The Visual Computer

# Merging polyhedral shapes with scattered features

Marc Alexa

Interactive Graphics Systems Group (GRIS),
Department of Computer Science, Darmstadt
University of Technology, Rundeturmstr. 6, 64283
Darmstadt, Germany
e-mail: alexa@gris.informatik.tu-darmstadt.de

The paper presents a technique for merging two genus-0 polyhedra. Merging establishes correspondences between vertices of the models as a first step in a 3D morphing process. The technique allows for the specification of scattered features to be aligned. This is accomplished with the following three steps:

1. initial embeddings of the polyhedra on unit spheres are computed,
2. the embeddings are deformed so that user-defined features (vertices) coincide on the spheres, and
3. an overlay of the subdivisions is computed and the aligned vertices are fused in the merged model.

**Key words:** Polyhedra – Scattered features – Morphing

## 1 Introduction

It is a property of nature that objects change their shapes. Metamorphosis of 3D objects is useful in applications of computer graphics such as animation and modeling.
Right from the first approaches to shape metamorphosis, it turned out to be difficult to handle 3D objects. For the purpose of animation, the task could be reduced to image morphing, and a number of solutions to this problem are known. However, while the metamorphosis problem seems to be solved for images, it is still open for 3D models.
Among the large number of representations for 3D shapes, facet-based representations are popular and widespread. For this reason, a technique to transform one facet-based representation of a shape to another that also maintains this representation in the intermediate steps is desirable.
Such a technique must allow the user to provide information about the appearance of the morph. More specifically, in many situations, the models share common features that should be preserved during the transformation. The user can identify these features and provide information about their location and correspondence (for instance, vertex-vertex correspondence of a few vertices). The algorithm should exploit this information as much as possible and provide an adequate transformation.

### *1.1 Related work*

We compare our work only to publications sharing the idea of topological merging or feature alignment. A more general discussion of morphing techniques for 3D objects can be found in a recent survey [16]. Wolberg [25] explains the basics of image morphing techniques and reviews the current development [26].
Kent et al. [14] introduce the idea of topological merging as a way of establishing correspondence for a wide class of genus-0 polyhedra; in an earlier version, Kent et al. [13] treat only star-shaped polyhedra. The basic idea is to project the polyhedra onto a sphere and to compute the overlay of these projections. The resulting model contains the topology of both models and can be deformed into the shape of the source models. Any interpolation scheme yields a transformation. It turns out that the basic part of the algorithm – mapping the vertex-edge graph of the polyhedron to a sphere – is difficult for arbitrary polyhedra. Carmel and Cohen-Or [3] use curve evolution for this task, but admit some implementation

The Visual Computer (2000) 16:26–37

problems in the 3D case. Recently, this problem was solved in principle [1, 16].
Feature alignment is now common in image morphing. Naturally, image morphing extends to raster-based 3D representations, such as volume representations, distance fields, etc. For these object representations, techniques that handle feature alignment have been proposed [4, 17].
Feature alignment in polyhedral morphing has been considered in several publications: Lazarus and Verroust [15] mainly align major axes. Carmel and Cohen-Or [4] use a two-part transform, an affine transformation and a deformation inspired by image warping, to align features. Kanai et al. [12] allow the user to specify a peripheral cycle and merge the vertex-edge graphs in the plane.
Another development is the decomposition of the model into patches homeomorphic to a disk [5, 10]. These are the only works to our knowledge in which scattered features can be aligned in polyhedral morphing. However, specification of features is more time consuming and sometimes less intuitive than vertex-vertex correspondences. Moreover, it is known from image morphing [19] that dissecting the surface causes smoothness problems at the boundaries. In our approach, each feature potentially influences the whole model so that boundary discontinuities cannot appear.

### *1.2 Overview*

The general idea of our approach follows that of [14]. We try to find a polyhedral shape that results from merging the two input shapes. This shape can represent both of the source shapes, and a metamorphosis of the shapes might be produced by interpolating the vertices. While interpolating the vertices is interesting in its own right (the so-called vertex-path problem), we consider only the merging problem.
In contrast to other work, our approach takes into account an arbitrary number of features (points) that must remain aligned during the transformation. We use vertices for the specification of features. This is no restriction, as one might introduce additional vertices on every face of the model. The complete process consists of the following three steps:

1. Find an embedding of the polyhedral model on the unit sphere. The embedding should reflect the spatial relationships of the model's vertices.
2. Given two sphere embeddings, deform these embeddings so that the aligned vertices have the same position on the unit sphere.
3. Compute an overlay of the two graphs.

With this scheme, aligned vertices are fused in the merging step. Thus, they have to remain aligned during the transformation. We discuss each of the three steps in detail in the upcoming sections.
For the purpose of transformation, the merged model has to be deformed to represent the source models, which is covered in Sect. 5. Section 6 reports on implementation issues and results of the approach. Figure 1 gives an illustration of the overall procedure.

## 2 Embedding the polyhedron on a sphere

We consider genus-0 polyhedra because they are the only models that have a topologically equivalent embedding on a sphere. This embedding is defined by a graph reflecting the relationship between the vertices. The graph of genus-0 polyhedra is necessarily planar (this dates back to Euler), i.e., it can be drawn in the plane so that no two edges intersect. It is known that every planar graph has a straight-line embedding, i.e., the edges could be represented by straight, non-intersecting lines – this is independently established by [8, 21, 23].
As the first step of the procedure, the polyhedron is triangulated. In terms of graph theory, we make the graph maximal. This assures that the embedding of the graph is unique and, therefore, no extra effort has to be made to preserve the topology during the mapping onto the sphere.
The general idea of our approach is to adopt an algorithm for producing a straight-line embedding of a planar graph to produce a "straight-arc" embedding of the graph on a unit sphere. We define the straight-arc embedding to be the one where an edge connecting two vertices is represented by the shortest possible path along the surface of the sphere. This shortest path is the part of the great circle passing through the two endpoints on the sphere. The existence of such an embedding for planar graphs follows from Steinitz's theorem on convex polytopes in three dimensions [22].
At present, only two methods for embedding an arbitrary genus-0 polyhedron on a sphere have been published: [20] extract a base tetrahedron from the

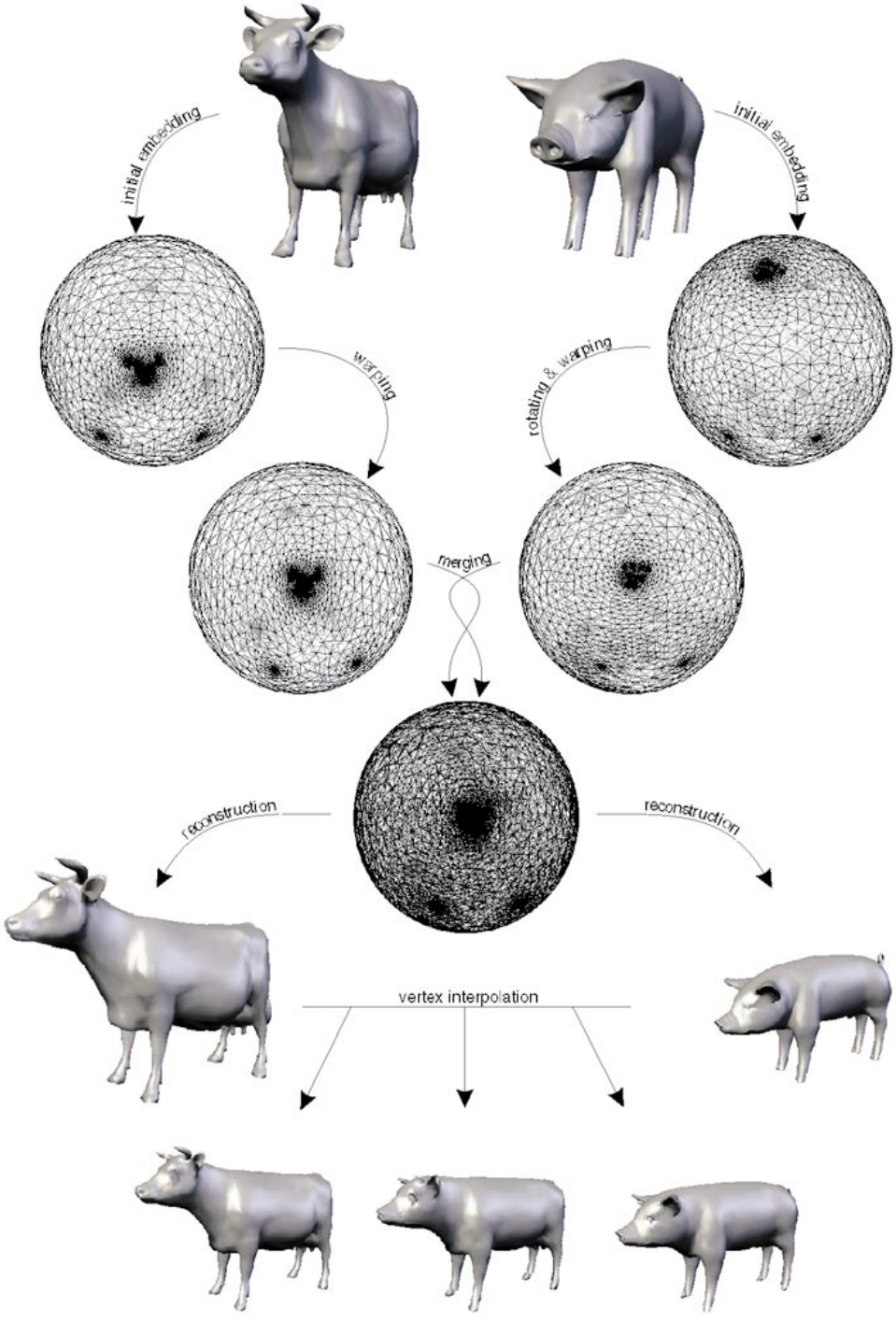



**Fig. 1.** The generation of a morph: first, the models are embedded on a sphere. Second, the embeddings are warped to align the user defined feature vertices. Third, the warped embeddings are merged. The resulting vertex-edge graph is induced back onto the original models. A morph sequence is obtained by interpolating the vertex positions

polyhedron and place the remaining vertices so that the polyhedron remains convex. Alexa [1] extends a classic technique to draw planar graphs to work on a sphere. All other approaches in the literature are limited to special classes of polyhedra. Here, we present a much improved version of Alexa's work, which is simpler and more robust than the original version.

### 2.1 Embedding algorithm

We begin with a 3D model in a facet-based representation consisting of faces $F$, edges $E$, and vertices $V = \{\boldsymbol{v}_0, \boldsymbol{v}_1, \dots\}$, such that $|V| - |E| + |F| = 2$. For each face, we have information about the vertices that are incident upon this face and their linear order along the boundary of the face. We want to compute a distribution of $V$ such that the mesh topology equals that of the model, and all vertices lie on the surface of a unit sphere: $\forall i.\ \| \boldsymbol{v}_i \| = 1$.
In the first step of the procedure, the model is triangulated. This assures that the embedding of the graph is unique [e.g., [11]] and, therefore, no extra effort has to be made to preserve the mesh topology during the mapping onto the sphere. We use a variation of spring embedding. In spring embedding algorithms, one tries to minimize a potential defined as

$$W_{\text{pot}} = \sum_{\{i,j\} \in E} \kappa_{i,j} \left\| \boldsymbol{v}_i - \boldsymbol{v}_j \right\|^2 .$$

While algorithms based on this paradigm produce nice results in the plane – see the comparison by Eck et al. [6] – the adoption to a spherical shell is nontrivial. In the planar case, the embedding is supported by a fixed peripheral cycle of the graph. Obviously, a peripheral cycle does not make sense on a closed manifold. However, if none of the vertices is fixed, the minimum energy state is reached when all vertices coincide.
The solution to this problem is as follows. We start with a reasonably equal distribution of the vertices on the sphere. Actually, we use the projection through the center of mass as the starting configuration. During the relaxation towards a minimum energy state, we penalize longer edges. Because the collapse of the vertices into one point has to pull at least one triangle over the equator, penalizing long edges effectively prevents the vertices from collapsing.
More precisely, in each step of the relaxation process, vertex $\boldsymbol{v}_i$ is moved.

$$\boldsymbol{p}_i = c \frac{\sum_{\{i,j\} \in E} \left(\boldsymbol{v}_i - \boldsymbol{v}_j\right) \left\| \boldsymbol{v}_i - \boldsymbol{v}_j \right\|}{\#\{(i,j) | \{i,j\} \in E\}} .$$

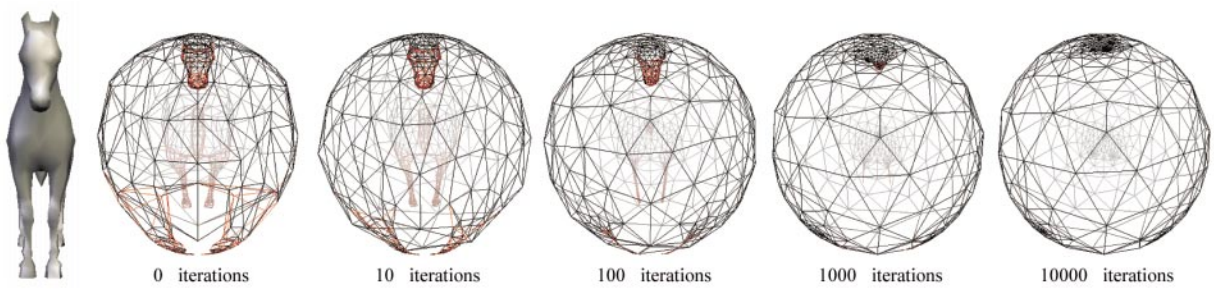
Multiplying $\left(\boldsymbol{v}_i - \boldsymbol{v}_j\right)$ by its length results in a quadratic weight for the edge lengths, such that longer edges are shortened. The constant $c$ is used to control the overall move length. With $c = 1$, the relaxation runs robustly, but not very efficiently. For very short edges, the quadratic weight makes moves very short and, thus, convergence very slow. Therefore, we scale up the move length proportionally to the inverse of the longest edge incident upon one vertex. The result is a much faster convergence.
Because $\boldsymbol{v}_i - \boldsymbol{p}_i$ is not necessarily on the unit sphere, we set

$$\boldsymbol{v}_i^{r+1} = \frac{\boldsymbol{v}_i^r + \boldsymbol{p}_i^r}{\left\| \boldsymbol{v}_i^r + \boldsymbol{p}_i^r \right\|} ,$$

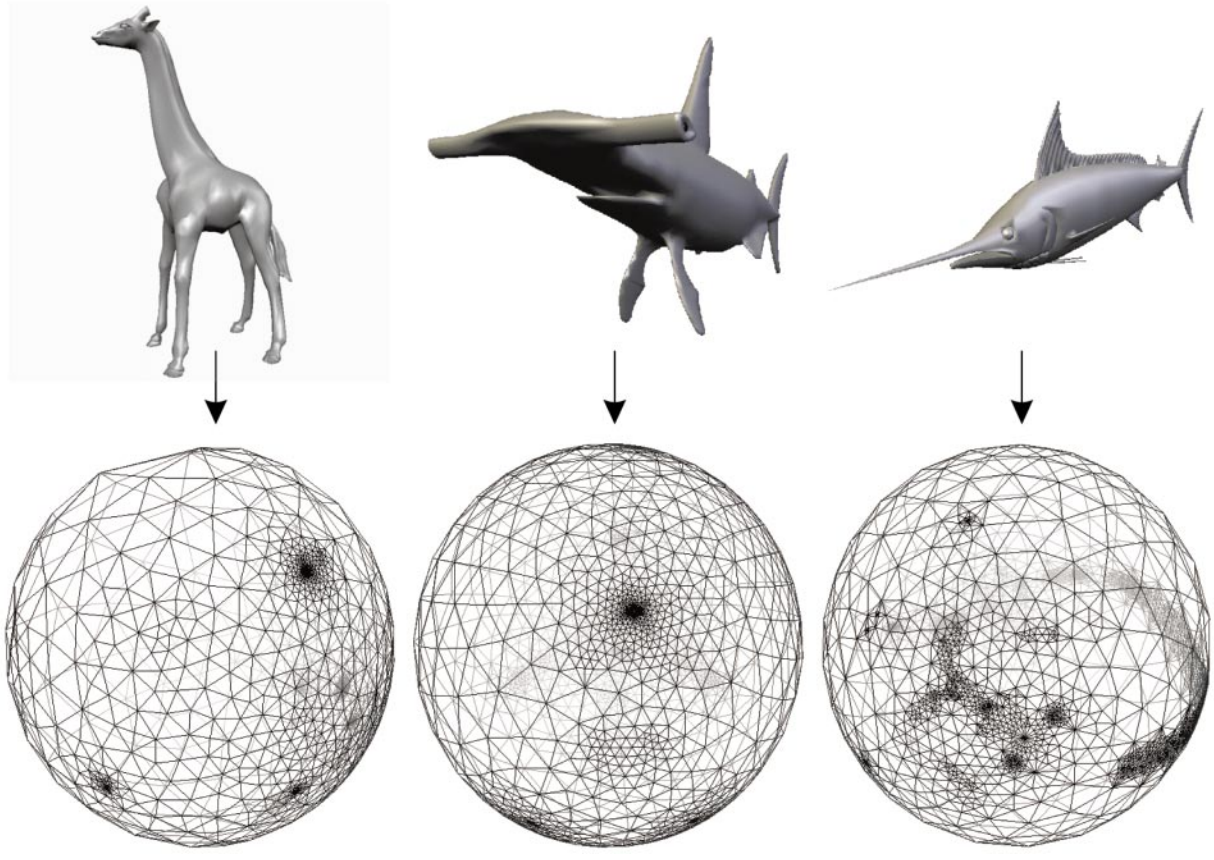
where $r$ indicates iterations in the relaxation process. A relaxation process for the polyhedral model of a horse is depicted in Fig. 2.

### 2.2 Termination of the relaxation

If we want to guarantee the topological correctness of the embedding, an epsilon bound of any kind is inadequate as the only termination criterion. Instead, the process is finished only when a valid embedding is found. The embedding is valid if and only if all faces are oriented the same way, i.e., the side that was on the outside of the model is on the outside of the sphere (obviously, the surface cannot fold back upon itself without at least one triangle being upside down).
We can check this condition by testing the orientation of three consecutive vertices along the boundary of each face. Here, orientation refers to whether the three vertices make a clockwise turn on the surface of the sphere. This can be computed by evaluating $\operatorname{sgn}((\boldsymbol{v}_0 \times \boldsymbol{v}_1) \cdot \boldsymbol{v}_2)$.
Thus, the relaxation is not terminated until the orientation of each face is the same as that of the original model. Because this test is rather expensive, it should only be done every $R$ iterations. We use $R = 10\,000$.
After the embedding is valid, a conventional epsilon bound is used as the final termination criterion. Actually, we use $\max_i (\| \boldsymbol{p}_i \|) < \varepsilon$.




2



3

**Fig. 2.** Embedding a polyhedral object on a sphere. Initially, the vertices are projected through an interior point of the model onto a unit sphere. The relaxation is finished when all faces are oriented correctly. Incorrectly oriented faces are surrounded by *red edges*

**Fig. 3a-c.** Sphere embeddings of polyhedral models of: **a** a giraffe; **b** a hammerhead shark; **c** a swordfish

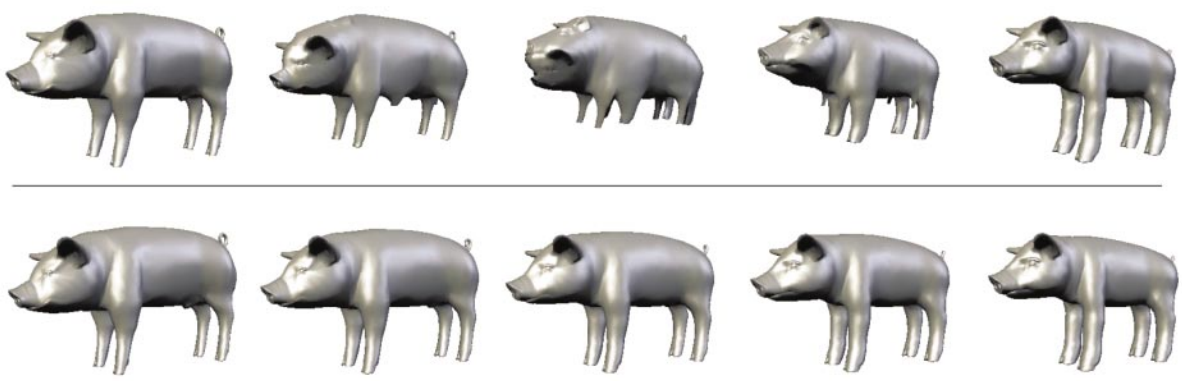


**Fig. 4a,b.** Morphs between the models of a young pig and a grownup pig: **a** no feature alignment is used, which leads to unpleasant effects (e.g., eight legs in the intermediate models); **b** the ears, eyes, hoofs, and the tail are aligned (a total of 17 vertex-vertex correspondences), yielding a smooth transformation

Figure 3 shows the sphere embeddings of various polyhedral models.

## 3 Aligning the Features

The necessity for aligning prominent features becomes evident even in very simple examples. Figure 4 shows two morphs between models of a young pig and a grownup pig. In Fig. 4a, no features were aligned, and the resulting morph is unacceptable. Figure 3b shows a morph produced with some features (ears, eyes, hoofs, and the tail) aligned. The result is obviously more pleasing. Surprisingly, the need of user guidance becomes more obvious when the shapes are similar. This is because we can envision a transformation, i.e., we expect common features of the models (head, legs, etc.) to be preserved. Nonetheless, this does not happen, of course, due to the different mesh topology of the models (in this example, the different mesh topologies are obvious from the different vertex counts of the models).
Typically, only the user knows what features would correspond in two models. In order to specify this correspondence, features are identified with vertices (which we refer to as feature vertices). A simple way to assure that features are preserved during the transformation is to place the corresponding feature vertices at (approximately) the same positions on the two spheres. Thus, in the merged model they are fused or at least very close to each other. This characteristic cannot change during any transformation. With our approach, the alignment of features is independent of the actual interpolation technique, but assured by the topology of the merged model.
Two steps are taken subsequently to achieve the goal of coinciding feature vertices on the sphere:

1. Rotate one sphere so that the squared distances of corresponding feature vertices are minimized.
2. Warp the surfaces of both spheres to further diminish the distances between corresponding feature vertices.

The first step can be sufficiently approximated by multidimensional numerical minimization. The second step is harder to tackle because of a general constraint throughout the process: the warp may not introduce incorrectly oriented faces, i.e. the embedding has to stay valid. This problem would be less difficult, if vertices as well as edges were warped. Since we need edge-edge intersection tests in the subsequent merging step, warping the edges is impractical. Instead, edges should (still) be defined as the shortest paths between vertices. That is, we warp only the vertices. Thus, even injective warping functions might introduce foldover.

The solution is to make several small, local deformations instead of searching for one warping function that is applied once to all vertices. This way, we can control the effect of the deformation (does it introduce foldover?) and reduce the amount of deformation when necessary. We apply a mapping for each of the feature vertices subsequently in order to move them towards the position of their corresponding vertices. Given a vertex at position $\boldsymbol{v}$ that should move to $\boldsymbol{w}$, we define a map $f$ as

$$f(\boldsymbol{x}) = \begin{cases} \boldsymbol{x} + c(d - \|\boldsymbol{x}-\boldsymbol{v}\|)(\boldsymbol{w}-\boldsymbol{v}) & \|\boldsymbol{x}-\boldsymbol{v}\| < d \\ \boldsymbol{x} & \|\boldsymbol{x}-\boldsymbol{v}\| \geq d \end{cases},$$

where $d$ is the radius of influence for the map. The constant $c$ is used to decrease the move length in case the deformation induces foldovers. We start with $c = 0.5$ in order to equally distribute the deformation to both models. After $f$ is applied to all vertices, they have to be normalized to be pulled back onto the surface of the unit sphere.

The map is applied to both models, and the necessary move is derived from the current positions. That is, we do not define an intermediate position for the feature vertices, as is common in image morphing. In case the deformation in one model introduces foldovers, coincidence can still be achieved by deforming the other model.

The complete process starts with a large value of $d$ in order to make more global deformations. If the features are not aligned and subsequent deformations due to feature vertex placement make no progress, $d$ is decreased. That is, if global deformations are not sufficient to align the feature vertices, more local deformations are used. Note that the alignment of all feature vertices is not necessarily possible because of topological restrictions.

Now we give a more formal overview of the algorithm.

### 3.1 Algorithm for aligning corresponding vertices

We begin with two embeddings with vertices $V^1$, $V^2$ on the sphere, as well as two ordered sets of features represented as a number of indices in the sets $T^1$, $T^2$ (we assume $|T^1| = |T^2|$ ). We denote the $i$th element of the vertex sets as $\boldsymbol{v}_i^1$, $\boldsymbol{v}_i^2$ and the $i$th element of the feature sets as $t_i^1$, $t_i^2$. The goal of the following procedure can be written as

$$\forall i.\ \boldsymbol{v}^1_{t_i^1} = \boldsymbol{v}^2_{t_i^2}\ .$$

1. Rotate the first sphere so that the summed squared distance

   $$s = \sum_i \left\| \boldsymbol{v}^1_{t_i^1} - \boldsymbol{v}^2_{t_i^2} \right\|^2$$

   of the corresponding feature vertices is minimized.
2. Define the displacement necessary to move the $i$th feature of the first sphere:

   $$\boldsymbol{p} = \boldsymbol{v}^1_{t_i^1} - \boldsymbol{v}^2_{t_i^2}\ .$$

   Then apply the following map to all vertices

   $$\forall j.\boldsymbol{v}_j^1 = \begin{cases} \boldsymbol{v}_j^1 + c\boldsymbol{p}\left(d - \left\|\boldsymbol{v}_j^1 - \boldsymbol{v}^1_{t_i^1}\right\|\right) & \left\|\boldsymbol{v}_j^1 - \boldsymbol{v}^1_{t_i^1}\right\| < d \\ \boldsymbol{v}_j^1 & \left\|\boldsymbol{v}_j^1 - \boldsymbol{v}^1_{t_i^1}\right\| \geq d \end{cases}$$

   and normalize the vertices again. After each mapping, check if the embedding is still valid (check the orientations of the faces as in the previous section). If not, decrease $c$. Repeat the map for all features $t_i^1$.
3. Deform the second sphere, i.e., do step 2 moving vertices $\boldsymbol{v}_i^2$ using $t_i^2$ features .
4. Repeat steps 2 and 3 until the positions of the vertices remain unchanged.
5. If the feature vertices do not coincide, and $d$ is greater than zero, decrease $d$ and go back to step 2.

## 4 Merging the embeddings

Given two embeddings, we need to produce an embedding that contains the faces, edges, and vertices of both source models. The problem is known as *map overlay*. Several algorithms have been proposed for this problem, but they handle mainly the case of planar graphs – see a textbook [2]. In general, the planar map overlay has the complexity $O(n \log n + k)$, where $n$ is the number of edges and $k$ is the number of intersections. If the two subdivisions are connected (as in our case) the planar overlay can be computed in $O(n+k)$ [9]. There seem to be only a few publications about the overlay of subdivision on the sphere, and it is not clear whether (and which of the) solutions for the planar case are applicable in our context.

Kent et al. [14] give an algorithm for the sphere overlay problem, which needs $O(n + k \log k)$ time. We give a new solution to this particular problem, which reports the intersection of two spherical subdivisions in the optimum time of $O(n+k)$. Furthermore, our algorithm exploits the topological properties of both subdivisions, which are used to guarantee the correct order of intersections.
The algorithm consists of two main parts: first, finding all intersections, and second, constructing a representation for the merged model. In both cases, a sophisticated data structure is needed to represent the models. We use the doubly connected edge list [18]. This data structure contains information about each face, each directed edge, and each vertex. The following information is stored for the various types:

1. The face record contains a pointer to an arbitrary half edge on its boundary.
2. The edge record contains pointer to: the vertex it is originating, the face it bounds, its twin (the half edge connecting the same vertices, but pointing in the opposite direction), and the next half edge along the boundary of the bounded face.
3. The vertex record contains information about the location in space and a pointer to an arbitrary edge that has this vertex as its origin.

The reader might want to consult a textbook [2] to get more information about this structure and especially about why these data are sufficient for all algorithmic tasks.
*Note*. In the following we assume that none of the vertices of one embedding lies on a vertex or an edge of the other graph. We can assure this *general position assumption* by using a symbolic perturbation scheme like the one described by [6].

## *4.1 Finding the intersections*

In the algorithm two geometric functions are needed: one to decide if and where two edges intersect on the sphere, and a second to decide whether a point lies inside a face. We cannot adopt planar techniques in either case

### 4.1.1 Edge-edge intersection

An edges is defined as the shortest path between two points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ on the sphere. This is the shorter arc of the circle with the same center as the sphere passing through the points (in the following we assume a unit sphere). Obviously, every two of those circles intersect (or they are the same). The points of intersection can be computed as follows. Each of the circles defines a plane; the intersection of the planes cuts the sphere in the intersection points we are searching for. The intersection of the planes is orthogonal to both normals of the planes, and the normal of a plane is orthogonal to the unit vectors $\boldsymbol{p}_1,\ \boldsymbol{p}_2$. Thus, we find the intersection of two circles based by $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ and $\boldsymbol{q}_1$, $\boldsymbol{q}_2$ to be

$$\boldsymbol{r} = \pm\left(\boldsymbol{p}_1 \times \boldsymbol{p}_2\right) \times \left(\boldsymbol{q}_1 \times \boldsymbol{q}_2\right) .$$

To decide whether the two shorter arcs intersect, we have to solve the following two equations for $s_p, s_q$:

$$t_p \boldsymbol{r} = \boldsymbol{p}_1 + s_p \left(\boldsymbol{p}_2 - \boldsymbol{p}_1\right)$$

$$t_q \boldsymbol{r} = \boldsymbol{q}_1 + s_q \left(\boldsymbol{q}_2 - \boldsymbol{q}_1\right) .$$

The two arcs intersect, if and only if $0 \le s_p, s_q \le 1$. The point of intersection is already given in $\boldsymbol{r}$.

### 4.1.2 Point in face

Since the model was triangulated, every face is described by three vertices $\boldsymbol{v}_1,\ \boldsymbol{v}_2,\ \boldsymbol{v}_3$. We can assume that vertices are ordered counterclockwise around the triangle as viewed from outside the sphere (the order is known from the next pointers in the edge list). We define three planes by two subsequent vertices and the origin. The intersection of the positive half spaces of these planes contains exactly the face. A point lies in the positive half space of a plane if the sign of the dot product with the normal of this plane is positive. Thus, a point $\boldsymbol{p}$ lies inside a spherical triangle bounded by $\boldsymbol{v}_1,\ \boldsymbol{v}_2,\ \boldsymbol{v}_3$ if and only if:

$$\left(\boldsymbol{v}_1 \times \boldsymbol{v}_2\right) \cdot \boldsymbol{p},\ \left(\boldsymbol{v}_2 \times \boldsymbol{v}_3\right) \cdot \boldsymbol{p},\ \left(\boldsymbol{v}_3 \times \boldsymbol{v}_1\right) \cdot \boldsymbol{p} \ge 0 .$$

### 4.1.3 Traversing to find intersections

Now that we can answer these basic geometric questions, the algorithm can be explained. We assume two embeddings consisting of vertices $V^1,\ V^2$, edges $E^1,\ E^2$, and faces $F^1,\ F^2$. For every edge $e^1$ in $E^1$we need a list of edges in $E^2$ that intersect $e^1$ and vice versa. These lists are generated by traversing the graph. Both of the graphs are traversed in almost the same way, so we describe the process only for traversing and generating the intersection lists for $E^1$.

The basic idea is to traverse the breadth of the graph first. Choose an arbitrary vertex $v^1 \in V^1$ and search the face $f^2 \in F^2$ that contains $v^1$. Start with the edge $e^1$ contained in the record of $v^1$. For the traversal, a stack of edges to inspect is maintained. For each half edge pushed onto the stack, the face that contains the origin of the half edge is also pushed. Thus, $e^1$ and $f^2$ are pushed onto the stack. Then the following procedure is repeated until the stack is empty:

1. Pop a working edge $e^1$ and a face $f^2$.
2. Mark the edge $e^1$ and $\tilde{e}^1$ its twin used.
3. Initialize a variable $c$ that stores the edge of $E^2$ that was intersected last.
4. Test whether $e^1$ intersects with one of the boundary edges $e_1^2,\ e_2^2,\ e_3^2 \neq c$ of $f^2$.
5. If $e^1$ intersects $e_i^2 \neq c$ then replace $f^2$ with the face stored in the twin of $e_i^2$, that is, the face on "the other side" of $e_i^2$ and go back to step 3. If no intersections are reported, a topologically ordered list of intersections for the half edge $e^1$ was generated.
6. $f^2$ is (now) the face that contains the endpoint of $e^1$. Thus, push all unused edges originating from this endpoint together with $f^2$ on the stack.

This procedure is done a second time with the two sets (superscripts one and two) exchanged. It is useful to generate the new vertices in one of the traversals, which makes the traversals slightly different. We need information about the relative positions of all vertices later. Therefore, the three vertices making up a face in one graph are stored for all vertices in the other graph (this is done when the edge and face are popped from the stack). For the newly generated vertex, the four vertices incident upon the two intersecting edges are stored.

It is easy to see that the algorithm is linear in the number of edges plus intersections. There are three intersection tests for each edge, and for each intersection of such an edge, three additional intersection tests are necessary. These are constant costs per edge and per intersection. In the initialization of the traversal all faces are tested to see whether they contain the starting vertex. This is also linear in the number of edges.

### 4.2 Constructing a representation

We want to compute a list of vertices $\hat{V}$, edges $\hat{E}$, and faces $\hat{F}$ from the vertices $V^1,\ V^2$, edges $E^1,\ E^2$, and faces $F^1,\ F^2$, as well as the intersection lists generated in this procedure. Both subdivisions are given as doubly connected edge lists, and a doubly connected edge list should be constructed for the merged model. We start with initializing $\hat{V} = V^1 \cup V^2$ and $\hat{E} = E^1 \cup E^2$. All the vertices originating from the intersections in the first part of the algorithm are already generated. Contrary to the vertices in $\hat{V}$, these vertices do not contain valid information about a half edge.

The basic idea is to process all intersection lists for the edges in $E^1$. For each intersection, the intersecting edges have to be cut. Because the original edges are changed in this step, the intersection lists have to be updated or new intersection lists have to be created. The invariant we like to maintain is that the intersection lists are valid for the current list of edges. The intersections for one edge are processed in the order given by the intersection list. Thus, the intersections inserted in the edge from $E^1$ are in the topologically correct order. It will become clear later on, where the information about the edges from $E^2$ (the intersection lists for te edges) is used, that these edges are also cut topologically correctly.

First we explain how to cut two edges (four half edges). The intersection list of $e^1$ contains information about an intersection with an edge $e^2$. Because the faces of the subdivisions are convex (triangles), and $e^1$ is leaving the face that is bounded by $e^2$, the situation is always the same: $e^1$ and $e^2$ make a clockwise turn.

The four edges already in $\hat{E}$ ($e^1,\ e^2$ and their twins) keep their origin. The four new edges will have the new vertex as their origin. It remains to update the next and twin pointers in the records of all eight edges.

The intersection list of $e^2$ and of some edges in $E^1$ might no longer be valid. Some edges in $E^1$ cut $e^2$ in the part that is now represented by the newly generated edge originating from the new vertex (we call this edge $\bar{e}^2$). To find these edges, the information in the intersection list of $e^2$ is inspected. The edges in this intersection list are browsed until $e^1$ (which has to be in the list) is found. $e^1$ can safely be removed from the intersection list. All edges in the intersection list after $e^1$ now belong to $\bar{e}^2$. That is, these edges are removed from the list and a new intersection list is generated for $\bar{e}^2$. Then the intersection lists for all edges in this new intersection list have to be updated. These intersection lists originally pointed to $e^2$ and now have to point to $\bar{e}^2$. After these updates, the invariant holds again.

After all intersections are processed in this way, we have a valid vertex and edge lists of the embedding. It remains to compute the records for the faces. Faces are easily found by following cycles of half edges (we exploit the next pointer of each edge: loop through all edges). If the edge does not contain information about a face, follow the next pointers of the edges until the first edge is found again. Create a new face, set the record of the edges to this face and the record of the face to an arbitrary edge. After all edges are processed, the complete information about faces is created. Note that faces created from intersecting triangles are not necessarily triangles themselves, but they are convex.

## 5 Reconstructing the source models

In most of the applications, the purpose of the merged model is to serve for a transformation between the original polyhedra. Thus, the merged model has to be deformed to have exactly the shape of the source models. We refer to vertices, edges, and faces as in the previous sections.

Note first of all that the information about edges and faces remains unchanged. Only the vertex positions have to be set. Of course, the vertices of $\hat{V}$ that are elements of $V^1$ can be set to their original values. For the other vertices, some additional information is needed. This information was computed while finding the intersections. We distinguish between vertices of $V^2$ and the new vertices. The vertex $\nu^2$ of $V^2$ belongs to a face in $F^1$. Remember that we stored the three vertices $\nu_1^1,\ \nu_2^1,\ \nu_3^1$ around that face. We compute the barycentric representation of $\nu^2$ in the basis of $\nu_1^1,\ \nu_2^1,\ \nu_3^1$. This barycentric representation is used to interpolate between all attributes of $\nu_1^1,\ \nu_2^1,\ \nu_3^1$. This way, not only the position is easily computed, but color values or texture coordinates might also be interpolated.

If a vertex $\nu$ does not belong to $V^1$ or to $V^2$, it was generated due to an edge-edge intersection. The four vertices of the two intersecting edges were stored for this vertex. In each of the reconstruction processes, only the two vertices that belong to the original model are needed. In this case, the vertices $\nu_1^1,\ \nu_2^1$ of the intersecting edge $e^1$ are used. As in the proceeding step, a barycentric representation is calculated (this time with respect to $\nu_1^1,\ \nu_2^1$). Again, this barycentric representation is used to interpolate all attributes of the supporting vertices $\nu_1^1,\ \nu_2^1$.

**Table 1.** Timing information about the initial embedding step

| Model | Vertices | Edges | Faces | Time in seconds |
|---|---|---|---|---|
| Cow | 2911 | 8714 | 5805 | 126.5 |
| Dolphin | 420 | 974 | 556 | 12.3 |
| Duck | 629 | 1597 | 970 | 9.4 |
| Giraffe | 4197 | 9537 | 5342 | 427.5 |
| Horse | 674 | 1535 | 863 | 23.3 |
| Pig | 3522 | 7689 | 4169 | 277.9 |
| Piglet | 3584 | 7869 | 4287 | 113.1 |
| Shuttle | 310 | 701 | 393 | 9.4 |

**Table 2.** Timing information about the deformation step

| Models aligned | Features | Aligned | Time in seconds |
|---|---|---|---|
| Shark / shuttle | 4 | No | 7.0 |
| Horse / giraffe | 16 | Yes | 51.5 |
| Horse / pig | 14 | Yes | 34.1 |
| Horse / pig | 22 | Yes | 53.0 |
| Giraffe / pig | 23 | No | 117.1 |

Furthermore, the faces of the merged model are not necessarily triangles, and more than three points are not necessarily coplanar, which might cause problems during a transformation. Thus, the models should be triangulated prior to a transformation.

## 6 Implementation and results

We have implemented the procedures explained in Java/Java3D using a native function in C for the relaxation step. Table 1 gives some execution and timing information for the initial embedding. Times are average execution times, not CPU seconds, for a Sun Ultra 10 in a regular network with normal workload. The initial embedding step makes use of a native function coded in C.

In order to make the feature specification feasible, we have designed a GUI that allows us to choose vertices of the source models. These feature vertices are aligned in the second step. The deformation of two models towards the alignment of their features is coded completely in Java. Due to the nature of the algorithm, the cost increases linearly with the number of features. Table 2 shows timing information for some of the models with different numbers of features to be aligned. Information is given as to whether the features could be completely aligned without creating foldovers.

**Table 3.** Timing and complexity information about the merging step

| | Giraffe | Horse | Pig | Shark | Swordfish |
|---|---|---|---|---|---|
| Giraffe | | 4.7 s | 8.1 s | 7.5 s | 6.8 s |
| Horse | 7152 | | 4.2 s | 3.6 s | 3.2 s |
| Pig | 12 509 | 6418 | | 6.7 s | 8.3 s |
| Shark | 11 825 | 6585 | 11 540 | | 6.8 s |
| Swordfish | 11 365 | 5448 | 14 645 | 11 243 | |

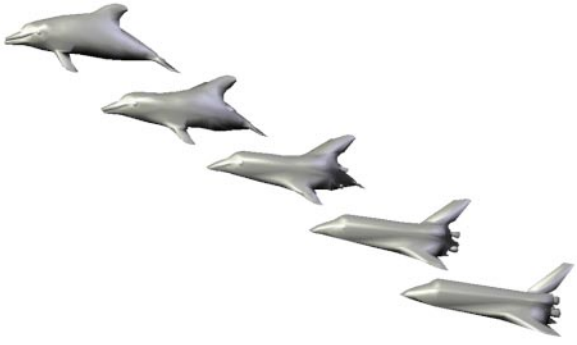


**Fig. 5.** Morphing between the models of a dolphin and a space shuttle

In the third step the sphere subdivisions are merged. The algorithm is also completely coded in Java. Of course, the merging step does not depend on the number of features directly, but on the number of intersecting edges, which might be influenced by the deformation of the embedding. Table 3 gives information about the merging step. The timing information (given in the upper triangle matrix) has to be seen with respect to the number of vertices of the output models (given in the lower triangle matrix).
Finally, we present another transformation obtained by linearly interpolating the vertex positions. Figure 5 shows the transformation from a dolphin to a space shuttle with feature alignment. Only four corresponding vertices were defined. Choosing the vertices in the GUI was done in less than 1 min.

## 7 Conclusions

We have presented a technique to merge arbitrary genus-0 polyhedra. The merging procedure is capable of aligning scattered features given as corresponding vertices. This allows for smooth transitions between polyhedra so that common features are preserved.

## References

1. Alexa M (1999) Merging polyhedral shapes with scattered features. Proceedings of Shape Modelling International'99, IEEE Computer Society Press, Los Alamitos, CA, pp 202–210
2. de Berg M, van Kreveld M, Overmars M, Schwarzkopf O (1997) Computational geometry: algorithms and applications. Springer, Heidelberg
3. Carmel E, Cohen-Or D (1998) Warp-guided object-space morphing. Visual Comput 13:465–478
4. Cohen-Or D, Levin D, Solomovici A (1998) Three-dimensional distance field metamorphosis. ACM Trans Graph 17:116–141
5. DeCarlo D, Gallier J (1996) Topological evolution of surfaces. Proceedings of Graphics Interface '96, Canadian Human-Computer Communications Society, Toronto, Canada, pp 194–203
6. Eck M, DeRose T, Duchamp T, Hoppe H, Lounsberry M, Stuetzle W (1995) Multiresolution analysis of arbitrary meshes. (SIGGRAPH '95) Comput Graph 29:173–182
7. Edelsbrunner H, Mücke EP (1990) Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. ACM Trans Graph 9:66–104
8. Fary I (1948) On straight lines representation of planar graphs. Acta Sci Math Szeged 11:229–233
9. Finke U, Hinrichs K (1995) Overlaying simply connected planar subdivisions in linear time. Proceedings of the 11th Annual ACM Symposium on Computational Geometry, pp 119–126
10. Gregory A, State A, Lin MC, Manocha D, Livingston MA (1998) Feature-based surface decomposition for correspondence and morphing between polyhedra. Proceedings of Computer Animation '98, Philadelphia
11. Harary F (1969) Graph theory, Addison-Wesley, Reading, Mass
12. Kanai T, Suzuki H, Kimura F (1997) 3D Geometric metamorphosis based on harmonic map. Visual Comput 14:166–176
13. Kent JR, Parent RE, Carlson WE (1991) Establishing correspondences by topological merging: a new approach to 3-D shape transformation. Proceedings of Graphics Interface '91, Calgary, Canadian Information Processing Society, pp 271–278
14. Kent JR, Carlson WE, Parent RE (1992) Shape transformation for polyhedral objects. (SIGGRAPH '92) Comput Graph 26:47–54
15. Lazarus F, Verroust A (1994) Feature-based shape transformation for polyhedral objects. Proceedings of the 5th Eurgraphics Workshop on Animation and Simulation, Oslo, Eurographics Association, pp 1–14
16. Lazarus F, Verroust A (1998) Three-dimensional metamorphosis: a survey. Visual Comput 14:373–389
17. Lerios A, Garfinkle CD, Levoy M (1995) Feature-based volume metamorphosis. (SIGGRAPH '95) Comput Graph 29:449–456

18. Muller DE, Preparata FP (1978) Finding the intersections of two convex polyhedra. Theor Comput Sci 7:212–236
19. Ruprecht D, Müller H (1995) Image warping with scattered data interpolation. IEEE Comput Graph Appl 15:37–44
20. Shapiro A, Tal A (1998) Polyhedron realization for shape transformation. Visual Comput 14:429–444
21. Stein SK (1951) Convex maps Proc Am Math Soc 2:464–466
22. Steinitz E, Rademacher H (1934) Vorlesung über die Theorie der Polyeder. Springer, Berlin
23. Tutte WT (1963) How to draw a graph. Proc London Math Soc 3:743–768
24. Wagner K (1936) Bemerkungen zum Vierfarbenproblem. Jber Deutsch Math-Verein 46:26–32
25. Wolberg G (1990) Digital image warping. IEEE Computer Society Press, Los Alamitos
26. Wolberg G (1998) Image morphing: a survey. Visual Comput 14:360–372

MARC ALEXA is a scientific researcher and PhD candidate in the Department of Computer Science at the Darmstadt University of Technology, Germany. He received a MS degree in Computer Science with honors from Darmstadt University of Technology. His research interests include shape and image transformation and their application to other fields, e.g. modeling, information visualization, and computer vision.