Core-based morphing algorithm for triangle meshes

Martina Málková*
Ivana Kolingerová†
Jindřich Parus‡
Department of Computer Science and Engineering, University of West Bohemia

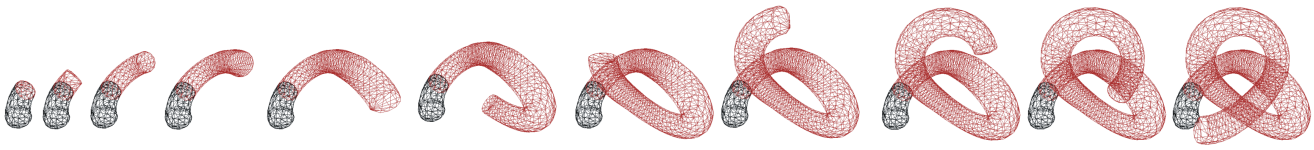


Figure 1: *Morphing of a knot - the knot (target mesh) grows out from the piece of rope (source mesh) in a natural way*

Abstract

This paper presents a method for the metamorphosis of genus-0 triangle meshes based on their intersection. It is an extension of our previous 2D algorithm [Málková, 2007]. Our algorithm is designed to simulate growing processes, therefore it is useful for morphing objects, where the user expects some parts of the latter object to grow out from the former one (e.g. a head with and without horns). The user can influence the algorithm's behavior by changing the mutual position of the objects, while the results are easily predictable.

CR Categories: I.3.5 [Computational Geometry and Object Modeling]: Boundary representations— [I.3.7]: Three-Dimensional Graphics and Realism—Animation

Keywords: morphing, triangle meshes, mesh intersection

1 Introduction

The goal of morphing is to compute a shape transformation (represented by an animation) between two shapes. There are usually many different ways how to deform one shape to another. The goal is to choose such a transformation which is visually plausible. Since morphing has a wide use in many types of applications (computer animation, design, special effects in movies, image compression and data visualisation), each of them requires different behaviour of the resulting morph and so defines the visual plausibility in a different way. Sometimes we might want to explode the initial shape into particles and form the final shape from the particles; sometimes we might want a continuous transformation during which the volume is preserved, etc. The algorithms usually concentrate on one type of application, trying to meet all its expectations. The most often expectation is to preserve the common features of similar shapes, so most algorithms put stress on this area. However, in some cases it is not possible to align all features (e.g., a body with and without a tail). In such cases, the user usually expects the nonaligned part to grow out of the rest, which is something that he/she can hardly obtain (or sometimes cannot obtain) from the current algorithms. To fill this area, we designed a new morphing algorithm, which has a "growing-like" nature, i.e., we focuse on a shape transformation that mimics growing (or disappearing) of some parts. Unlike most other algorithms, ours is not limited by star-shaped objects and can be used for all genus-0 objects.

After defining the input shapes, the computation of a morph consists of solving two distinct subproblems - establishing the correspondence of vertices and setting the vertex paths of the corresponding vertices, together with the dynamics of the transformation, describing how the vertices move in time. Although the properly computed vertex path can lead to establishing the expected transformation between the corresponding vertices, the crucial problem here is to automatically establish such correspondence that is similar to the one the user would enter manually.

Most algorithms let the user influence the result by adding some constraints or by manually defining correspondence of some concrete vertices. This process can be long and not intuitive, the user sometimes cannot easily predict which vertices he/she should define to achieve his expectations from the result. Our algorithm offers an intuitive tool for influencing the vertex correspondence - the user can achieve different results by changing the mutual position of the objects, while the result is always easily predictable.

The main steps of our algorithm are as follows: after the mutual position of the objects is defined, it computes the intersection of the objects, defining the *core* of the morphing. The core is a fixed part that is constant and will not change its shape during the morphing process. When transforming between the source and the target object, parts of the source object disappear in the core (i.e., they die away) whereas parts of the target object grow out of the core. The process of disappearing can be viewed as a reversed process of growing, thus algorithmically we need to solve only one process.

The paper describes an extension of our polygon morphing algorithm into 3D, where its input objects are triangle meshes. The meshes are assumed to be closed, however the presented algorithm is not limited only for closed meshes, if we are able to compute or manually define their intersection.

The paper is organized as follows. Section 2 describes related work in the area of mesh morphing. Section 3 describes the method it-

*mmalkov@kiv.zcu.cz
†kolinger@kiv.zcu.cz ; Work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Centre for Computer Graphics)
‡jparus@kiv.zcu.cz

self and discusses two possible strategies of the process dynamics. The following Section 4 shows results of the algorithm and demonstrates typical use of the techniques described in Section 3. The last Section 5 describes conclusions and suggestions for future work.

2 Related Work

Warping and morphing have a long tradition in Computer Graphics and the full description is out of the scope of this paper. We refer reader to [Gomes et al., 1999] for a detailed description. Here we describe the work related to mesh morphing. The techniques described deal with morphing of 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 [Alexa, 2002], most of the algorithms consist of three main 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. However, as is discussed in [Gomes et al., 1999], such solution have problems when computing rotational morphing; when we morph between two line segments, both of them of the same length but one rotated, the line segment shorten during the morphing process, which is something we do not expect.

In [Kent and Carlson, 1992], the authors first project the objects onto a unit sphere. The projection used depends on the type of the input objects and is discussed for the most of the genus-0 objects(convex and star-shaped objects, objects of revolution and extruded objects). The 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.

Alexa uses the idea of Kent et al. and presents another correspondence-based algorithm for morphing polyhedra [Alexa, 2000]. After triangulating the polyhedron, he examines the spherical projection. Because the method is designed for all genus-0 meshes, the projection may produce some overlapping edges (*foldovers*). 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, but there is still some possibility that the relaxation will degenerate. The resulting embeddings are merged to get the supermesh. 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, and the positions of the vertices of the target mesh are computed by using the barycentric coordinates.

The algorithm presented in [Ahn et al., 2004] tries to enhance the correspondence-based morphing by decreasing the number of vertices of the supermesh. It uses the spherical embedding from the previous Alexa's approach (and therefore it is limited by the same class of objects) to find the correspondence between M_0 and M_1. After projecting M_0 and M_1 onto the unit sphere, M_0' (and similarly M_1') is constructed by mapping the vertices of M_1 onto the surface of M_0. Additional edge swapping of the created edges is done to reduce the difference between M_0' and M_1. Then, the sequence of connectivity transformations between M_0' and M_1' is computed by using an adaptation of the algorithm from [Hanke et al., 1996], resulting in a graph of edge swaps that needs to be done during the transformation. Each swap is realized by a *geomorph* [Hoppe, 1996] to make it smooth. The disadvantage of this approach 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.

A noncorrespondence based approach is described in [Cohen-Or et al., 1998]. The method needs the user to define corresponding control points on the input objects first. These points are used 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. The warped object is rasterized into a binary discrete volumetric representation and converted into a 3D distance field by a method presented in [Levin, 1987]. 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. The main benefit of this method is that it does not require the input objects to be of the same topological genus.

3 The Proposed Solution

3.1 General Idea

For better understanding of the following text, let us briefly recall the 2D algorithm, from which the 3D method was extended [Málková, 2007]. The input of the 2D algorithm are two simple polygons. The intersection of the input polygons is computed, resulting in a simple polygon called *core*. The core is the only part that does not change during the morphing process, the parts of the input polygons that are outside the core either grow out from the core or disappear in the core. Such parts are also simple polygons, whose boundary consists of a polygonal chain C_{in}, containing vertices and edges common to the part and the core, and a polygonal chain C_{out} containing vertices and edges lying outside the core. The chains C_{in} and C_{out} meet at exactly two vertices, called intersection vertices. The goal is to compute the morphing sequence between C_{in} and C_{out}. For each vertex of C_{out}, so called *vertex path* is computed. The vertex path consists of positions, defining where the vertex travels during the morphing process. If the vertex path contains only two positions (the initial position and the final position of the vertex), it represents *vertex-to-vertex correspondence*. But, for example, to represent the growth of non-convex shapes without self-intersections, the vertex path needs to contain more positions. The vertex paths define both the vertices correspondences (the first and the last positions in the path are the ones of the corresponding vertices) and their key positions during the animation. To view the final morph, we only need to interpolate between the positions in vertex paths according to the given time.

Brief description of our 3D extension is quite similar. As the input, we have two objects A, B defined by their triangle mesh boundaries $\delta A, \delta B$. The intersection of both input objects (*core* C) is computed. The boundary of the intersection is also a triangle mesh δC. The parts of the objects that are outside the core are computed as the differences $P = A - C, Q = B - C$. Each part's boundary consists of a mesh C_{in} containing vertices and triangles common to the part and the core, and a mesh C_{out} containing vertices and triangles lying outside the core. Those two meshes meet at $1..n$ closed polygonal chains, called *intersection chains*. The goal is to compute the morphing sequence between C_{out} and C_{in}. Again, the vertex path is computed for each vertex of C_{out} and the final morph at a given time is computed by interpolating between the coordinates in the vertex path of each vertex.

Now, let us describe the algorithm in detail. As the input, we have two objects A, B. The core is computed as $C = A \cap B$. It can consist of $C = (0, \ldots, c-1)$ disjoint parts. 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 in the following text that the core consists of only one part.

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 solving 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 simply reversing the time plan.

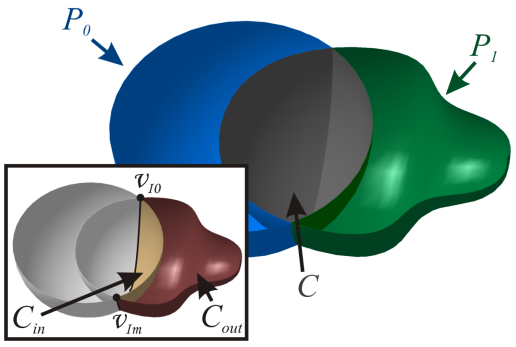
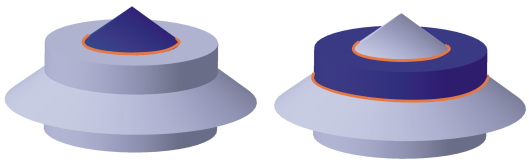


Figure 2: *General terms: Core C, part $P_i = C_{out} \cup C_{in}$, intersection vertices v_{Ii}.*

δP_i consists of two surfaces C_{in}, C_{out} (shown in Figure 2). The surfaces C_{in}, C_{out} are separated by closed polygonal chains, let us call them *intersection chains* I. There can be an arbitrary number of intersection chains $I = \bigcup I_i, i = 0 \ldots n$. Figure 3 shows an example of a part with one intersection chain (Figure 3a) and two intersection chains (Figure 3b), which are the most often cases.

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.

Any method can be used for morphing C_{out} to C_{in}, ours uses a concept of so-called *topological distance* of a vertex. If we define a distance $d(v_i, v_j)$ between vertices v_i and v_j as the minimal number of edges on the mesh between v_i and v_j, we can compute the topological distance of a vertex v_i as its minimal distance $d_{min} = min(d(v_i, v_j), v_j \in I_k), k = 1 \ldots n$, where n is the number of intersection chains. The topological distance therefore



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

Figure 3: *There can be an arbitrary number of the part's intersection chains (orange).*

represents the length of the shortest path from the vertex to any intersection chain.

The topological distances are 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 it the new distance and put it in the stack. The algorithm is shown in Figure 4. An example of computed topological distances can be seen in Figure 5.

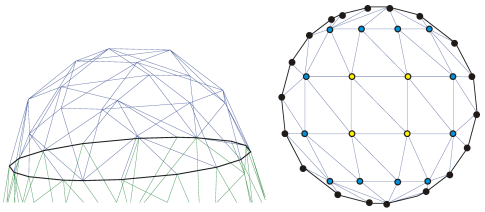
Input: List of vertices of the intersection chains of the current part $I = \bigcup v_{Ii}$, list of vertices $C_i = (v_1, \ldots, v_N)$, for which the topological distance should be computed. Stack $S = \emptyset$.

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: The algorithm for computing topological distances.



(a) (b)

Figure 5: *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.*

After the topological distances are computed for both C_{in} and C_{out}, we compute the vertex paths of the vertices of C_{out}. These vertex paths can be a simple one-to-one correspondence between the vertices of C_{out} and C_{in}, in which case they contain only the positions of the corresponding vertices, but they can also be more complex. For the vertex path computation, we extended our 2D method called *Perimeter growing*, which searches the positions of the vertices among the existing vertices of C_{out}. This method will be described in Section 3.2 in detail.

After the computation of vertex paths, we need to merge the parts and the core into the resulting supermesh. The reason why we cannot just take the whole parts and the core and produce the final morphing sequence is that during the animation, 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}). While moving, some vertices of C_{out} can cross an edge of C_{in} - and at that time, the vertices and edges that were inside and should not be visible, are now visible (see Figure 6).

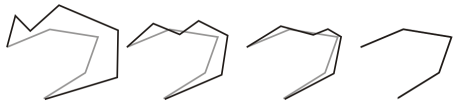


Figure 6: *Not removing C_{in} may result in unwanted self-intersections: C_{in} (grey), C_{out} (black) (in 2D for simplicity)*

When the result is merged, the animation can run by using the time plan. The time plan computes the actual velocity of each vertex depending on a time and the chosen dynamics of the animation. The different dynamics setting and their behavior will be discussed in Section 3.3.

3.2 Perimeter Growing

As the vertex paths are represented as a list of desired positions of the vertex during the time, the goal of the Perimeter Growing is to compute such a list for each vertex of a given part that the vertex paths contain only the positions of existing vertices of the part. Therefore during the final animation, all the vertices follow the boundary of the part. To try to avoid self-intersections, the vertices should take the shortest possible way they can to reach their corresponding vertices from C_{in}. Such an approach should lead to a morphing sequence, where C_{out} continuously grows out of C_{in}, following the perimeter of the original part.

To find a vertex path of a vertex v_i, we first need to find out where should the path lead - decide about the corresponding vertex. In 2D, the correspondences were solved from the intersection vertices, depending on the topological distance of each vertex and the count of the vertices of C_{in} and C_out. In a special case when C_{in} and C_{out} had the same number of vertices (Figure 7a), a vertex with d_i from C_{in} corresponded to a vertex with d_i from C_{out}. In other cases, either more vertices from C_{out} corresponded to one vertex from C_{in} (Figure 7c, or some vertices from C_{out} were duplicated to cover all the vertices from C_{in} (Figure 7b). The vertex paths were set up only for vertices from C_{out}, each of them containing the coordinates of the corresponding vertex from C_{in} as its last element.

As each vertex of the part in 2D had only two neighbors, only two vertices had the same topological distance and so knowing the topological distance was enough to be able to solve the correspondence. However, in 3D, the solution is not so clear, because more than two vertices can have the same topological distance. Let us discuss the following example: In Figure 8, 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.

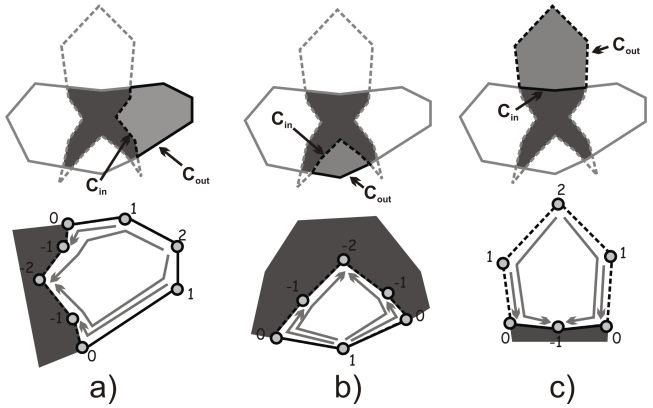


Figure 7: *Solving correspondences in 2D when a) C_{out} and C_{in} have the same number of vertices, b) C_{in} has more vertices than C_{out}, c) C_{out} contains more vertices than C_{in} (dark grey: core, grey: processed part, arrows are connecting the corresponding vertices)*

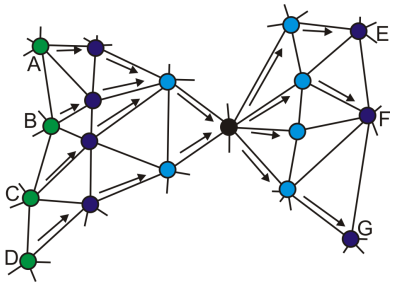


Figure 8: *Correspondence problem at one intersection vertex (black, other vertices are colored according to their topological distance: ± 1 light blue, ± 2 blue, 3 green).*

However, we are able to solve the correspondence problem to the intersection vertices while computing the topological distances. When a vertex v_i assigns to its neighbor v_j some topological distance, v_j is assigned the vertex path containing the vertex path of v_i extended by the coordinates of v_i (Figure 11, step 2).

This way we are able to compute the transformation from C_{out} to the intersection line. In the same manner, we can compute the vertex paths for C_{in} and therefore its transformation to the intersection line as well.

So although we are not able to solve the correspondence between C_{in} and C_{out} directly, we can piece the resulting morphing sequence from three parts (see Figure 9): In the first part, $(0, t - \epsilon)$, vertices of C_{out} move towards the intersection vertices. At $t - \epsilon$, the resulting morph appears to contain only the intersection vertices (but there are also the vertices of C_{out}, which are at the same positions as the intersection vertices), connected by edges defined by C_{out}. Let us call it S_0. In the last part, $(t + \epsilon, 0)$, vertices of C_{in} move from the intersection line to their original positions (their vertex paths are reversed). At $t + \epsilon$, the resulting morph appears to have only the intersection vertices, connected by edges defined by the C_{in}. Let us call it S_1. During the time $(t - \epsilon, t + \epsilon)$, we need to morph from S_0 to S_1, which are two triangular meshes with the same vertices, but a different connectivity. 2ϵ is the time amount spent on the connectivity change, its value depends on how fast we want the change to run over.

The last part to solve is to transform the connectivity of S_0 to S_1 by an edge swap sequence, as is presented in [Ahn et al., 2004].

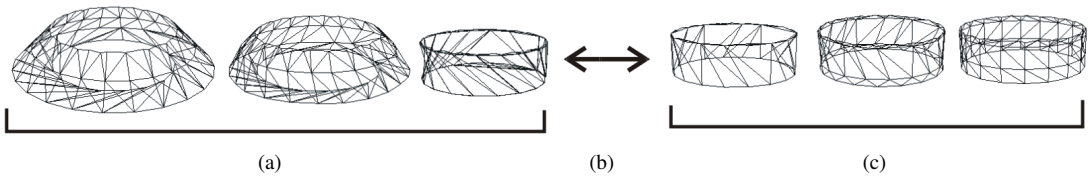


Figure 9: *The resulting morphing sequence consists of three parts: (a) the outside part morphs towards the intersection vertices, (b) connectivity change, (c) the inside part morphs from the intersection vertices.*

The continuous swap is created by applying geomorphs [Hoppe, 1996] (Figure 10). As geomorphs can be used only to swap one edge, the dependency graph of edge swaps is constructed to allow the transformation to proceed in a minimal possible time.

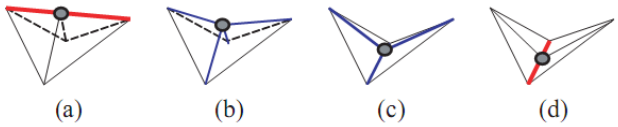


Figure 10: *Using geomorph to swap the edges: (a) create the vertex at the center of the edge to be swapped, (b)&(c) the vertex moves during the geomorph, (d) the edge has been swapped so the vertex can be removed*

(From [Ahn et al., 2004])

The pseudocode of the perimeter algorithm is shown in Figure 11. Note that the algorithm has only small changes from the one computing the topological distances (Figure 5): the assignments of the vertex path in steps 1,2.

Input: Part P_i in the form of three lists: List of vertices of the intersection chains $I = \bigcup v_{Ii}$, list of vertices on the core C_{in} and list of vertices outside the core C_{out}. Stack $S = \emptyset$.

Output: Part P_i, where each vertex of C_{out} and C_{in} has its vertex path computed in a form of a list of coordinates $pv_i = v_0, \ldots, v_{n-1}$

The algorithm:

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

Figure 11:The Perimeter algorithm.

The main advantage of the algorithm is its simplicity and usage for arbitrary types of parts. The algorithm produces best results when the mesh is uniform, which leads into its often use together with some remeshing algorithm.

3.3 Time Plan

After computing the vertex paths and creating the supermesh, the task is to compute the positions of vertices according to the current time (create the final animation or the morphed shape). The way the vertex paths are handled is called the time plan. The time plan decides when each vertex travels and how it travels. The former offers at least two possibilities - first, the vertex moves all the time and if it has n positions in its path, it has $(n-1)/t$ time to travel between each pair of positions, where t means the whole amount of time. Second possibility is to assign to each vertex a different amount of time, according to the number of positions it has in its vertex path. The vertex with the longest vertex path travels all the time, and the other vertices move only during their portion of the time.

Along the time distribution, the time plan decides about the trajectory of the vertex between the key positions defined by the vertex paths. The actual position of the vertex between two positions from its vertex path can be computed either by a linear interpolation, or by any other interpolating method.

The choice of the concrete time plan depends on the output of the method for the vertex path computation and on our expectations of its final behavior. For the Perimeter growing, we decided to use the traditional linear interpolation along with assigning each vertex different amount of time depending on the length of its vertex path.

As was already told, the Perimeter growing produces two supermeshes, one for the time $(0, \frac{t}{2} - \delta)$ and the other for $(\frac{t}{2} + \delta)$. Let us denote the amount of time for one supermesh t_s. Each vertex v_i in the supermesh has $d_i + 1$ coordinates in its vertex path (where d_i is its topological distance). The vertex with the longest vertex path is the one with d_{max}. The maximal topological distance for each part P_i can be found during the distances computation. The amount of time for each vertex is then computed as $t_i = (0, \frac{t_s}{d_{max}})$. The resulting time sequence is shown in Figure 12 (top view of the part from 5 is used). First, all the vertices with $d_i > 0$ travel to the positions of the ones with $d_i = 1$, where such vertices stop and the others continue to the positions of the vertices with $d_i = 2$ and so on.

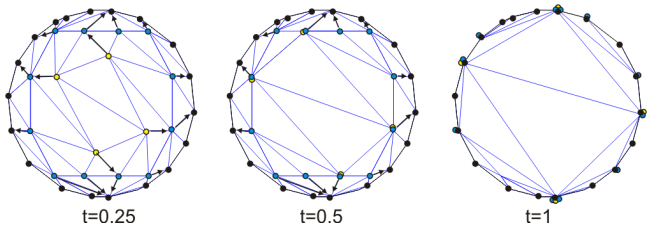


Figure 12: *The morphing sequence for one part, where we used the time plan with linear interpolation and different amounts of time for each vertex (the vertex paths are computed by the Perimeter growing).*

4 Results

In this section, we will discuss the cases where the use of our algorithm brings benefits. As the output of our algorithm depends on the mutual position of the meshes and therefore on the shapes

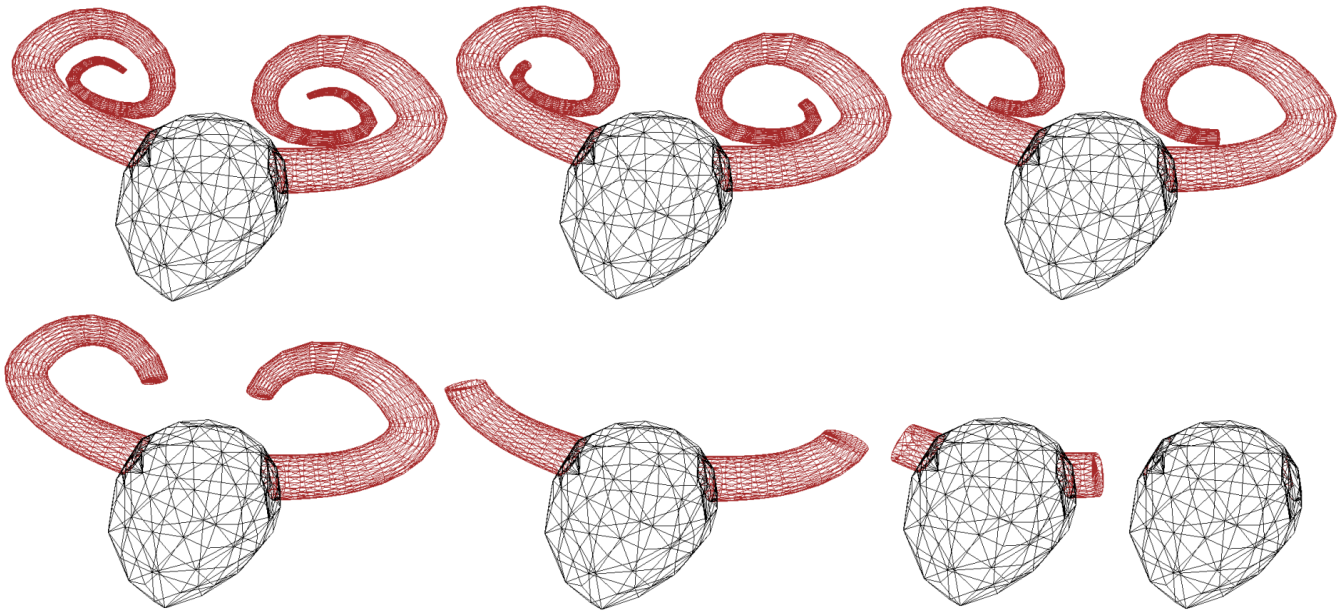


Figure 13: *Two horns (parts, red) morph from the head (core, black).*

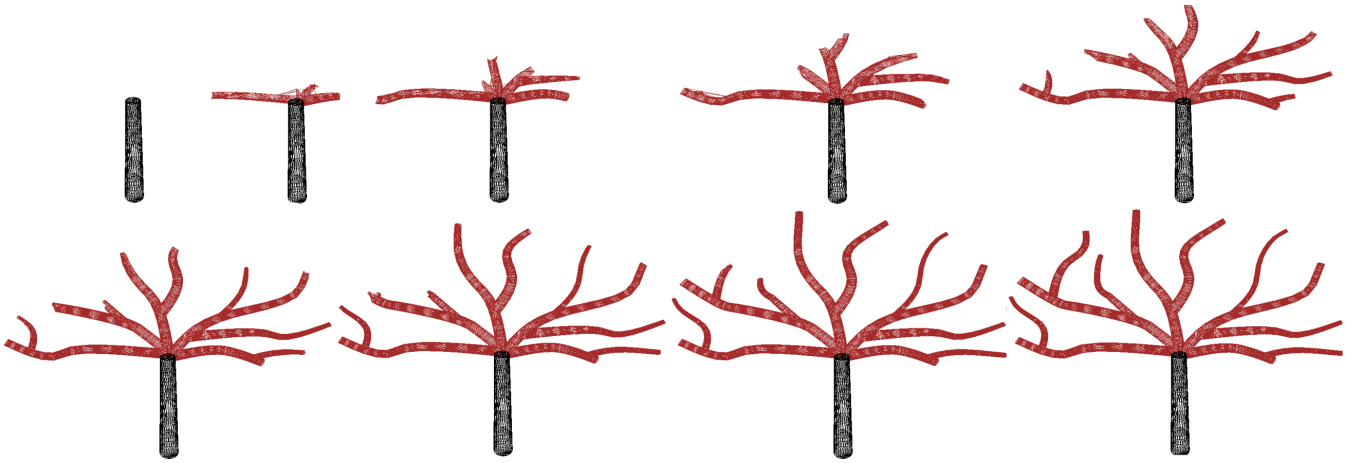


Figure 14: *An example of morphing a complexly branched part (red)*

of the parts P, Q, we will discuss its behaviour according to the shapes of the parts and not the types of meshes. The presented method (Perimeter growing) is used independently for each part, so we will show its behavior on one part's shape at a time.

Let us recall, that we should not use our algorithm, when we do not want the output to grow out from the mesh intersection, but we suppose the algorithm to align the corresponding features and morph between them (i.e. morph between faces with different expressions or bent and straight finger). Our algorithm is designed only for creating the growing behavior of the part. In special cases, the growing behavior can be similar to the feature one, but in general it is a completely different output.

The presented Perimeter growing method produces better results for thin and long parts (independently on the number of branches and curves of the part), because it "cuts" the top of the part during growing. Figures 1, 13 show examples of morphing a long curved part, consisting of only one branch. Also more branches are not a problem, as Figure 14 shows, where the algorithm produces only small artefacts around the branch joinings.

In all the examples, the core C was similar to one of the meshes, which suggests the idea of the use of our algorithm - to add something else to an existing shape and make it grow from it, or similarily let something of an existing shape disappear in the rest.

Animation videos of the examples can be seen at http://herakles.zcu.cz/research/morphing.

In the presented examples, we followed two current limitations of the algorithm. The former is, that only two input meshes are considered. The latter has been already discussed in Section 3 - the intersection of the input meshes should consist of only one part, if it does not, the algorithm chooses one of the parts and consider it the only one (Figure 15a,b). However, this solution always leads to self-intersections in the area of the other parts. To avoid the self-

intersections, we would have to split such parts of the objects that connect two (or more) parts of the core. This would lead to a proper result without self-intersections, however, the first object would first split into several objects, which would then merge to form the target object. We did not want to achieve this type of effect, however it would be possible future work in case of its practical need.

The algorithm itself is able to deal with more than two objects, but there are some limitations. It handles the special case, when the objects have the same intersection (Figure 15c). We only need to define which objects are to disappear in the core and which ones are to grow out from it. When there are distinct pairs of objects and each pair has an intersection consisting of only one part (Figure 15d), the algorithm can be used iteratively for each pair. However, if the input objects intersect at several places while not having any part of the intersection common for all of them (Figure 15e), the algorithm will not work.

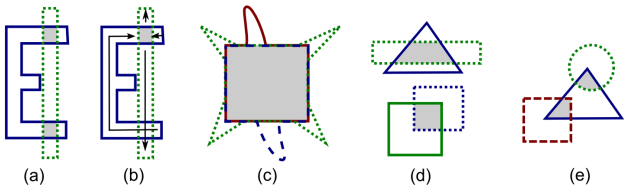


Figure 15: *Limitations of the algorithm (shown in 2D for simplicity): (a,b) more parts of the core (grey) lead to choosing only one, (c)more input objects with the same core, (d) distinct pairs of objects which can be solved separately, (e) several cores without a common one*

Although the algorithm itself is not much usable for more input objects, its nature offers an extension, where one object is given as the core, and the other objects are considered to be the parts that either grow out or disappear from the core. Each object is required to have an intersection with the core object, and this intersection is clipped out. In such a way, the algorithm can have an infinite number of input objects. The idea is sketched in Figure 16 (the core object is filled with grey, other objects are only outlined). This extension has not been implemented yet and belongs to our future work. By using this different approach to define the input meshes, the user would be able to first model the core (i.e., a body) and then add the parts separately (i.e., horns, a tail), placing them at a desired position.

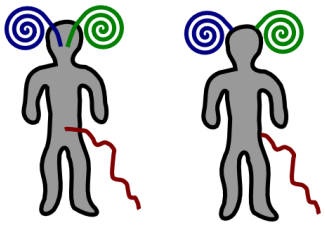


Figure 16: *A different approach to defining input meshes, where the user explicitly defines the core (filled with grey) and the parts (in 2D for simplicity): (a) input objects (b) clipped input objects representing the parts*

5 Conclusion and Future Work

We have presented a novel approach to morphing meshes. The goal of our approach was to offer a tool for animating growing processes, which is an area of morphing that is not explored. Our algorithm solves the problem of morphing between two meshes in an unusual way. 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. This way, the morphing is established as a process of disappearing of the parts of the latter object and growing of the parts of the former object.

We presented one method, called Perimeter growing, for computing the transformation of each part. This method computes the correspondences of vertices and the vertex paths at the same time. It is based on the topological distances of the vertices. As the topological distances do not cover the euclidean distance information, the input meshes should be uniform to obtain best results. The method has strong advantages when we use it for long curved parts, but it is usable also for other shapes of the parts.

The experiments confirmed that our algorithm (in combination with the Perimeter growing method) is suitable for the cases, where the user expects some parts of the object to grow out from the intersection or disappear in it. It can be used 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.).

There are still some implementation parts to finish. To be able to use the Perimeter algorithm, we need to implement the Ahn's algorithm [Ahn et al., 2004] 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.

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

Ahn, M., Lee, S., and Seidel, H. (2004). 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. ACM.

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

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

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

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

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

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

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

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

Málková, M. (2007). A new core-based morphing algorithm for polygons. Proceedings of CESCG.