The Visual Computer

Interactive surface decomposition for polyhedral morphing

Arthur Gregory,
Andrei State,
Ming C. Lin,
Dinesh Manocha,
Mark A. Livingston

Department of Computer Science, University of North Carolina at Chapel Hill, CB 3175 Sitterson Hall, Chapel Hill, NC 27599, USA
e-mail: {gregory,andrei,lin,dm,livingst}@cs.unc.edu,
http://www.cs.unc.edu/~geom/3Dmorphing

We present a new approach for establishing correspondence for morphing between two homeomorphic polyhedral models. The user can specify corresponding feature pairs on the polyhedra with a simple and intuitive interface. Based on these features, our algorithm decomposes the boundary of each polyhedron into the same number of morphing patches. A 2D mapping for each morphing patch is computed in order to merge the topologies of the polyhedra one patch at a time. We create a morph by defining morphing trajectories between the feature pairs and by interpolating them across the merged polyhedron. The user interface provides high-level control, as well as local refinement to improve the morph. The implementation has been applied to several polyhedra composed of thousands of polygons. The system can also handle homeomorphic non-simple polyhedra that are not genus-zero (or have holes).

Key words: Metamorphosis – Surface decomposition – Animation – 3D polyhedral morphing – Geometric features

Correspondence to: A. Gregory

1 Introduction

Image and object morphing techniques have gained increasing importance in the last few years. Given two objects, metamorphosis involves producing a sequence of intermediate objects that gradually evolve from one object to the other. The techniques have been used in a number of applications, including scientific visualization, education, entertainment, and computer animation. Morphing, whether in two or three dimensions, generally consists of two basic phases that establish a correspondence between the images or objects and interpolate between them, in conjunction with blending their colors or textures. We present a new approach for establishing correspondence for morphing between two homeomorphic polyhedra. Initially, the user selects some corresponding elements called *feature pairs*. Although we borrow this term from previous morphing algorithms for images or 3D volumetric models [4, 31], our concept of a feature is closer to the sparse control mesh used in [14]. Our algorithm includes a simple and intuitive user interface for feature specification and automatically generates a *feature net*. Based on the feature nets, the algorithm decomposes the boundary of the polyhedra into morphing patches, computes a mapping for each morphing patch to a 2D polygon, merges them, and constructs a *merged polyhedron* with a topological connectivity that contains both of the input polyhedra. In order to create a morph, the merged polyhedron has a *morphing trajectory* for each vertex to move along from one input polyhedron to the other. The overall complexity of the algorithm is $O(K(m+n))$, where K is a user-defined constant and m and n correspond to the number of vertices in the two input polyhedra. Some of the principal attributes of our approach are:

- *Simple user interface*. The users only need to select a few corresponding pairs of vertices on the two polyhedra to define the feature nets. The trajectories along which these features travel during morphing are currently represented as Bézier curves.
- *Fine user control*. The algorithm not only provides the user with high-level control in terms of specifying the features and trajectories, it also provides a mechanism to locally refine the morph or the animation sequence.
- *Generality*. The algorithm is applicable to all genus-zero polyhedra and makes no assumptions about convexity or star shape. Furthermore, it

The Visual Computer (1999) 15:453–470

can also be applied to *non-simple* homeomorphic polyhedra, that are not genus-zero.

Organization. The rest of the paper is organized in the following manner. In Sect. 2, we survey related work in this area. We give an overview of our approach in Sect. 4, present the algorithm to compute the correspondence in Sect. 5, and address the morphing trajectories and other interpolation issues in Sect. 6. We describe the user interface in Sect. 7. Then, we discuss the implementation in Sect. 8 and also highlight its performance on various models. We analyze the algorithm in Sect. 9 and mention areas for future research in Sect. 10.

2 Related work

The problems of establishing correspondence between graphical objects for shape transformation and morphing have been widely investigated in computer animation, graphics, and computational geometry for more than a decade. Most of the earlier work in the area focused on image metamorphosis, though a number of approaches have also been proposed for 3D volumetric models and 3D polyhedral models. Surveys of some of these techniques can be found in [17, 29]. An extended abstract of this paper appears in [18].

2.1 Two-dimensional morphing

Given two images, the problem of constructing a metamorphosis from one image to the other has been extensively studied in computer graphics and image processing. The set of algorithms can be classified into those that operate on raster images [4, 13, 30, 47, 48] and those that operate on vector-based or geometric representations [19, 40, 42, 45]. Feature-based image morphing can be achieved by energy minimization [40] or feature-based constraints [42]. Beier and Neely presented an elegant feature-based approach [4]. The features may be points or lines [4, 44] or to snakes [30]. Mappings that have been used include (bi)linear mappings [4, 44] and spline-based mappings or free-form deformations [30] in conjunction with a weighting function that effectively controls the range over which a feature has influence. Ranjan and Fournier [37] presented an approach which uses circles to partition the objects. Other algorithms for transforming one image into another are based on 2D particle systems to map the pixels [39].

It is possible to generate 2D images from a 3D model and apply 2D morphing algorithms to these. In this case, the intermediate stages of the morph are images. For many applications in animation and design, the 3D models themselves not their images, should be transformed [9, 25]. Furthermore, if the viewpoint or the lighting parameters are changed, the 2D morph has to be recomputed. However, 3D morphing is independent of viewing or lighting parameters. Having a 3D representation also allows the use of computer animation techniques such as keyframing.

2.2 Three-dimensional volume morphing

Given two volumes, 3D volume morphing involves producing a sequence of volumes to transform them. A number of approaches have been published [8, 11, 20, 31, 36]. These include the use of Fourier transforms that warp linearly in Fourier space [23]. This is a simple approach and requires minimal user specification, but inhibits intuitive understanding of the morph. Lerios et al. [31] have presented a 3D extension of Beier and Neely's [4] approach. It allows the user to specify a set of features and permits fine user control. Cohen-Or et al. [12] introduce a technique based on distance field interpolation.

Three-dimensional polyhedral models can be voxelized to enable the use of 3D-volume morphing. However, the intermediate stages of the morph are volumes, and converting them into geometric models produces topologically complex objects. Given a geometric surface description of the model, we can use current graphics systems for fast rendering, and we can also use various geometric algorithms for applications such as physically based simulation or 3D object manipulation. Moreover, for feature-based approaches, it is simpler and more intuitive to design a user interface based on the geometric model as opposed to the volumetric model. For example, a user can pick any vertex, edge, face, or contour of the original polyhedron as a feature, which is not easy with a voxel-based representation. Therefore, approaches based on voxelization followed by 3D volume morphing have limitations as well.

2.3 Three-dimensional shape transformations and metamorphosis

Several approaches related to establishing correspondence between 3D polygonal objects for shape transformation and metamorphosis have been proposed. Physically based modeling techniques based on deformations [3, 41] and particle systems [38] can be used for object metamorphosis. Hong et al. [22] present an approach for polyhedral models that matches the faces with closest centroids. Chen and Parent [7] present an algorithm to transform piecewise linear 2D contours and extend it to 3D cylindrical objects. Bethel and Uselton [6] add degenerate vertices and faces to two polyhedra until they have a common vertex neighborhood graph. Kaul and Rossignac [25] transform a pair of polyhedra by using their Minkowski sums. Hodgins and Pollard [21] have presented an algorithm for interpolating between control systems of dynamic models. Wyvill [49] has described an approach for implicit surfaces. Parent [34] has presented an approach that splits the surface of the model into pairs of sheets of faces and recursively subdivides them until they have the same topology. Parent [34, 35] has also described a method for deformation of polyhedral objects based on implicit functions. Kent et al. [26, 27] have presented a shape transformation algorithm for genus-zero polyhedra that involves projecting the models onto a sphere. Chen et al. [9] have produced 3D morphs of cylindrical images. Galin and Akkouche [16] have presented an algorithm for blob metamorphosis based on Minkowski sums. Lazarus and Verroust [28] have proposed a method based on skeletal curves. Shapiro and Tal [46] propose a polyhedron realization algorithm for shape transformation. Alexa [1] presents a technique for merging two genus-zero polyhedra. Kanai et al. [24], as well as Bao and Peng [2], have presented algorithms for shape transformation of genus-zero polyhedra using harmonic maps. DeCarlo and Gallier [14] have proposed a morphing technique that establishes correspondence by allowing the user to divide the surface into triangular and quadrilateral patches that can be projected onto a plane. Our overall approach shares their theme. However, we improve upon several restrictions in their technique, making it easier for the user to specify correspondence between complicated models. For example, we remove the requirement that the user-specified surface patches must be triangular or quadrilateral, and that each can be directly projected onto a plane.

3 Terminology

The term polyhedron refers to an arrangement of polygons such that two and only two polygons meet at each edge. It is possible to traverse the surface of the polyhedron by crossing its edges and moving from one polygonal face to another until all polygons have been traversed by this continuous path [33]. Furthermore, each vertex is adjacent to at least three edges. Topology refers to the vertex/edge/face connectivity of a polyhedron. Simple polyhedra are all polyhedra that can be continuously deformed into a sphere. Non-simple polyhedra are topologically equivalent to a solid object with holes in it. In this paper, we assume that each face of a polyhedron is homeomorphic to a closed disk. The genus g of a polyhedron is the maximum number of non-intersecting loops that do not divide its surface into two regions. Moreover, polyhedra satisfy the Euler-Poincaré formula: $v-e+f-2(1-g)=0$, where v, e, f, and g are the number of vertices, edges, faces, and genus of the polyhedron, respectively. The genus of a simple polyhedron is zero.

4 Overview

Given two homeomorphic polyhedra, our goal is to generate a morph that results in a smooth and gradual transition from one polyhedron to the other. One key aspect of our system is to allow the user to identify the important features of each polyhedron and specify a correspondence between them. The rest of the algorithm consists of a combination of techniques that can produce the desired result from the given user input.

Our algorithm decomposes the problem of morphing two polyhedra into morphing corresponding pairs of surface patches. Given the user's specification, the algorithm automatically partitions each polyhedron into a series of morphing patches, each of which is homeomorphic to a closed disk. Based on this decomposition, our approach is applicable to non-simple polyhedra as well. There are many

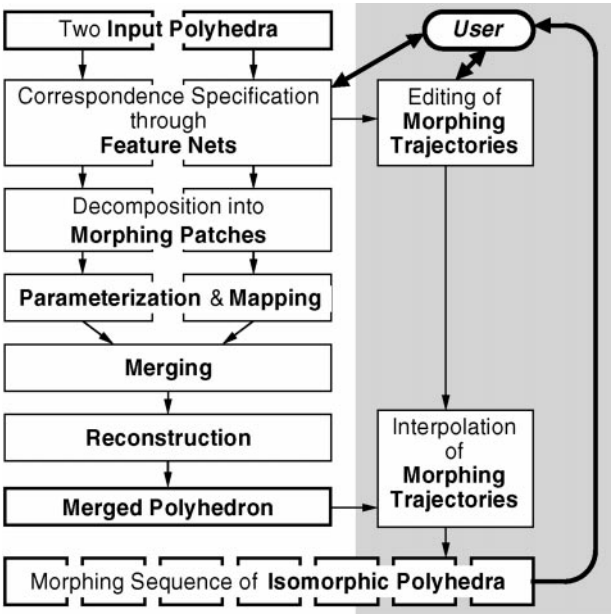


Fig. 1. Overview of the polyhedral morphing algorithm

other advantages to this approach. It is simpler to compute a 2D parameterization for one patch at a time. We use such mappings to merge the topologies of the polyhedra. Moreover, it allows us to use a number of algorithms from computational geometry. These include computing arrangements of lines, triangulations of polygons, and planar straight-line graphs, and determining point locations in planar subdivisions. All these techniques are used for establishing correspondence between the two polyhedra.

Geometric algorithms are prone to robustness and accuracy problems. These involve dealing with degenerate configurations and inaccuracy problems due to finite precision arithmetic. Since an important component of the morphing algorithm is to merge the topologies of two polyhedra, we need to make sure that the algorithm maintains valid data structures and topology at each stage. In order to develop a robust implementation of the algorithm, we have, at times, opted for simpler geometric algorithms, which may not have the best asymptotic performance.

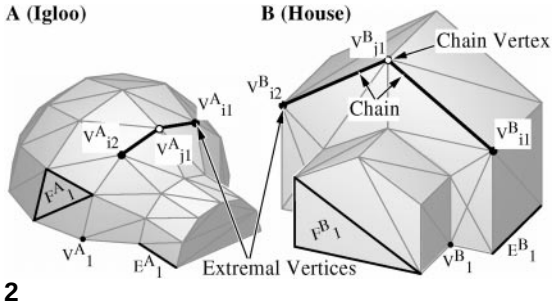
An overview of our approach is given in Fig. 1. Given the user input, the algorithm consists of two phases: establishing a correspondence between the two polyhedra and interpolating corresponding vertex locations.

4.1 Correspondence

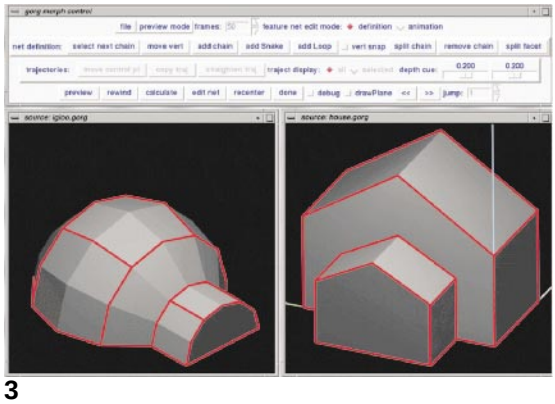
- *Feature net specification*. The user specifies a network of corresponding chains on the surfaces of the two input polyhedra by specifying the vertices of their endpoints as shown in Figs. 2 and 3. The interior edges of the chains are then computed as the shortest path along the edges between the specified endpoints. The feature net is a subgraph of the vertex/edge connectivity graph of each polyhedron.
- *Decomposition into morphing patches*. Based on the feature nets, the algorithm decomposes the surface of each polyhedron into the same number of morphing patches, each being homeomorphic to a closed disk.
- *Mapping*. A pair of corresponding morphing patches are mapped to a 2D polygon.
- *Merging*. The algorithm merges the topological connectivity of morphing patches in the 2D polygon.
- *Reconstruction*. Using the results from merging, the algorithm reconstructs the facets for the new morphing patch and generates a merged polyhedron with the combined topologies of the original two.
- *Local refinement*. The user can make local changes to the feature net, such as splitting chains, moving extremal vertices, deleting chains or extremal vertices, or adding new ones, and then recompute the merged polyhedron.

4.2 Interpolation

- *Trajectory specification*. The user specifies the trajectories for the vertices of the feature net to follow during the morph. The morphing trajectories for the remaining vertices of the merged polyhedron are computed from these.
- *Morph generation*. The algorithm makes use of the trajectories and interpolates the surface attributes to generate a morph.
- *Local control*. The user can modify the trajectories and generate a new morph. This step does not involve recomputation of the merged polyhedron, as shown in the shaded "feedback loop" of Fig. 1.



2



3

Fig. 2. Input polyhedra with user-specified correspondences

Fig. 3. User interface for igloo-house morph showing completed feature net (*red*) with morphing patches

5 Correspondence between polyhedra

In this section, we present the algorithm that computes the correspondence between two polyhedra. Given two homeomorphic polyhedra, $\boldsymbol{A}$ and $\boldsymbol{B}$, we represent their vertices as $\boldsymbol{V}^A = V_1^A,\ V_2^A,\ \ldots,\ V_m^A$ and $\boldsymbol{V}^B = V_1^B,\ V_2^B, \ldots,\ V_n^B$, respectively. Superscripts represent the corresponding polyhedron. The edges and faces of the polyhedra are represented as $\boldsymbol{E}^A$, $\boldsymbol{E}^B$, $\boldsymbol{F}^A$ and $\boldsymbol{F}^B$, respectively. The output is a merged polyhedron with the topology of both input models, for which each vertex has a location on the two input models.

The system ensures that each face of the input and output polyhedra is a triangle. Otherwise, the system triangulates the face. The boundary and topology information for each polyhedron is represented by an *adjacency graph*. Given the polyhedra, the system computes a circularly ordered set of edges for each vertex. For each edge, the system stores incident vertices as well as left and right adjacent faces. Each facet contains a counterclockwise-ordered list of three vertices and three edges. The vertices and edges of the adjacency graph represent the vertex/edge connectivity information of the polyhedron. We will use the symbols G^A and G^B to represent the adjacency graph of two polyhedra. Furthermore, we assign a weight to each edge of this graph. The weight corresponds to the euclidean distance between the two vertices defining the edge. Based on the user's specification, the correspondence algorithm marks some of the vertices and edges in these graphs. To start with, each edge and vertex in these graphs is unmarked.

To illustrate the correspondence algorithm described in this section, we will make use of Figs. 2–4 and 6–11. In our illustration, polyhedron $\boldsymbol{A}$ corresponds to a model of an igloo and polyhedron $\boldsymbol{B}$ corresponds to a model of a house (Fig. 2). Upper case letters denote 3D objects, and lower case letters represent 2D objects.

5.1 Specifying corresponding features

The user selects a pair of unmarked vertices on each of the input polyhedra to be in correspondence denoted by $\{V_{i1}^A,\ V_{i2}^A\}$ and $\{V_{i1}^B,\ V_{i2}^B\}$, respectively (Fig. 2). The algorithm computes a shortest path between these vertex pairs in the adjacency graph using only unmarked vertices and edges. Let the shortest paths correspond to $\{V_{i1}^A,\ V_{j1}^A,\ \ldots,\ V_{jk}^A,\ V_{i2}^A\}$ and $\{V_{i1}^B,\ V_{j1}^B,\ \ldots,\ V_{jl}^B,\ V_{i2}^B\}$, as shown in Fig. 2. All the intermediate vertices and edges on the shortest paths in each graph are marked. We call such a shortest path a *chain*. Moreover, the user-selected vertices are referred to as the *extremal vertices* of a chain. The selected vertices and chains are used to formulate a feature net for each polyhedron. We will represent the feature nets as N^A and N^B. They are subgraphs of G^A and G^B, respectively. The user needs to specify a sufficient number and arrangement of chains to partition the boundaries of the polyhedra. The algorithm imposes some constraints on the user. Each extremal vertex must be adjacent to at least two chains, and each chain must have a connected patch on each side. As a result, N^A and N^B have the same number of chains and extremal vertices, and the user has specified a mapping between each extremal vertex and chain. In this way, the two feature nets define a bijection.

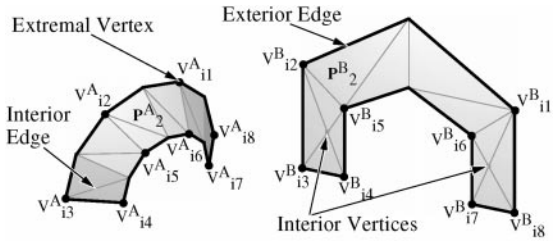


Fig. 4. Two selected morphing patches

5.2 Decomposition into morphing patches

The feature nets are used to decompose the boundary of each polyhedron into the same number of morphing patches. A morphing patch (Fig. 4) is simply a subset of a polyhedron that is homeomorphic to a closed disc, thus simplifying the geometric computations necessary to compute a morph. The vertices and edges are partitioned into exterior and interior vertices and edges. The exterior vertices of the morphing patches are those on the specified feature net (Fig. 4).

The decomposition algorithm has two steps. First, the perimeters of the morphing patches are computed by traversing the feature nets. Second, the interiors of the morphing patches are computed. Here is an overview of the algorithm to partition the feature net into morphing patch perimeters.

```
Partition polyhedron into morphing patch
perimeters(){
  For each extremal vertex in the feature
  net Vi {
     For each chain Cj adjacent to Vi
     that does not already have an adjacent
     patch clockwise from Vi {
        1. OppVert = the extremal vertex at
        the other end of Cj from Vi
           CurrentChain = Cj
           While (OppVert != Vi ){
            2. CurrentChain =the next
                            clockwise chain
                            adjacent to
                            OppVert from
                            CurrentChain
            3. OppVert =the extremal vertex at
                       the other end of
                       NextChain from OppVert
        }
     }
  }
}
```

The computation of the perimeter, as detailed in the pseudo-code above, uses the circular ordering of edges at each vertex. Beginning at a vertex and chain of the feature net, the algorithm walks through the tightest clockwise loop of extremal feature net vertices it can find. From the first extremal vertex and chain, it moves to the extremal vertex at the other end of the chain (step 1). Next it uses the circular ordering of edges at that vertex to proceed to the closest clockwise chain (step 2). Then it follows that chain to the vertex at the other end (step 3). This process continues until it comes back to the original vertex, hence traversing the perimeter of a morphing patch. Note that this is possible because the chains may not cross. Since there is a bijection between the feature nets, this process is performed simultaneously on both of the input polyhedra. For example, in step 2, the next corresponding pair of clockwise chains can be determined by examining the underlying graph of only one of the input models.

The interior of a morphing patch is computed with a *depth-first search* algorithm modified to traverse through faces of the graph instead of vertices. It starts with an arbitrary edge on the perimeter of the morphing patch, and determines which adjacent face is interior to the patch by choosing the one with the same ordering as that of the exterior vertices. Then it crosses the face that is on the interior of the morphing patch and recursively branches out to the faces on its other two edges. The recursion stops at an edge that has already been traversed or is part of the perimeter. This process is repeated until all vertices, edges, and faces of the original polyhedra have been partitioned into morphing patches. These morphing patches are represented by $P_1^A, \ldots, P_K^A$ and $P_1^B, \ldots, P_K^B$. The system checks that the interior of each patch is homeomorphic to a closed disk, requiring additional specification from the user if that is not the case.

For a genus-zero polyhedron, no morphing patch could contain a hole. If the input polyhedron has a genus greater than zero, the user needs to specify the features in such a manner that each morphing patch is homeomorphic to a disk. Such a decomposition is always possible, as we will see later for a cup and torus example.

5.3 Mapping

Given a morphing patch, our goal is to compute a parameterization over a convex polygonal region in two dimensions. Construction of a parameterization

for complex shapes over a simple domain is an important problem that occurs in various applications.

5.3.1 Desiderata

A mapping algorithm requires a parameterization that is intuitive from the user's point of view. When the user describes a pair of corresponding morphing patches on the input polyhedra, s/he should be able to intuitively imagine how the interiors will be mapped without any detailed understanding of the system. After some experimentation, it appears that we want the mapping to have the following property. Given any two triangular faces F_i^A and F_j^A of a morphing patch, let them map to the triangles f_i^A and f_j^A, respectively, in the 2D polygon. The ratio of the areas between F_i^A and F_j^A should be close to that of f_i^A and f_j^A. Individual triangles are not as important here as the fact that the mapping minimizes the distortion of the area across the patch as a whole. This leads to a more predictable morph for the interiors of corresponding morphing patches. For example, in Fig. 5, one would not expect that the corresponding morphing patches to split apart during the morph, but instead one patch should bend into the other. The solution is equivalent to taking a uniform coordinate system on the 2D polygon as if it were composed of rubber sheets with no potential energy, and placing it on the surface of the 3D morphing patch so that the energy is minimized.

5.3.2 Previous approaches

A number of algorithms have been proposed by Kent et al. [26], Maillot et al. [32], and Eck et al. [15]. One possible solution is to use harmonic maps. They minimize the metric distortion and preserve the aspect ratios of the triangle, but can introduce area compression [15]. In Fig. 5, it is area compression that causes the intermediate model, HP, to have two peaks instead of one. A harmonic map may not produce a desirable mapping. It would treat the morphing patch as if it were composed of triangular rubber sheets with no potential energy in three dimensions, and minimize their total energy after being placed into two dimensions.

5.3.3 Our approach

We currently use a divide-and-conquer approach with an area preservation heuristic. An example is shown in Fig. 6. Note that, if the ear had been mapped with a harmonic map, all the triangles would be very small in the middle; instead, the areas are more uniform at the expense of distorting the shape.

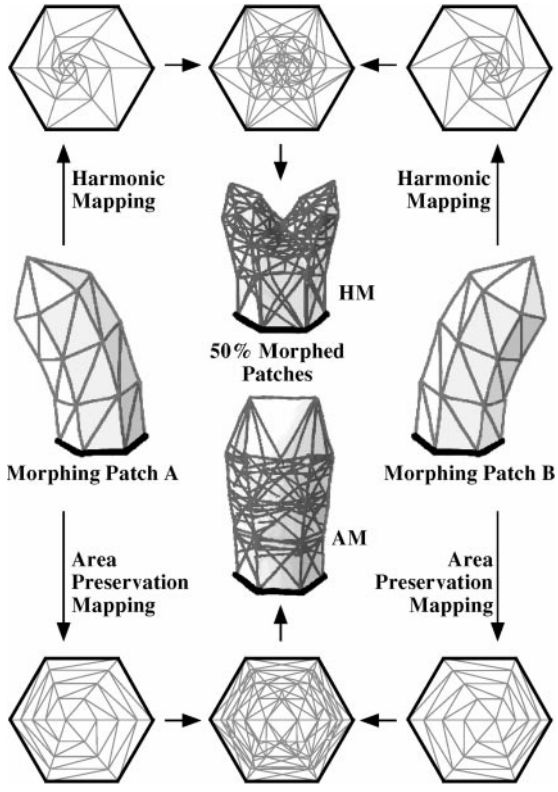


Fig. 5. Comparison between the area-preservation and harmonic mappings. A corresponding pair of morphing patches ***A*** and ***B*** are mapped to a 2D polygon using both a harmonic mapping, and an area preservation mapping. The two mappings are then merged and reconstructed, and the resulting fifty percent morphs are shown for the patch, ***HM*** and ***AM***, respectively. Note that, from a user standpoint one would expect the intermediate patch to look more like ***AM*** than ***HM***. During the morph of ***AM***, patch ***A*** appears to bend over into patch ***B***, whereas during the morph of ***HM*** the tip of patch ***A*** shrinks into the patch ***B***'s side while the tip of patch ***B*** grows out of patch ***A***'s side

Given a pair of morphing patches, P_i^A and P_i^B, we compute a mapping from the surface of the patch to a regular 2D polygon. Let these morphing patches consist of m_i extremal vertices. We map the morphing patch into a regular 2D polygon, inscribed in the unit circle, with m_i edges. We represent the regular 2D polygon as p_i.

To compute a mapping, our algorithm first establishes a bijection between the chains of the two

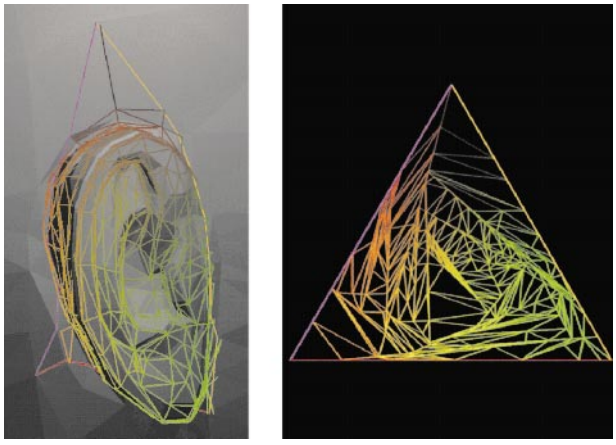


Fig. 6. A morphing patch containing an ear mapped to a 2D polygon with our algorithm

feature nets by splitting edges on the respective chains on the two models. The splitting criterion is based on edge lengths. The splitting of edges does not require the splitting of triangles. After splitting, the algorithm has ensured that the corresponding patches have the same number of exterior edges and vertices.

The extremal vertices of the morphing patches are mapped to the vertices of p_i. Each chain of the morphing patch is mapped to an edge of p_i. All the external vertices lying in the interior of a chain are mapped onto the edges of p_i. The 2D coordinates of the vertices along the chains are interpolated on the basis of the arc length of the chain.

The next step is to compute a mapping for the interior vertices of P_i^A and P_i^B. We use a simple recursive technique that tries to preserve the ratio of areas of the triangles and is based on a greedy heuristic. The algorithm divides a morphing patch by selecting two exterior vertices V_i^A and V_j^A that do not lie on the same chain. Next, it computes a shortest path across the interior of the morphing patch. This path is then adjusted so that the ratio of the surface area on each side is as close as possible to the ratio that it will have once mapped to the 2D polygon. The vertices and edges lying on this path are mapped to the interior of p_i, along the segment connecting V_i^A and V_j^A. On the basis of this path, the algorithm recursively divides the morphing patch and maps the subpatches to p_i. This process is then repeated for P_i^B. A pseudo-code description of the algorithm to map the interior of a morphing patch to a 2D polygon is given here.

```
MapInterior(Perimeter){
 if Perimeter surrounds any vertices
 (encompasses > 1 facet){
    1 SplitPath = the shortest path between two
         Perimeter vertices
         that approximately divides
         the Perimeter in half;
    2 If there is no subdividing path between
         Perimeter vertices, then split the
         Perimeter edges until there is one;
    3 Optimize SplitPath to preserve the area
         ratio;
    4 Interpolate the 2D coordinates at the
         endpoints of SplitPath along its
         interior;
    5 LPerimeter =SplitPath + part of
                  Perimeter on its left;
      RPerimeter = SplitPath + part of
                  Perimeter on its right;
    6 MapInterior(LPerimeter);
                  MapInterior(RPerimeter);
    7 Remove any extra vertices and edges added
         in step 2.
 }
}
```

The shortest path found in steps 1 and 2 tends to preserve the area ratio on each side from the 3D model to the 2D polygon if the portion of the morphing patch enclosed by the perimeter does not have much curvature. Otherwise, the algorithm modifies the path between V_i^A and V_j^A in step 3, until the ratio of the surface area of the morphing patch on either side of the 3D path is as close as possible to the areal in the 2D polygon on either side of the line it is mapped to.

At this point, the algorithm has computed a parameterization for each morphing patch such that each triangular face F_j^A and F_k^B has been mapped to a corresponding triangle, f_j^A and f_k^B, respectively, in p_i. The next step of the algorithm is to compute a mapping for each interior vertex of P_i^A to P_i^B (and vice versa).

Given an interior vertex of P_i^A, the algorithm locates the triangle, say f_k^B, that contains the image of that interior vertex. Furthermore, the algorithm computes the corresponding point in f_k^B and represents it with barycentric coordinates in terms of vertices of f_k^B. The barycentric coordinates are then applied to the vertices of F_k^B to compute the corresponding point on P_i^B.

This process is repeated for all the interior vertices of P_i^A and P_i^B. The time complexity of this part of the algorithm depends on the complexity of locating the triangle for each interior vertex of P_i^A. A simple

search procedure would be linear in the number of triangles. However, using efficient data structures for planar point location [5], which involves linear time preprocessing, this search time can be reduced to be logarithmic in the number of triangles.

5.4 Merging

The algorithm has so far produced mappings into p_i such that the vertices $V_j^A \Rightarrow v_j^A$, $V_j^B \Rightarrow v_j^B$, and edges $E_j^A \Rightarrow e_j^A$, $E_j^B \Rightarrow e_j^B$. The edges e^A and e^B will in general intersect. We compute the intersections, split the intersecting edges, and create new vertices (Fig. 7).

Let n_e be the total number of edges, and let k_e be the number of edge pairs that actually intersect. In the worst case, k_e can be $O(n_e^2)$. Efficient and optimal algorithms of complexity $O(n_e \log n_e + k_e)$ have been proposed by Clarkson and Shor [10] to compute the intersections. However, we are not aware of any robust implementations of these algorithms.

In our application, we encounter many degenerate edge configurations. These include almost coincident edges and vertices. Motivated by simplicity and robustness, we used an algorithm of complexity $O(n_e^2)$, which checks all edge pairs for overlap. Since the intersection computations can fail on edges that are coincident, we handle the case in which edges lie on the same mapped path separately. To avoid creating an invalid topology, we first calculate all the edge intersections, and sort the intersection points on each edge before creating the output edges. After intersection computation and splitting, we denote the set of all vertices and edges in p_i by x^{AB} and g^{AB}, respectively.

5.5 Reconstruction

After computing the intersection of all the edges, the algorithm produces a planar straight-line graph (PSLG) [5] from those intersections. The PSLG is constructed from the x^{AB} and g^{AB}. The next step is to compute a triangulation of these PSLGs. Though good theoretical algorithms of linear complexity are known [43], it is unclear if they can handle PSLGs (Fig. 8) that have almost collinear edges or have very small angles between them. To handle such cases robustly, we use a simple divide-and-conquer algorithm. From the counterclockwise ordering of the winged edge data structure of one of the input models superimposed on the other, we recursively subdivide the connected edges and vertices from the merging step into the smallest counterclockwise cycles possible. Since the nature of the intersections in the merging step guarantees these cycles are convex, from this point we can triangulate the regions. After this step has been performed on all the morphing patches, we get a merged polyhedron, as shown in Figs. 9 and 10.

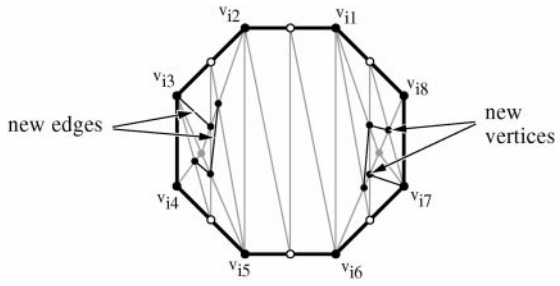


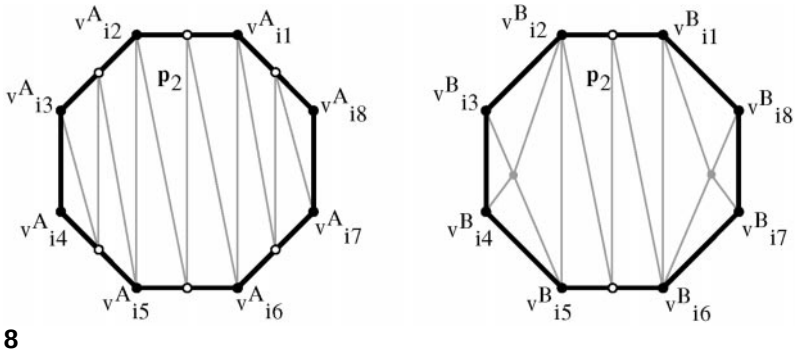
Fig. 7. The 2D polygons are combined into one polygon with merged topology

6 Interpolation

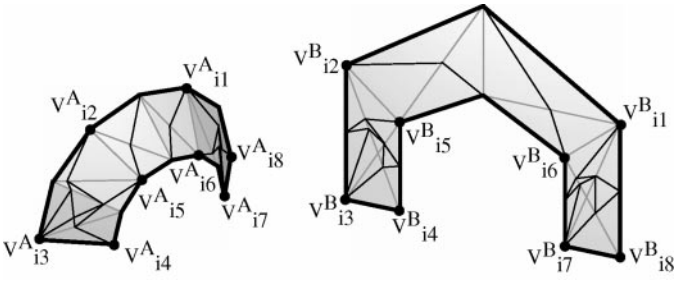
At the end of the correspondence process, the merged polyhedron has the combined topological connectivity of polyhedra ***A*** and ***B***. Each vertex on polyhedron ***A*** has a corresponding vertex on polyhedron ***B***. In this section, we discuss issues for interpolating between the two polyhedra to generate a morph.

6.1 Aligning the input models

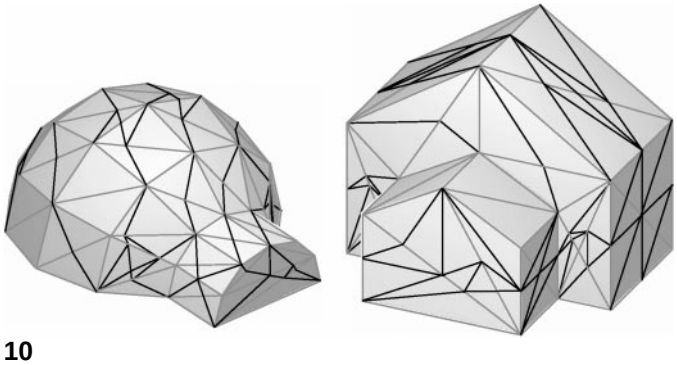
In many cases the location, orientation and size of the input polyhedra are quite different. For example, in Fig. 11, the igloo is much smaller than the house and has been positioned centrally on the ground, against the back wall of the house. The user must do the scaling, positioning, and orienting the input polyhedra with respect to each other in preparation for 3D morphing. This influences the appearance of the morph directly. This requirement is quite similar to the preparation required for 2D image morphing [4]. Note that if the user changes the relative scale, position, or orientation of the two input polyhedra, only



8



9



10

Fig. 8. The morphing patches are mapped to regular 2D polygons
Fig. 9. The output after merging and reconstruction steps
Fig. 10. The reconstructed merged polyhedron has the combined topology of the two input polyhedra

the morphing trajectories must be re-specified and interpolated (shaded feedback loop in Fig. 1); the correspondence specifications remain valid.

6.2 Morphing trajectories

During morphing, the vertices travel from their positions on ***A*** to their respective positions on ***B*** along morphing trajectories. Kent and colleagues [26] suggest using Hermite interpolation between the corresponding vertices with the tangents pointing along normal directions. Similarly, we allow the user to represent the trajectory as a Bézier curve for each pair of extremal vertices. Initially, the trajectories are specified by the user for each extremal vertex. The trajectories are represented as cubic Bézier curves and denoted by $B_V(t)$ for each vertex V. The two endpoints lie on ***A*** and ***B***, respectively. The user specifies the tangent directions at each endpoint. Based on the tangents, the algorithm computes the

control points for each Bézier curve using Hermite interpolation. The user can modify the tangents for each trajectory belonging to an extremal vertex of the feature net, as shown in Fig. 11 (which shows the tangent vectors as green line segments). Starting from the morphing trajectories of the extremal vertices, the system computes trajectories for all other vertices of the merged polyhedron. Various methods are used for interpolating between chain vertices and interior vertices:

1. *Chain vertices*. For each chain vertex of a morphing patch, only the two adjacent extremal vertices of the chain are used. The algorithm computes weighting factors based on their distances along the chain (or arc length) from the chain vertex.
2. *Interior vertices*. For each interior vertex of a morphing patch, all extremal vertices lying on the boundary of that morphing patch are used. The algorithm computes weighting factors for each extremal vertex based on their distances along shortest paths toward the interior vertex. The weighting function is similar to that of Beier and Neely [4].

In both these cases, the weighting factors are applied to the tangent vectors at each endpoint of a trajectory. For trajectory endpoints on polyhedron ***A***, the endpoint vectors on ***A*** are averaged. For trajectory endpoints on ***B***, the endpoint vectors on ***B*** are averaged. This process results in interpolated tangent vectors for trajectories at each vertex, which are then used to compute the two inner Bézier control points (also shown in green in Fig. 11). Note that this may result in morphing trajectories that are not straight lines, even if the user specifies all extremal vertex trajectories as straight lines.

The speed at which a vertex travels along the morphing trajectory is determined by sampling based on the "frame" number in the morphing sequence. For example, if the morph is to have 100 frames, then at frame 30, each point V will be at the position $B_V(0.3)$. Beyond this, the user may want to specify a nonlinear mapping between the frame number and the value of t, in order to control the speed at which morphing takes place. The algorithm also allows the user to individually modify this mapping for individual extremal vertices, in order to make some parts of the polyhedrons morph sooner or later than others. This is analogous to the techniques used for 2D image morphing [4].

6.3 Interpolating surface attributes

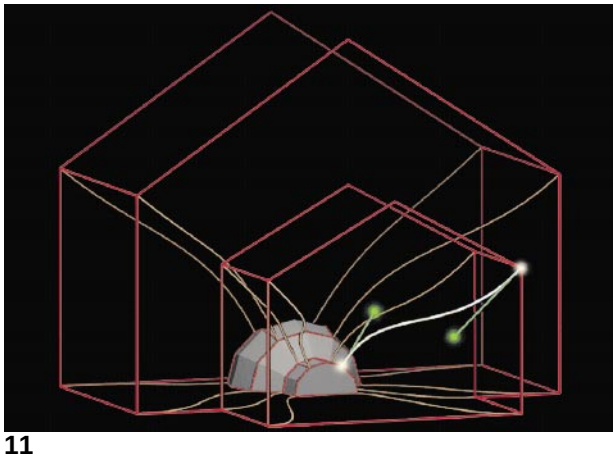
In addition to the morphing trajectories required by the algorithm, other attributes of the input polyhedra need to be interpolated to generate a good morph. These include vertex colors, lighting coefficients, normal vectors, etc. Interpolation of these surface attributes occurs during the mapping and merging steps. Separate values are computed for the attributes of ***A*** as well as for the attributes of ***B***. During morphing between ***A*** and ***B***, the attributes are linearly interpolated between their values corresponding to each polyhedron.

Normals are a case that requires special attention. They are problematic, not only because they can be used to represent a smooth surface, but because they can also define creases or hard edges in a model. This case is handled by storing four normals for each edge: one for each vertex for the face on each side of the edge. Hence, the merged polyhedron will have eight normals per edge (four for each source model). Now, by simply interpolating, a crease can be morphed into a smooth surface. This can be observed in Fig. 12.

7 User interface

The user interface is one of the most important aspects of a morphing system. Although it is easy for the user to conceptualize a morph between two objects, it can be rather difficult to design a system that allows the user to express this easily. Our system achieves this goal by allowing the user to draw the key correspondences on the surfaces of the input models, and to specify the paths that the corresponding features will follow during the morph, taking advantage of graphics hardware to allow real-time interaction.

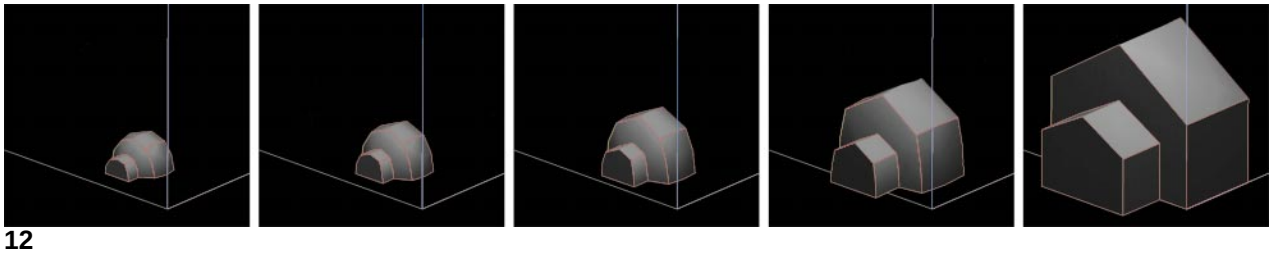
The user specifies corresponding chains of the feature nets for input models ***A*** and ***B*** by selecting the chains' endpoints. In order to enforce a bijection between the two feature nets, the system requires the user to specify the feature net vertices in corresponding pairs. The interior of a chain is computed as the shortest path of unmarked vertices and edges between its endpoints, which are then marked so that another chain cannot cross or overlap it. In the case



11

Fig. 11. S-shaped morphing trajectories for the extremal vertices of the feature nets. The highlighted (*white*) trajectory is shown with its Bézier control points, which are the fixed endpoints of the trajectory. The *green* control points can be moved by the user

Fig. 12. Morph of the igloo into the house



12

that a path of unmarked vertices and edges between a chain's endpoints is not available, the system creates new vertices and edges by splitting the necessary face(s).

As a simple extension to creating a single pair of corresponding chains, we also allow the user to create *multichains* and *loops*. When creating a multichain, after the user has specified the first corresponding feature vertex pair, each additional vertex pair s/he specifies makes a pair of chains connecting to the last vertex pair. A loop is simply a multichain with the property that the last pair of corresponding vertices is connected to the first pair with an additional pair of chains.

Once the corresponding pairs of chains have been specified, we allow several techniques for *local* refinement. These include splitting a chain into two chains at a selected vertex on one of the input models, removing a chain, and moving extremal vertices of the feature net. Figure 3 shows the user interface and the feature nets on the two input models.

After a corresponding pair of extremal vertices have been specified on polyhedron $\boldsymbol{A}$ and polyhedron $\boldsymbol{B}$, the user can control the morphing trajectory. The user can position the tangents of the trajectory, whose endpoints are the location of the vertices on the two input models as shown in Fig. 11. By default, the morphing trajectory is a straight line. After the user specifies the feature net, the rest of the morph is calculated as explained in Sects. 5 and 6. Another additional feature of the interface allows the user to easily adjust it by refining the feature net *locally*. As already mentioned, it is not necessary to recalculate the merged polyhedron if the user only edits the morphing trajectories shown in Fig. 11.

8 Implementation and performance

We have implemented the system in C++ using the OpenGL and Tcl/Tk libraries. It features a graphical user interface for specifying features and trajectories and for refining the morph.

The input polyhedra are specified in a shared vertex representation. The adjacency graph of each polyhedron is stored so that each vertex has a list of edges stored in counterclockwise order, each edge contains the incident vertices and two facets, and

each facet contains three vertices and three edges, also stored in counterclockwise order. Furthermore, the system ensures that each polyhedron has valid topology and that it satisfies the Euler–Poincaré formula.

When the user specifies the extremal vertices of a chain in the feature net, the system computes the path connecting them using Dijkstra's shortest path algorithm. It starts with one of the extremal vertices as the start vertex and incrementally computes shortest paths to other vertices of the polyhedron. It stops when it has computed the path to the other extremal vertex. Since the endpoints of chains are typically close and the shortest path consists of a few edges, the system can compute these paths fast enough for interactive response.

Our implementation also utilizes a number of geometric algorithms for triangulating planar straight-line graphs, edge intersections, and point location. As mentioned in Sect. 4, we have opted for simplicity and robustness rather than efficiency or algorithms with optimal asymptotic performance.

8.1 Performance improvement

The merging algorithm described in Sect. 5.4 computes all intersections between the mapped edges of each morphing patch. Based on the decomposition algorithm described in Sect. 5.2, a morphing patch of a large polyhedron may consist of thousands of edges. The number of intersections (and thereby the combinatorial complexity of the merged polyhedra) grows with the number of edges and, in the worst case, is a quadratic function of the number of edges. As a result, the merging and reconstruction steps can become a bottleneck in the overall computation. To overcome this problem, we subdivide each morphing patch into smaller subpatches such that each subpatch consists of at most Q edges. A typical value for Q in our implementation is 100.

We subdivide the patches with a recursive divide-and-conquer algorithm. It is quite similar to the mapping algorithm presented in Sect. 5.3 and starts with computing a path between the external vertices of a patch. The system computes a corresponding path on the other patch, and tries to preserve the ratio of the areas on either side of each chain. At the same time, it maintains the bijection between the feature nets. Hence it divides each patch into subpatches. This procedure is applied recursively, till each subpatch has less than Q edges.

The division of morphing patches into subpatches reduces the overall computation time, as well as the size of the polyhedra that are obtained after the merging and reconstruction steps. Furthermore, it also speeds up the interpolation algorithm.

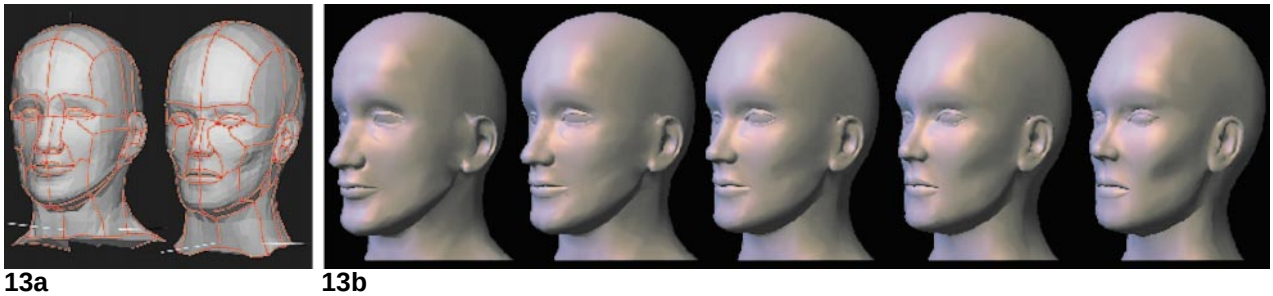
8.2 Results

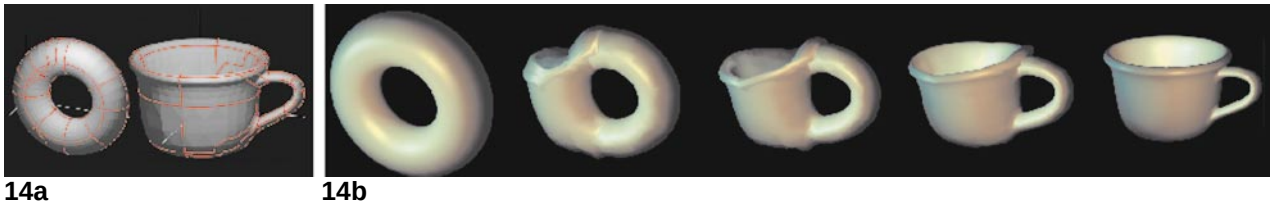
Our system has been applied to a number of complex polyhedral models and used to create several morphs successfully. These include simple polyhedra (Fig. 10) as well as non-simple polyhedra corresponding to a torus and a cup (Fig. 14). We present the results in Table 1. It includes the complexity of input and output models, their genus, the number of extremal vertex pairs specified, and the number of morphing patches. The table also reports the times required on a SGI Onyx 2 with 195 MHz R10 000 by a user to specify the features, the trajectories, and the time to compute the merged polyhedra.

9 Analysis

In this section, we analyze our algorithm. We first present an asymptotic bound on its running time and then analyze the results produced by it.

Assume we have two polyhedra with m and n vertices. The algorithms for checking the topology of a polyhedra and constructing the adjacency graph take at most $O(m+n)$ time. Let the number of extremal vertex pairs specified by the user be k. For large models, k is much smaller than m and n. The time to compute the feature nets is dominated by the shortest-path computation algorithm. In the worstcase, the shortest-path algorithm can take $O(km+kn+m\log m+n\log n)$ time, but in practice it is much less because the length of shortest path is typically small. The number of morphing patches can be at most $O(k)$. The computation of feature nets and morphing patches involves use of a depth-first search, and its overall time is bounded by $O(k(m+n))$. After the subdivision algorithm, presented in Sect. 8.1, each morphing subpatch can have up to Q edges and the number of morphing subpatches can be $O((m+n)/Q)$. The complexity of


13a 13b


14a 14b

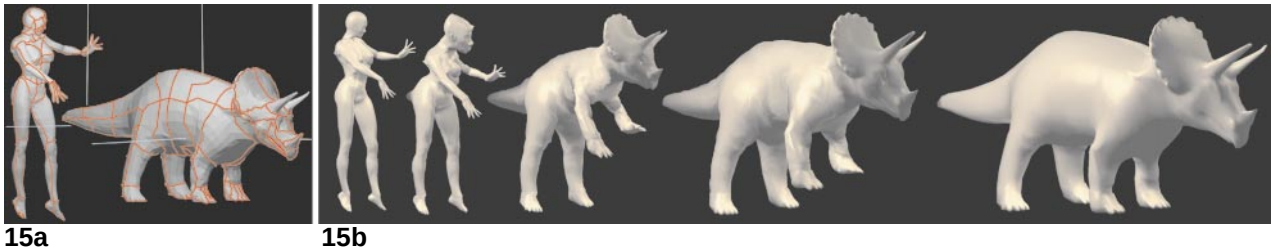

15a 15b

Fig. 13. a Human-heads morph. The male head consists of 3426 triangles. The female head consists of 4020 triangles. The feature nets (*red*) consists of 134 extremal vertices on each of the two polyhedra. All morphing trajectories are straight lines; **b** The merged polyhedron (computed in approx. 30 sec on an R10 000 CPU) as it morphs between the heads

Fig. 14. a Doughnut-cup morph. The doughnut consists of 8452 triangles. The feature nets (*red*) consists of 63 extremal vertices on each of the two polyhedra. Most morphing trajectories are straight lines, except for a few around the rim of the cup, where material was "routed" along curved paths to avoid self-intersections; **b** The merged polyhedron (computed in approx. 1 min on an R10 000 CPU) as it morphs between cup and doughnut

Fig. 15. a Human-triceratops morph. The human consists of 17 528 triangles. The triceratops consists of 5660 triangles. The feature nets (*red*) consists of 185 extremal vertices on each of the two polyhedra. Most morphing trajectories are straight lines except a few around the rather flat tail, which was made to "inflate" slightly to avoid self-intersections; **b** the merging polyhedron (computed in approx. 2.5 min on an R10 000 CPU) as it morphs from human to triceratops

merging and reconstructing each pair of morphing subpatches is $O(Q^2)$. The interpolation algorithm needs to compute the length of the shortest path from each interior vertex of a morphing subpatch to each extremal vertex of the morphing subpatch. We make use of single-source shortest-path algorithms and compute the paths for each extremal vertex on the boundary of the morphing patch. In the worst case, it can take $O((m+n)k \log Q)$ time. As result, the overall complexity of the algorithm is $O(K(m+n))$, where $K = \max\{k \log Q,\ Q\}$.

Our approach does not suffer from the ghosting problems seen in image and volume morphing [4, 31]. However, a similar scenario can occur: self-intersection. We can check for it automatically with collision detection algorithms, but this is prohibitively expensive. Furthermore, self-intersection may be desirable, and in some cases even necessary in order to allow some part of a morphing object to reach its target position. Hence, it is currently the responsibility of the user to "reroute" portions of a morphing polyhedron by controlling the morphing trajectories to

avoid self-intersections. (For example, we prevented the two sides of the triceratop's flat tail in Fig. 15 from temporarily passing through each other in this way.)

The visual quality of a morph created with our system is quite subjective. As with conventional 2D image morphing, it is influenced primarily by the number of detailed feature correspondences. Beyond that, controlling the morphing trajectory strongly contributes to a smoothly flowing appearance during morphing. Finally, similarly to 2D techniques, morphs between objects that are similar in appearance (for which one intuitively notices the corresponding pairs of features by simple visual inspection) result in smoother transitions than morphs between vastly different objects (cf. the human heads morph in Fig. 13 and the cup-doughnut morph in Fig. 14).

10 Current limitations and future work

The specification of the feature net suffers from two limitations. First, the feature net must be connected. It would be nice to remove this restriction by automatically dividing the feature net into connected components, then adding enough chains to connect them so that each morphing patch is homeomorphic to a disk. Second, the chains connecting the feature net vertices are currently restricted to lie on the edges of the source models. It would be beneficial to allow the user to draw on the surface instead of the graph of the polyhedron. It would also be useful to remove the restriction that the input models have to be homeomorphic – one could imagine that the user merely specifies the correspondence, and then the system creates a polyhedron that can appear to have the topology of either of the input models.

One of the largest limitations of the current system is the user interface for controlling the morphing trajectories. It can be very difficult to control the shape of the intermediate models by positioning the tangents of a cubic curve. We envision a vast improvement that would allow the user to sculpt the intermediate models and have the system automatically calculate the morphing trajectories that include the points for the intermediate models sculpted by the user.

The performance of the mapping algorithm highlighted in Sect. 5.3 varies with the triangulation of the morphing patch. Since it uses a greedy heuristic, the algorithm may not be able to preserve the area ratios. This can result in some noticeable distortions in the morph, especially when the mapping algorithm introduces area compression for one of the patches. Currently, the user can work around this by adjusting the morphing trajectories. There is considerable literature in Graph Drawing on planar embedding of planar graphs. We plan to apply some of those techniques to our problem.

Texture coordinates can be interpolated with other surface attributes of the input polyhedra. However, this will yield correct results only if both the polyhedra use a common texture map. In such a case, the mapped (unique) texture will seem to flow smoothly across the surface of the polyhedron during morphing. Our algorithm currently handles this case. In many cases, each of the input polyhedra may have different texture maps. One possible solution is to use a weighted blend between the textures of $\boldsymbol{A}$ and $\boldsymbol{B}$, controlled by the morphing interpolation factor t. This would result in an effect similar to an image fade-over between the textures of $\boldsymbol{A}$ and $\boldsymbol{B}$. Our algorithm could easily be extended to handle this situation. However, we believe that a more powerful effect could be attained by allowing the user to perform a controlled

Table 1. Performance of our algorithm on four pairs of input polyhedra

Models		Triangles		Output triangles	Morphing patches	User specification time	Time to compute merged polyhedron
House–igloo	Fig. 3	82	40	214	10	$\sim$ 5 min	$<$ 1 s
Human–triceratops	Fig. 15	5660	17 528	97 900	86	$\sim$ 6 h	2.5 min
Human–heads	Fig. 13	3426	4020	32 520	67	$\sim$ 3 h	30 s
Donut–cup	Fig. 14	4096	8452	61 701	50	$\sim$ 4 h	1 min

conventional 2D morph in texture space together with the 3D polyhedral morph specified by our algorithm.

Morphing between animated models. In traditional 2D morphing, an image sequence of a moving actor can be morphed into another sequence showing a different moving actor. This is typically accomplished by (tediously) respecifying correspondences for each pair (or at least for many pairs) of frames within the image sequences [4]. An extension of our approach could handle this problem in the following way: once correspondences have been specified for a computer-animated character, they can remain attached to the character's topology and carry over throughout the animation sequence. In other words, the correspondence features and the character are animated together. As for the morphing trajectories, they would have to be specified, at least for the first and last frames of the animation, and would have to be interpolated for all other frames.

11 Summary

We have presented a new approach for establishing a correspondence for morphing between two homeomorphic polyhedra, which includes a simple, intuitive user interface. It has been successfully applied to a number of polyhedral models, including ones that are not genus-zero. We believe it is versatile enough to produce visually pleasing 3D morphs, once it is coupled with an effective method for specifying the interpolation between the two models.

Acknowledgements. This research has been funded in part by an Alfred P. Sloan Foundation Fellowship, Armly Research Office (ARO) Contract CAAH04-96-1-0257, National Science Foundation (NSF) Career Award CCR-9625217, Office of Naval Research (ONR) Young Investigator Award (N0014-97-1-0631), Honda, Intel, and NSF Center for Computer Graphics and Scientific Visualization. We thank R. and S. Gregory for additional support, Rhinoceros for making the beta version of their modeling program freely available, S. Nelson for the triceratops model we found on the web, T. Gaul for video editing, and H. Fuchs.

References

1. Alexa M (1999) Merging polyhedral shapes with scattered features. Proceedings of Shape Modeling International
2. Bao H, Peng Q (1998) Interactive 3D morphing. Comput Graph Forum 17:C23–C30
3. Barr A (1984) Global and local deformations of solid primitives. ACM Comput Graph 18:21–30
4. Beier T, Neely S (1992) Feature-based image metamorphosis. (SIGGRAPH '92) Comput Graph :35–42
5. Berg MD, et al. (1997) Computational geometry: algorithms and applications. Springer, Berlin, Heidelberg, New York
6. Bethel E, Uselton S (1989) Shape distortion in computer-assisted keyframe animation. In: Magnenat-Thahlmann N, Thalmann D (eds) State of the art in computer animation. Springer, Berlin, Heidelberg, New York, pp 215–224
7. Chen E, Parent R (1989) Shape averaging and its applications to industrial design. IEEE Comput Graph Appl. 9:47–54
8. Chen M, Jones MW, Townsend P (1995) Methods for volume metamorphosis, In: Wilbur Y.P.a.S. (ed) Image processing for broadcast and video production
9. Chen D, State A, Banks D (1995) Interactive shape metamorphosis. Proceedings of the 1995 Symposium on Interactive 3D Graphics, pp 43–44
10. Clarkson KL, Shor PW (1989) Applications of random sampling in computational geometry, II. Discrete Comput Geom 4:387–421
11. Cohen-Or D, Levin D, Solomovici A (1996) Contour blending using warp-guided distance field interpolation. in: IEEE Visualization '96
12. Cohen-Or D, Levin D, Solomovici A (1998) Three-dimensional distance field metamorphosis. ACM TOG 17:116–141
13. Covell M, Withgott M (1994) Spanning the gap between motion estimation and morphing. Proceedings of the IEEE International Conferences on Acoustics, Speech and Signal Processing 5:213–216
14. DeCarlo D, Gallier J (1996) Topological evolution of surfaces. Proceedings of Graphics Interface'96,
15. Eck M, et al. (1995) Multiresolution analysis of arbitrary meshes. (SIGGRAPH '95) Comput Graph 173–182
16. Galin E, Akkouche S (1996) Blob metamorphosis based on Minkowski sums. Comput Graph Forum 15:143–153
17. Gomes J et al. (1998) Warping & morphing of graphical objects. In: Barsky BA (ed) Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann
18. Gregory A, et al. (1998) Feature-based surface decomposition for correspondence and morphing between polyhedra. Proceedings of Computer Animation
19. Guibas L, Hershberger J (1994) Morphing simple polygons, Proceedings of the 10th Annual ACM Symposium on Computational Geometry, pp 267–276
20. He T, Wang S, Kaufmann A (1994) Wavelet-Based volume morphing. Proceedings of IEEE Visualization 85–91
21. Hodgins J, Pollard N (1997) Adapting simulated behaviors for new characters. (SIGGRAPH '97) Comput Graph 153–162
22. Hong T, Magnenat-Thalmann N, Thalmann D (1988) A general algorithm for 3D shape interpolation in a facet-based representation. Proceedings of Graphics Interface '88, pp 229–235
23. Hughes J (1992) Scheduled Fourier volume morphing. (SIGGRAPH '92) Comput Graph 43–46
24. Kanai T, Suzuki H, Kimura F (1997) 3D Geometric metamorphosis based on harmonic maps. Pacific Graph 97–104

25. Kaul A, Rossignac J (1991) Solid-interpolating deformations: construction and animation of PIPs. Proceedings Eurographics 493–505
26. Kent J, Carlson W, Parent R (1992) Shape transformation for polyhedral objects. (SIGGRAPH '92) Computer Graph 26:47–54
27. Kent J, Parent R, Carlson W (1991) Establishing correspondences by topological merging: a new approach to 3D shape transformation. Proceedings of Graphics Interface '91 pp 271–278
28. Lazarus F, Verroust A (1994) Feature-based shape transformation for polyhedral objects. The 5th Eurographics Workshop on Animation and Simulation
29. Lazarus F, Verroust A (1998) Three-dimensional metamorphosis: a survey. Visual Comput 14:373–389
30. Lee S, et al. (1995) Image metamorphosis using snakes and free-form deformations. (SIGGRAPH '95) Comput Graph 439–448
31. Lerios A, Garfinkle C, Levoy M (1995) Feature-based volume metamorphosis. (SIGGRAPH '95) Comput Graph 449–456
32. Maillot J, Yahia H, Veroust A (1993) Interactive texture mapping. (SIGGRAPH'93) Comput Graph 27–34
33. Mortenson ME (1985) Geometric modeling. Wiley, New York
34. Parent R (1992) Shape transformation by boundary representation interpolation: a recursive approach to establishing face correspondences. J Visualization Comput Anim 3:219–239
35. Parent R (1995) Implicit function based deformations of polyhedral objects. Implicit Surfaces '95,
36. Payne B, Toga A (1992) Distance field manipulation of surface models. IEEE Comput Graph Appl 12:65–71
37. Ranjan V, Fournier A (1996) Matching and Interpolation of Shapes using unions of circles. Comput Graph Forum 15:129–142
38. Reeves WT (1983) Particle systems – a technique for modeling a class of fuzzy objects. ACM Trans Graph 2:91–108
39. Rosenfeld M (1987) Special effects production with computer graphics and video techniques. SIGGRAPH '87 Course Notes No. 8
40. Sederberg T, Greenwood E (1992) A physically based approach to 2D shape blending. (SIGGRAPH '92) Comput Graph 26:25–34
41. Sederberg T, Parry S (1986) Free-form deformation of solid geometric models. (SIGGRAPH '86) Comput Graph 20:151–160
42. Sederberg T, et al. (1993) 2D Shape blending: an intrinsic solution to the vertex path problem. (SIGGRAPH '93) Comput Graph 15–18
43. Seidel R (1991) A simple and fast randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Comput Geom Theory Appl 1:51–64
44. Seitz S, Dyer C (1996) View morphing: synthesizing 3D metamorphoses using image transforms. (SIGGRAPH '96) Comput Graph 21–30
45. Shapira M, Rappoport A (1995) Shape blending using the star-skeleton representation. IEEE Comput Graph Appl 15:44–50
46. Shapiro A, Tal A (1998) Polyhedron realization for shape transformation. Visual Comput 14:429–444
47. Wolberg G (1990) Digital image warping. IEEE Computer Society Press, Los Alamitos, Calif
48. Wolberg G (1989) Skeleton-based image warping. Visual Comput 5:95–108
49. Wyvill W (1990) Metamorphosis of implicit surfaces. (SIGGRAPH '90) Course Notes No 23 – Modeling and Animation with Implicit Surfaces

ARTHUR GREGORY is a researcher at the University of North Carolina, Chapel Hill. He received his B.S. in Math Science from UNC in 1997. His research interests include the visual and haptic components of virtual reality, 3D morphing, geometric and physically-based modeling, and computer animation.

ANDREI STATE is a researcher in the Department of Computer Science at the University of North Carolina (UNC) at Chapel Hill. He has held this position since 1991. From 1991 to 1993 he was a graphics system designer for the VISTAnet Gigabit Network project. Prior to that he worked as a software engineer at Thomson Digital Image and on CATIA at Dassault Systemes, both in Paris, France. He received his Dipl.-Ing in Aerospace Engineering in 1988 from the University of Stuttgart and his MS in Computer Science in 1991 from UNC. Andrei is the principal designer of the UNC augmented reality (AR) visualization systems. His current research interests include AR technology and 3D morphing techniques.

MING C. LIN received her B.S., M.S., Ph.D. degrees in Electrical Engineering and Computer Science in 1988, 1991, 1993 respectively from the University of California, Berkeley. She is currently an assistant professor in the Computer Science Department at the University of North Carolina (UNC), Chapel Hill. Prior to joining UNC, she was an assistant professor in the Computer Science Department at both Naval Postgraduate School and North Carolina A&T State University, and a Program Manager at the U.S. Army Research Office. She received the NSF Young Faculty Career Award in 1995, Honda Research Initiation Award in 1997, and UNC/IBM Junior Faculty Development Award in 1999. Her research interests include real-time 3-D graphics for virtual environments, applied computational geometry, physically-based modeling, robotics and distributed interactive simulation. She has served as a program committee member for many leading conferences on virtual reality, computer graphics, and computational geometry. She was the general chair of the First ACM Workshop on Applied Computational Geometry and the co-Chair of 1999 ACM Symposium on Solid Modeling and Applications. She is also a guest editor of the International Journal on Computational Geometry and Applications, the co-editor of the book "Applied Computation Geometry", and the Category Editor of ACM Computing Reviews in Computer Graphics.

DINESH MANOCHA is currently as associate professor of computer science at the University of North Carolina at Chapel Hill. He received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1987; M.S. and Ph.D. in computer science at the University of California at Berkeley in 1990 and 1992, respectively. He received Alfred and Chella D. Moore fellowship and IBM graduate fellowship in 1988 and 1991, respectively, and a Junior Faculty Award in 1992. He was selected an Alfred P. Sloan Research Fellow, received NSF Career Award in 1995 and Office of Naval Research Young Investigator Award in 1996, and Hettleman Prize for scholarly achievement at UNC Chapel Hill in 1998. His research interests include geometric and solid modeling, interactive computer graphics, physically-based modeling, virtual environments, robotics and scientific computation. His research has been sponsored by DARPA, NSF, ARO, ONR, Sloan Foundation, Intel and Honda. He has published more than 95 papers in leading conferences and journals on computer graphics, geometric and solid modeling, robotics, symbolic and numeric computation, virtual reality, molecular modeling and computational geometry. He has served as a program committee member for many leading conferences on virtual reality, computer graphics, computational geometry, geometric and solid modeling and molecular modeling. He was the program co-chair for the first ACM Siggraph workshop on simulation and interaction in virtual environments and program chair of first ACM Workshop on Applied Computational Geometry. He was the guest editor of special issues of International Journal of Computational Geometry and Applications.

MARK A. LIVINSTON graduated from Duke University in 1993 with a BA in computer science and in mathematics. He earned the MS and Ph.D. degrees from the University of North Carolina at Chapel Hill in 1996 and 1998, respectively. His thesis work was in the area of augmented reality. He currently works for Hewlett-Packard Labs in Palo Alto, California on computer vision and computer graphics algorithms.