and centerline 4

2.3 Weiler-Atherton's algorithm 6

3 State of the art **8**

3.1 Polygon morphing algorithms 8

3.2 Mesh morphing algorithms 11

3.3 Summary 15

4 Our solution **16**

4.1 General algorithm independently on the space 16

4.2 2D algorithm 18

4.2.1 Perimeter growing 19

4.2.2 Half-line growing 22

4.2.3 Projection growing 24

4.2.4 Merging 26

4.2.5 Improvements 27

4.3 3D algorithm 29

4.3.1 Perimeter growing 30

4.3.2 Projection growing 32

4.3.3 Projection plane 39

4.3.4 Parts with more than one intersection chain 41

4.3.5 Dividing the part into convex or star-shaped parts 41

5 Results **43**

5.1 Results for polygons 43

5.1.1 Parts of a spiral type 43

5.1.2 Convex parts 44

5.1.3 Long and more or less straight parts 45
5.1.4 Summary 46
5.2 Results for meshes 46
5.2.1 Convex parts 46
5.2.2 Parts whose centerline contains only one curve 47
5.2.3 Star-shaped part 48
5.2.4 Non-convex part, having a convex inside and convex outside . 49
5.2.5 Summary 51

6 Conclusion **52**

6.1 Future work 52

A Proceedings of CESCG 2007: A new core-based morphing algorithm for polygons **55**

B User manual **62**

C Implementation **68**

D Other results **70**

1 Introduction

From a general point of view, morphing can be described as a process when one object is continuously changed into another. In nature, morphing process represents a growth of animals or plants. In movies, morphing is one of the tools for special effects, the types of objects used there depend only on the author's imagination. But morphing is not only the whole process (animation), it can be also a way to achieve new shapes or patterns, which is usable in the area of design.

There is no definition of how should one object morph into another. Many effects can be achieved when morphing between two objects, and the "best" morph can be determined only visually. Usually, the decision what is the best morph also depends on the current application - we might need to preserve the most of the appearance of the first object, or to preserve the volume no matter what is the shape, or we might want to achieve some other effect during the morphing process. Therefore each application implements one area of morphing effects and the user has to choose the application according to which effect he wants to achieve. Most applications concentrate on morphing objects that are similar or have common features (e.g. one object is only transformed, both are four-legged animals, one face is smiling and one not) and then try to preserve these features during the morphing process. In our approach, we focused on growing processes. Such processes are common in nature (e.g. a seed grows into a plant), or can be found in movies' special effects (e.g. horns grow from a head of an evil person and a tail grows from his back, hundreds of fingers grow from the body of an alien).

The core idea of our work is to compute an intersection of two objects. The intersection then stays unchanged during the morphing process, while the rest of the first object disappears in it and the rest of the second object grows from it to form the final shape. This idea was first introduced in [18], where morphing between volumetric objects was described. We used their idea of the intersection of two objects to construct an original algorithm for morphing objects given by boundary representation.

During our research, we first concentrated on solving the problem in two dimensions - between arbitrary polygons (the first part of our research has already been published [15]). The research in 2D was done in cooperation with Ing. Jindřich Parus[1], who contributed by some ideas and advice also in 3D. With the experience from 2D, we continued to construct the algorithm in 3D, where we concentrated on objects in boundary representation.

In this text, after an overview of related terms, we will concentrate on other solutions related to our work (Section 3), along with their advantages and disadvantages. In Section 4, the whole algorithm in 2D and consequently in 3D will be described. The results of both 2D and 3D versions of the algorithm will be shown in Section 5. The last section (Section 6) will conclude the whole text, summarising what has been achieved and giving ideas of possible future extensions.

[1]jparus@gmail.com

2 Related terms

A *polygon* is an ordered set of vertices v_i, $i = 0, \ldots, n-1$. An edge e_i of polygon is a line segment with endpoints v_i, v_{i+1}. A *simple polygon* is a polygon whose consecutive edges e_i, e_{i+1} intersect only in the endpoint v_{i+1}. An unclosed sequence of edges is called a *polygon chain.* A *closed polygonal chain* is a polygonal chain, where also p_0 and p_{n-1} are connected by a line segment.

A *polytope* is the generalisation of polygon in two dimensions, polyhedron in three dimensions, and polychoron in four dimensions. A *convex polytope* is defined as the intersection of half-spaces.

A *topological distance* $d(v_i, v_j)$ between vertices v_i and v_j in E^n is the number of edges in the shortest path from v_i to v_j.

An object O is *convex*, when a line segment connecting its two arbitrary vertices lies completely inside or on O. An object P is *star-shaped*, when there exists at least one point p inside P, where a line segment connecting p with an arbitrary vertex of P lies completely inside or on P.

According to [3], a *topology* of an object refers to the vertex/edge/face network. An object is *Euler-valid* if its topology fulfils the formula $V - E + F = 2 - 2G$, where V are the vertices of the object, E edges, F faces and G is the number of passages through the object (*genus*).

2.1 2D Voronoi diagram

Voronoi diagram was first introduced in [22]. In the plane, the Voronoi diagram of a set of points S is the partition of the plane which associates a region $V(p)$ with each point p from S in such a way that all points in $V(p)$ are closer to p than to any other point in S.

The region $V(p)$ is the interior of a (in some cases unbounded) convex polytope called the Voronoi cell for p. The Voronoi diagram is then the set of such polytopes, which subdivides the whole plane. An example of a Voronoi diagram is in Figure 2.1.

2.2 Medial axis, medial surface and centerline

As is described in [7], the medial axis of a closed curve in 2D is a set of points that have at least two closest points on the boundary, i.e., the set of centres of circles which touch the curve at more than one point. The medial axis in 2D consists of a set of curves. The analogy in 3D - set of centres of spheres which touch the surface at more than one point - produces so called *medial surface.* Figure 2.2 shows examples of the medial axis in 2D and the medial surface in 3D.

The medial axis and the medial surface are also known as the *skeleton.* The main disadvantage of the medial axis is that small changes in the shape can produce large

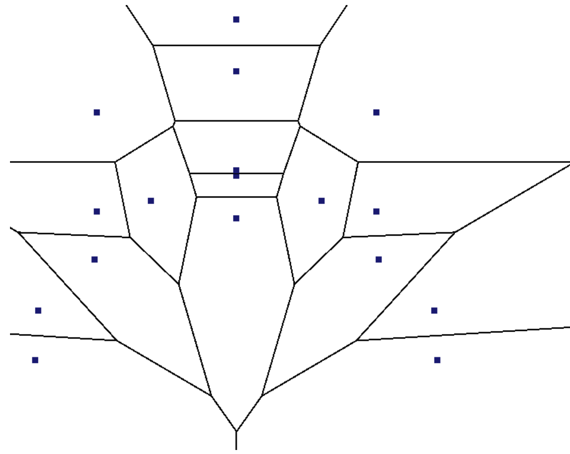


Figure 2.1: Voronoi diagram of a point set in 2D

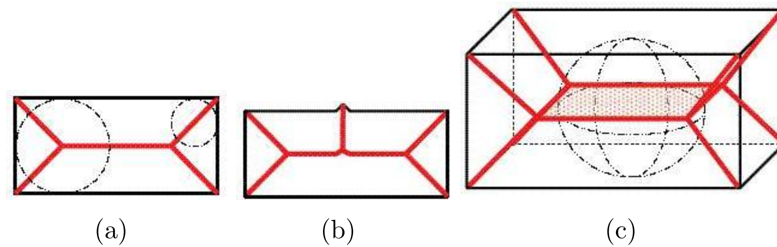


Figure 2.2: (a,c) The medial axis in 2D and (b) the medial surface in 3D, a few examples of inscribed circles and a sphere (from [7])

changes in its medial axis (Figure 2.2c). Also, as the medial surface consists not only of the set of curves, but also surfaces, some applications, such as virtual navigation, need it to be simplified to contain only curves. Such simplified representation of the medial surface, consisting only of curves, is called *a centerline* or *a curve skeleton*. Examples of centerlines of 3D objects are shown in Figure 2.3.

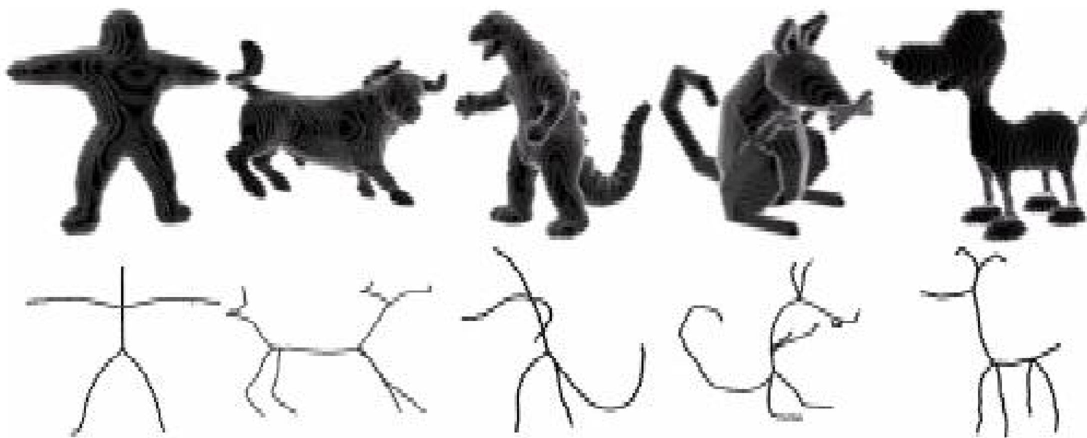


Figure 2.3: Examples of centerlines of different 3D objects (from [7])

2.3 Weiler-Atherton's algorithm

Weiler-Atherton's algorithm [23] is used to compute the intersection of simple polygons P and Q. The algorithm is described in Figure 2.4 An example of the process is shown in Figure 2.5. The polygons P, Q should have the same orientation, in our case we will assume a clockwise orientation. Two lists L_P,L_Q are created from the vertices of polygons P, Q (Figure 2.5a). The intersections of the polygons are computed and inserted into the lists at the proper positions, the lists are linked at the intersections (represented by an arrow at i_0); the list of inbound intersections I_P with respect to P is generated (Figure 2.5b). The first intersection i_0 from the list I_P is taken and found in L_P (marked red in Figure 2.5c). The search starts from i_0 to the right, switching lists at each intersection (green arrows), and ends when i_0 is reached again (the result is filled green). Because $I_P \neq \emptyset$, the intersection i_3 is taken and the process is repeated. Because I_P is empty after that, there are two disjoint parts in the result.

Input: A polygon Q, defined by vertices $q_0, \ldots, q_{m-1}$, a polygon P, defined by vertices $p_0, \ldots, p_{n-1}$.

Output: The intersection of P and Q: $L = (Q \cap P)$.

The algorithm:

1. Create lists L_Q,L_P from the vertices of the polygons P,Q.
2. Label the items of L_Q,L_P whether they are inside or outside the other polygon.
3. Compute all intersections of the polygons and insert them into both lists to the proper positions (between the vertices defining the edge they are lying at), mutually linking the lists at the intersections.
4. If there are no intersections, there are only three possible solutions by testing the position of an arbitrary vertex v of P and an arbitrary vertex w of Q:
   - v is inside Q (whole P is inside Q) - return P.
   - w is inside P (whole Q is inside P) - return Q.
   - v is outside Q and w is outside P ($P \cap Q = \emptyset$) - return $\emptyset$.
5. Generate a list of inbound intersections $I_P = (i_0, \ldots)$ with respect to P. Create a list $L = \emptyset$, which will contain the resulting intersection.
6. Take the first intersection i_i from the list, set $I_P = I_P \backslash i_i$ and find i_i in L_P. Put i_i at the end of the list L.
7. Continue from i_i to the right through the list, and put all the visited vertices to L. Each time the intersection vertex is reached, it is removed from I, and its link is followed to the other list, until the intersection vertex i_i is reached.
8. If $I_P = \emptyset$, the algorithm is finished - return L. Otherwise, continue by 6.

Figure 2.4: The Weiler-Atherton algorithm.

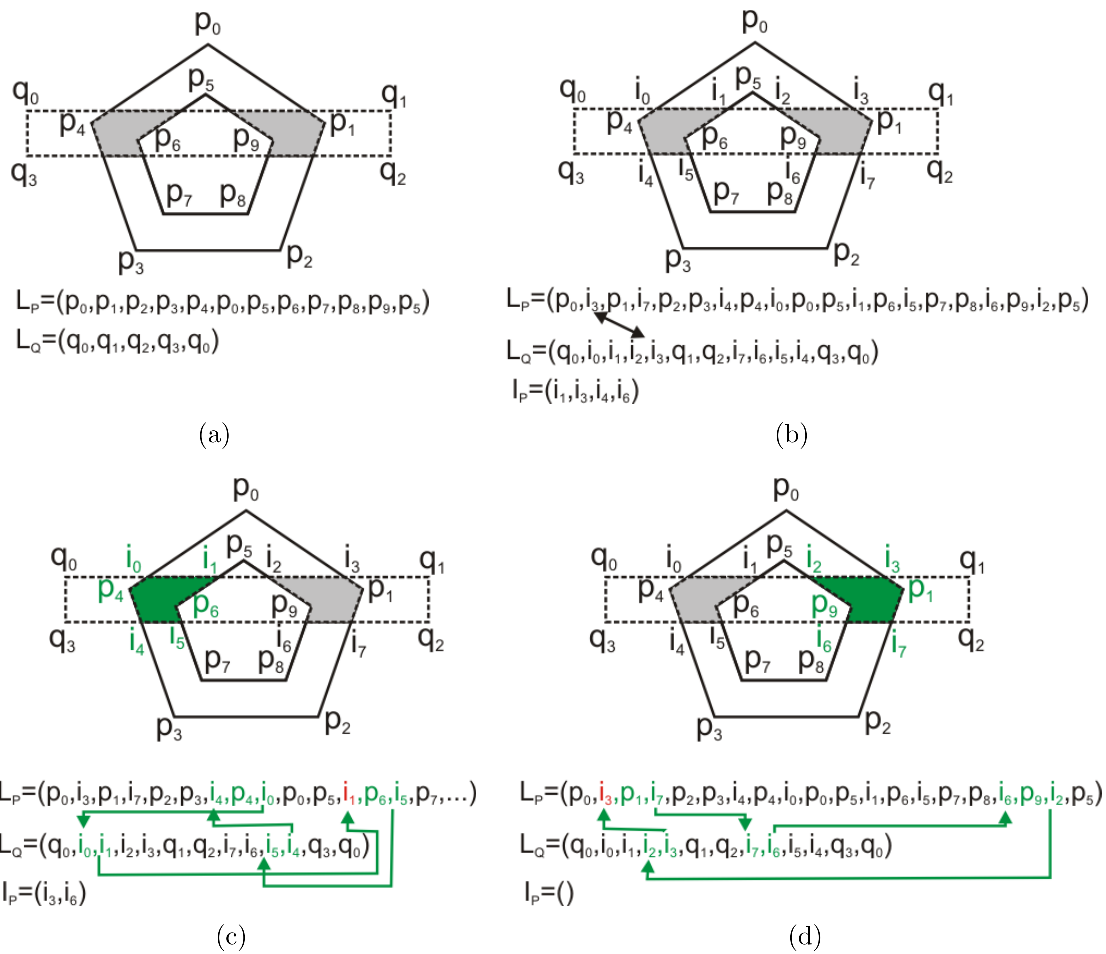


Figure 2.5: An example of clipping by Weiler-Atherton's algorithm.

3 State of the art

Because we concentrated both on 2D and 3D versions of our algorithm, algorithms for morphing polygons will be described here as well as those for morphing meshes.

Most of the algorithms that will be described here are "correspondence-based", meaning they first find a correspondence between vertices of the source and the target polygon. They usually need to add vertices to both source and target polygon to make the best correspondence (according to their requirements). When the correspondence is established, the trajectories between corresponding vertices need to be found. This step can be very complicated, however, most solutions use a simple linear interpolation here. As is discussed in [9], this simple choice has some disadvantages when computing rotational morphing. The problem is shown in Figure 3.1 on morphing between two line segments, both of them of the same length, but one of them rotated. If we use the linear interpolation for computing the trajectory of corresponding vertices, the line segments shorten during the morphing process, which is something we do not expect.

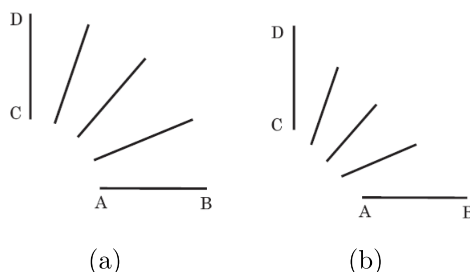


Figure 3.1: An expected morphing sequence (a) and a morphing sequence using linear interpolation (b), from [9]

3.1 Polygon morphing algorithms

The first approach worth mentioning is ***A physically based approach to 2-D shape blending*** [**17**] by **Sederberg et al.**. The polygon edges are modelled as wires with some material properties (modulus of elasticity, stretching stiffness constant). To get the resulting polygon, we need to bend or stretch the wire. The goal is to minimise the amount of work needed to create the target polygon.

The stretching work can be influenced by the user, who can define a constant representing the material and cross sectional area of the wire. Because the wire is not only stretched in our case, but also compressed, it can happen that the target length is zero (edge collapsed into a vertex) - such a case is penalized to prevent it, the amount of penalization can be also altered by the user.

In the case of the bending work, two conditions need to be avoided in the bending of the shape: first, the angle between two parts of the wire should never be zero, and second, the angle should change monotonically during the time. Such cases are penalized by other two constants.

After the correspondence is found, the trajectories between the corresponding vertices are computed by a simple linear interpolation of their coordinates. The algorithm duplicates some vertices during the correspondence establishing (when an edge of the target polygon collapses to a single vertex of the source polygon, such vertex needs to be duplicated to be able to setup the one-to-one correspondence). One of the negatives of this algorithm is that it only duplicates vertices and does not add vertices anywhere on the edges. Therefore, it is dependent on the distribution of the vertices. According to the results obtained, 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 ***2-D Shape Blending: An Intrinsic Solution to the Vertex Path Problem*** **[16] Sederberg et al.** describe a computation of trajectories, where *intrinsic parameters* (e.g. edge lengths or internal angles) are interpolated rather than the vertex positions directly. The polygons are converted to the so-called edge-angle representation [9]. In the edge-angle representation a polygon is described by one fixed vertex, one edge incident to the fixed vertex, the length of each edge and the internal angle of each vertex. The advantage of this representation is that except for the fixed vertex and edge it is invariant to rigid transformation. The absolute vertex coordinates are extracted from interpolated intrinsic parameters. This interpolation scheme avoids edge collapsing and non-monotonic angle changes. This technique was used for generating in-betweens for the animation based on key frames. The concept of interpolation of intrinsic parameters was also further used for morphing of planar triangulations in [20, 21]. The intrinsic interpolation avoids local self-intersections; however, it does not avoid global self-intersections, which may appear especially in the case of highly dissimilar and complicated shapes.

Another optimization-based algorithm was presented by **Zhang** in ***A Fuzzy Approach to Digital Image Warping*** **[24]**. It uses a similarity function to obtain the vertex correspondence.

In ***Shape Blending Using the Star-Skeleton Representation*** [19] **Shapira and Rappoport** outline a morphing using the star skeleton. The algorithm first decomposes the source and the target polygons into star-shaped polygons. Then it constructs the skeletons of the decompositions. The skeleton is a planar graph which joins star-points of neighboring star-shaped polygons, i.e., it is a dual graph to the star-shaped decomposition. Important is that skeletons of the source and the target polygon have to be isomorphic, which requires an isomorphic star-shaped decomposition. Then, the interior and the boundary of the polygon can be expressed relatively to the skeleton. During the morphing, the skeletons are interpolated and the intermediate shapes are reconstructed from the interpolated skeletons. The difference between this approach and previous approaches [16, 17] is that this approach takes into consideration also the interior of the polygon and not only the boundary. The problem of this approach is that it relies on an isomorphic star-shaped decomposition which might be difficult to compute, especially in the case of dissimilar shapes. This technique also does not include finding correspondence between

vertices, so it needs to be found by using some other method or manually specified.

Alexa et al. [4] introduced an approach called ***As-Rigid-As-Possible Shape Interpolation***. The basic idea is to compute a compatible triangulation of input polygons. The compatible triangulation is a dissection of the source and the target polygon so that the triangulations are isomorphic, i.e., we have one-to-one correspondence between triangles in the source triangulation and triangles in the target triangulation. Then, for each triangle an affine transformation which transforms a source triangle to the target triangle is computed. By interpolation of the affine transformation a source triangle is transformed to the target triangle. The transformation of one triangle influences the transformation of adjacent triangles as well; therefore, the transformations for the whole triangulations are computed in the least square sense. Similar approaches were also described by Surazhsky and Gotsman in [20, 21]. A challenging issue of approaches based on compatible triangulation is an extension of this idea in 3D, where it requires computing compatible tetrahedronization of input 3D objects. This approach guarantees that the in-between shapes do not self-intersect. The main problem here is the computation of isomorphic triangulation of the input shapes, because the quality of the triangulation influences the quality of the resulting morphing animation.

Another approach is called ***2D merging*** [9]. It is a "generalization" of an algorithm which was originally developed for 3D meshes, e.g., [2, 13]. 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.

In ***Warp-guided object-space morphing*** [5] **Carmel and Cohen-Or** showed an algorithm which combines a 2D merging and a polygon evolution. A user first specifies several anchor points to define a correspondence between features. Using the anchor points a warp function is computed. The warp function warps anchor points of a source polygon towards anchor points of a target polygon. Once the source polygon is warped a polygon evolution technique is used to evolve the source and the target polygon to a convex shape. The convex shapes are projected to circles. Using the merging technique the circles are merged and new vertices are projected back to the original polygons. This technique was extended to 3D (see the next section).

Johnstone and Wu [12] described an approach to morph two separate polygons into one in ***Morphing two polygons into one***. The 2-to-1 morphing is a fundamental case in morphing between different numbers of polygons. The basic idea is to merge the two polygons into one and then use some 1-to-1 polygon morphing technique to morph between the merged polygon and a target polygon. The key step is the merging. During the merging the two polygons are morphed towards each other until they meet at one point. Then a curve evolution technique is used to morph the two polygons connected in some point into a more natural shape which

is later morphed towards the target shape.

3.2 Mesh morphing algorithms

In 3D, the input objects for morphing can be implicit, volumetric or in boundary representation. Our work deals with objects in boundary representation, therefore we will discuss only one work from the field of volumetric objects, the one that gave birth to the core idea of our algorithm.

In ***Cellular Automata for 3D Morphing of Volume Data*** [18], **Semwal et al.** 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.

Further techniques deal with morphing of meshes, more specifically triangular meshes. However, we can use those techniques for polygonal meshes as well if we triangulate them first.

As Alexa describes in his overview of mesh morphing algorithms [3], most of these methods work in three following steps:

1. **Establish a correspondence between the meshes.** Decide which vertex of the mesh M_0 corresponds to which one of the mesh M_1. This is usually the crucial step of the whole process.

2. **Generating a *supermesh.*** A supermesh is a mesh that represents both M_0 and M_1.

3. **Creating paths $V(t), t \in < 0, 1 >$ for the vertices.** Usually the algorithms use an interpolation of corresponding vertices, mostly a simple linear interpolation.

Kent et al. propose in ***Shape transformation for polyhedral objects***[13] an algorithm that uses both the topology and the geometry of the input objects. First, both objects are projected onto a unit sphere. The vertex to vertex correspondence is established by merging the topologies of the input objects. The merging process is done by clipping the projected faces of one model to the projected faces of the other. The paths for the corresponding vertices are created by either a linear interpolation, or using a Hermite spline with its tangent vectors equal to the vertex normals.

The main problem discussed in their article is the projection onto the unit sphere. Several methods for the projection were proposed, depending on the type of the input objects. For the star-shaped objects, the center point (arbitrary point from the kernel of the object) of the object is found and then the vertices are moved to the surface of the sphere in the direction of a vector defined by their position and the position of the center point. The convex objects are projected in the same manner, only the center point is an arbitrary interior point of the object. Another type of

objects are so-called *objects of revolution*. Such objects consist of a set of planar contours arranged at angular increments around an axis (axis of revolution), e.g., a glass constructed by rotating a curve around its axis. The contours are projected onto a longitudinal arc of the sphere by several methods, where among the best is the method of Ekoule [8]. Next type of objects, *extruded objects*, are created by moving a planar polygon along the straight line. The ends of the object are capped by two copies of the polygon. Projecting such object is done by mapping the two caps to its convex hull by Ekoule's method and then projecting the resulting object in the same way as were the convex objects.

Another approach for the projection was to treat the surface model as a flexible object, and inflate the object with air until it is convex. To ensure that the simulation will produce the convex model, the vertices already lying on the convex hull were fixed. The convex model was again projected to the sphere.

The methods of projection are discussed for the most of the genus-0 objects, the authors only suggest how to project the other types of objects (replacing a sphere by a representative manifold, or cutting the objects).

Alexa uses the idea of Kent et al. and presents another correspondence-based algorithm for morphing polyhedra in his ***Merging polyhedral shapes with scattered features*** [2]. First, the polyhedron is triangulated. Then the approximation of the smallest enclosing sphere (*a circumsphere*) of the model is computed, the model is transformed such that the circumsphere is transformed to a unit sphere. Then the spherical projection is examined. Because the method is designed for all genus-0 meshes, there can be overlapping edges (*foldovers*) in the projected result. To remove the foldovers, the relaxation process is introduced. The relaxation works iteratively, moving in each step each vertex to the center of its neighbors' positions in the previous step. Some vertices on the sphere need to be fixed to avoid the vertices converge to one position. Such vertices are called *anchors*. At least four anchors are needed (in case of three anchors the vertices would converge into the triangle, as is shown in Figure 3.2a). Even with four anchors the embedding might collapse (see Figure 3.2b). Also, because the anchors have fixed positions, they

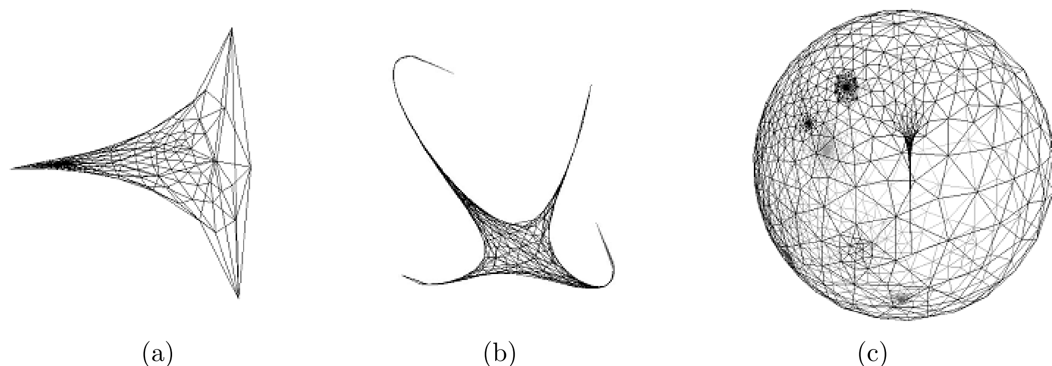


Figure 3.2: Problems of the sphere embedding: (a) collapsed embedding fixed with three vertices (b) collapsed embedding fixed with four vertices (c) foldovers (from [2])

can cause the foldovers themselves. Their solution is simple: As anchor vertices, they choose a random regular tetrahedron with vertices on the unit sphere. Then

they perform the relaxation until the largest movement of any vertex in one step is smaller than a predefined constant. If the relaxation collapsed, they move back to the original mesh and choose a different tetrahedron. If not, they fix the vertices diametric to the tetrahedron used and relax again to remove the possible foldovers made by the original tetrahedron's vertices. They switch those two tetrahedra and perform relaxation until there are no foldovers.

If the user has specified any vertex correspondence, the embeddings are deformed so that the corresponding vertices have the same position on the sphere.

The resulting embeddings are merged. All edge intersections are found (edges are here the shortest path between two points on the sphere), new vertices are inserted at their positions and the corresponding edges are cut. The result of this process is a merged mesh (a supermesh), which is not necessarily a triangle mesh, there can be non-triangle (but still convex) faces. Also, it is not guaranteed that more than three points lie on the same plane, so the supermesh needs to be triangulated after the merging.

The supermesh is deformed to have the shape of the source (and equivalently target) mesh by setting its vertex positions. The vertices of the source mesh remain the same, but as the vertices of the target mesh do not exist on the source mesh, their positions are computed by using the barycentric coordinates - the barycentric coordinates of the vertex v in the triangle v'_1, v'_2, v'_3 in the supermesh are computed and used to find the position of v in the triangle v_1, v_2, v_3 in the original object. Also we need to compute the position of the vertices that were created due to the edge intersection. Each such vertex lied at the intersection of two edges - one of the source and one of the target mesh. When deforming the supermesh to have the shape of the source mesh, we use the source edge, and compute the barycentric coordinates with respect to the vertices defining this edge. And again, the barycentric coordinates are used to find the position of the vertex in the source mesh.

Ahn et al. try to enhance the correspondence-based morphing by decreasing the number of vertices of the supermesh in **Connectivity transformation for mesh metamorphosis** [1]. They use a spherical embedding from Alexa's approach [2] to find the correspondence between M_0 and M_1. both M_0 and M_1 are mapped onto the unit sphere. M'_0 (and similarly M'_1) is constructed by incrementally mapping each vertex v_1 of M_1 onto the surface of M_0: first, we find the face f_0 that contains the mapped position of v_1. Then, v_1 is added to M_0 and connected to the three vertices of f_0 (see Figure 3.3b). Then they swap some of the created edges (not the original meshes of M_0) to reduce the difference between M'_0 and M_1 (Figure 3.3c).

Then, the sequence of connectivity transformations between M'_0 and M'_1 is computed by using an adaptation of Hanke and Ottmann's algorithm [10]: For each edge of M'_0 and M'_1, they check if there exist the corresponding edge on the opposite mesh, and if not, they compute an error (based on Euclidean distance) that occurs if they swap the edge to get the correct position. In such a way, they build the priority queue of edge swaps, sorted according to the computed errors. Because some edge swaps are dependent on the other ones, they construct the transformation dependency graph to perform all swaps in the shortest possible time. After the de-

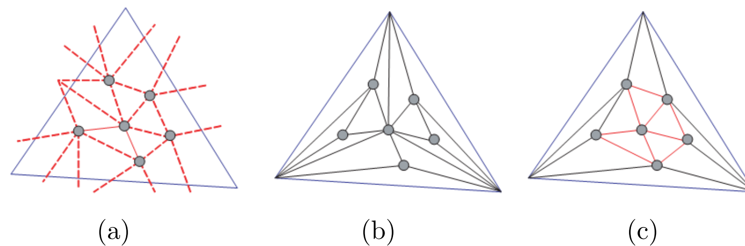


Figure 3.3: Mapping the target vertices onto the source mesh: (a) original configuration of target vertices mapped onto the source triangle (b) result of simple embedding (c) enhanced result after edge swaps (from [1])

pendency graph construction, they compute the exact time portions for each swap. The resulting in-between meshes are constructed by transforming the vertices of the supermesh according to the vertex-to-vertex correspondence established at the beginning, and incrementally swapping the edges according to the plan established by the dependency graph. Each swap is realized by a *geomorph* [11] to make it smooth: A vertex is inserted at the intersection of the two edge positions, and it moves towards one of the vertices of the target edges. When it reaches the vertex's position, it is removed.

Because of the used spherical embedding, the approach can be used only for genus-0 objects. Another disadvantage is the need of computations during the creation of the in-between meshes, which slows down the resulting animation. On the other side, the resulting meshes contain much smaller number of vertices in comparison to other approaches using a fixed connectivity. The visual results are claimed to be similar to Alexa's approach.

Cohen-Or et al. describe a non-correspondence based approach in their ***Three-dimensional distance field metamorphosis*** [6]. Their method needs the user to define corresponding control points (anchor points) on the input objects first (their number depends on their complexity). Then they use the corresponding points to define such warp function $\{W_t\}_{t=[0,1]}$ that $W_1(M_0)$ approximates M_1 as well as possible. The warp function consists of a rigid (rotation, translation) and elastic transformation of M_0. Then it generates a signed 3D distance field by rasterizing the warped object into a binary discrete volumetric representation and converting it into a distance field by a method presented in [14]. Both M_0 and M_1 are represented as discrete distance field (DF) volumes, and the intermediate object (supermesh) is constructed by generating its DF-volume and extracting its surface. The quality of the resulting morph highly depends on a proper warp - if the corresponding points of the two objects are correctly aligned by the warp, it produces the expected results. Otherwise, it may produce results that are far away from the expected ones, sometimes containing parts that unexpectedly disappear and reappear. Also the creation of volumetric representation can consume a large storage space for meshes with a large number of triangles. Then the method requires the object to be simplificated before it is converted into a distance field. The main benefit of this method is that it does not require the input objects to be of the same topological genus.

3.3 Summary

Unlike for morphing images, there are no rules how to measure the quality of the resulting morph for meshes. The criteria are dependent on the expectations on the user. However, there are some non-written criteria that the authors of the presented articles probably tried to follow. First, the in-between shapes should not contain self-intersecting edges. Second, the common features of the input objects (e.g. a head, legs, a tail) should not change during the morphing process and no other features should be introduced (no one expects a rib growing out of a dog's back when morphing it to a horse).

Because all the algorithms presented focused on the overall plausibility of the resulting morph as was described, and not on the special effect of growing, the comparison with these algorithms cannot be open-minded. If we compare such algorithm with ours for an object, where we expect growing or disappearing of some part, our algorithm designed for such effect should be better than the one designed to follow the overall quality of the morph. On the other hand, if we compare the results for two input objects that are about the same but only transformed, our algorithm could produce different than desired results, because no growth or disappearing is expected, and we put all our efforts on this effect, while repressing the need of the common features preservation. However, we will compare our algorithm with a few others in Section 5 to see how exactly the use of our algorithm achieves the growing effect in comparison with the other algorithms that were not designed for it.

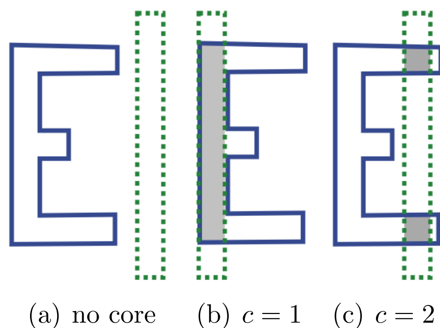
4 Our solution

First (Section 4.1), we will describe the general structure of our algorithm independently on the space. In Section 4.2, the 2D algorithm will be described in detail, and in Section 4.3, the 3D algorithm will be involved from the knowledge from 2D. The general idea of the 2D algorithm and one concrete method (*Perimeter growing*) was already published in our paper [15].

4.1 General algorithm independently on the space

Brief description of the algorithm is as follows: First, the intersection of both input objects is computed. This intersection is considered a morph base - part which does not change during the morphing process. The parts of the input objects which are not in the intersection will either grow up or disappear in it. For each vertex of each part, so-called topological distance is computed, representing its distance from the intersection. The last step is to compute trajectories for those particular vertices, which are not part of the core, according to their topological distance.

Now, let us describe the algorithm in detail. As the input, we have two objects A, B. We compute an intersection $C = A \cap B$. In the further text, we will call such an intersection *a core*. The core can consist of $C = (0, \ldots, c-1)$ disjoint parts (see Figure 4.1, shown in 2D for simplicity). Because our algorithm is dependent on the existence of core, it cannot solve the case when $C = \emptyset$. It is designed for the case when $c = 1$. However, we can solve the problem of multiple parts by choosing one representative part as a core. Therefore, let us suppose that the core consists of only one part.

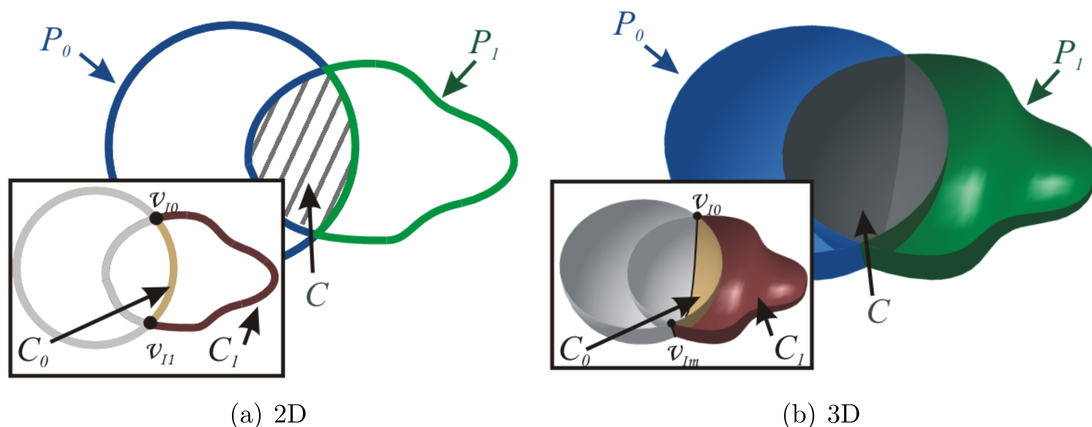


(a) no core (b) $c = 1$ (c) $c = 2$

Figure 4.1: Different number of core parts (from [15])
(solid line: source polygon, dotted: target polygon, grey: core)

When the core C is computed, we compute the difference $P = A - B$, which will disappear in the core, and the difference $Q = B - A$, which will grow out of the core. In the following description, we will concentrate only on computing the disappearing of one part P_i, $P = \bigcup P_i$. The other parts from P are computed equivalently, as well as the parts Q, $Q = \bigcup Q_j$. We achieve the growing of part Q_i by just reversing the time.

Because we work with objects in a boundary representation, the computed core and parts are defined by a set of vertices and edges: $C = (V_C, E_C)$, $P_i = (V_{Pi}, E_{Pi})$. We can divide the vertices and edges of P_i into two sets C_{in}, C_{out}, where C_{in} consists of vertices and edges common for the core C and the part P_i ($V_{Cin} = V_C \cap V_{Pi}$, $E_{Cin} = E_C \cap E_{Pi}$), C_{out} is the part of P_i which remains after removing C_{in} from P_i ($V_{Cout} = V_{Pi} \backslash V_{Cin}$, $E_{Cout} = E_{Pi} \backslash E_{Cin}$). Intersection vertices v_{Ii} are such vertices of C_{in} that lie at its "edge", meaning at least one of their neighbors belongs to C_{out}. By morphing C_{out} to C_{in} we achieve the effect of disappearing of the part P_i in the core C. Figure 4.2 shows the discussed terms, note that the fill is used to distinguish between the terms, not to denote volumetric objects. Also notice that in 2D, there are two intersection vertices, but in 3D, there are n intersection vertices. Therefore we introduce an *intersection chain*, which is a closed polygonal chain, whose vertices are the intersection vertices and its edges are such edges of C_{in} that connect the intersection vertices. Part of the intersection chain is sketched in Figure 4.2b by a solid black line between v_{I0} and v_{Im}.



(a) 2D (b) 3D

Figure 4.2: General terms: Core C, part $P_i = C_{out} \cup C_{in}$, intersection vertices v_{Ii}.

The morphing between C_{in} and C_{out} will be described using a vertex path for each vertex of C_{out} excluding the intersection vertices. The vertex path of a vertex v_i is a list of couples (t_j, p_j), $j = 0, .., l-1$, where t_j is a time and p_j is a position of the vertex v_i at the time t_j, and l is the number of the elements in the vertex paths, its value depends on the specific algorithm. A vertex path is usually defined on a canonical time interval $[0, 1]$. The vertex path has at least two elements, i.e., the initial position of the vertex at the time $t = 0$ and a final position of the vertex at the time $t = 1$. A movement of a vertex is obtained by computing intermediate positions of a vertex. The intermediate positions are computed by interpolating the position values p_j within the vertex path. We can use an arbitrary interpolation technique, e.g., a piecewise linear interpolation or cubic spline interpolation. Once a vertex path representing the growing process is computed, it can be interpreted in a reversed order to represent the growing process.

The computation of a vertex path depends on the specific method of our algorithm. Most of the methods use a concept of a topological distance, which is computed with respect to the intersection vertices. The concrete computation of

the topological distance depends on the space, so it will be discussed later.

As already told, the vertex paths are computed only for the vertices of C_{out}. That is because the final object (supermesh) will contain only the vertices of C_{out} of each part, the intersection vertices and sometimes some parts of the core. The reason why we cannot just take the whole parts and the core and produce the final morphing sequence is that the vertices of C_{in} rest at their positions during the time, while the vertices of C_{out} change their positions (travel towards their corresponding vertices in C_{in}). During this change, some vertices of C_{out} can cross an edge of C_{in} - and at that time, the vertices and edges that were inside and so were not visible, are now visible producing self-intersections (see Figure 4.3).

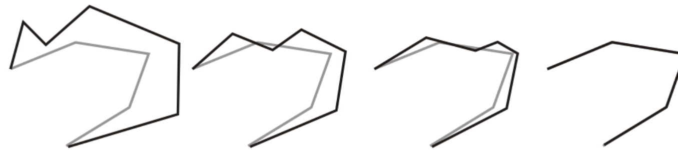
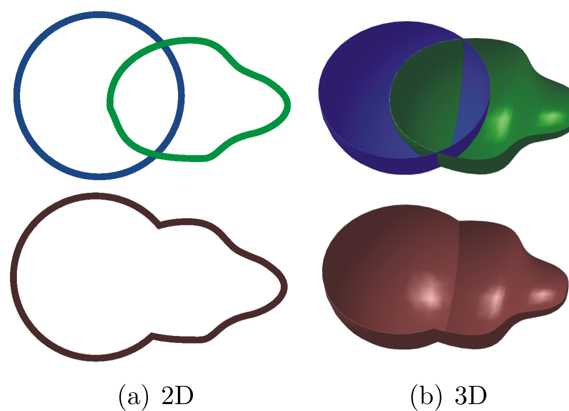


Figure 4.3: Not removing C_{in} may result in unwanted self-intersections: C_{in} (grey), C_{out} (black)

The merging process is shown in Figure 4.4. The final object consists of vertices of only C_{out} from each part and the intersection vertices. Sometimes it can contain parts of the core - it happens when the input objects A and B share some edges and vertices. The merging algorithm is also space dependent, so it will be described in the corresponding sections.



(a) 2D (b) 3D

Figure 4.4: Merging (top: the input objects, bottom: the merged result)

4.2 2D algorithm

In 2D, the input objects A, B are simple polygons. A part P_i consists of two polygon chains C_{in}, C_{out} (described in the previous section). The polygon chains C_{in}, C_{out} are separated by intersection vertices v_{I0}, v_{I1} (Figure 4.2a). An intersection vertex lies in the intersection of the input polygons A and B. If $P_i \neq A$ and $P_i \neq B$ then P_i has two intersection vertices. By morphing the polygon chain C_{out} to the polygon

chain C_{in} we achieve the effect of disappearing of the part P_i in the core C. Hereby, we decompose the polygon morphing problem into several polygon chain morphing problems.

The topological distance $d(v_i, v_j)$ between vertices v_i and v_j is the minimal number of edges on the polygon between v_i and v_j. We compute the topological distance with respect to the intersection vertices. Because there are always two intersection vertices, we use the minimal topological distance $d_{min}(v_i) = min(d(v_i, v_{I0}), d(v_i, v_{I1}))$. Intuitively, the topological distance establishes an order in which the vertices will grow in order to form the whole part. The vertices with a smaller topological distance will finish earlier than vertices with larger topological distance. This avoids self-intersections during the growing process. To distinguish between topological distances of vertices lying on the polygon chains C_{in} and C_{out}, we add the negative sign to the vertices lying on the polygon chain C_{in}. Then d_{min}, d_{max} are minimal and maximal topological distances of the part P_i.

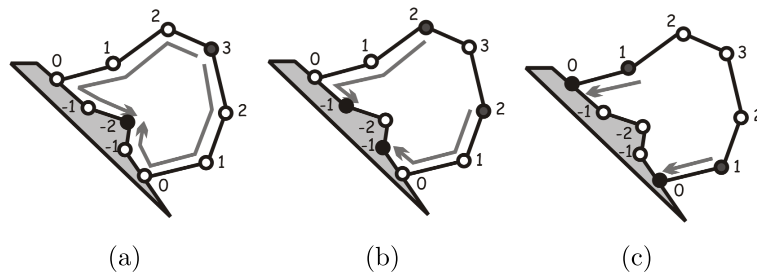
In the following text we will describe three methods how to compute the vertex paths to be able to morph between polygon chains C_{in}, C_{out} and simulate a process of disappearing of part P_i in this way. By simultaneous growing and disappearing of all parts we will achieve the effect of morphing between two simple polygons.

4.2.1 Perimeter growing

The first method to be described is called *Perimeter growing*, because all the vertices v_i lying on C_{out} travel along the perimeter of the part P_i, i.e., their vertex path contains only vertices of P_i. The problem is to determine at which vertex v_j with the topological distance d_j is the specific vertex path supposed to end. Vertex path of a vertex v_i with the topological distance d_i contains vertices with topological distances $(d_{i-1}, d_{i-2}, ..., d_0, d_{-1}, ..., d_j)$ (see Figure 4.5). There are two rules concerning the last vertex of the vertex path, vertex v_j. First, it must belong to the polygon chain C_{in}. Second, we need at least one vertex path to end at each vertex that belongs to the polygon chain C_{in} (to form the shape of the other polygon). Therefore, we use the following approach: the vertex path of the vertex with d_{max} always ends at the vertex with d_{min}. The vertex path of the vertex with d_{max-1} should end at the vertex with d_{min+1}. Generally, a vertex path of the vertex with d_{max-i} should end at the vertex with d_{min+i} (Figure 4.5).

However, such a vertex does not always lie on C_{in}. If we denote n_0, n_1 the number of vertices of C_{in}, C_{out} respectively, we can distinguish the following three cases:

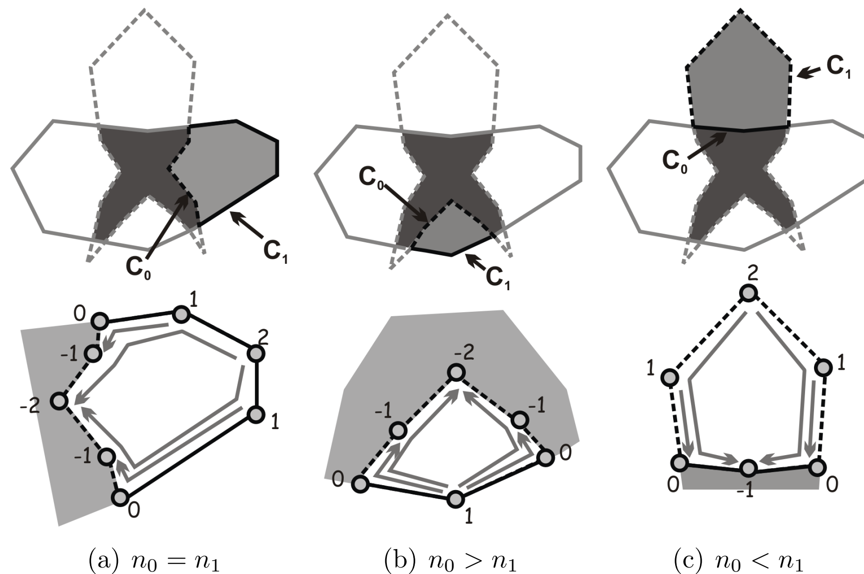
- $n_0 = n_1$ (Figure 4.6a)
  Each vertex of C_{out} ends its path at one vertex of C_{in}.
- $n_0 > n_1$ (Figure 4.6b)
  There are some vertices of C_{in} that do not belong to any vertex path. It means that some of the vertices of C_{out} need to be duplicated. In such a case, we use such vertices of C_{out} that have the topological distance equal to one and duplicate them as many times as is necessary to cover all the vertices of C_{in} that are left.



(a) (b) (c)

Figure 4.5: Vertex paths ($d_{max} = 3, d_{min} = -2$): (a) vertex path for the vertex with $d = d_{max}$ ends at the vertex with $d = d_{min}$ (b) for the vertex with $d = d_{max-1}$ it ends at the vertex with $d = d_{min+1}$ (c) and so on

- $n_0 < n_1$ (Figure 4.6c)
  Some vertices of C_{out} cannot end their paths at the supposed vertex. In such a case their vertex paths end at the intersection vertices.



(a) $n_0 = n_1$ (b) $n_0 > n_1$ (c) $n_0 < n_1$

Figure 4.6: Three possible inputs for computing vertex paths (dark grey: core, grey: selected part, a full line: the first polygon, a dashed line: the second polygon, light grey: a part of the core, grey arrows: vertex paths)

The algorithm is shown in Figure 4.7. The pointers i, j to the list C_{out} are set to the vertex (vertices) with the maximal topological distance (step 2). The pointers k, l to the list C_{in} are set to the vertex (vertices) with the minimal topological distance (step 3). During the computation, the pointers i, k are heading backward through their lists, and j, l forward, until they reach the edge of the list.

The step 4 computes vertex paths for the vertices at pointers i, j. After the vertex path is computed, the pointers are tested whether they have reached the edge of the list (step 5). If $i > 0$ and $k > 0$ (step 1), any of them has not done so, therefore all the pointers are moved and the algorithm comes back to the path assignment. If $i = 0$ and $k = 0$ (step 4), all of them have done so and so all the vertices of C_{out} have their vertex path computed and the algorithm is finished. If

$i > 0$ and $k = 0$ (step 2), it means that there are more vertices of C_{out} than of C_{in} (Figure 4.6b). In such a case, the rest of the vertices of C_{out} head towards such vertices of C_{in} that have the topological distance of -1, in our case, the first and the last vertex of C_{in} - the ones that k, l pointer at. Therefore we will not move k, l, but only i, j. The most complicated case is if $i = 0$ and $k > 0$ (step 3), meaning there are more vertices of C_{in} than of C_{out} (Figure 4.6c). In such a case, we duplicate such vertices of C_{out} that have the topological distance of 1, so the first and last vertex in the list C_{out}, the ones that i, j pointer at (step 6).

Input: Part R_i: list of vertices of $C_{in} = (w_0, \ldots, w_{m-1})$, list of vertices of $C_{out} = (v_0, \ldots, v_{n-1})$, the two intersection vertices v_{I0}, v_{I1}.

Output: Part R_i with a changed list of vertices of $C_{out} = (v_0, \ldots, v_{p-1})$, where $p \geq n$ (some vertices were duplicated), and each vertex of C_{out} has a vertex path, where the number of elements in its path depends on its topological distance,maximal and minimal topological distance of the part.

The algorithm:

1. The list C_{in} is sorted in the order the vertices lie on the polygon chain from v_{I0} to v_{I1}. The list C_{out} is sorted in the same manner.
2. Find such i, j that $i \leq j$ and vertices v_i,v_j are the vertices with the maximal topological distance. If there is only one such vertex ($i = j$), duplicate the vertex v_i and put it in C_{out} between v_i and v_{i+1}. Set $j = i + 1$.
3. Find such k, l that $k \leq l$ and vertices w_k,w_l are the vertices with the minimal topological distance. There is no need of duplicating if $w_k = w_l$.
4. The vertex path of a vertex v_i is set as the couples $(v_i, v_{i-1}), (v_{i-1}, v_{i-2}), \ldots, (v_0, v_{I0}), (v_{I0}, w_0), \ldots, (w_{k-1}, w_k)$. The vertex path of a vertex v_j is set as the couples $(v_j, v_{j+1}), (v_{j+1}, v_{j+2}), \ldots, (v_{n-1}, v_{I1}), (v_{I1}, w_{m-1}), \ldots, (w_{l+1}, w_l)$.
5. Test i, k:
    1. If $i > 0$ and $k > 0$, set $i = i - 1, j = j + 1, k = k - 1, l = l + 1$ and continue by 4.
    2. If $i > 0$ and $k = 0$, set $i = i - 1, j = j + 1$ and continue by 4.
    3. If $i = 0$ and $k > 0$, set $k = k - 1, l = l + 1$ and continue by 6.
    4. If $i = 0$ and $k = 0$, the algorithm is finished.
6. Duplicate the vertex v_{i+1} and put it in C_{out} between v_i and v_{i+1}. Set $i = i + 1$ (pointer i to the position of the new vertex), and because a new vertex was inserted before v_j, set also $j = j + 1$ to maintain the index the same.
7. Duplicate v_{j-1} and put it between v_j, v_{j-1}. Set $j = j - 1$. Continue by 4.

Figure 4.7:The Perimeter growing.

Because the vertex path computed by this method follows the perimeter of the part, the results always seem as if something was really growing from the core. Two things are ruining the nice effect. The former thing is the top of the growing part, which is always a straight line connecting the vertices with the same topological distance. This causes that the method is not suitable for parts, where some vertices with the same topological distance are wide apart (Figure 4.8a). On the other hand, it has really good results for the parts that are narrow and/or highly non-convex,

like parts of a spiral or curly type or long and straight parts (Figure 4.8c,d). The latter thing is that following the shape of the part is not always what we wanted - for example if we have a part as in Figure 4.8b), the part will first grow from the core and then come back a little, and until that it will continue growing. But that coming back is something we do not expect.

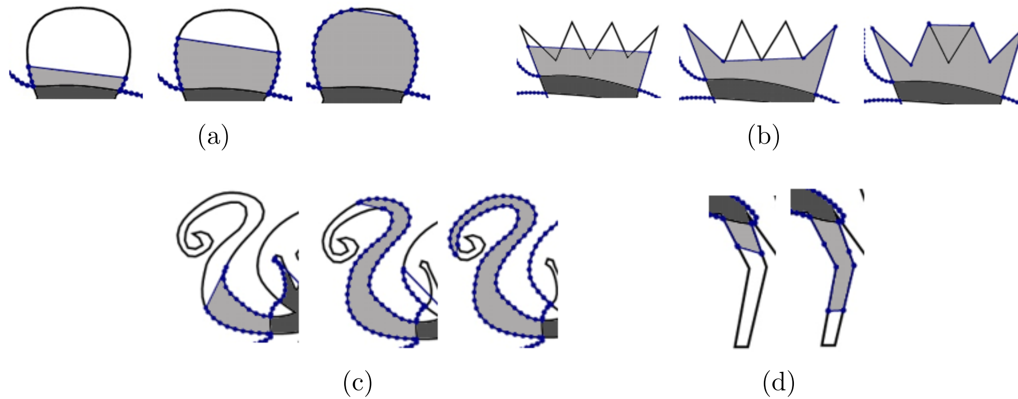


Figure 4.8: When not to use (a,b) and when to use (c,d) the Perimeter growing (dark gray: part of the core, light gray: part growing out, black: input polygons)

4.2.2 Half-line growing

The method called *Half-line growing* is similar to the Perimeter growing with a slight variance: Instead of using the vertices of C_{out} as the elements of the vertex paths, we use the midpoints of line segments defined by the vertices with the same topological distance (Figure 4.9). Let us denote m_i a midpoint of the line segment $v_i v_j$ where $d_i = d_j$. Then, the vertex path of a vertex v_i with the topological distance d_i contains vertices $(m_{i-1}, m_{i-2}, ..., m_0, ..., m_{j-1}, v_j)$. The vertex v_j is computed according to the rules described in Section 4.2.1. The results for all three cases with a different relation between n_0, n_1 are shown in Figure 4.10. When $n_0 = n_1$ (Figure 4.10a), each vertex of C_{out} ends its path at one vertex of C_{in}. If $n_0 > n_1$ (Figure 4.10b), some of the vertices of C_{out} need to be duplicated (those with the topological distance equal to one). If $n_0 < n_1$ (Figure 4.10c), some vertices of C_{out} cannot end their paths at the supposed vertex (so they end at the intersection vertices).

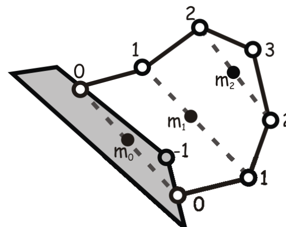
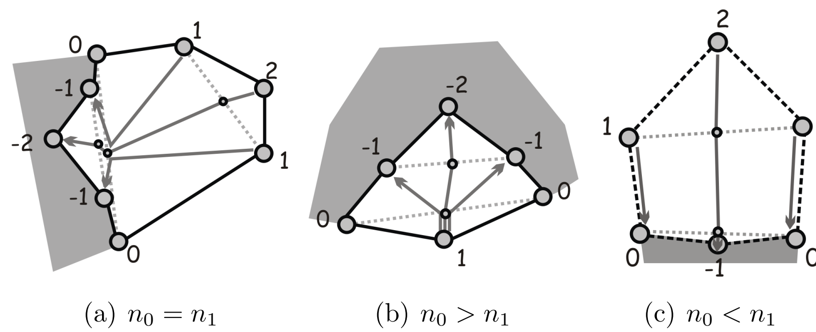


Figure 4.9: Midpoints (black): points in the middle of the line segment (dashed) connecting the vertices with the same topological distance

The algorithm is shown in Figure 4.11. The pointers i, j to the list C_{out} are set to the vertex (vertices) with the maximal topological distance (step 2). The pointers k, l



(a) $n_0 = n_1$ (b) $n_0 > n_1$ (c) $n_0 < n_1$

Figure 4.10: Three possible inputs for computing vertex paths (light grey: a part of the core, grey arrows: vertex paths)

to the list C_{in} are set to the vertex (vertices) with the minimal topological distance (step 3). During the computation, the pointers i, k are heading backward through their lists, and j, l forward, until they reach the edge of the list. All midpoints of the line segments defined by the vertices with the same topological distance are computed.

The step 7 computes vertex paths for the vertices at pointers i, j. After the vertex path is computed, the pointers are tested whether they have reached the edge of the list (step 8). If $i > 0$ and $k > 0$ (step 1), any of them has not done so, therefore all the pointers are moved and the algorithm comes back to the path assignment. If $i = 0$ and $k = 0$ (step 4), all of them have done so and so all the vertices of C_{out} have their vertex path computed and the algorithm is finished. If $i > 0$ and $k = 0$ (step 2), it means that there are more vertices of C_{out} than of C_{in} (Figure 4.10c). In such a case, the rest of the vertices of C_{out} head towards such vertices of C_{in} that have the topological distance of -1, in our case, the first and the last vertex of C_{in} - the ones that k, l pointer at. Therefore we will not move k, l, but only i, j. The most complicated case is if $i = 0$ and $k > 0$ (step 3), meaning there are more vertices of C_{in} than of C_{out} (Figure 4.10b). In such a case, we duplicate such vertices of C_{out} that have the topological distance of 1, so the first and last vertex in the list C_{out}, the ones that i, j pointer at (step 9).

Input: Part R_i: list of vertices of $C_{in} = (w_0, \dots, w_{m-1})$, list of vertices of $C_{out} = (v_0, \dots, v_{n-1})$, the two intersection vertices v_{I0}, v_{I1}.

Output: Part R_i with a changed list of vertices of $C_{out} = (v_0, \dots, v_{p-1})$, where $p \geq n$ (some vertices were duplicated), and each vertex of C_{out} has a vertex path, where the number of elements in its path depends on its topological distance,maximal and minimal topological distance of the part.

The algorithm:

1. The list C_{in} is sorted in the order the vertices lie on the polygon chain from v_{I0} to v_{I1}. The list C_{out} is sorted in the same manner.
2. Find such i, j that $i \leq j$ and vertices v_i,v_j are the vertices with the maximal topological distance. If there is only one such vertex ($i = j$), duplicate the vertex v_i and put it in C_{out} between v_i and v_{i+1}. Set $j = i + 1$.
3. Find such k, l that $k \leq l$ and vertices w_k,w_l are the vertices with the minimal topological distance. There is no need of duplicating if $w_k = w_l$.

4. Create a list of midpoints of C_{out}, $M_1 = \{m_0^1, \ldots, m_{i-1}^1\}$, m_{i-p}^1 is a midpoint of a line segment given by v_{i-p} and v_{j+p}.
5. Create a list of midpoints of C_{in}, $M_0 = \{m_0^0, \ldots, m_{k-1}^0\}$, m_{k-p}^0 is a midpoint of a line segment given by w_{k-p} and w_{k+p}.
6. Compute a midpoint m_I of a line segment given by the intersection vertices v_{I0}, v_{I1}.
7. The vertex path of a vertex v_i is set as the couples $(v_i, m_{i-1}^1), (m_{i-1}^1, m_{i-2}^1), \ldots, (m_0^1, m_I), (m_I, m_0^0), \ldots, (m_{k-1}^0, w_k)$.
   The vertex path of a vertex v_j is set as the couples $(v_j, m_{j+1}^1), (m_{j+1}^1, m_{j+2}^1), \ldots, (m_{n-1}^1, m_I), (m_I, m_{m-1}^0), \ldots, (m_{l+1}^0, w_l)$.
8. Test i, k:
   1. If $i > 0$ and $k > 0$, set $i = i - 1, j = j + 1, k = k - 1, l = l + 1$ and continue by 4.
   2. If $i > 0$ and $k = 0$, set $i = i - 1, j = j + 1$ and continue by 7.
   3. If $i = 0$ and $k > 0$, set $k = k - 1, l = l + 1$ and continue by 9.
   4. If $i = 0$ and $k = 0$, the algorithm is finished.
9. Duplicate the vertex v_{i+1} and put it in C_{out} between v_i and v_{i+1}. Set $i = i + 1$ (pointer i to the position of the new vertex), and because a new vertex was inserted before v_j, set also $j = j + 1$ to maintain the index same.
10. Duplicate v_{j-1} and put it between v_j, v_{j-1}. Set $j = j - 1$. Continue by 7.

Figure 4.11:The Half-line growing.

The Half-line growing method is suitable for similar cases as was the Perimeter growing, with the slight difference that the top line of the growing part is not straight, but it is in the shape of a spire. This can result in better outputs for growing prickles or anything that is sharp, because the spire is there from the beginning, showing the future shape of the part (Figure 4.12).

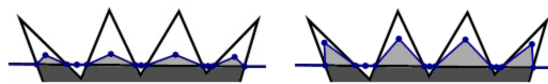


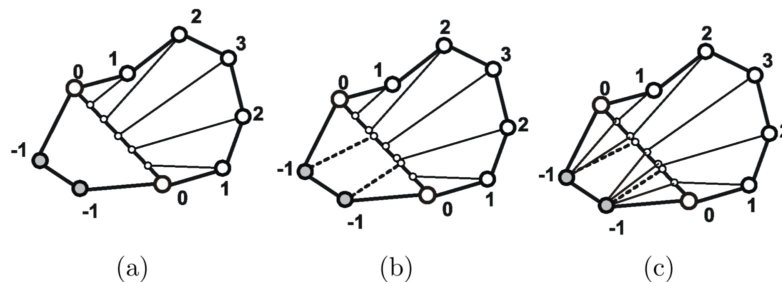
Figure 4.12: When to use the Half-line growing (dark gray: part of the core, light gray: part growing out, black: input polygons)

4.2.3 Projection growing

The last method to be described is the *Projection growing*. Its first difference from the previous methods is that all vertices of C_{out} have the same number of elements in their vertex paths. We call *One step projection* such a method, where there are only two elements in each vertex path of a vertex - the vertex itself and its destination. The *Two step projection* represents a method, where the vertex path has also one element in the middle (the assignment of this element is done in 4.15,step 2). Both of them work in the same way. First, the vertices of C_{out} are mapped (projected) onto the line segment defined by the intersection vertices (vertices with zero topological distance). An equidistant mapping is used - the line segment is divided into

$n + 1$ parts, where n is the number of the vertices of C_{out}. We assign the vertices of C_{out} chronologically to the new vertices on the line segment (Figure 4.13a, Algorithm 4.15,step 3). The next step is to map the vertices of C_{in} in the same way (Figure 4.13b, Algorithm 4.15,step 4). Then we sort the projected vertices of C_{in} and C_{out} into one sorted list in the order in which they appear on the line segment defined by the intersection vertices. The last step is to go through this sorted list as follows (Figure 4.13c, Algorithm 4.15,steps 6-8):

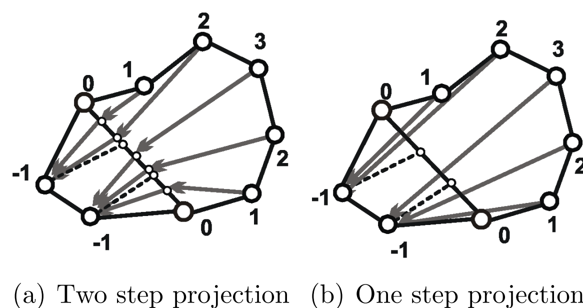
1. Go through the list until a vertex of C_{in} is reached. All the vertices of C_{out} that are before this vertex will have it in their vertex path.
2. Until the next vertex of C_{in} is reached, all the vertices of C_{out} will have the recent vertex of C_{in} in their vertex path.
3. Repeat step 2 until the end of the list is reached.



(a) (b) (c)

Figure 4.13: Computing vertex paths in the Projection growing (a) mapping the vertices of C_{out} onto the line segment between the intersection vertices (b) the same mapping of the vertices of C_{in} (c) choosing the vertices of C_{in} for the vertex paths of the vertices of C_{out}(dashed: mapped vertices, thin lines: vertex paths)

For the *Two step projection* method, the vertex into which the vertex v_i was mapped also belongs to the vertex path of v_i. For the *One step projection* method, only the vertex v_i itself and the assigned vertex of C_{in} are in the vertex path of v_i (Figure 4.14).



(a) Two step projection (b) One step projection

Figure 4.14: Vertex paths (grey)

Input: Part R_i: list of vertices of $C_{in} = (w_0, \ldots, w_{m-1})$, list of vertices of $C_{out} = (v_0, \ldots, v_{n-1})$, the two intersection vertices v_{I0}, v_{I1}.

Output: Part R_i with a changed list of vertices of $C_{out} = (v_0, \ldots, v_{k-1})$, where $k \geq n$ (some vertices were duplicated), and each vertex of C_{out} has a vertex path containing three positions in case of the two step projection method - its own position, its projected position and the position of its corresponding vertex - or only two positions for the one step projection (not containing the projected position).

The algorithm:

1. Compute the line segment l_i defined by the vertices v_{I0} and v_{I1}.
2. For all the vertices of C_{out} put their coordinates as the first element in their vertex paths (only for the two-step projection).
3. Map ("project") the vertices of C_{out} onto l_i: divide l_i on $n+1$ equidistant parts (line segments) $(p_0, p_1), \ldots, (p_i, p_{i+1}), \ldots, (p_n, p_{n+1})$, and assign them chronologically to the vertices: the projected position of v_i is p_{i+1}.
4. Map ("project") the vertices of C_{in} onto l_i in the same way: divide l_i on $m+1$ equidistant parts $(q_0, q_1), \ldots, (q_i, q_{i+1}), \ldots, (q_m, q_{m+1})$. The projected position of w_i is q_{i+1}.
5. If the first vertex in C_{in} is not neighbor of v_I0, reverse the list C_{in}, so the vertices v_i appear on l_i starting from v_{I0}, check the list C_{out} in the same manner and merge them into one list C while maintaining the order.
6. Go through the list C until a vertex v_i is reached. All the vertices of C_{out} that are before this vertex will have v_i in their vertex path.
7. Until the next vertex of C_{in} is reached, all the vertices of C_{out} will have the recent vertex of C_{in} in their vertex path.
8. Repeat step 7 until the end of the list is reached.

Figure 4.15:The projection growing.

The projection growing is a method that provides its outputs somewhere between the Perimeter (or Half-line) growing and typical algorithms based on the correspondence. There is still dependence of the outputs on the core, however, the parts do not follow any shapes of the original polygons but grow directly from the core. That results in its usability when the shape of the part is convex or when it is non-convex, but not curled or spiral (Figure 4.16).

4.2.4 Merging

As was already figured in the general description, after we handle separately each part, we want to merge them, so that the result is one polygon with a vertex path for each vertex. Also remember that vertex paths of the growing parts must be reverted because we considered only the disappearing. The merging process in 2D is motivated by Weiler-Atherton algorithm for polygons intersection [23], recall Section 2. It processes part by part by copying vertices with positive topological distance to the new list of vertices. It skips between the adjacent parts at the intersection vertices. The new list of vertices forms the new polygon. The details of the merging process are in Figure 4.17.

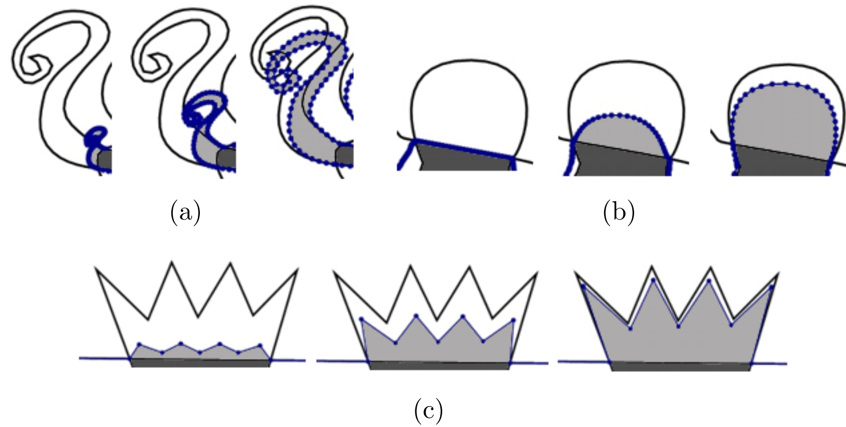


Figure 4.16: Where not to use (a) and where to use (b,c) the Projection growing (dark gray: part of the core, light gray: the part growing out, black: input polygons)

Input: List of parts $R = \bigcup R_i$, lists of vertices of each part $l_i = (v_0, \ldots, v_{n-1})$, list of vertices of the core $C = (C_{in}, \ldots, c_{m-1})$. The lists l_i and C are circular (so the next vertex to v_{n-1} is v_0 and the previous vertex to v_0 is v_{n-1}). Each part has a different number of vertices in its list, but each part shares exactly two vertices with two other parts (the intersection vertices) or with the core.

Output: One list of vertices containing such vertices v_j from the lists l_i that have $d_j \geq 0$.

The merging algorithm:

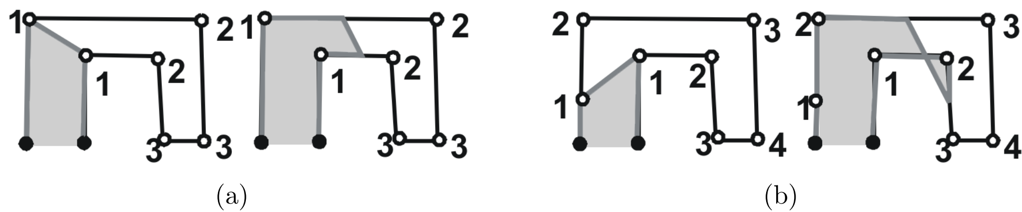
1. Choose an arbitrary part R_i from the list of input parts (for example the first one). Start from the first vertex in R_i. Go through l_i until the first intersection vertex v_j is found. Add v_j to the resulting list (which now contains only v_j).
2. Check the vertex v_{j+1} if its topological distance is positive. If so, continue forward, otherwise backward, in l_i. Add each visited vertex to the resulting list until the next intersection vertex v_k is added. Delete the part R_i from the list of parts.
3. Because v_k was the intersection vertex, there are three possibilities:
   - One of the parts in the list contains it - in such a case use this part and continue by 2.
   - The list of parts is empty (v_k is the intersection vertex from step 1). In such a case, the algorithm is finished.
   - The input polygons A and B shared some vertices and edges, and therefore no part in the list contains v_k. In such a case, we find v_k in the core list C - let us denote the found v_k as c_j to know that it is in the list C. We check the vertex c_{j+1} if it belongs to l_i of the part where v_k was. If so, go backward, otherwise forward, in the list C. Add each visited vertex to the resulting list until the next intersection vertex c_k is added. Denote c_k as v_k and continue by step 3.

Figure 4.17:The merging algorithm.

4.2.5 Improvements

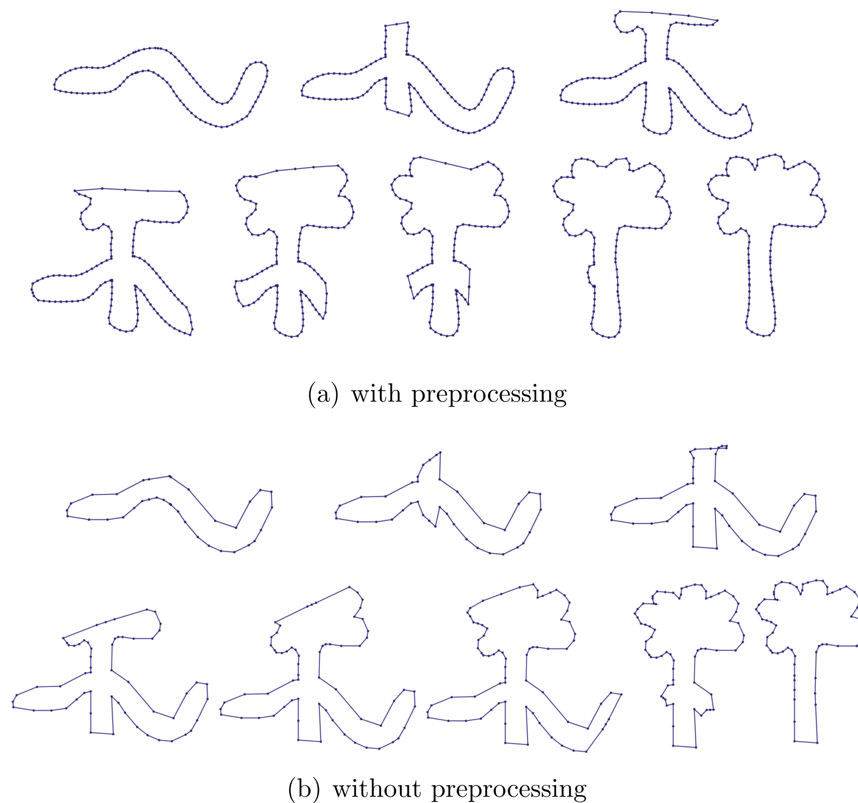
Both Perimeter and Half-line algorithm do not take into account lengths of edges of the poly line they morph, they compute only with the topological distance of the

points. That results in a different behavior for the same-shaped polygons with a different number of vertices (see Figure 4.18). A solution is to include a preprocessing part to this algorithm, when the polygons are "resampled", so that all their edges are of the same length. If we include such resampling into our algorithm, the result is even better (see Figure 4.19). Not only it results in a smoother and controlled movement of each part, but also the segments that are about the same length have approximately the same number of vertices.



(a) (b)

Figure 4.18: Adding one point to a polygon results in different behavior



(a) with preprocessing

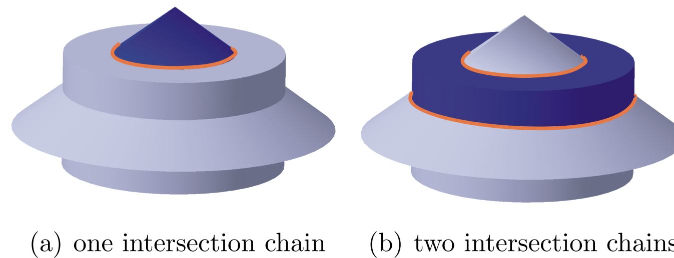
(b) without preprocessing

Figure 4.19: Morphing between a snake and a tree (half-line algorithm)

Another improvement comes from the fact that each part P_i is computed separately. It means that the parts do not have to be computed by the same method, the methods can be arbitrarily combined. As each method is suitable for a different kind of shapes (as is discussed in Section 5), combining the methods (so that we can choose what method to use for each part, not globally) can result in a more interesting output.

4.3 3D algorithm

In 3D, the input objects A, B are triangle meshes. A part P_i consists of two surfaces C_{in}, C_{out} (shown in Figure 4.2). The surfaces C_{in}, C_{out} are separated by polylines, let us call them *intersection chains*. Unlike in 2D, there are not always two intersection chains - the number of the intersection chains can vary from one to infinity. The intersection chains are always closed. Figure 4.20 shows an example of a part with one intersection chain (Figure 4.20a) and two intersection chains (Figure 4.20b), which are the most often cases, however, there can be parts with also more than two intersection chains. By morphing the surface C_{out} to the surface C_{in} we achieve the effect of disappearing of the part P_i in the core C.



(a) one intersection chain (b) two intersection chains

Figure 4.20: There can be an arbitrary number of the part's intersection chains (orange).

In 3D, we compute the topological distance with respect to the vertices lying on the intersection chains and separately for C_{in} and C_{out}. We also use the negative topological distances for C_{in} as we did in 2D.

The topological distance of a vertex v_i is computed by the *Breadth-first search* algorithm, where the search begins at the intersection chain(s). The vertices of the intersection chain(s) are assigned a topological distance of zero, and put into the stack. Than one by one, the vertices are popped from the stack, and for each vertex v_i (with a topological distance t_i), we go through its neighbors. If the neighbor has greater topological distance than $v_i + 1$, we assign to it the new distance and put it in the stack. The topological distances are computed separately for C_{in} and C_{out} of each part (we could search the whole part together, but we want to assign negative topological distances to C_{in}). The algorithm is shown in Figure 4.21. An example of computed topological distances can be seen in Figure 4.22.

Input: List of vertices of the intersection chains of the current part $I = \bigcup v_{Ii}$, list of vertices $C_i = (v_1, \ldots, v_N)$, for which the topological distance should be computed. Stack $S = \emptyset$.

Output: List of vertices $C_i = (v_1, \ldots, v_N)$ with assigned topological distances $(d_1, \ldots, d_N)$.

The algorithm:

1. Initialization: For all $v_{Ii} \in I$: $d_{Ii} = 0$, push v_{Ii} in S. For all $v_j \in C_i$: $d_j = \infty$.
2. Pop a vertex v_i from the top of the stack. Go through all its neighbors, for each neighbor v_j: If $d_j > d_i + 1$, $d_j = d_i + 1$ and push v_j in S.
3. If $S \neq \emptyset$, continue by 2. Otherwise the algorithm is finished.

Figure 4.21:The algorithm for computing topological distances (breadth-first search).

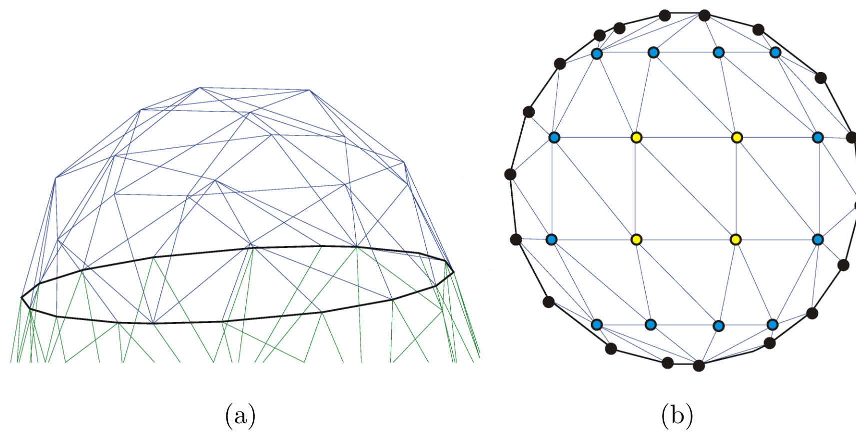


Figure 4.22: An example of topological distances in 3D
(a) side view of the part (blue), the intersection chain (black)
(b) top view - computed topological distances: - intersection points ($d_i = 0$, black), $d_i = 1$ (blue), $d_i = 2$ (yellow).

In the following text we will describe how the extension of the Perimeter and Projection growing to 3D can be one. We will also discuss why not to extend the Half-line growing.

4.3.1 Perimeter growing

In 2D, the vertices v_i lying on C_{out} travel along the perimeter of the part P_i. That remains the same in 3D. Next, the first direction of a vertex v_i is in 2D given by its neighbor with the topological distance $d_i - 1$. There is only one case when there are two such neighbors - when C_{out} consists of an odd number of vertices, the vertex with the highest topological distance has two neighbors with $d_{max} - 1$. In that case, we duplicate such a vertex. But in 3D, a vertex v_i can have many neighbors with $d_i - 1$ (see Figure 4.23).

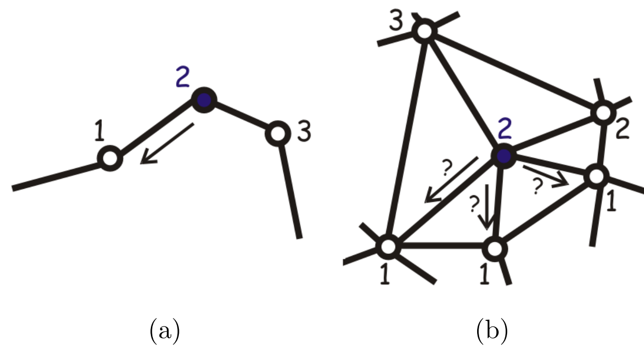


Figure 4.23: The first difference in 2D and 3D for Perimeter growing. The direction of a vertex v_i in 2D is clear, but there are many possible directions for v_i in 3D.

The solution is to compute the vertex path during computing the topological distance: When a vertex v_i assigns the topological distance $d_i + 1$ to a vertex v_j, its vertex path and the vertex itself is copied to the vertex path of the vertex v_j.

Unlike 2D, we compute the vertex path also for the part C_{in}, which lies on the core. We actually computed them also in 2D, during the path assignment - when we assigned a vertex with d_{max} a vertex path to a vertex with d_{min}, we actually computed a vertex path for d_{max} (ending at the intersection vertex), a vertex path for d_{min} (also ending at the intersection vertex) and merged them into one vertex path for only d_{max}. But because the vertex paths were so clear (each vertex had always only one direction to go), we did not need to precompute them. We do need it now, because if we did not remember during the topological distance computation from which vertex the vertex was assigned (and so its vertex path), we would loose the information forever.

So now we know how all the vertices reach the core, but we need only the vertices of C_{out} to travel in the future. Intuitively, we want to assign the vertex paths of C_{in} to such vertices of C_{out} that end at the same intersection vertex.

So we want to go through all the intersection vertices and separately for each to solve the correspondence problem for the vertices that end at their position. For one intersection vertex v_{Ii}, we need to deal with three possibilities, which are the same as they were in 2D version, so let us discuss what happens if we want to use the same solution as was in 2D. If there are no negative vertices ending at v_{Ii}, all the positive vertices end at v_{Ii}. If there are no positive vertices ending at v_{Ii}, we use v_{Ii} and duplicate it as many times as is the number of the negative vertices ending at its position. The case when both negative and positive vertices end at v_{Ii} is shown in Figure 4.24. In such a case, the vertices with d_{max} should end at the vertices with d_{min}, the vertices with $d_{max}-1$ at the vertices with $d_{min}+1$ and so on. However, here is the problem that more vertices have the same topological distance. In Figure 4.24, we have four vertices with d_{max} (A, B, C, D) and three vertices with d_{min} (E, F, G), and we need to solve the correspondence problem within them. Visually, we can decide that A should end at E; B, C at F and D at G. But unfortunately, we do not have any other information about the vertices than their topological distance, and so we are unable to decide this automatically. We could try to decide it according to the neighbors, but we still would not know whether A should end at E or G. Or worse, also E and G could be neighbors and then neighboring information would not help much.

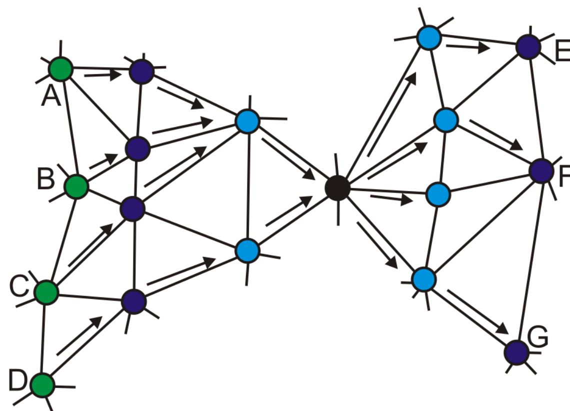
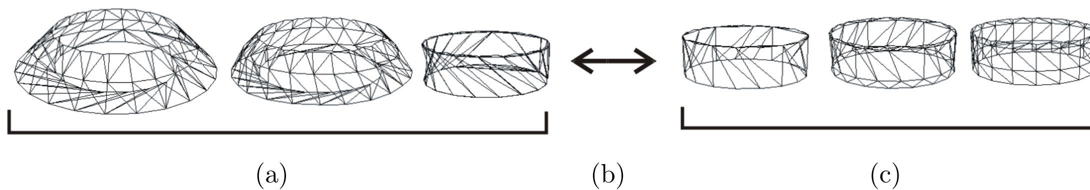


Figure 4.24: Correspondence problem at one intersection vertex (black, other vertices are colored according to their topological distance: ±1 light blue, ±2 blue, 3 green).

In spite of this correspondence problem, we can achieve the resulting morph by changing our expectations of the supermesh - according to 2D, the supermesh should be obtained by merging the parts. But if we leave this solution, we can define the resulting morphing sequence to consist of three parts (see Figure 4.25): In the first part, $(0, t-\delta)$, the outside of the part (vertices with $d > 0$) is morphing towards the intersection vertices. At $t-\delta$, the resulting morph appears to contain only the intersection vertices (but there are also the vertices of the outside part, which are at the same positions as the intersection vertices), connected by edges defined by the outside part. Let us call it S_0. In the last part, $(t+\delta, 0)$ the inside of the part (vertices with $d < 0$) is morphing from the intersection vertices towards their positions. At $t+\delta$, the resulting morph appears to have only the intersection vertices, connected by edges defined by the inside part. Let us call it S_1. During the time $(t-\delta, t+\delta)$, we need to morph from S_0 to S_1, which are two triangular meshes with the same vertices, but a different connectivity. This part is not solved in this work, but possible solution is to follow the solution of [1]. First, find the edges which should be swapped. Because some swaps are dependent on the other ones, a dependency graph is constructed (otherwise, we would have to morph only one edge at a time). Then according to the dependency graph, exact time portion is computed for each edge swap. The swap itself is realised smoothly as is described in [11] and summarised in Section 3.2.



(a) (b) (c)

Figure 4.25: The resulting morphing sequence consists of three parts: (a) the outside part morphs towards the intersection vertices, (b) connectivity change, (c) the inside part morphs from the intersection vertices.

4.3.2 Projection growing

The projection growing in 2D resulted in the effect that all the vertices of the positive part headed towards certain points of the line segment connecting the two intersection vertices, and then continued towards their corresponding vertices in the negative part. That is an effect we want to follow in 3D - to compute the vertex paths of all the vertices of the positive part to make them first travel to some plane given by the intersection vertices, and then continue towards their corresponding vertices in the negative part. In this section, we will discuss how to compute such vertex paths and what geometrical problems occur.

When using projection growing, we actually establish a correspondence between C_{out} and C_{in}, but we do it in the simplest way. In 2D, we used the line segment l defined by the intersection vertices, onto which we equidistantly mapped the vertices of C_{out} (Figure 4.13). Each vertex v_i of C_{out} then "occupied" one part of the line

segment l_i, and his corresponding vertices were those vertices of C_{out} that were mapped into l_i.

In 3D, the equivalent for line segment l_i would be probably an area on a plane bounded by the intersection chain. However, it is not that simple for two reasons: first, there is not always one intersection chain, and second, even if there is only one intersection chain, it does not always lie on a plane. So we need to choose a projection plane somehow dependent on the intersection vertices (vertices of the intersection chain). For our first experiments we decided to use a plane given by three arbitrary points of the intersection chain(s). When we are projecting the part onto the plane, we would expect that the plane is rotated in a way that the orthogonally projected part covers the largest possible area on the plane (so there will be the least loss of information). Figure 4.26 shows four examples of the resulting plane, when the vertices at the positions $0, \frac{1}{3}, \frac{2}{3}$ in the intersection chain were chosen. Let us judge them according to our expectations. Figures 4.26a,c show parts where the vertices of the intersection chain lie in one plane, and the resulting projection plane is satisfactory. Figure 4.26b shows an example of a part with two intersection chains. In such a case, we choose two vertices from the first intersection chain and one from the second, to ensure that the plane will cut the part. We will discuss such parts in Section 4.3.4. Figure 4.26d shows that also for the case when the vertices that do not lie at the same plane the result is quite satisfactory. We will further discuss the importance and choice of the projection plane in Section 4.3.3 along with other expectations that should be fulfilled.

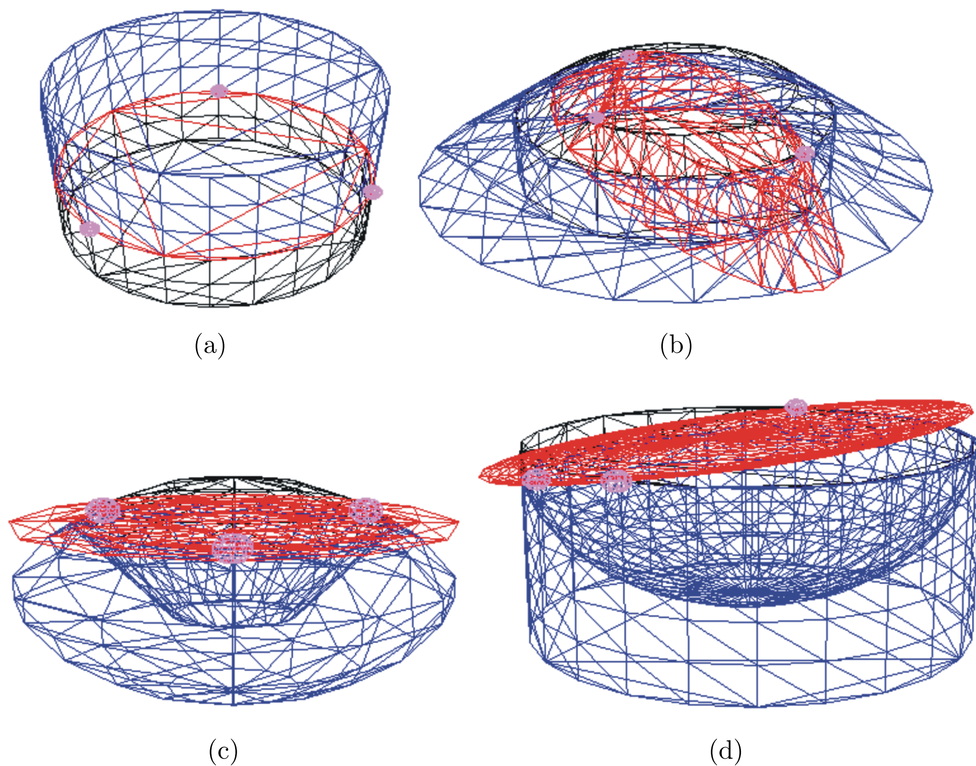
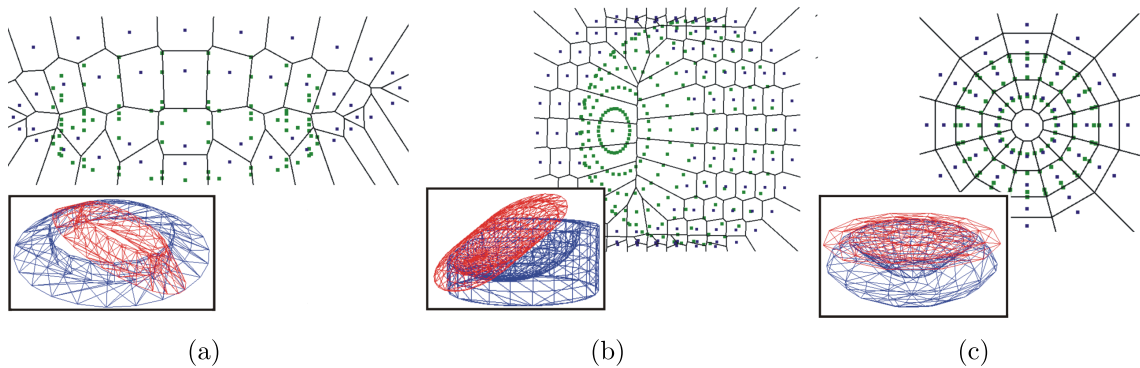


Figure 4.26: The construction of a projection plane: the part (blue), the core (black filled with gray), three chosen points of the intersection chain (violet), the part projected onto the plane (red).

When we already know the plane, we need to project the points onto it. Here, we cannot use the simple equidistant mapping as we did in 2D, because we do not have a bounded area here. So we use orthogonal projection of the vertices to the plane for our first experiments. We will discuss our choice of projection in Section 4.3.3. We also project both vertices of C_{in} and C_{out} as it was in 2D. But now, we need to decide how to define the area which will occupy one vertex of C_{out} (an equivalent to a line segment l_i in 2D). We decided to use Voronoi diagram of the projected vertices of C_{out}, which will divide the plane into cells $C = \cap c_i$, where the closest vertex of each vertex in a cell is the vertex that is defining this cell (the examples of Voronoi diagrams of the parts are in Figure 4.27).



(a) (b) (c)

Figure 4.27: Voronoi diagrams of the projected parts
(bottom left corner: a part (blue), projected part (red); top - diagram of the part: vertices outside the core, forming the diagram (blue), vertices on the core (green))

The projected vertices of C_{out} are defining the cells of the resulting Voronoi diagram, and the projected vertices of C_{in} are lying in the cells. An important fact is that it can happen that more vertices of C_{out} define only one cell of the Voronoi diagram - that happens when they are exactly above themselves (as is in Figure 4.29). In each cell, there are n vertices $V = (v_0, \dots v_{n-1})$ defining the cell (vertices originally of C_{out}, most of the cases $n = 1$) and m vertices $W = (w_0, \dots, w_{m-1})$ (vertices originally of C_{in}) lying in the cell.

The Voronoi diagram is used to establish the vertex-to-vertex correspondence between C_{in} and C_{out}. We need to assign vertices of C_{in} to the vertices of C_{out} (put their coordinates in the vertex paths of vertices of C_{out}). The cell of the Voronoi diagram defines the area, where all the vertices of C_{in} are closer to the vertex (or vertices) of C_{out} defining the cell than to the other vertices. We will use this information to establish the correspondence - each vertex of C_{out} will head towards the vertices of C_{in} that projected into its cell. However, some problems occur here. Some vertices do not have any vertex in their cells. In that case, the intuitive solution would be to find the nearest vertex in the neighbourhood cells and let them have it in their vertex paths. We do not mind, if more vertices of C_{out} head towards one vertex of C_{in}, they simply have it both in their vertex paths. Another problem is when some vertex of C_{out} has more than one vertex of C_{in} in its cell. As people cannot go to two places at the same time, the vertex also cannot have two ending vertices in its path. The solution is to duplicate the vertex and so the duplicated

vertex can head towards the second vertex of C_{in}.

In the previous explanation, we expected that there is only one vertex defining the cell. Let us now discuss what happens in the cell exactly, taking into account the possibility of more vertices of C_{out} defining the cell. So three cases can happen in each cell:

1. $m \leq n$, $m \neq 0$, meaning there are less or equal vertices of C_{in} than of C_{out}, but there is at least one (so we do not need to search the neighbors).
2. $m = 0$, meaning there are no vertices of C_{in} in the cell and so we need to search the neighbors if they do have any.
3. $m > n$, meaning there are more vertices of C_{in} in the cell that the vertices of C_{out} can handle, so some vertices of C_{out} need to be duplicated.

The first case has another two possibilities: The former one is that there is one vertex w_i of C_{in} in a cell defined by vertices $v_j, \ldots, v_k$ of C_{out}. That means the correspondence establishment is simple here - we add w_i to the vertex path of all the vertices $v_j, \ldots, v_k$. Figure 4.28 shows an example of the correspondence computation for a part of a cylinder. In the Voronoi diagram (Figure 4.28c), each cell contains more than one vertex of C_{out}, but only one vertex of C_{in}. The resulting morph is as we would expected - all the vertices that are above some intersection vertex head towards it (Figure 4.28d).

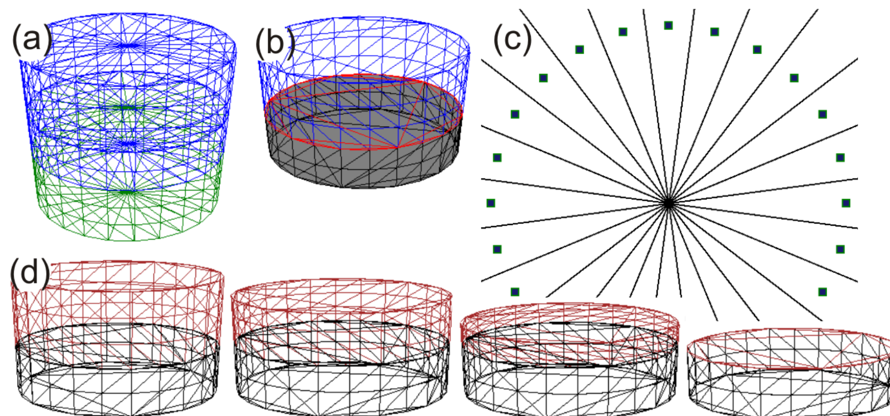


Figure 4.28: Computing projection when $m = 1, n > 1$
(a) input objects - cylinders, (b) core (black filled with gray) and one part (blue) projected onto the plane (red) given by three arbitrary points of the intersection chain, (c) Voronoi diagram of the part: vertices outside the core (blue), vertices on the core (green), (d) examples of the resulting morph the part (red), the core (black)

But if there are also more vertices of C_{in}, we need to decide which vertex of C_{out} will head towards which one of C_{in}. Because the vertices of C_{out} are at the exactly same position, we cannot decide depending on their projected coordinates. But we can decide according to their original positions - according to "how far" they were from the intersection vertex. This actually recalls our solution in the perimeter growing - there we add the "furthest" vertices of C_{in} (vertices with the

lowest topological distance) to the vertex paths of the "furthest" vertices of C_{out} (vertices with the highest topological distance). That is exactly how we will solve the correspondence here. So how to do it algorithmically: we sort the lists W, V according to the topological distances; $w_0, \ldots, w_{m-1}$ upwardly (so that the vertex with the lowest topological distance is first) and $v_0, \ldots, v_{n-1}$ in a descending order (so the vertex with the highest topological distance is first). We add the vertex w_i (vertex at the position i in the sorted list of W) to the vertex path of the vertex v_i (vertex at the position i of the sorted list of V). Because $m \leq n$, the list V may be longer than the list W. In such a case, we add w_{m-1} to the vertex paths of the rest of vertices in V. Figure 4.29 shows an example of the correspondence computation for a part, where projecting C_{out} results in a Voronoi diagram, where cells filled with grey contains more vertices of C_{in} and also more vertices of C_{out}. The cells that are not filled contain no vertices of C_{in} and will be discussed in the following paragraph. The resulting morph in (Figure 4.29d) shows that the correspondence is set up as we would expected.

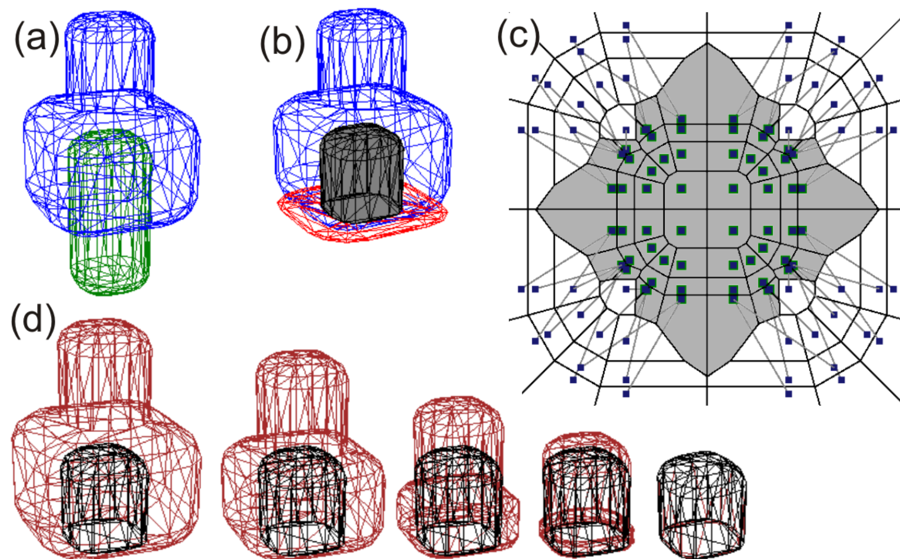


Figure 4.29: Computing projection when $m > 1, n > 1, m \leq n$ (yellow cells in (c), upper part of the part in (a,b)) and when $m = 0$ (white cells in (c))
(a) input objects, (b) core (black filled with gray) and one part (blue) projected onto the plane (red), (c) Voronoi diagram of the part: vertices outside the core (blue), vertices on the core (green), corresponding vertices connected by gray lines, (d) examples of the resulting morph the part (red), the core (black)

If there are no vertices of C_{in} in the cell, we already told that we try to find the closest vertex of C_{in} in the neighbourhood cells. First, we look in the neighbourhood cells of our cell, if they do not have any vertex inside either, we look in the neighbour's neighbours and so on until we find some cell containing at least one vertex of C_{in}. If there is a large area of cells without any vertex inside them, this process can be quite long, even if we do not use the neighbors that have been already searched. An example of this case is in Figure 4.29: in the Voronoi diagram of the part (4.29c), each vertex is connected by a gray line with the vertex that was found as the nearest vertex in its neighbourhood, and therefore added to its vertex path.

If there are more vertices defining one cell with $m = 0$, we still find only one nearest vertex and add it to the vertex paths of all the vertices of the cell.

The worst case is when $m > n$, meaning that there are more vertices of C_{in} than vertices of C_{out}. As was already told, we solve such a case by duplicating as many vertices of C_{out} as needed (so $m-n$ times). If there is more than one vertex defining the cell, we duplicate the vertex with the lowest topological distance (as we did in 2D Perimeter growing). After the duplication, we assign the vertices of C_{in} to the vertex paths of the vertices defining the cell and the duplicated vertices. But as the duplicated vertices are not inserted into the mesh, they would not be visible in the resulting object. So we need to place them somehow somewhere in the mesh C_{out}.

We tried two methods of inserting the duplicated vertices into the mesh C_{out}. Both of them work with the projected coordinates of the vertices of C_{out} and C_{in}.

The first method redistributes the triangles of the vertex v_0 (the original vertex which was duplicated) among the duplicated vertices and itself. The vertex v_0 is the only one that is included in the mesh. Figure 4.30a shows v_0 with its triangles and path to one of the vertices of C_{in} (the other two vertices of C_{in} will be the target of the duplicated vertices).

Then we add also the other the vertices of C_{in} to the vertex paths of the duplicated vertices, and move them half way according to their paths (Figure 4.30b). We move also the vertex v_0. That is done to be able do distinguish the vertices in the space and so reassign the triangles correctly.

For each vertex $v_0, ..., v_{m-n}$ we construct a vector $\vec{w_0}, ..., \vec{w_{m-n}}$, which is defined by the original position of the vertex v_i and its new position. Then we sort the vectors clockwise according to their angle with $\vec{w_0}$ (Figure 4.30b).

For each triangle a_i, b_i, v_0 we construct two vectors $\vec{v_0 a_i}, \vec{v_0 b_i}$ and find their positions in the sorted field $\vec{w_0}, ..., \vec{w_{m-n}}$: we find such a vector $\vec{w_i}$ that has a larger angle with the vector $\vec{w_0}$ than has $\vec{v_0 a_i}$. If there are more such vectors, we choose the one with the lowest angle. We also find the vector $\vec{w_j}$ that fulfils the same conditions for $\vec{v_0 b_i}$. The vectors $\vec{w_i}, \vec{w_j}$ determine the vertices to which the triangle will be assigned. If $i = j$ (Figure 4.30c) or $j = i + 1$ (Figure 4.30d), the new triangle will be a_i, b_i, v_i. Otherwise (Figure 4.30e), we need to add new triangles to the object, because we have to assign one triangle to more vertices.

A more complex example of such a situation is shown in Figure 4.31. We solve it in the following way: First, we choose a vertex v_k with the largest angle $a_i v_k b_i$ and we assign our triangle to the vertex v_k (Figure 4.31a). We continue recursively for the vertices to the left of v_k and to the right (Figure 4.31b), until there are no vertices left.

Figure 4.30f shows the situation when all the triangles were redistributed. There are still some triangles missing - first, the so-called *border* triangles. The neighbor triangles used to end at the same vertex v_0, and now they end at different vertices v_i, v_j. Therefore, we also need to add the new triangles (border triangles) v_i, v_j, a_i. We keep track of what triangles to add during the redistributing process.

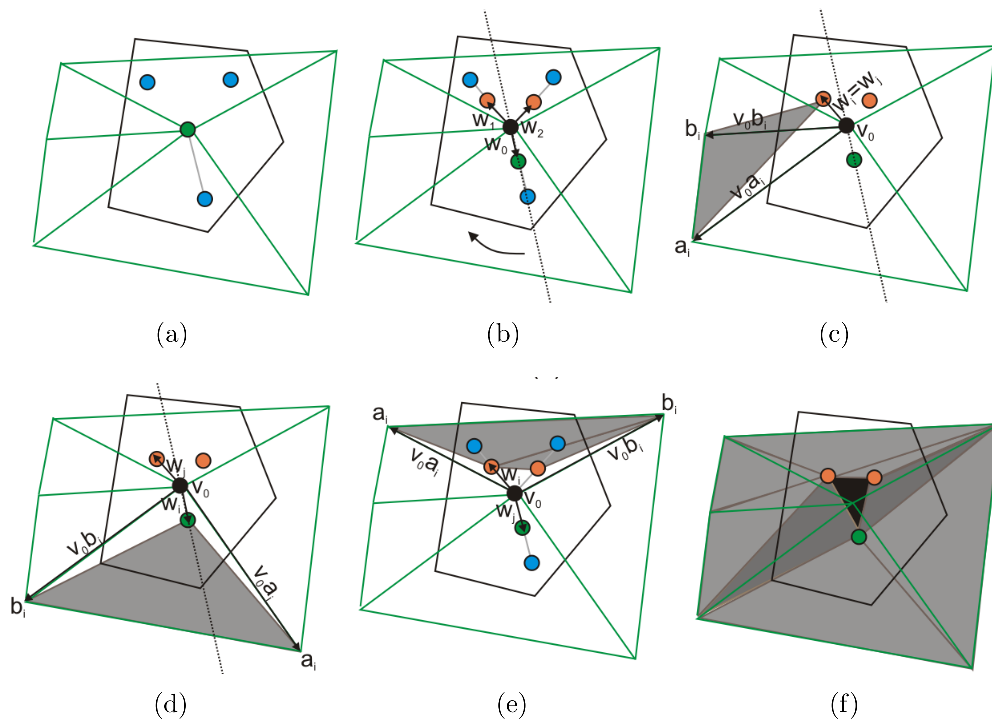


Figure 4.30: Duplicating the vertex

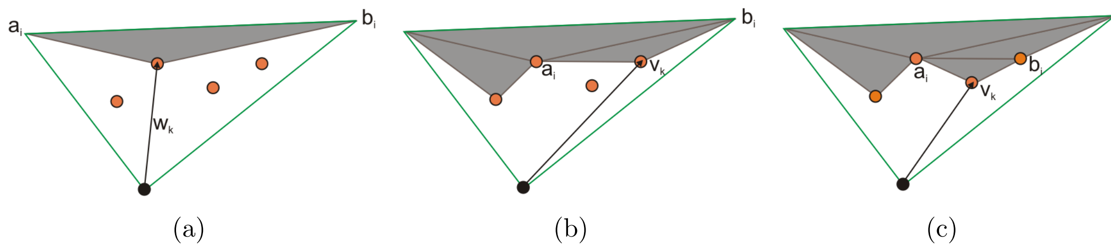


Figure 4.31: Assigning one triangle to more vertices

After the border triangles are added, there is still an empty convex space bounded by the vertices $v_0, ..., v_{m-n}$. This space needs to be triangulated (Figure 4.30g), for example by the Delaunay triangulation.

It may seem that everything is done, however, there are still two situations that need to be solved. First, we need to solve the case when two vertices v_i, v_j has about the same angle with v_0. We decided to solve such a case by using only the vertex which has a larger distance from the original position of v_0, and taking another vertex into account during the triangulation process.

Another problem is when the vertices $v_1, ..., v_{m-n}$ are not surrounding the vertex v_0, in the context, than if $m - n > 2$, v_0 needs to lie in the convex hull of $v_1, ..., v_{m-n}$. If this rule is not fulfilled, self-intersections may occur, because some neighbors' edges may intersect the area where the edges of v_0 are redistributed.

Because this process is highly dependent on the current situation in the cell, we decided to implement another, much simpler way of duplicating the vertex. The

vertex v_0 is duplicated again and the duplicated vertices are again put half way towards their vertex paths. But the triangles are reassigned in a different way - for each v_i, a triangle of v_0 in which it lies is found. Sometimes, it does not lie in any triangle of v_0, in that case, we need to search in the neighbors' triangles. Then this triangle is split into three new triangles, taking them into account when searching the triangle of v_{i+1}. This division is not very suitable for cells, where there are more vertices to duplicate, because its result is not smooth. In such cases, using the first way of duplicating would be better. This way of duplicating may also result in self-intersections, if a vertex v_i moves between several triangles during the morphing process (only the triangle in its half way is taken in account).

When we assign paths to all the vertices, the Projection method is done and we can merge the parts to get the resulting object. However, the paths are not always assigned correctly. To be correct, they need both the positive and the negative part to project in such a way, that the vertices in the neighbouring cells in the Voronoi diagram are also neighbors in the original mesh. This can be achieved by choosing the right projection plane (which is discussed in the next Section), but only if both C_{in}, C_{out} are convex or star-shaped. If not, we need to divide them into convex or star-shaped parts, which will be discussed in Section 4.3.5.

The resulting supermesh contains the merged parts, each part consisting of only vertices of the outside (those with $d >= 0$). When the vertices of the part are at their original positions (the first position in their paths), they exactly form their part of the original shape. But when they are at the target positions (last position in their paths) they do not form the original shape exactly. That is because we did solve only the vertex paths, not the connectivity. As was discussed for the Perimeter growing, two shapes with the same vertices does not have to have the same edges (connectivity). Therefore, the vertices form only the approximation of the original shape at their target positions. The quality of the approximation depends on the fineness of the mesh (the finer the mesh is, the better is the approximation). To morph the outside part into the exact shape of the inside part, we would have to include the change of the connectivity during the morphing process as well, as was discussed in 4.3.1.

4.3.3 Projection plane

One of the key steps of the Projection growing method is to choose a "good" projection plane, meaning that when we are projecting the vertices C_{in} and C_{out}, we want to avoid the case when two vertices, that are not neighbors, project into the same cell or a neighbouring cells. This can happen for example when the part has some unconvexities. We will try to deal with the non-convex parts later, so for this moment, let us suppose that the part is convex. But also convex parts can produce such a case, when the projection plane is chosen inconveniently.

We decided to use the orthogonal projection to project the vertices of P_i onto the plane. Such projection is usable only for convex parts and several special types of star-shaped parts. For the star-shaped parts, the perspective projection would be better, so it would be useful to decide what projection to use according to the current

type of the part. But for the first experiments with the algorithm, we decided to start with the orthogonal projection.

Because we will use the orthogonal projection, the position of the plane is not important, we only need to find a proper normal vector. However, to make the examples more evident, we place the projection plane nearby the intersection chain to be able to compare the results.

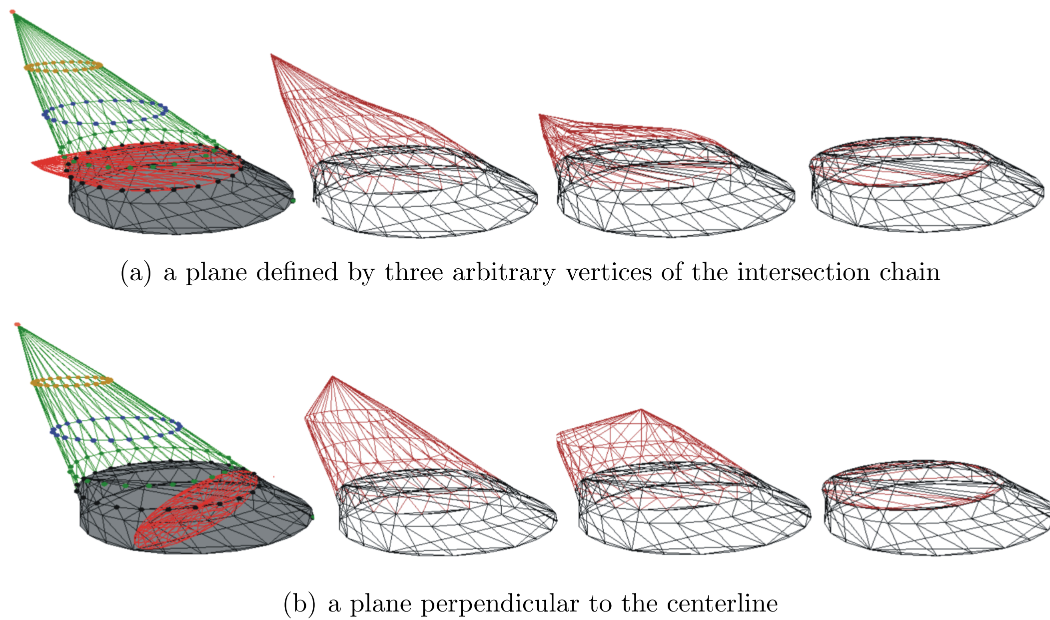
To describe the best direction of the projection plane, we will use a concept of a *centerline*. For a general part, the centerline would contain one or more curves. As we need it to contain only one curve, which is a straight or only slightly curved line, we will compute only with convex parts (how to deal with the non-convex parts is described in Section 4.3.5). If we project the convex part to the plane perpendicular to such centerline, we achieve the best projection we can.

However, computing a medial axis would be inadequately time consuming, and also we do not have fully convex parts, so there would be even more artefacts, and we would probably not be able to automatically decide which vertices to use. So we need to find some approximation of the centerline.

Our approximation uses the topological distances of the vertices. If we compute a gravity point g_i of vertices with the same topological distance d_i, we get a vertex somewhere near the centerline. So computing such a gravity point for each topological distance would give us an approximation of the whole centerline. As was already said, the vertices of the centerline should lie on one line for the convex parts. Therefore the computed gravity points should lie approximately on one line too. So we can use only two of them to compute the line we need. We decided to use the gravity points g_0 and g_m, where $m = \lceil \frac{max}{2} \rceil$, max is the maximal topological distance of the part. We did not use the gravity point at the maximal topological distance d_{max} itself, because vertices with $d = d_{max}$ can be only at one side of the part and produce a gravity point lying too far from the centerline. We use the maximal topological distance only if $max = 1$.

The difference in the resulting morph when the projection plane is chosen from three arbitrary vertices of the intersection chain (as was in 4.3.2) and when the plane is perpendicular to our approximation of centerline is shown in Figure 4.32. Notice that in the example, the part $C_{in} = \emptyset$. We will involve the part C_{in} in our computations now.

So we know the approximate centerline of the part C_{out}. If the part C_{in} also contains some vertices, we can compute the approximate centerline also for C_{in}. But we cannot project each part on a different projection plane. A simple solution is to compute an average between the two normal vectors of the planes. If angle of the two normal vectors is obtuse, we reverse the direction of one of the vectors and compute the average after this reversion.



(a) a plane defined by three arbitrary vertices of the intersection chain

(b) a plane perpendicular to the centerline

Figure 4.32: The resulting morph is highly dependent on the choice of the projection plane: projected part (red), core (grey), part (green), resulting morph sequence (brown)

4.3.4 Parts with more than one intersection chain

As was already discussed, the part does not always contain only one intersection chain. An example of a part with two intersection chains was given in Figure 4.20. However, also a part with more than two intersection chains can occur. Such parts must be handled differently than the ones with one intersection chain because of a problem with projection - with one intersection chain, we are able to find a projection plane, where each part projects in a way that its edges do not intersect in the projection. However, we are not able to project parts with more than one intersection chain to a plane to satisfy such requirement. One solution would be to project such parts on a sphere with a centre in the gravity point of the part. To follow our technique, we would then need to construct a Voronoi diagram on the sphere and assign the vertices as was described above. Or better, use the technique from [1] to find the correspondence of the part's vertices by projecting onto the sphere. To non-violently follow our technique, we decided to find a projection plane and split the part by the plane into two subparts, which are handled separately by the method for parts with one intersection chain. The normal vector of the projection plane is found in the same way as was described in the previous chapter.

4.3.5 Dividing the part into convex or star-shaped parts

When computing the projection plane, we presumed that the part was convex, but we would like our algorithm to be usable to more general parts. If a part P_i is not convex, it can be divided into several convex parts $P_{i0}, \ldots, P_{in-1}$. Than we construct a graph of the created parts: Each part P_{ij} represents a node of the graph, and the parts that are neighbours are connected by an edge. The core is represented by a

node connected to the parts neighbouring it.

Then we compute a topological distance of each part with respect to the core (which has a topological distance of 0). We start with the part with a maximal topological distance, compute the projection plane and project the vertices to the plane. The projected coordinates are only put to the vertices' paths, and the projected vertices are now considered as the vertices of the part's neighbour in the graph. In such a way, we project all the vertices of the parts with the topological distances greater than one, and then we have one or more parts with a topological distance of one. We will handle such parts together, because otherwise, we would have to decide which part corresponds with which part of Q_i. The rest parts are handled the same way as were the convex parts. The whole process is sketched in Figure 4.33, for simplicity in 2D.

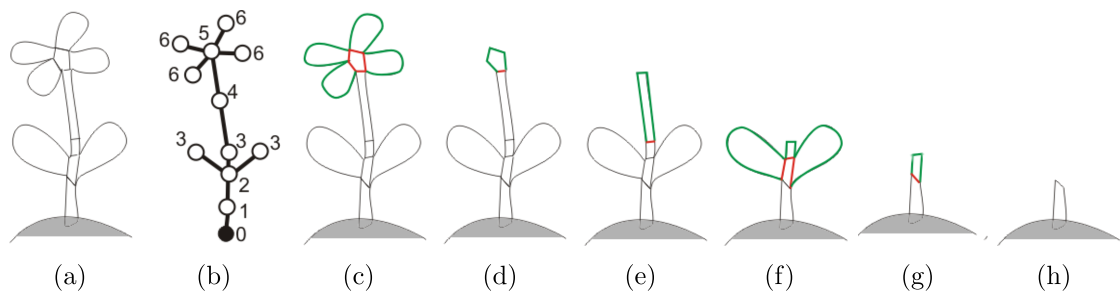


Figure 4.33: Projecting a non-convex part by dividing it into convex pieces: (a) original part (grey: core), (b) topological distances of the parts, (c-g) projecting the parts, (h) the resulting convex part

5 Results

First, results from the 2D algorithm[2]will be discussed as well as their comparison with the Sedeberg's and Greenwood's algorithm[3] [17] and with the Carmel's and Cohen-Or's algorithm[4] [5], well-known correspondence-based algorithms. In the next part, the results from the 3D algorithm[4] will be shown along with their evaluation. Other interesting results in 3D are in D.

5.1 Results for polygons

In the following examples we will demonstrate a behaviour of different methods of vertex path computation of our algorithm for different shapes. Although it is possible to morph each part of a shape independently, in our examples we will morph all parts using only one method, so that we can clearly demonstrate its suitability for the given shape and the given type of effect.

As already mentioned, our algorithm is completely suitable for the cases when one would expect some parts of the polygon B to grow out of the polygon A (or some parts of A disappearing in B). Those parts can be horns, prickles, fingers or tails, usually in situations when the user wants them to appear (grow out), or disappear in something. However, our algorithm is not suitable for similar polygons which are only transformed (e.g., moved, rotated, scaled), where the user expects the polygon only to move according to the transformation, not to change its shape.

Examples 5.1.1 and 5.1.2 show the case where the user expects some parts of the polygon to grow or to disappear, the example 5.1.3 shows morphing of completely different polygons.

5.1.1 Parts of a spiral type

For the shape of a spiral type, the Perimeter (Figure 5.1) or the Half-line growing are better then the Projection (Figure 5.2), where the result contains many self-intersections.

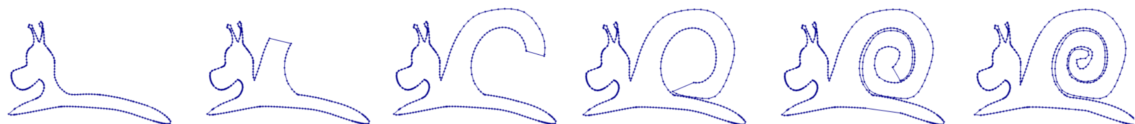


Figure 5.1: The Perimeter growing

[3]An implementation by P. Celba, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, `http://iason.zcu.cz/~kolinger/vyukaUK.html/Metamorfoza_Celba.zip`

[4]An implementation obtained from `http://w3.impa.br/~morph/software/softw-2d-morphing.html`

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

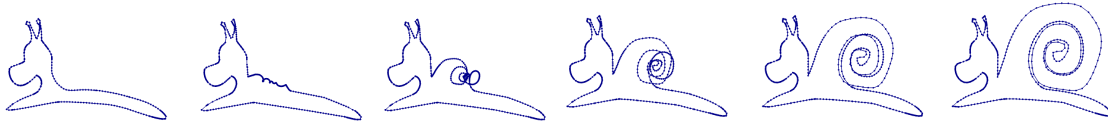


Figure 5.2: The Projection growing

Although the Projection growing is not producing results we would expect when we want something of a spiral type to grow out of the core, it can sometimes give interesting aesthetical results when we fill the polygons by some color (because the overlapping parts have the color of the background).

Carmel's and Cohen-Or's algorithm (Figure 5.3) and Sedeberg's and Greenwood's algorithm (Figure 5.4) give results containing many self-intersections here.

Figure 5.3: The Carmel's and Cohen-Or's algorithm

Figure 5.4: The Sedeberg's and Greenwood's algorithm

5.1.2 Convex parts

For parts which are convex or nearly convex the Projection growing (Figure 5.5) is more suitable then the Perimeter or Half-line growing (Figure 5.6).

In the example of a butterfly and an alien, the body of the butterfly is similar to the alien, so one would probably expect the wings of the butterfly to disappear in the body of the alien, and the eyes (at the end of the antenna) to grow out from the antenna of the butterfly.

Both Carmel's and Cohen-Or's algorithm (Figure 5.7) and Sedeberg's and Greenwood's algorithm (Figure 5.8) result in many self-intersections in this case.

5.1.3 Long and more or less straight parts

When the shape of the parts is long and more or less straight, the Projection (Figure 5.10), Perimeter and Half-line (Figure 5.9) growing have results of a similar

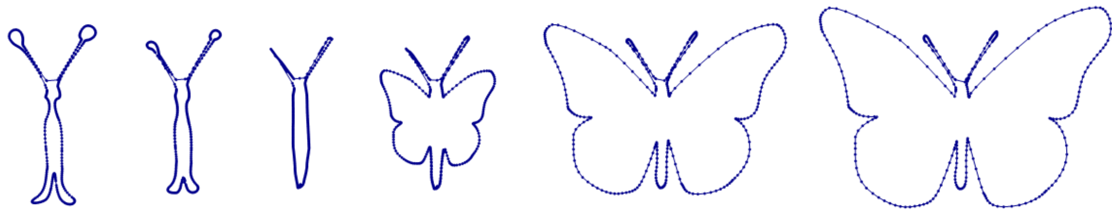


Figure 5.5: The Projection growing

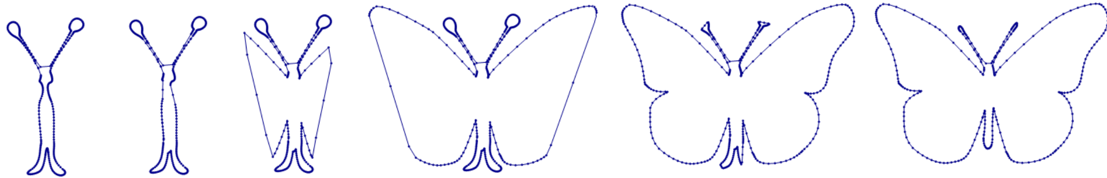


Figure 5.6: The Half-line growing

Figure 5.7: The Carmel's and Cohen-Or's algorithm

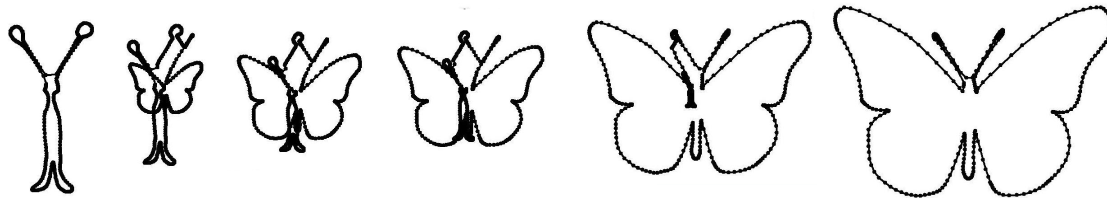


Figure 5.8: The Sedeberg's and Greenwood's algorithm

quality. In this example, such a case appears at octopus's fingers. For the other parts, the Projection growing appears to be more suitable.

The case shown in this particular example is not the case, where one would naturally expect the growing behaviour, because the octopus and the shark do not have any similar part. However, growing of the octopus's fingers is probably the only way how to morph from the shark's stomach into them without an intersection.

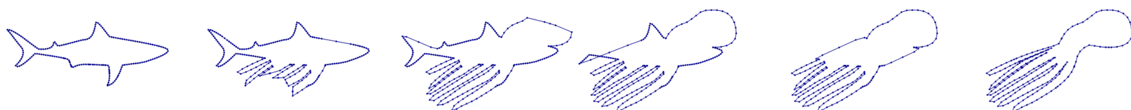


Figure 5.9: The Half-line growing

The result Carmel's and Cohen-Or's algorithm (Figure 5.11) is quite similar as our growing algorithm for the case of octopus's fingers. The Sedeberg's and

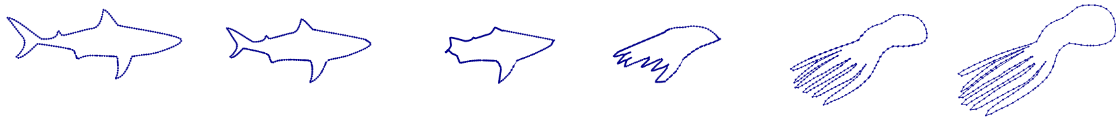


Figure 5.10: The Projection growing

Greenwood's algorithm (Figure 5.12) experiences a few intersections here.

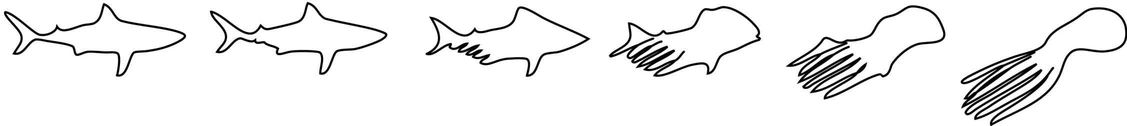


Figure 5.11: The Carmel's and Cohen-Or's algorithm

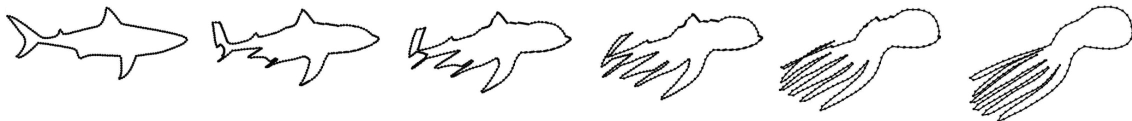


Figure 5.12: The Sedeberg's and Greenwood's algorithm

5.1.4 Summary

In the previous three examples we showed that our algorithm produces expected results for the case when some parts of one polygon are supposed to grow out of the other polygon or disappear in it. However, as we can see in the last example (Section 5.1.3), the algorithm can be also used in some cases where growing is not expected and still produce acceptable results. On the other hand, it is not very suitable for the polygons that are similar or nearly similar with dissimilarities of non-growing type (like faces with different expressions, bent and straight finger etc.).

5.2 Results for meshes

In the following examples, we will demonstrate a general behaviour of the Perimeter and Projection growing in 3D.

5.2.1 Convex parts

If the parts are about convex, the Projection growing produce better or about the same results as the Perimeter growing. There are two things making the results of the Perimeter growing worse - first, the case discussed already for 2D - that it "flattens" the top (in the sense of the highest topological distance) of the part during

the process. Second, its results are dependent on the distribution of the vertices and therefore their topological distances. It can produce different results for about the same parts, as is shown in Figure 5.13 - the figure's hands morph nicely, however, the resulting morph of its legs, which are quite similar as the hands, is worse. The use of Projection growing for the same input is shown in Figure 5.14.

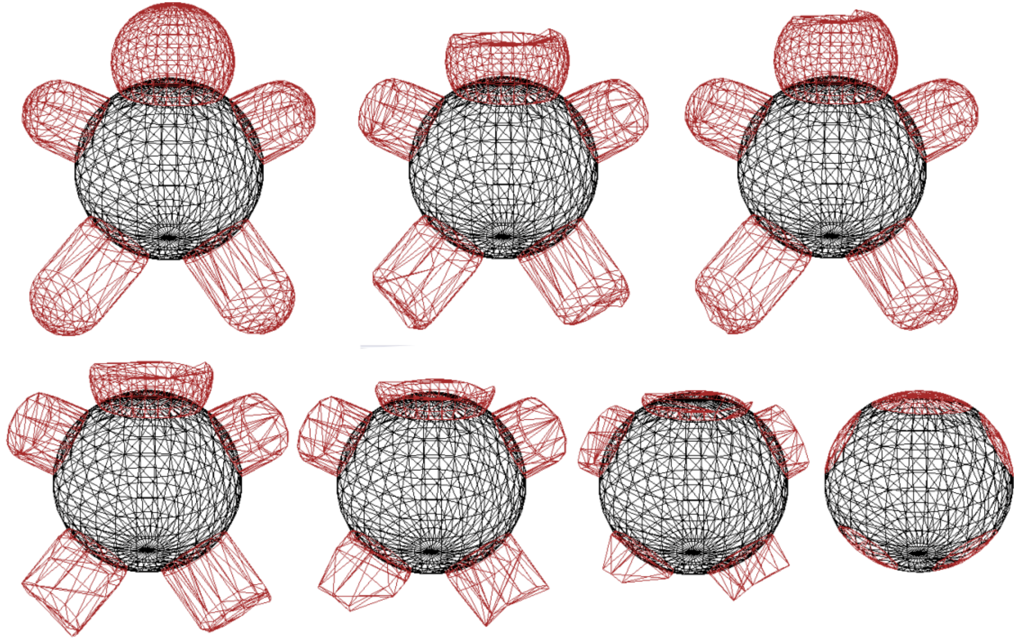


Figure 5.13: The Perimeter growing

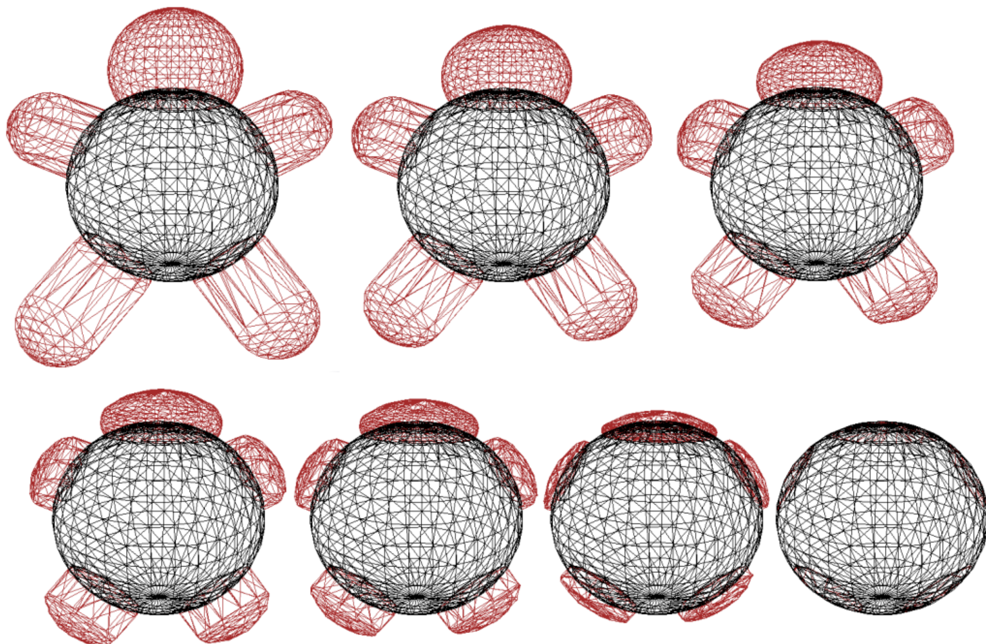


Figure 5.14: The Projection growing

5.2.2 Parts whose centerline contains only one curve

If the centerline contains only one curve, it means that the part is convex, or that it is somehow curved, but does not have any branches. In such a case, the Perimeter growing is the best choice, containing none or not much intersections. Figure 5.15 shows an example of morphing between a piece of a rope (M_0) and the whole rope (M_1) - the core and M_0 are identical, therefore there is only one part (part of M_1) to morph out of the core. The Projection growing in its raw form that was presented produces many self-intersections in this case (Figure 5.16). That is because our implementation expects a part, which is about convex, and searches the best projection plane and projects onto it based on this expectation. However, a different projection plane or a different type of projection would not help here to remove the self-intersections. We would need to split the part into several (about) convex pieces as was discussed in Section 4.3.5 to achieve a result without self-intersections or with a small number of them.

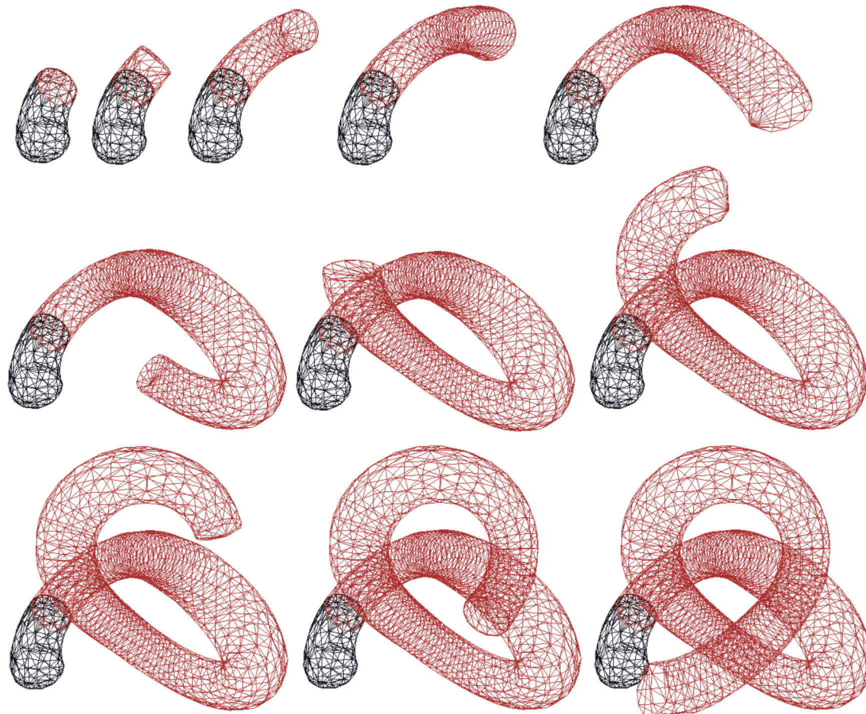


Figure 5.15: The Perimeter growing

5.2.3 Star-shaped part

If the part is star-shaped, the Projection growing has usually better results than the Perimeter growing, which flattens the tops of the convex pieces of the star-shaped part, or can produce some unwanted connections between the neighbourhood pieces. The Figures 5.18 and 5.19 shows an example of about star-shaped part (the part in Figure 5.17d), where the Perimeter growing produces some unwanted edges between

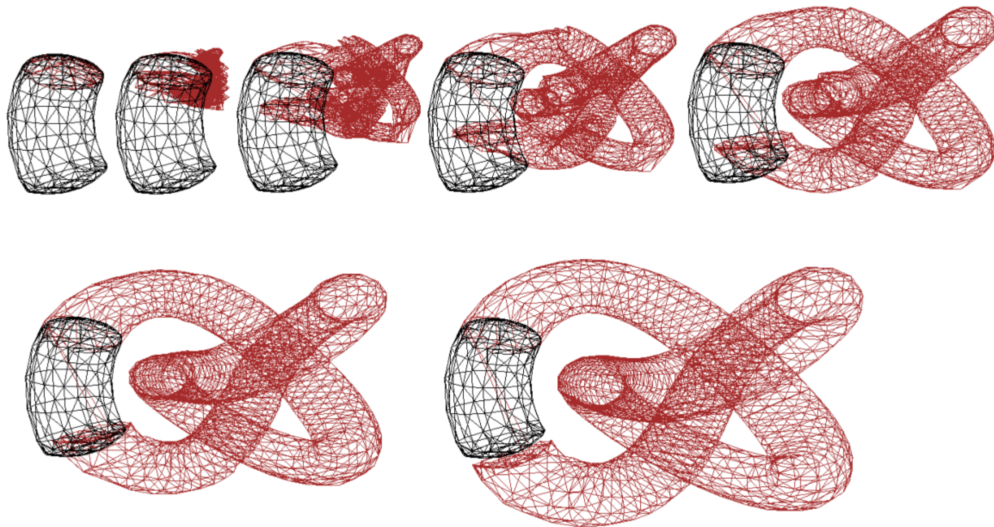
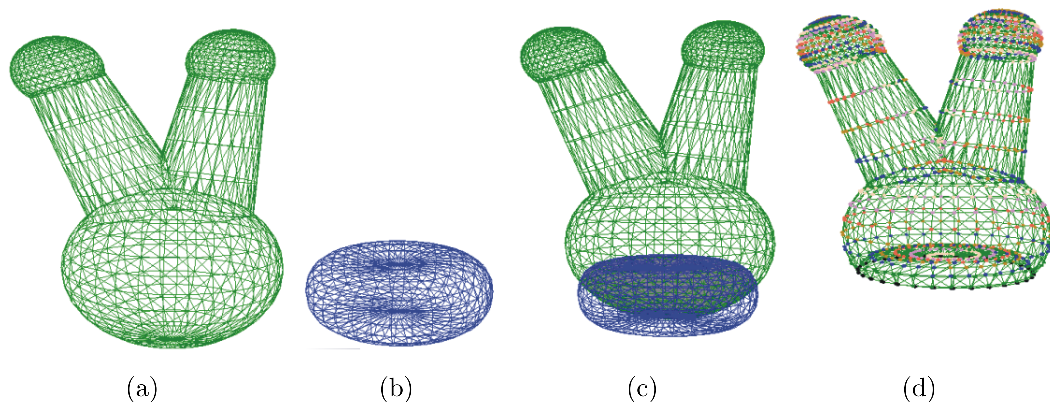


Figure 5.16: The Projection growing

the two convex pieces, and the Projection growing produces self-intersections in the area of the balls situated at the top. In case the balls would not be there, the result would be satisfying enough.

5.2.4 Non-convex part, having a convex inside and convex outside

The Figures 5.18 and 5.19 also describe a case of a part, where its outside is convex, its inside is convex, but the part together is not convex (Figure 5.17e), because the outside and the inside are on the same side from the intersection chain. In this case, the Projection growing is definitely better, because when we use the Perimeter growing, the vertices first travel towards the intersection vertices, and then back to the vertices of the outside part. However, when we use the Projection growing, the vertices of the outside part travel straight towards their corresponding vertices of the inside part.



(a) (b) (c) (d)

Figure 5.17: (a,b) the input meshes (c) mutual position of the meshes, the core is filled with gray (d,e) the toplogical distances of the part's vertices (black-intersection vertices)

The Figure 5.18 also shows another interesting fact about the Perimeter growing. That is what always appears in its result, when the two meshes share one core, and after subtracting the core from the meshes, each one of them consists only of one part. At a time $t = 0.5$, the vertices of the first part are at the intersection chain as well as the vertices of the second part. Because the parts share the same intersection chain, that results in a mesh defined only by the vertices of the intersection chain, in this case a flat area bounded by the intersection chain. This result can be affected by using a slightly different time periods for the parts to make the vertices not meet at the intersection chain.

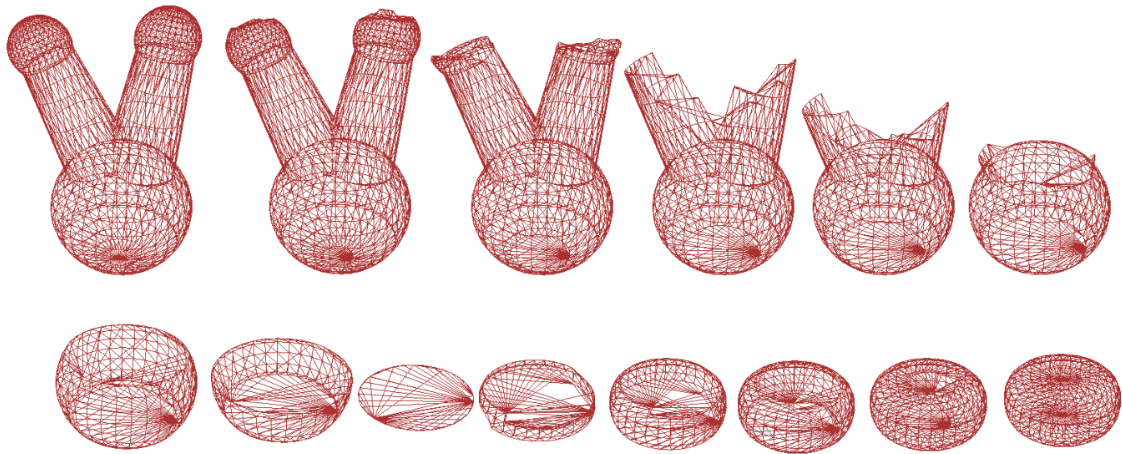


Figure 5.18: The Perimeter growing

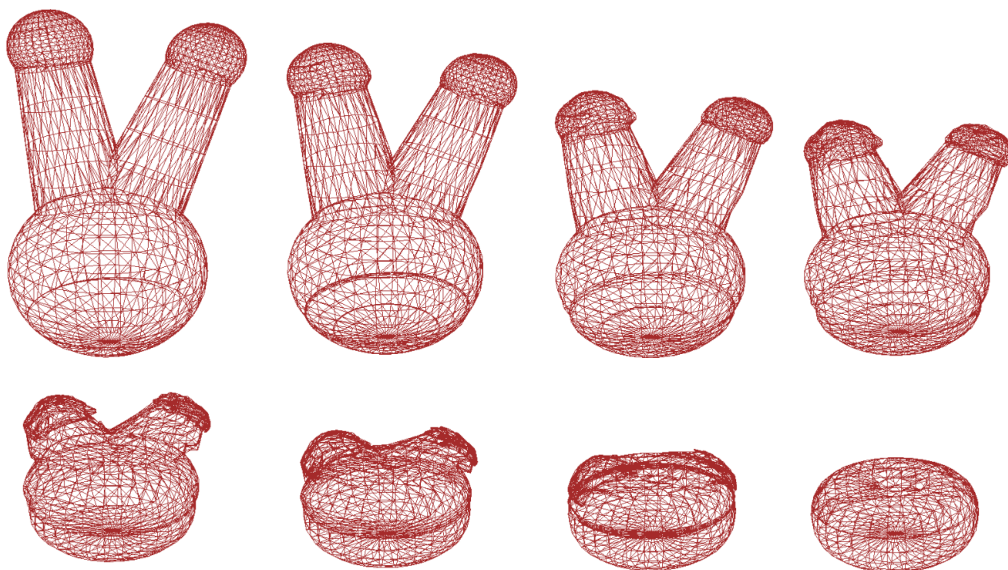


Figure 5.19: The Projection growing

If the part is of some other, more complicated shape, but contains only one intersection chain, the Perimeter growing is usually better than the raw implementation of the Projection growing. If the part contains more intersection chains, the decision is even harder and depends on the exact shape of the mesh.

5.2.5 Summary

We have shown that the Perimeter growing produces best results for long and curved shapes, but can be used also for the convex shapes and more complicated shapes containing only one intersection chain. The Projection growing produces best results for convex shapes, because it was designed for them. However, it can be extended by cutting the non-convex shapes into convex pieces, or by using a different projection method (i.e. perspective projection). A disadvantage of the Perimeter growing is its dependence on the distribution of the vertices.

6 Conclusion

We have presented a novel algorithm for morphing geometrical objects in boundary representation. The algorithm is different from the others in its field in a way that it does not compute the correspondence between the whole two objects, but it first computes an intersection of the objects, subtracts the intersection from the original objects and so it decomposes them into several parts lying outside the intersection. Then it computes the correspondence separately for each part - correspondence between the vertices lying outside the intersection and the vertices lying on the intersection.

Although we did not avoid the correspondence computation as was first expected, we can base our computation on a knowledge that the two pieces do not intersect, and share a set of vertices (the intersection chain). We presented three methods for the correspondence computation. Along with the correspondence, the methods deal with the vertex path problem, a problem that is usually solved by only linearly interpolating between the corresponding vertices. Our methods construct the vertex paths containing several vertices describing the vertex positions during the time. This way, we can produce a not self-intersecting results also for the curved parts, where the simple linear interpolation would be useless for any combination of corresponding vertices.

The experiments confirmed that our algorithm is suitable for the cases, where the user expects some parts of the object to grow out from the intersection or disappear in it. Our algorithm can be suitable also for some other than grow-like cases, but it is not useful for the objects that are of the same shape and they are only transformed (rotated, translated etc.).

6.1 Future work

There are still some parts that are not completed in our 3D algorithms. The Ahn's algorithm [1] should be implemented to continuously change the connectivity between the two in-between meshes in Perimeter growing. The same algorithm should be used to handle the connectivity in the Projection growing, so that its result for the pieces lying on the core are precise, not only approximations as it is now. And last, the merging algorithm should be established for the 3D parts.

In both 2D and 3D, the projection method could be improved to produce better results for the non-convex parts by splitting the part into about convex pieces, as was discussed in Section 4.3.5.

The intersection of the two input objects has been computed manually in 3ds max 7, so part of the future work will be to implement this computation to be able to change the positions of the input objects in our program.

References

[1] M. Ahn, S. Lee, and H. Seidel. Connectivity transformation for mesh metamorphosis. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 75–82, New York, NY, USA, 2004. ACM.

[2] M. Alexa. Merging polyhedral shapes with scattered features. *The Visual Computer*, 16(1):26–37, 2000.

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

[4] M. Alexa, D. Cohen-Or, and D. Levin. As-rigid-as-possible shape interpolation. *Computer Graphics Forum*, pages 157–164, 2000.

[5] E. Carmel and D. Cohen-Or. Warp-guided object-space morphing. *The Visual Computer*, 13:465–478, 1997.

[6] D. Cohen-Or, A. Solomovic, and D. Levin. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics*, 17:116–141, 1998.

[7] N. D. Cornea, D. Silver, and P. Min. Curve-skeleton applications. *IEEE Visualization Conference (VIS)*, page 13, 2005.

[8] A. B. Ekoule, F. C. Peyrin, and C. L. Odet. A triangulation algorithm from arbitrary shaped multiple planar contours. *ACM Trans. Graph.*, 10(2):182–199, 1991.

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

[10] S. Hanke, T. Ottmann, and S. Schuierer. The edge-flipping distance of triangulations. *j-jucs*, 2(8):570–579, aug 1996.

[11] H. Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM.

[12] J. K. Johnstone and X. Wu. Morphing two polygons into one. *40th Annual Southeast ACM Conference*, 2002.

[13] J. R. Kent and R. E. Carlson, W. E.and Parent. Shape transformation for polyhedral objects. *Computer Graphics*, 26:47–54, 1992.

[14] D. Levin. Multidimensional reconstruction by set-valued approximation. *Clarendon Press Institute Of Mathematics And Its Applications Conference Series, Algorithms for approximation*, pages 421–431, 1987.

[15] M. Málková. A new core-based morphing algorithm for polygons. Proceedings of CESCG, 2007.

[16] 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.

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

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

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

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

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

[22] G. Voronoi. Nouvelles applications des paramétres continus á la théorie des formes quadratiques. *Journal fur die Reine and Angewandte Mathematic*, 133:97–178, 1907.

[23] K. Weiler and P. Atherton. Hidden surface removal using polygon area sorting. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 214–222. ACM, 1977.

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

A Proceedings of CESCG 2007: A new core-based morphing algorithm for polygons

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.

*tina.malkova@centrum.cz

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

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.

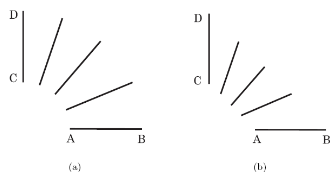


Figure 1: An expected morphing sequence (a) and a morphing sequence using linear interpolation (b), from [3]

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.

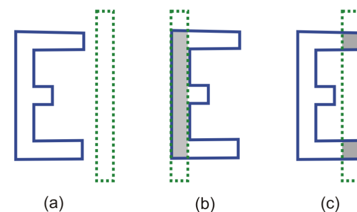


Figure 2: Different number of core parts (Continuous line: source polygon, dotted: target polygon, grey: core)

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.

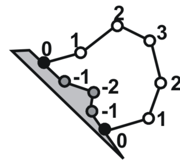


Figure 3: Topological distances of vertices (White: vertices outside the core, black: intersection vertices, grey: vertices inside the core, light grey: part of the core)

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.

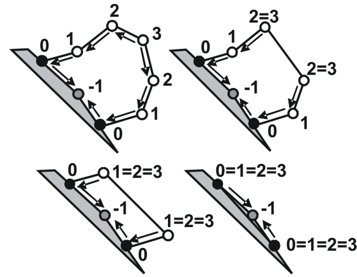


Figure 4: The perimeter algorithm

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.

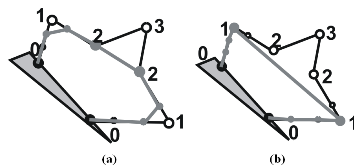


Figure 5: The perimeter algorithm with constant time (Grey: the current polygon and positions of the vertices)

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).

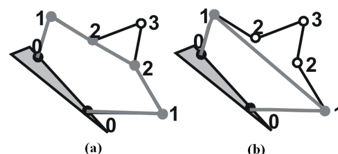


Figure 6: The perimeter algorithm with constant velocity (Grey: The current polygon and positions of the vertices)

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.

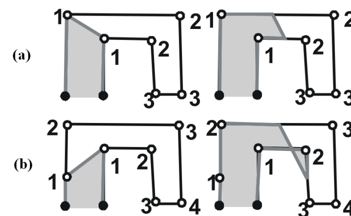


Figure 7: Self-intersection (Bottom: extra point added; grey: the current polygon)

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.

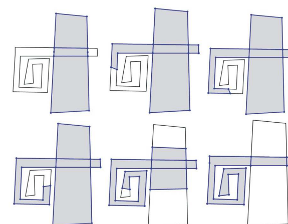


Figure 8: The perimeter algorithm with the const. velocity (Grey: the current polygon, black: source and target polygons)

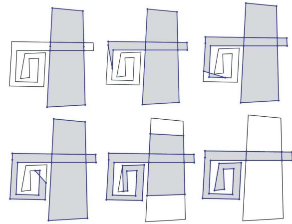


Figure 9: The perimeter algorithm with the const. velocity - an extra point added

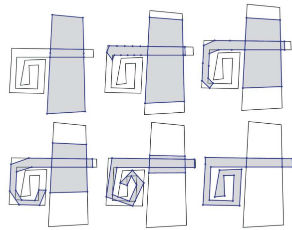


Figure 10: The perimeter algorithm with the const. time

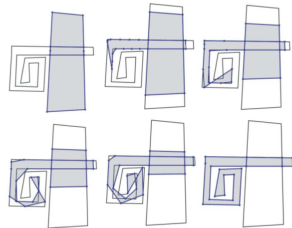


Figure 11: The perimeter algorithm with the const. time - an extra point added

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.

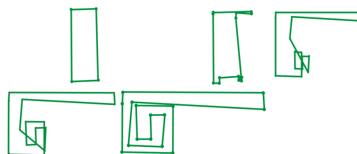


Figure 12: Sederberg's algorithm

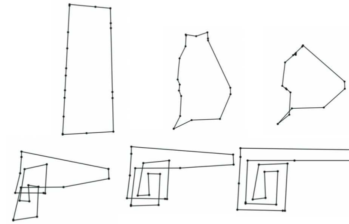


Figure 13: 2D merging algorithm

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.

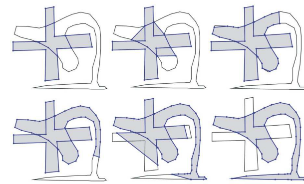


Figure 14: The perimeter algorithm with the const. velocity

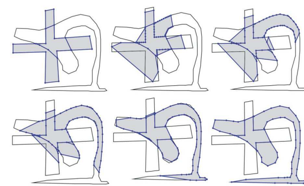


Figure 15: The perimeter algorithm with the const. time

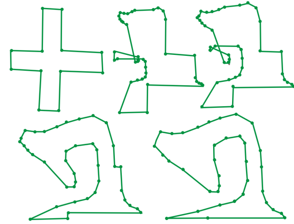


Figure 16: Sederberg's algorithm

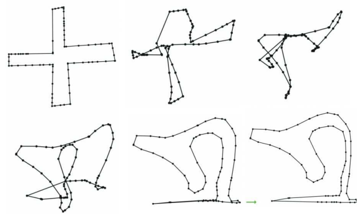


Figure 17: 2D merging algorithm

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).

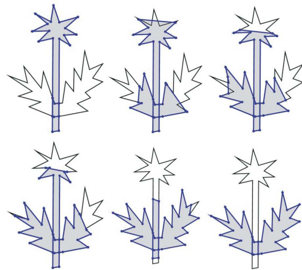


Figure 18: The perimeter algorithm with the const. velocity

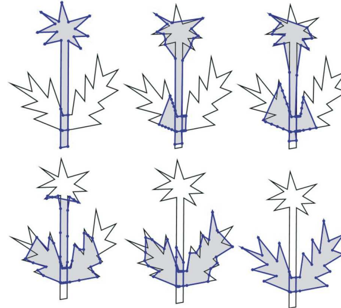


Figure 19: The perimeter algorithm with the const. time

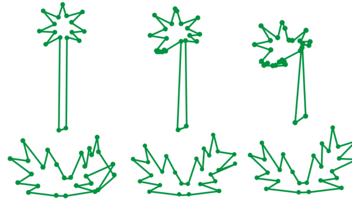


Figure 20: Sederberg's algorithm

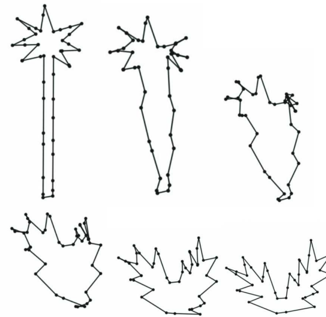


Figure 21: 2D merging algorithm

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.

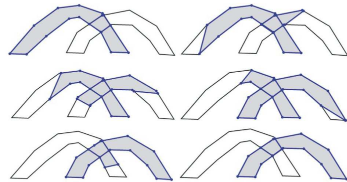


Figure 22: The perimeter algorithm with the const. velocity

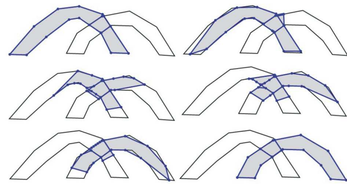


Figure 23: The perimeter algorithm with the const. time

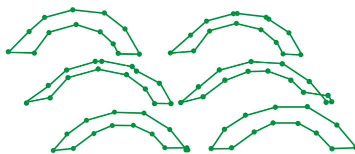


Figure 24: Sederberg's algorithm

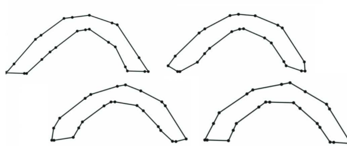


Figure 25: 2D merging algorithm

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.

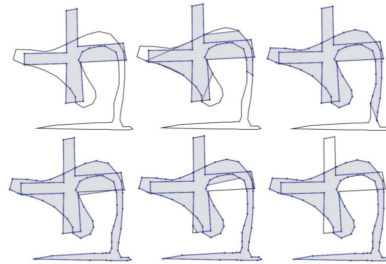


Figure 26: The perimeter algorithm with the const. velocity

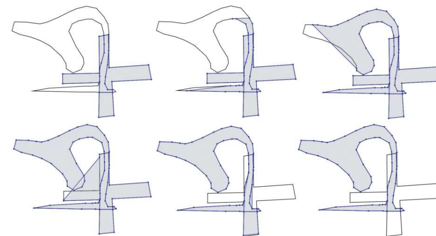


Figure 27: The perimeter algorithm with the const. velocity

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.

B User manual

2D program - PolygonMorphing.exe

Running the program

To successfully run the program, you need the following files:

- *PolygonMorphing.exe* (the program itself)
- *AviFile.dll* - a library for exporting resulting animations to *.avi
- *gpc.dll* - a library for computing boolean operations on polygons

Main window

After successfully running the program, the program's main window appears (Figure B.1). The numbered items are described in the following table.

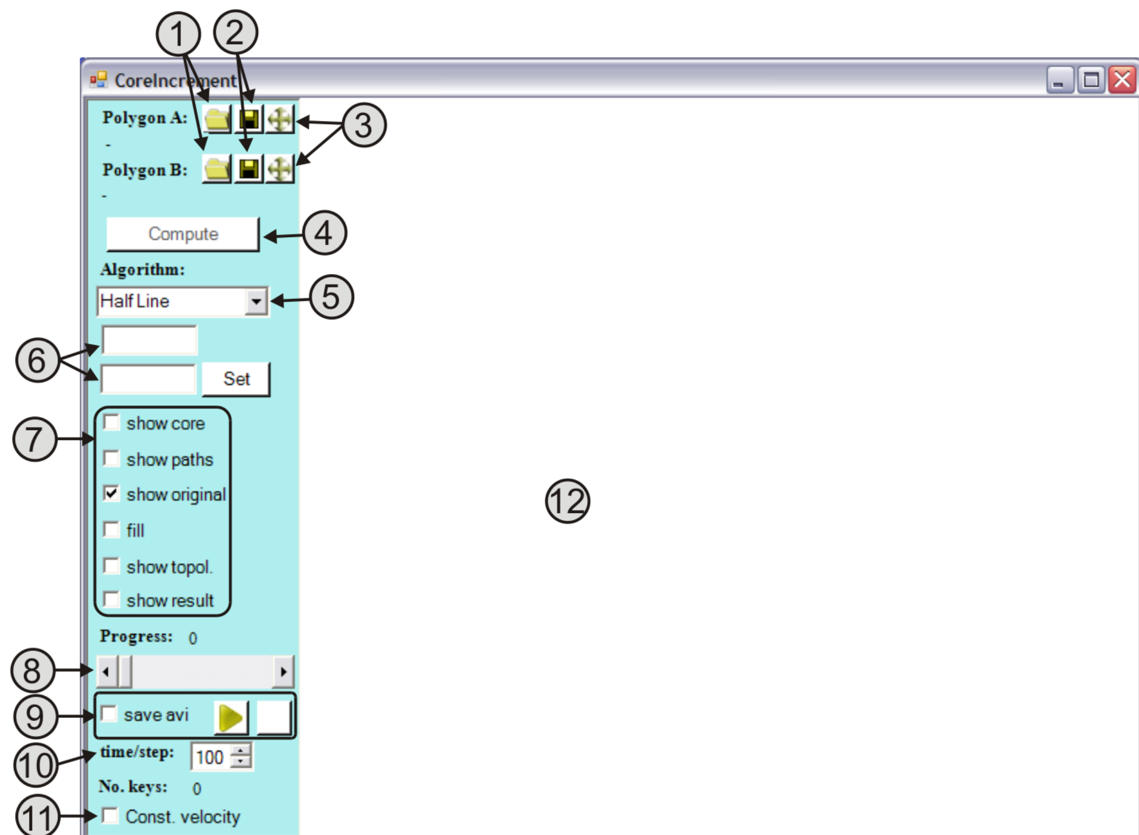


Figure B.1: The program's main window

number	description
1	load input polygons (Section B)
2	export polygons into a different format (Section B)
3	enable moving the polygon by mouse dragging
4	compute the resulting morph by a method in (5)
5	select method for the vertex path computation
6	mix different methods for one set of input polygons (Section B)
7	influence what is rendered on the canvas (12)
8	animation scroll bar, the current time is shown above the bar
9	playing exporting an animation (Section B)
10	influence the precision of the animation scroll bar (8)
11	if checked, all the vertices travel at the same time
12	the canvas where everything is rendered

Input and output files

As input files, our program accepts files in .pol and .ps formats. It can export the input polygons into .pol, .txt and .cel.

.pol format
.pol format is a simple text file, where on the first line is the number of vertices of the polygon, on the second line is "closed" if the polygon is closed, and on the rest lines, there are coordinates of the polygon's vertices.

```
70
closed
-6.20689651770484E-0001 -3.90804597701149E-0001
-6.66666663012742E-0001 -4.65517241379310E-0001
-6.80839640679162E-0001 -5.10870770160431E-0001
-6.95402295039153E-0001 -5.57471264367816E-0001
...
```

.ps format
.ps format is also a text file, but can be quite complicated. Our program accepts only a file with one polygon, where some of its line segments do not have to be lines, but also curves. However, the curve is considered as a line with the curve's endpoints. Therefore, you need to refine the curves before exporting from the program where the polygon was created (we used a free drawing program *Inkscape* to create our input polygons).

```
0 842 translate
0.8 -0.8 scale
0 0 0 setrgbcolor
[] 0 setdash
1 setlinewidth
0 setlinejoin
0 setlinecap
```

```
gsave [1 0 0 1 0 0] concat
0 0 0 setrgbcolor
[] 0 setdash
2.363097 setlinewidth
0 setlinejoin
0 setlinecap
newpath
65.350638 335.53387 moveto
65.350638 335.28149 65.350638 333.82109 65.350638 331.54657 curveto
65.350638 329.27204 65.350638 326.18335 65.350638 322.67437 curveto
65.350638 319.16538 65.350638 315.23609 65.350638 311.28034 curveto
...
```

polygons in .txt format
Polygons in .txt format are required by the implementation of [5]. The file is almost similar to our .pol.

```
##_POLYGONAL_CURVES_FILE

Number of vertex:
74
258.000000 385.000000
252.000000 379.000000
247.000000 371.000000
245.000000 365.000000
...
```

.cel format
Polygons in .cel format are required by the Celba's implementation of [17]. The format is also nearly similar - first line contains the number of vertices, the following lines the vertices themselves, only as integers.

```
341
486 358
486 358
486 358
487 359
...
```

Running and exporting an animation

To run the animation, you simply push the button "play" in the animation field (Figure B.1(12)). If you also want the animation to be exported into .avi format, you need to check "save avi" checkbox before clicking the button. The resulting file will be saved in the same directory as the input polygons are located.

The button right to the "play" button is used to export the animation as a sequence of six images. If you want such a result, first you need to define six *keys* - values of time used to render the results. You do so by scrolling the animation scroll bar and pressing 'K' whenever you want the time to be saved as a key. If you change your mind, you can reset the whole sequence by pressing 'R'. The number of keys entered is shown in the left bottom corner of the window. When you have entered the sixth key, you can view the resulting image by pushing the white button.

Mixing different algorithms for different parts

The part mixing is not very user-friendly. There are two input fields, the first for the parts of the first input polygon, the second for the second one. In each field, you need to put a sequence of numbers delimited by semicolons. Each i-th number denotes the method for computing the vertex part of i-th part. The method's numbers are as they appear in the select box: 0-Half-line, 1-Perimeter, 2-Projection, 3-One step projection. So if you put in for example the sequence `0;0;1;0`, the third part will be computed by the Perimeter growing, and the rest by the Half-line growing method.

3D program - CoreMorph.exe

Running the program

To successfully run the program, you need the following files:

- *CoreMorph.exe* (the program itself)
- *FortuneVoronoi.dll* - a library for computing 2D Voronoi diagrams
- *sphere.WRL* - coordinates of a unit sphere
- *input files* - files containing the input objects, the sets of input objects are in the directory *objects* and was computed in 3ds max 7.
  - *a.WRL, b.WRL* - the input objects themselves
  - *core.WRL* - the intersection of the objects
  - *a0.WRL,...,a[n-1].WRL* - n distinct parts (result of $A - core$)
  - *b0.WRL,...,b[m-1].WRL* - m distinct parts (result of $B - core$)

Main window

After successfully running the program, the program's main window appears (Figure B.2).

On the canvas, the intersection (black) and the result computed by the Projection growing (brown) is shown. The scene on the canvas can be rotated by holding the

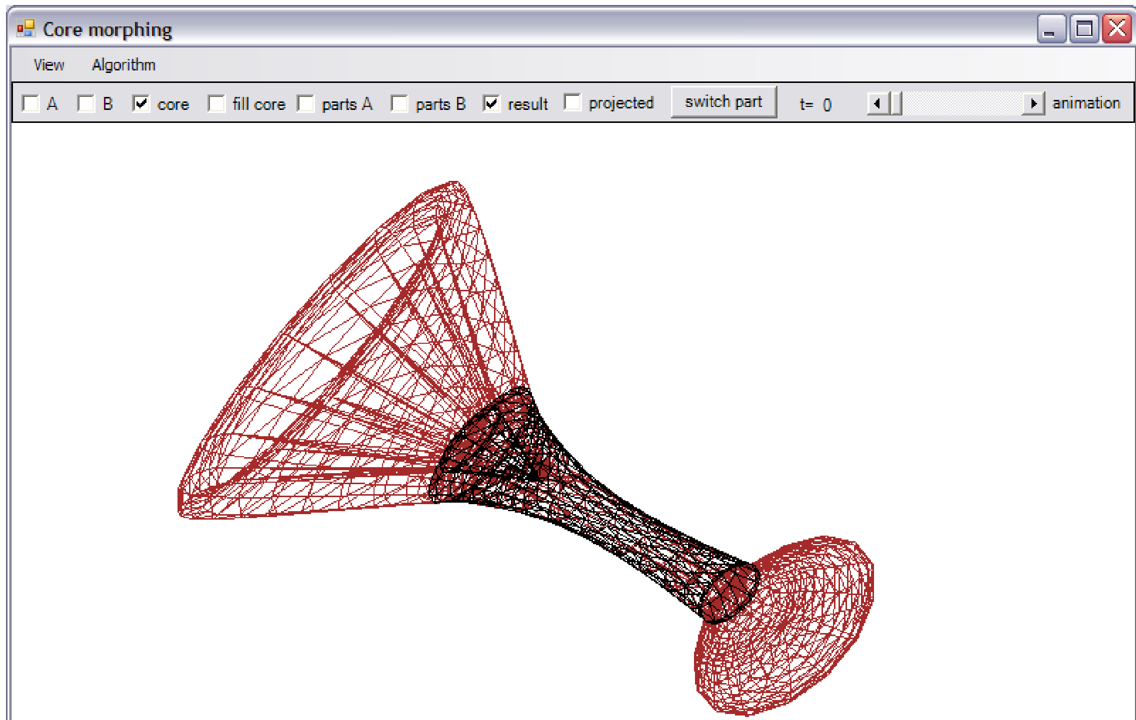


Figure B.2: The program's main window

right mouse button and moving the mouse, translated by holding the left mouse button and moving the mouse (or using the $ASDW$ keys), and scaled by scrolling the mouse wheel (or using the +- keys).

By using the checkboxes in the toolbar, we can influence what will be shown on the canvas (from the left):

- A - the first input object
- B - the second input object
- *core* - the intersection
- *fill core* - if the intersection will be filled
- $partsA$ - the input parts $a0, a1, ..., a[n-1]$
- $partsB$ - the input parts $b0, b1, ..., b[m-1]$
- $result$ - the resulting morph
- $projected$ - this checkbox belongs to the Projection growing method. It shows the projected part $a0$ (or $b0$ if $a0$ does not exist). The showed projected part can be switched by the button *switch part*.

In the right part of the toolbar, there is an animation slide, which is used to influence the time. At the time 0, the resulting object should be A, at the time 100 it should be B. The current time is shown to the left of the slide.

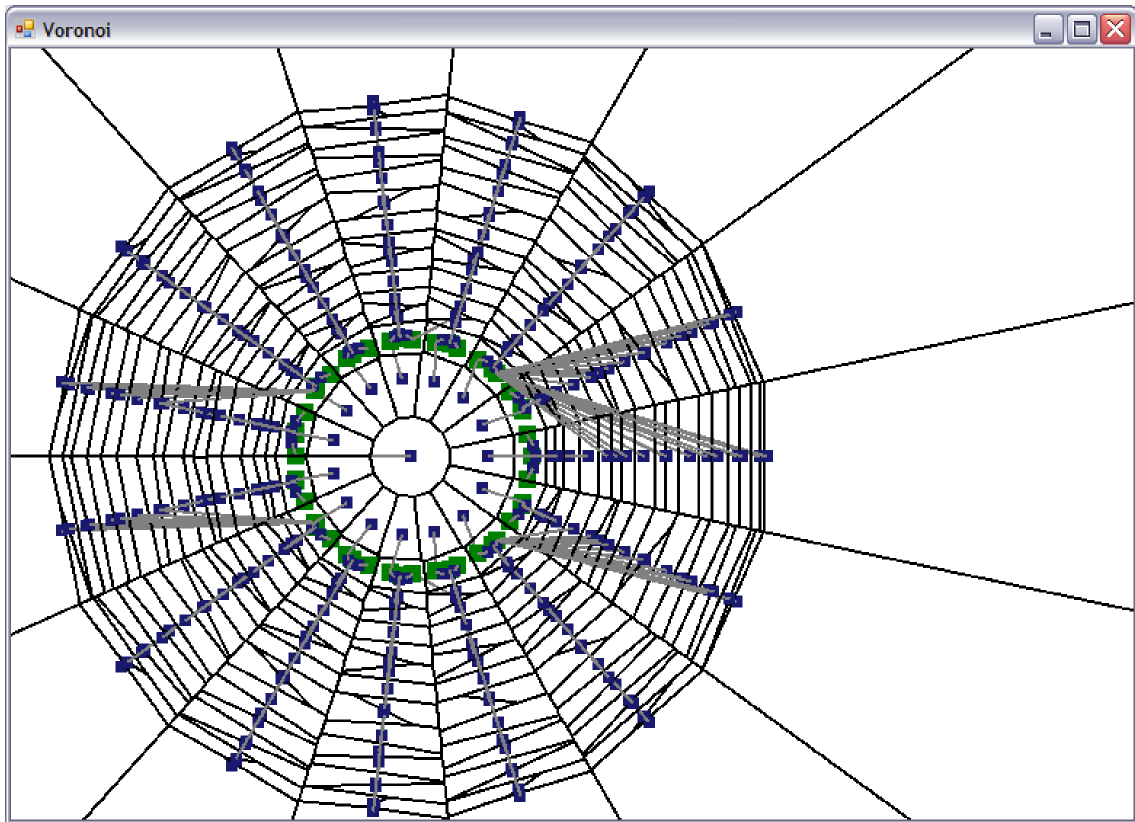


Figure B.3: Window with Voronoi diagram

The window's main menu consists of the following:

- *View*
  - *Voronoi diagram* - shows a new window (Figure B.3 with computed Voronoi diagram for the current part (as was discussed for the check-box *projected*, you can switch the displayed part by the button *switch part*). In the Voronoi diagram, the blue points are the projected vertices of the outside part (lying outside the core), the green points are the projected vertices of the inside part (lying on the core), the grey line segments connect the corresponding vertices, the black line segments show the voronoi diagram of the outside part.
  - *Algorithm* - here you can switch between the Projection growing and the Perimeter growing. Notice, that for the Perimeter growing, no projection plane or Voronoi diagram are computed, the projected parts with their Voronoi diagrams shown are computed from the Projection growing.

C Implementation

Two applications were created, the former on the 2D version on our algorithm, the latter on its 3D version. Both programs were written in C#, using Microsoft Visual Studio 2005.

2D version

Clipping

Clipping polygons is done by a *C* library *gpc.dll*[5], which has a port for *C#* - class *GpcWrapper*.

Data structures

The program works with clipped polygons, which are saved as an object *MyPolygon*, containing a list of objects *Part*. Each part stores its minimal and maximal topological distance, and a list of its vertices. Each vertex then stores its coordinates, topological distance and its vertex path, which is implemented as a list of coordinates. For the data to be consistent, each vertex have the vertex path, but those with a negative topological distance do not have any items in it.

Visualization

The visualization is done using the *GraphicsPath* object, to be independent on *DirectX*. That brings problems when exporting the animation to .avi, because when using DirectX, we can export its output directly to files, but when using the GraphicsPath object, we do not have a direct access to the output. This is not solved very professionally, because the program screenshots and saves the window.

3D version

Clipping

Because we did not found any *C#* library for clipping objects in boundary representation, the clipping is done in *3ds max* and the program loads already clipped polygons. As the input objects, the application uses WRML objects (**.WRL*), as a format of objects that can be exported from 3ds max, and easily altered.

[5] `http://www.cs.man.ac.uk/~toby/alan/software`

Data structures

Most of the objects are stored as a list of vertices and list of triangles' indices, to be easily loaded, saved and rendered. Only the clipped polygoons are saved in a different way, because some additional information needs to be stored during the computation. We store them as an array of *Part*s, where each part again stores the information of the maximal and minimal topological distance, list of its vertices, and here also the list of indices (triangles). The vertices again store their coordinates, their topological distance and their vertex path. New information stored in each vertex are its neighbors - to be able to quickly assign the topological distances and reassign the triangles when duplicating the vertex. For the method of Projection growing, the vertices also store their projected coordinates.

Voronoi diagram construction

The Voronoi diagram is computed by using a FortuneVoronoi library[6]. The library has some computational problems after DirectX is initialized, therefore we are not able to offer the user an interface to load the objects during the application run.

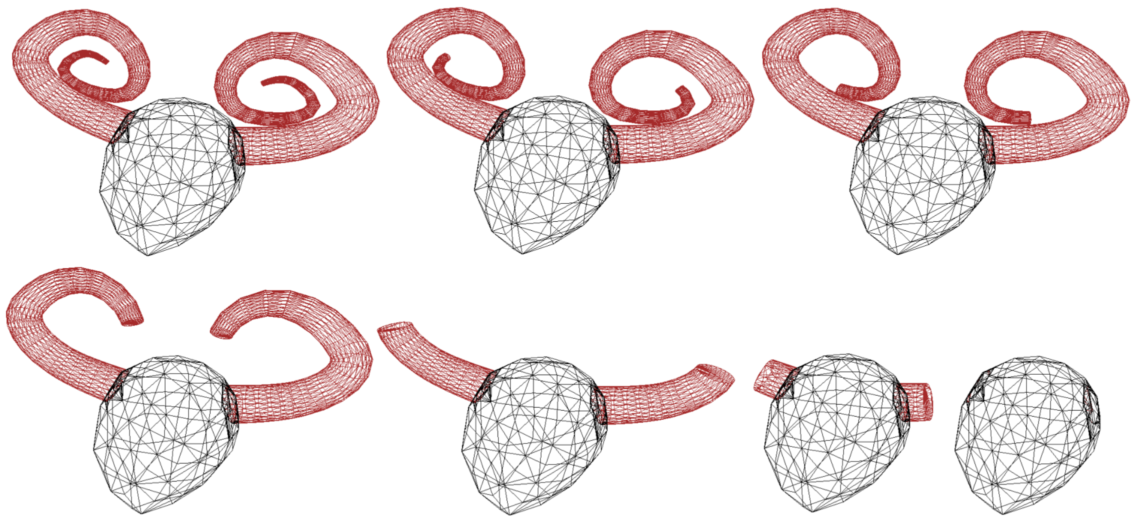
Visualization

Visualization is done by using DirectX 9.0c. The objects stored as a list of vertices and indices are easily to visualize (using vertex and index buffers), and the computed objects with vertex paths are visualized by altering one additional object with a vertex and index buffer - index buffer do not need to be altered, but we rewrite the vertex buffer each time the time changes.

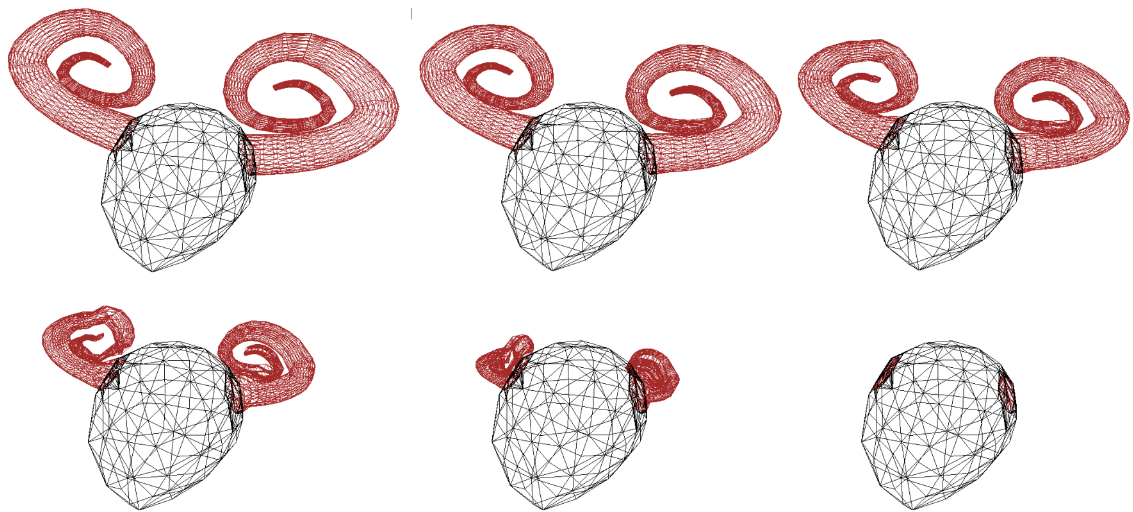
The vertices are visualized using a unit sphere. For each vertex, the sphere is moved to the vertex's coordinates, its color is set according to the vertex's topological distance, and it is rendered.

[6] `http://www.codeproject.com/KB/recipes/fortunevoronoi.aspx`

D Other results

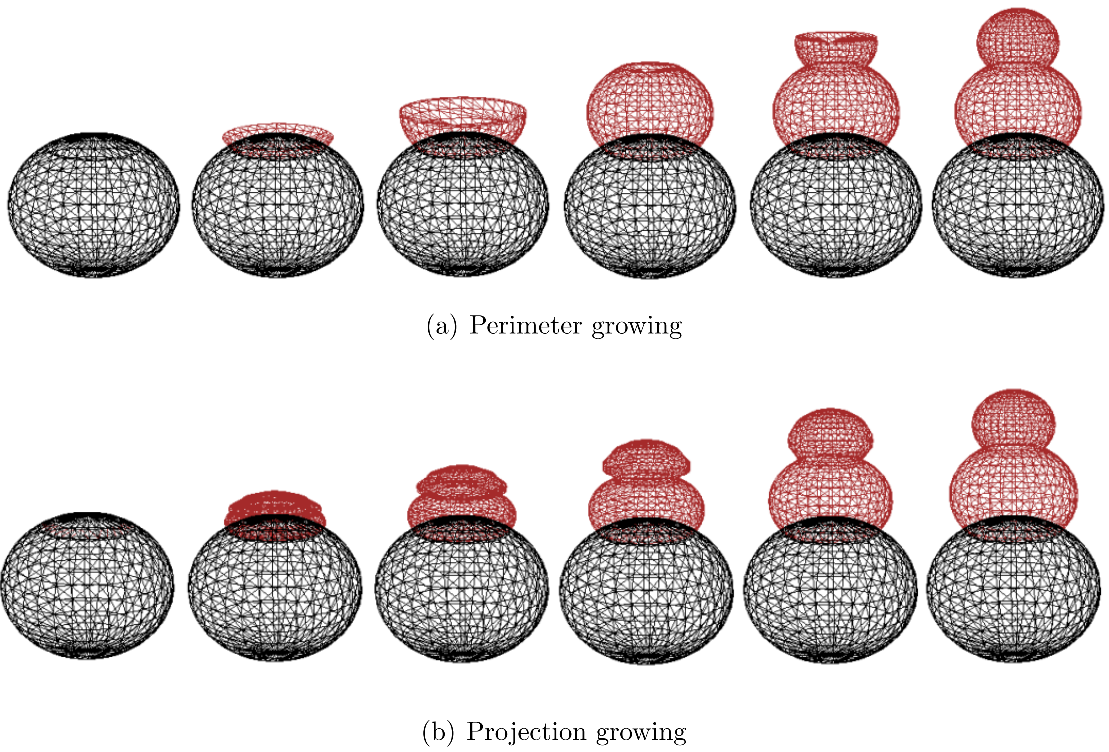


(a) Perimeter growing



(b) Projection growing

Figure D.1: Another example showing that the Perimeter growing is better for long curved parts



(a) Perimeter growing

(b) Projection growing

Figure D.2: Projection growing does not always produce bad results for the non-convex parts

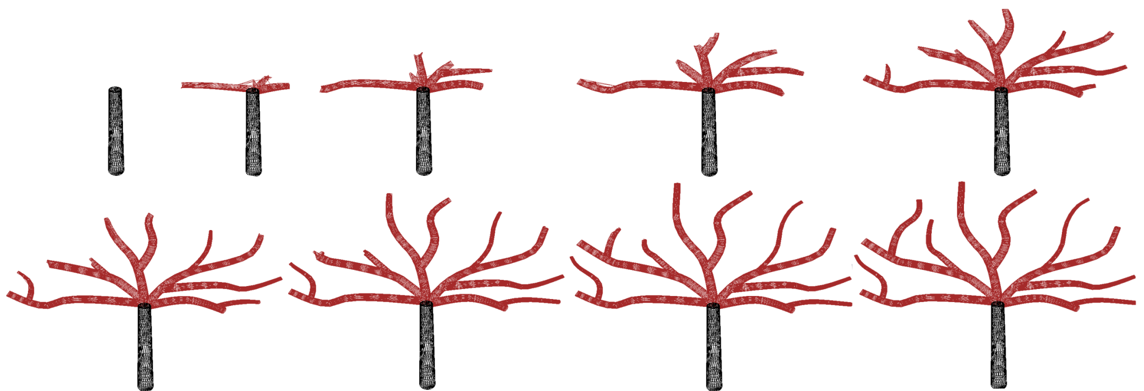


Figure D.3: Perimeter growing for more complex parts