Eurographics/SIGGRAPH Symposium on Computer Animation (2003)
D. Breen, M. Lin (Editors)

Interactive Control of Component-based Morphing

Yonghong Zhao, Hong-Yang Ong, Tiow-Seng Tan and Yongguan Xiao

School of Computing, National University of Singapore, Singapore

Abstract

This paper presents an interactive morphing framework to empower users to conveniently and effectively control the whole morphing process. Although research on mesh morphing has reached a state where most computational problems have been solved in general, the novelty of our framework lies in the integration of global-level and local-level user control through the use of components, and the incorporation of deduction and assistance in user interaction. Given two polygonal meshes, users can choose to specify their requirements either at the global level over components or at the local level within components, whichever is more intuitive. Based on user specifications, the framework proposes several techniques to deduce implied correspondences and add assumed correspondences at both levels. The framework also supports multi-level interpolation control — users can operate on a component as a whole or on its individual vertices to specify trajectories. On the whole, in the multi-level component-based framework, users can choose to specify any number of requirements at each level and the system can complete all other tasks to produce final morphs. Therefore, user control is greatly enhanced and even an amateur can use it to design morphing with ease.

Categories and Subject Descriptors (according to ACM CCS): *I.3.6 [Methodology and Techniques]: Interaction techniques, I.3.7 [Three-Dimensional Graphics and Realism]: Animation, I.3.8 Application*

1. Introduction

Morphing (or metamorphosis) of polygonal meshes involves the creation of a smooth transition from a source mesh to a target mesh. For two meshes, there are numerous ways to transform one to the other. Algorithms for morphing are mainly evaluated by criteria related to the ease of user control and the visual quality of morphing sequence. Morphing is such an aesthetic problem that fully automatic methods cannot meet all the needs that arise in all applications. Therefore, user interaction is important and unavoidable.

A mesh morphing process basically consists of two steps: establishing the *correspondence* where each vertex of a source mesh is mapped to a vertex of a target mesh, and calculating the *interpolation* where trajectories are defined for all corresponding vertices. Research on mesh morphing has reached a state where most computational problems in these two steps have been solved in general [2, 14].

However, a similar claim cannot be made for user control. Efficiency of user interaction has not received much attention by previous morphing works; they usually concentrate on the computational issues and overlook the interactive process of specifying and modifying user requirements. There was no good scheme to make user interaction intuitive, flexible and efficient. For example, to specify high-level correspondences, such as pairing a leg of a duck with a leg of a dinosaur, users generally had to express such a requirement indirectly in terms of many low-level vertex correspondences. This is usually neither obvious nor natural, especially when two original meshes are very different in shapes.

The presented framework seeks to enhance user control throughout the whole morphing process. It is termed a component-based framework as it utilizes *components* of objects to achieve such a goal. The technical uniqueness of our framework lies in multi-level user control through the use of components, and deduction and assistance to make user control easy; Figure 1 (in the color plate) shows an example of user control. Specifically, major contributions of this framework are as follows:

- **Multi-level correspondence control**

Global-level and local-level user specifications interact with each other, and enable our users to specify their requirements in either level whichever is more intuitive. Users can directly specify *global-level* correspondences by pairing components to represent their high-level requirements, without resorting to the more tedious low-level vertex pairing. Yet, when fine control is required, users can specify *local-level* correspondences over local features within a pair of corresponding components.

- **Interactive and flexible user control**

Due to the incorporation of multi-level user control, the framework can automatically deduce correspondences from one level to the other. Moreover, to make user interaction more flexible, several techniques are presented to provide assistance and deduction in user control. At the global level, we utilize a novel constraint tree to provide candidate counterparts for user-selected components, maintain user specifications after modifications to components and correspondences, and finally deduce correspondences over all components. At the local level, the framework derives those local-level correspondences not stated but implied by user specifications and adds assumed correspondences where appropriate to improve the morphing. In this framework all user specifications are respected and no system restriction are imposed on users.

- **Multi-level interpolation control**

Users can edit trajectories by operating on components as a whole. Trajectories over individual vertices can be deduced accordingly. This allows users to manipulate the interpolation quickly and easily at the global level. When local-level control is desired, trajectories of individual vertices can also be specified.

2. Previous Work

3D mesh morphing has been extensively studied in recent decades. Necessarily, we limit our review to focus on methods related to interactive morphing control.

Recent techniques allow users to specify vertex pairs and use them to calculate the complete vertex correspondence between two original meshes. To handle topological changes in morphing, DeCarlo and Gallier [6] allow users to partition two original meshes into triangular and quadrilateral patches. Gregory *et al*. [9] decompose mesh surfaces in a morph according to user-specified feature nets over them. In the work of Lee *et al*. [12], a user can specify vertex pairs on two original meshes and then the correspondence problem is solved through their simplified models. Kanai *et al*. [11] allow users to specify feature vertices and connect them subsequently. The system then constructs control meshes based on user input. Alexa [1] allows users to specify scattered features and then align them using a global mapping and warping method. Praun *et al.* [17] present an algorithm that establishes consistent parameterization for a set of meshes sharing a base domain. To morph one mesh into the other, the users are required to specify corresponding vertices in both meshes for each vertex in the base domain.

In the above works, users usually have to invest considerable effort to get final morphs, especially for meshes of complex shapes. In those morphing methods using patches [6, 9, 11], a user must specify a feature net by assigning feature vertex pairs and then identifying connectivity among those vertices. In [17], the users are required to specify corresponding vertices for every vertex in the common base domain. [1, 12] do not have such a requirement. However, they do not utilize the shapes of meshes in calculating correspondences; their users may need to spend significant effort to locally adjust vertex correspondences. Contrary to these, our framework allows its users to specify any number of requirements, and computes a morph through deduction. This greatly reduces workload and complexity in user interaction and makes designing morphs easy.

In addition, morphing design in these previous works is only a one-directional process. However, our framework supports a trial-and-error process, which is more flexible and convenient. A user can start to design a morph by specifying a small number of requirements, and then interactively improve those unsatisfactory parts of the morphing result through more specifications.

Lazarus and Verroust [13] provide high-level control in the morphing of star-shaped meshes by allowing users to control axes of objects. This method automatically establishes vertex correspondences at the stage of mesh re-sampling. Thus, users cannot control correspondences over vertices flexibly. Our work differs in that we provide multi-level user control without imposing any system-caused restrictions on users, and we can handle morphing complex meshes by using components.

On morphing objects of other kinds of representations, the following are some works related to our approach. Shapira and Rappoport [18] morph 2D polygons by partitioning them into star-shaped pieces. Our work is similar to theirs in partitioning complex objects into simpler forms for morphing. Unlike their work, our method does not have the requirement that objects must be compatibly decomposed and thus a user can design a morph more conveniently. Galin *et al.* [7, 8] address the soft object morphing using Minkowski sums for the interpolation of skeletal elements. Though our work is similar to theirs in using components, there are fundamental differences in the technicalities because of different object representations.

3. Components and Framework Overview

A mesh exists as a collection of polygons and contains no explicitly defined high-level information. To manipulate a mesh at a high level, one way is to decompose it into a collection of primitives, each of which comprises a group of polygons. Such a primitive is termed a *component*. Two connected components share some edges, which we call a *boundary*. A component can have a number of boundaries, each of which represents a *connection* with a *neighboring component*. For a mesh, we represent its components and the connections between them as a *connectivity graph*. In this graph, each component is represented as a

node, and a connection between two components as an edge connecting the two nodes of the components. Figure 2 illustrates the above definitions. We shall use this example throughout Section 4.

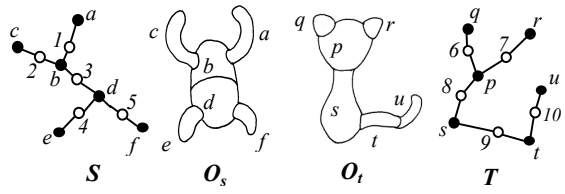

Figure 2: *The source object O_s, the target object O_t, and their respective connectivity graphs **S** and **T**. In this figure and subsequent figures, the connections of connectivity graphs are shown explicitly as white nodes for ease of illustration.*

In this framework, a component does not necessarily have to be semantically meaningful. When a user wants to manipulate some polygons as a whole, these polygons can be grouped as a component. The component decompositions of two original meshes need not be compatible in the sense of having the same number of components and the same connectivity among the components. A user can define components either manually using system-provided tools, or using some automatic methods [15, 16, 19] to obtain an initial decomposition. Besides, pre-defined components in meshes (e.g. OBJ or VRML files) can also be used. Figure 3 shows a typical workflow of our framework.

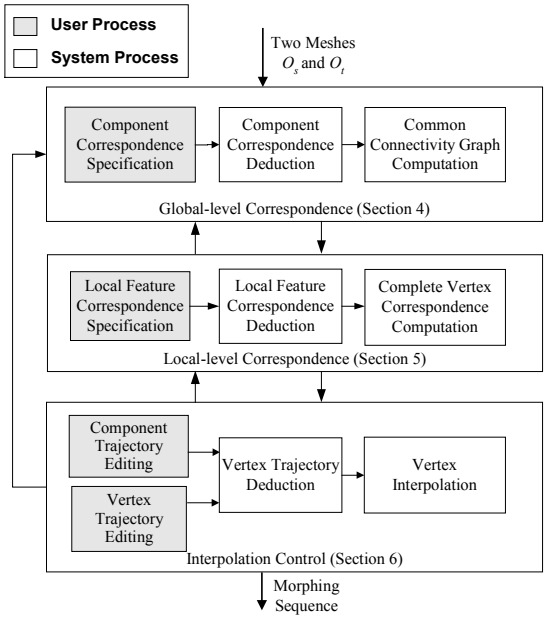

Figure 3: *Main steps in a typical workflow of our framework. At any step, a user can re-visit any previous step to modify the specifications. In addition, the user only needs to specify those requirements of interest, and the system can complete the remaining work through deductions.*

4. Global-Level Correspondence

Given O_s and O_t, users can specify correspondences over their components. The result of this global-level correspondence process is to establish correspondences between all components in O_s and O_t.

Section 4.2 introduces our method for recording user-specified correspondences. In this framework, not only can users specify one-to-one correspondences over components (i.e. correspondence of exactly one component of O_s to one component of O_t), they can also specify group-to-group correspondences, where a group of components in O_s is paired with a group in O_t. With the ability of specifying correspondences over component groups, the user can choose to defer decisions on component decomposition, thus allowing a user to experiment with the morph.

Section 4.3 organizes all recorded correspondences to keep the history of user specifications, enabling the support of undoing specifications and incremental modifications to component decomposition.

Section 4.4 introduces how the framework assists user specification by locating the probable counterparts of user-selected components. Such counterparts are those components, which have similar connectivity to the selection.

Section 4.5 provides details on the computation of the complete component correspondence. A user can choose to specify any number of correspondences, and the framework finally deduces component pairs accordingly with all user specifications respected. In achieving this, our framework provides a range of automations: from totally manual (where the user specifies detailed correspondences for every component) to semi-automated (where the user specifies only important correspondences) to fully automated (where the system computes all correspondences).

Some definitions are first introduced in Section 4.1.

4.1. Terminology

The source connectivity graph **S** and the target connectivity graph **T** are represented as the two graphs $G(V_S, E_S)$ and $G(V_T, E_T)$ respectively, where V_S and V_T are sets of components and E_S and E_T are sets of connections. For the example of **S** and **T** in Figure 2, we have $V_S = \{a,b,c,d,e,f\}$, $E_S = \{1,2,3,4,5\}$ and $V_T = \{p,q,r,s,t,u\}$, $E_T = \{6,7,8,9,10\}$. Note that when a user modifies the decomposition, for example, by merging two connected components, the connectivity graph is changed and updated automatically. Given **S** and **T**, a user can specify a global-level correspondence by associating a group of components of O_s with a group of components of O_t. Such a user-specified correspondence is termed a *constraint*.

The final product of this correspondence process is a *common connectivity graph* $\boldsymbol{M}=G(V_M, E_M)$, where V_M is the set of correspondence nodes and E_M is the set of correspondence edges. A *correspondence node* and *a correspondence edge* represent a pair of corresponding components (i.e. *component pairs*) and a pair of corresponding connections (i.e. *connection pairs*) respectively. A correspondence node has one of these forms: (v_s, v_t), (v_s, ζ_V) or (ζ_V, v_t) where $v_s \in V_S$, $v_t \in V_T$, and ζ_V denotes a *null-component* used when a component has no counterpart. Every v_s or v_t appears in exactly one correspondence node of $\boldsymbol{M}$. Similarly, a correspondence edge has one of these forms: (e_s, e_t), (e_s, ζ_E) or (ζ_E, e_t) where $e_s \in E_S$, $e_t \in E_T$ and ζ_E denotes a *null-connection* used when a connection has no counterpart. Every e_s or e_t appears in exactly one correspondence edge of $\boldsymbol{M}$. For the two connectivity graphs in Figure 2, a possible $\boldsymbol{M}$ constructed is shown in Figure 4.

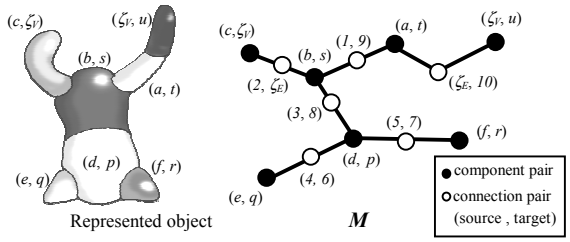

Figure 4: *Common connectivity graph* $\boldsymbol{M}$. *Note that the component c of the source, which is paired with a null component* ζ_V, *will gradually shrink during the morph. Conversely, the component u of the target will gradually grow.*

4.2. Correspondence Between Component Groups

To enable correspondence specification between groups of components, we need to keep track of all possible correspondences over components and over connections for both $\boldsymbol{S}$ and $\boldsymbol{T}$. Naïvely recording all these possibilities is generally inefficient in terms of storage and computation. Instead, we record and update them in an implicit and concise way as described below.

4.2.1. Permissibility

A *permissible component pair* is defined as a component pair that can possibly appear in the final $\boldsymbol{M}$. The set of all permissible component pairs is denoted as R_V. Note that $\{(v_s, \zeta_V) \mid v_s \in V_S\}$ and $\{(\zeta_V, v_t) \mid v_t \in V_T\}$ are always subsets of R_V, as it is always possible that a component of an object has no counterpart from the other object. In addition, a correspondence between two groups of components is denoted as $\langle X, Y\rangle$ where $X \subseteq V_S$ and $Y \subseteq V_T$. Specifically, $\langle X, Y\rangle$ means that both $(x, \overline{y})$ and $(\overline{x}, y)$ are not permissible for all $x \in X$, $y \in Y$, $\overline{x} \in V_S - X$ and $\overline{y} \in V_T - Y$. We say $\langle X, Y\rangle$ is *permissible* if, and only if, $\forall x \forall y$, $(x, y) \in R_V$. The permissibility of a correspondence over connections can be defined in a similar way. The set of all *permissible connection pairs* is denoted as R_E.

A combined notation for correspondences over both components and connections has the form $\langle \boldsymbol{P}, \boldsymbol{Q}\rangle$, in which $\boldsymbol{P} = G(X, E)$, where $X \subseteq V_S$, $E \subseteq E_S$, and $\boldsymbol{Q} = G(Y, F)$, where $Y \subseteq V_T$, $F \subseteq E_T$. Note that in $\boldsymbol{P}$, X may not contain all the nodes that the edges in E are incident to. Therefore, $\boldsymbol{P}$ may not be a usual graph and likewise for $\boldsymbol{Q}$. They are denoted as graphs here for the convenience of description. The definition of permissibility of correspondences is extended to this notation: $\langle \boldsymbol{P}, \boldsymbol{Q}\rangle$ is *permissible* if, and only if, $\langle X, Y\rangle$ is permissible and $\langle E, F\rangle$ is permissible.

Throughout the process of specifying global-level correspondences, all current correspondences between $\boldsymbol{S}$ and $\boldsymbol{T}$ are encapsulated in a *correspondence set* defined as $\mathbb{C} = \{\langle \boldsymbol{P}_i, \boldsymbol{Q}_i\rangle \mid i = 1, 2, ..., n\}$, where $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i\rangle$ is permissible and $\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_n$ is a partition of $\boldsymbol{S}$ and $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_n$ is a partition of $\boldsymbol{T}$. Let $\boldsymbol{P}_i = G(X_i, E_i)$ and $\boldsymbol{Q}_i = G(Y_i, F_i)$, we have R_V and R_E implicitly recorded as:

$$R_V = \bigcup_i \{(X_i \cup \{\zeta_V\}) \times (Y_i \cup \{\zeta_V\})\} - \{(\zeta_V, \zeta_V)\}$$

$$R_E = \bigcup_i \{(E_i \cup \{\zeta_E\}) \times (F_i \cup \{\zeta_E\})\} - \{(\zeta_E, \zeta_E)\}$$

4.2.2. Processing Constraints

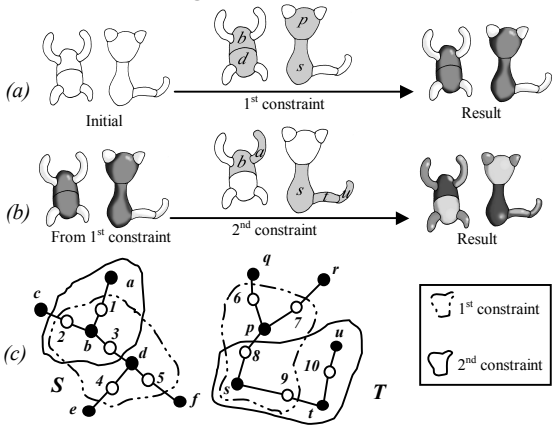

Figure 5: *Maintenance of* $\mathbb{C}$ *after applying two constraints to* $\boldsymbol{S}$ *and* $\boldsymbol{T}$ *in Figure 2. In (a) and (b), the component correspondences after each constraint are shown as the rightmost pictures (see also the color plate). In (c), the first constraint* $\langle \{b,d\}, \{p,s\}\rangle$ *which induces the connection correspondence* $\langle \{1,2,3,4,5\}, \{6,7,8,9\}\rangle$ *is circled in dashed lines while the second constraint* $\langle \{a,b\}, \{s,t,u\}\rangle$, *which induces the connection correspondence* $\langle \{1,2,3\}, \{8,9,10\}\rangle$ *is circled in solid lines.*

When a user specifies a constraint, pairing components is more intuitive than pairing connections. Therefore, in the current implementation, a constraint is a correspondence over components $\langle X, Y\rangle$, where $X \subseteq V_S$, $Y \subseteq V_T$. From such a constraint, the system induces another correspondence between all connections incident to components in X, denoted as the set E, and

all connections incident to components in Y, denoted as the set F. Both correspondences are jointly represented as $\langle \boldsymbol{P}, \boldsymbol{Q} \rangle$ where $\boldsymbol{P}=G(X, E)$ and $\boldsymbol{Q}=G(Y, F)$.

Initially, we have $\mathbb{C} = \{\langle \boldsymbol{S}, \boldsymbol{T} \rangle\}$, where every component of $\boldsymbol{S}$ can be possibly paired with every component of $\boldsymbol{T}$. Given a $\langle \boldsymbol{P}, \boldsymbol{Q} \rangle$, we partition each $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle \in \mathbb{C}$ into $\langle \boldsymbol{P}_i', \boldsymbol{Q}_i' \rangle$ and $\langle \boldsymbol{P}_i'', \boldsymbol{Q}_i'' \rangle$, where $\boldsymbol{P}_i' = \boldsymbol{P} \cap \boldsymbol{P}_i$, $\boldsymbol{Q}_i' = \boldsymbol{Q} \cap \boldsymbol{Q}_i$, $\boldsymbol{P}_i'' = \boldsymbol{P}_i - \boldsymbol{P}_i'$ and $\boldsymbol{Q}_i'' = \boldsymbol{Q}_i - \boldsymbol{Q}_i'$. Any trivial cases of $\langle G(\phi,\phi), G(\phi,\phi) \rangle$ are removed from $\mathbb{C}$. It can be shown that all the correspondences in $\mathbb{C}$ are permissible and $\mathbb{C}$ now successfully encapsulates $\langle \boldsymbol{P}, \boldsymbol{Q} \rangle$. Figure 5 shows our constraint processing upon two constraints for $\boldsymbol{S}$ and $\boldsymbol{T}$ in Figure 2.

4.3. Constraint Tree

To record the history of specified constraints, we use a binary tree termed *constraint tree*, where each $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ in $\mathbb{C}$ is represented as a leaf. Whenever we perform a partitioning of $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ upon a new constraint, we create a left child and a right child for this leaf, which correspond to the new correspondences $\langle \boldsymbol{P}_i', \boldsymbol{Q}_i' \rangle$ and $\langle \boldsymbol{P}_i'', \boldsymbol{Q}_i'' \rangle$ respectively. Obviously the i^{th} constraint is encapsulated in those nodes at the i^{th} level of the constraint tree, and the current correspondence set $\mathbb{C}$ is actually the set containing all the leaves.

The content of a parent node is always equal to the union of the contents of its children. The constraint tree for the example in Figure 5 is shown in Figure 6. Such a tree structure makes it easy to undo any constraint specification while other constraints are unaffected, as discussed in the next subsection.

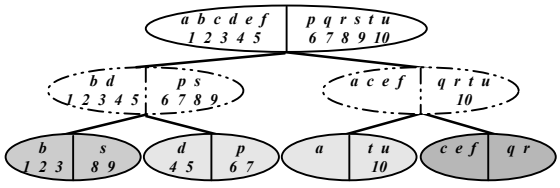

Figure 6: *Constraint tree for the example in Figure 5.*

4.3.1. Flexible Undoing

When a user wants to undo a specific i^{th} constraint, a naïve approach is to remove constraints in reverse order from the most recent one to the i^{th} constraint. As a result, the user will lose all those constraints that are specified after the i^{th} constraint. Using the constraint tree, this framework is able to remove solely the influences of the i^{th} constraint. For each pair of nodes n_l and n_r at level i, let n_p be the parent of n_l and n_r. We merge the two respective left (right, respectively) subtrees of n_l and n_r to be the new left (right, respectively) subtree of n_p. To merge two subtrees, we superimpose them and combine the contents of the superimposed nodes to obtain a new subtree having the same structure. This operation effectively removes the nodes at the level representing the unwanted constraint from the constraint tree.

For the example in Figure 6, if we remove the first constraint, we merge the leaf containing b and the leaf containing a to be the left child of the root and merge the other two leaves to be the right child of the root. The updated constraint tree is exactly the same as a constraint tree with only the second constraint.

4.3.2. Modifying Component Decomposition

When a user modifies the component decompositions of $\boldsymbol{O}_s$ and $\boldsymbol{O}_t$ in the process of specifying correspondences. In such a case, we need to update $\boldsymbol{S}$, $\boldsymbol{T}$ and the constraint tree while preserving all unaffected constraints. Such a modification can always be expressed as one of the two kinds of operations: splitting a component into two, and merging two connected components into one.

Splitting can be handled easily, as we only need to find the leaf that contains the component to be split, replace it with two new components and update their connections. The structure of the constraint tree is not affected.

Merging is more complicated if the components to be merged are not within the same leaf. In such a circumstance, the structure of the constraint tree has to be updated. First, we locate the leaves where the two components to be merged are located. Next, we search upward from the leaf level to find their nearest common ancestor. The constraint that is represented by the level below this common ancestor is then the one that separates the two components. Then, this constraint is removed via the flexible undoing discussed in the previous subsection. These three steps are repeated until all the constraints that cause the separation of the two components are removed. Finally, the components to be merged are replaced with the new component and the connections are updated accordingly.

4.4. Identifying Candidates

The constraint tree keeps track of all the permissible component pairs and connection pairs implicitly. Therefore, for a leaf $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$, all the components in $\boldsymbol{P}_i$ are naturally possible counterparts for any component in $\boldsymbol{Q}_i$, and vice versa. In a morphing, however, a user generally expects the connectivity among components be kept as much as possible. For example, if the head and the body of a cow have been paired with the head and the body of a triceratops respectively, the user may not regard the horns of the triceratops as good counterparts for a leg of the cow although they are in the same leaf. Therefore, the user usually expects a good counterpart to be *similar in connectivity* to the selected components. We call a counterpart that can meet such kind of user expectation a *candidate*. To identify candidates from the set of possible counterparts, we need to analyze the connectivity as follows.

Within a leaf, the components are placed into groups of maximally connected components. In Figure 7, for example, leaf ***II*** has one group of ***T*** where *t* and *u* are connected by *10* while leaf ***IV*** has two groups of ***T***, which contains *q* and *r* respectively. For a group within a leaf, a different leaf is called its *neighboring leaf* if the leaf contains a connection incident to any component within the group, or contains a component incident to any connection within the group. Consequently, we define the similarity in connectivity as follows: a group in ***S*** is similar in connectivity to a group in ***T*** if they are from the same leaf of the constraint tree and have the same set of neighboring leaves; see Figure 7 for an example.

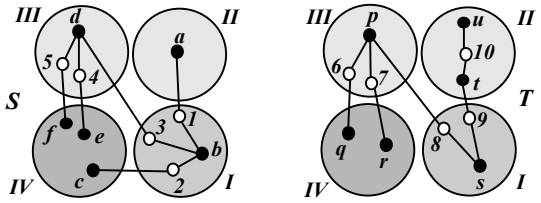

Figure 7: *Analysis of similarity in connectivity for the example in Figure 6. For clarity, the contents of **S** and **T** are shown separately and in the same shades as the leaves shown in Figure 6. In leaf **IV**, the group {e} in **S** is similar in connectivity to both the group {q} and the group {r} in **T** as they have the same set of neighboring leaves, {**III**}. On the other hand, the group {c} in **S** has no similar counterpart in **T**.*

The above definition can also be extended to provide a range of intermediate degrees of similarity in connectivity for two groups, by considering how much their sets of neighboring leaves overlap.

4.5. Constructing Common Connectivity Graph

Once the user finishes all the specifications about component correspondence, the framework constructs ***M*** as follows. We first refine the constraint tree and then construct correspondence nodes and correspondence edges of ***M*** from all its non-empty leaves.

The refinement step first processes the tree until each resulting leaf has at most one maximally connected group from ***S*** and at most one from ***T***, giving preference to groups which are similar in connectivity. Then, we pair individual components and connections, giving preference to those components connected to other leaves, until each resulting leaf contains at most one component or connection of ***S*** and at most one component or connection of ***T***. Figure 8 shows a possible result for the example in Figure 7 after the refinement step.

When the refinement step is done, the framework then calculates the common connectivity graph ***M*** from the current correspondence set $\mathbb{C}$ as follows. First, construct a correspondence node for the component pair in every $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$. ζ_V is paired with every component that has no counterpart. Then, construct a correspondence edge for connections in every $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ similarly. ζ_E is paired with every connection that has no counterpart. Wherever any connection from a correspondence edge is incident to any component from a correspondence node, we set this correspondence edge to be incident to this correspondence node. For the example in Figure 8, the computed ***M*** is shown in Figure 4.

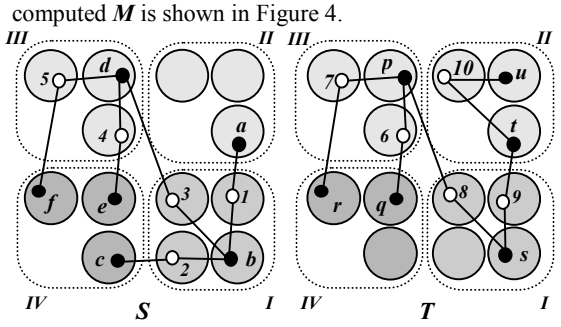

Figure 8: *In Figure 7, leaf **IV** contains multiple groups of maximally connected component. The components groups {e} and {f} can be arbitrarily paired with {q} and {r} as they both have leaf **III** as their only neighboring leaf in Figure 7. Then, the individual components and connections are paired, with preference to those directly connected to other leaves. The result is as shown in this figure.*

5. Local-Level Correspondence

For each component pair containing no null-component, users can specify and pair local features to control the morphing (Section 5.1). In addition to *user-specified* local-level correspondences, the framework deduces *implied* local-level correspondences according to user specifications (Section 5.2) and adds *assumed* ones where appropriate to improve the morphing (Section 5.3). An automatic patch-partitioning method (Section 5.4) is proposed here to create compatible patch layouts subsequently. These are used to establish the complete vertex correspondence for this component pair. For any component pair containing a null-component, its complete vertex correspondence is constructed automatically (Section 5.5).

5.1. User-Specified Local-level Correspondences

In this framework, a user can specify several types of local features within a component: a *feature vertex* is a mesh vertex, a *feature line* is a sequence of connected mesh edges and a *feature loop* is a closed loop of mesh edges. Then, the user can pair a local feature in one component with one in its corresponding component; the two local features then forms a *local feature pair*. Two end-vertices of a feature line are also treated as feature vertices. Note that after obtaining a morph, a user can still revisit this step to add or modify local feature pairs in order to improve the morph.

5.2. Implied Local-level Correspondences

A boundary between two connected components is shared by both of them. Thus, from those user-specified

global-level and local-level correspondences involving boundaries, the framework is able to deduce implied local-level correspondences to save user effort as discussed in the following.

First, correspondences over component boundaries can be derived from connection pairs in the common connectivity graph $\boldsymbol{M}$. Each boundary of a component represents a connection. Thus, from $\boldsymbol{M}$, we can deduce a set of corresponding boundaries (each boundary is then a feature loop). For the example of $\boldsymbol{M}$ in Figure 4, the boundary of component *d* connecting component *f* and the boundary of component *p* connecting component *r* form an implied local feature pair for component pair *d* and *p* as deduced from the correspondence edge (*5*,*7*). Also, this correspondence edge implies another local feature pair at boundaries in the component pair *f* and *r*.

Second, within a pair of components, if two corresponding local features are both specified at boundaries and the two neighbors at these two boundaries are also corresponding, the framework also records these two local features as a pair of local features in their neighbors.

5.3. Assumed Local-level Correspondences

A user can choose to specify just those local feature pairs of interest. The framework is then able to add assumed local feature pairs when the user has finished the specification so far. In the event that the user subsequently specifies other new feature pairs after the assumed features are added, the existing assumed features are removed and new ones are calculated where appropriate. In the following, we define the *distance between two vertices* as the length of the shortest path between them along the mesh edges, and the *distance between two local features* as the minimum distance between their vertices.

First, consider two corresponding components each have only one local feature. For example, in Figure 9(b) (in the color plate), the boundary of one tail and that of the other tail are corresponding, as deduced from $\boldsymbol{M}$. To better align the two components, the framework adds one more feature vertex pair at their tips. In such a component, the vertex farthest away from its boundary is computed here and treated as its *tip vertex*. Such assumed correspondence between tip vertices can help to avoid the "tip-shrinkage" problem mentioned in [9]. Thus, each component has at least two local features after all assumed tip vertex pairs are computed.

Second, for two corresponding boundaries l_s and l_t of a component pair (c_s, c_t), there should be at least two feature vertex pairs on them so that the system knows how they are aligned during morphing. When the user does not provide this, the framework will add assumed feature vertex pairs on l_s and l_t as follows.

The general idea to decide the alignment is to examine l_s and l_t, together with local features nearby, to obtain a relative orientation. This is an attempt to avoid undesirable twisting during a morph. For l_s, we locate its nearest local feature l_s' on c_s. Let v_s be the vertex of l_s on the shortest path from l_s to l_s'. Let l_t' be the corresponding local feature of l_s' on c_t, and let v_t be the vertex of l_t on the shortest path from l_t to l_t'. We set (v_s, v_t) as one assumed feature vertex pair, and obtain another assumed feature vertex pair formed by the two vertices on the boundaries farthest away from (or opposite to) v_s and v_t respectively. Though the treatment for l_s and l_t is asymmetric in this approach, our experiment has shown reasonably good outcomes in most cases, such as the one shown in Figure 9(a) (in the color plate).

Third, if there is more than one connection between two components in a mesh and the same between their corresponding components in the other mesh, the correspondences between their boundaries are not uniquely defined and are assigned arbitrarily in the computation of $\boldsymbol{M}$. When the user specifies correspondences between some boundaries, the framework assumes the correspondences for the remaining boundaries by re-computing $\boldsymbol{M}$. Figure 9(c) (in the color plate) illustrates this in the morphing from a mug to a donut.

5.4. Automatic Patch Partitioning

Given local feature pairs in corresponding components, there are various methods to establish complete vertex correspondence with all local feature pairs aligned. The common approach is to first partition the meshes into pairs of compatible patches and then perform mapping and merging for each pair of patches [6, 9, 11].

Praun *et al.* [17] present an algorithm to establish compatible patch layouts for meshes that share a common coarse model and respect features. Generally, a user needs to choose base domain and identify features for these meshes. Given such meshes, this method establishes correspondences between meshes and allows remeshes with the same connectivity. In our framework, however, users can choose to specify any number of local feature pairs for two corresponding components. As such, we propose an algorithm to automatically construct the common base domain based on all local-level correspondences within a component pair. Then the parameterization method in [17] is employed to establish compatible patch layouts for the component pair. Details are as follows.

First, we group all local features in each component into *feature groups*, each of which is a maximally connected group of local features. Second, a spanning tree is constructed by connecting all the feature groups in one component. Third, another spanning tree is

constructed to connect all the feature groups that are in the leaf nodes of the first spanning tree. At this stage, the component being processed is partitioned into patches according to these two spanning trees. Last, we treat the net formed by the two spanning trees as a base domain to guide the partitioning of the other component into a compatible patch layout. The user can choose to modify these patch layouts when necessary.

With compatible patch layouts for each pair of components, there are several methods for topological merging that can be used to create the complete vertex correspondence for each pair of patches, for example, barycentric and harmonic mapping [10, 20]. However, establishing the meta-mesh by overlapping the mapping of two meshes is known to be costly in computation. By working on components instead of the whole meshes, such computation in our framework is speeded up. Alternatively, the framework can also use other efficient methods such as multi-resolution remeshing in the place of topological merging.

5.5. Handling Null-components

For a component pair (c_s, c_t) where $c_s \in \boldsymbol{O_s}$ and $c_t=\zeta_V$ (or $c_s=\zeta_V$ and $c_t \in \boldsymbol{O_t}$, respectively), there will be a component disappearing (or growing, respectively) in the morphing sequence. Without loss of generality, we only discuss the case of component disappearing ($c_t=\zeta_V$). Let c_s' be a component connected to c_s at boundary l_s in $\boldsymbol{O_s}$ and (c_s', c_t') be a component pair. The system handles this in two ways according to user input.

In the first case, when the user specifies local-level correspondence for the component pair (c_s', c_t'), the user can assign a local feature l_t in c_t' to be the counterpart of l_s. In this case, the framework is responsible for producing a morph where c_s will gradually disappear into l_t. The following method is applied then to automatically construct a new component c at l_t to be the counterpart of c_s. First, it creates the topology (i.e. the mesh connectivity) of c by copying the topology of c_s. Note that vertices of both components naturally form *vertex pairs*, each of which comprise two corresponding vertices in c_s and c. Next, for each vertex pair (v_s, v_t), if v_s is at l_s, v_t is already a vertex of l_t; otherwise, v_t is set at the position of v where (u, v) is a vertex pair and u is the boundary vertex closest to v_s. By replacing c_t with c, the framework then establishes the complete vertex correspondence for this component pair.

In the second case, when the user has not assigned correspondence for the boundary l_s when the user continued into the process of patch partitioning, the system simply merges c_s into c_s' so that the counterpart of l_s is determined by the topological merging. Note that when $c_t'=\zeta_V$, the merging of c_s into c_s' is also applied.

6. Interpolation Control

Given the complete vertex correspondence for every component pair, various methods can be applied for vertex interpolation, for example, linear interpolation or as-rigid-as-possible interpolation [3] to obtain a morph. Based on the use of components, this framework enables a user to edit morphing trajectories at both the global and the local levels.

Through manual or automatic means, a user can construct a *skeleton* for a mesh by assigning for each component a *bone*, which is a polyline. It is not required that the bone for a component accurately represents the component's shape.

Although the use of skeletons is not novel, there are several challenges in using skeletons in our framework. The most obvious one is that the skeletons of two original meshes are usually very different in both their shapes and numbers of bones. Due to the one-to-one relationship between a component and its bone, we construct correspondences over bones directly from the common correspondence graph $\boldsymbol{M}$, by adding null-bones for null-components in $\boldsymbol{M}$. Subsequently, the morphing of skeletons can be computed by interpolating between every two corresponding bones.

Skeleton-based control has been used in many applications (see [4, 5, 13]). To bind mesh vertices to underlying bones in a morph, there are many possible implementations. One is to adapt the weighted vertex method to blend the associations between mesh vertices and the line segments of underlying bones. For each mesh vertex, we find the point on its underlying bone that is nearest to the vertex, and compute its arc-length values along the bone. By interpolating arc-length values and relative positions of mesh vertices with respect to their nearest points on the bones, each mesh vertex in the source can be transformed to its corresponding vertex in the target according to the complete vertex correspondence.

Due to the low computational cost of skeleton morphing, such a morph can serve as a fast means to get a general idea of how the final morph will look like. Therefore, users can decide whether to modify the global-level specifications before moving on to the more costly computation of mesh morphing.

As for user control in the interpolation step, a user can specify the position or orientation of a component by editing keyframes of the bone of the component. Accordingly, the system is able to deduce the trajectories of its vertices. Such kind of editing is usually much easier and more intuitive than editing trajectories of individual vertices. Figure 10 illustrates this using a morphing sequence where we combine the morph from a calf to a cow with the morph from the cow to a

triceratops. In each morph, we add two component keyframes to achieve the walking effect from the calf to the cow then to the triceratops. In addition, the user can also specify vertex trajectory at an intermediate frame. In that case, new vertex position will be used to update the arc-length and relative position of vertices at that frame. Thus, the user can achieve sophisticated morphing trajectories efficiently at both global and local levels.

Figure 10: *Component keyframe editing. In this morph, a calf is transformed into a cow, then into a triceratops while walking.*

7. Results

We implemented a prototype system for the component–based morphing framework on a Pentium IV 2GHz PC. The main focus of our experiments is to test the efficiency and effectiveness of user interaction reported in this paper. Below are samples of our experiments (Figures 11 to 14 are shown in the color plate). More results and video files are available at:
http://www.comp.nus.edu.sg/~tants/morphing.html.

• Trial-and-error morphing design using components (a duck and a dinosaur)
The user first cut each object into its tail, body and head. Then, the user paired solely the two heads, and the system produced a morph accordingly. Realizing that there were six legs in the intermediate objects, as shown in Figure 11(a), the user then went back to specify two legs for each object and pair their right legs, as shown in Figure 11(b). Maintaining all previous user specifications, the system updated the morph correspondingly and produced a better morph. The whole process took just a few minutes. It can be clearly seen that a user can design a morph at the global level without considering any mesh detail and can refine the specifications whenever necessary.

• Effective deduction in genus-1 morphing (a mug and a donut)
This example shows that our user can also design high-genus morphing with ease. In this example, the user first decomposed each object into two components, and then paired the body of the mug with a half of the donut. Because there are two connections between the two components in each object, the user then paired one boundary in the mug with one boundary in the donut, as shown in Figure 9(c). Based on these user specifications, the system completed other tasks through deduction and the whole morph is completed in less than 1 minute. Figure 12 shows deduced local feature pairs in (a) and computed patch layouts in (b). Note that we achieved this morph shown in (c) by just a few clicks, whereas in [9] and [12], to design a similar morph, the user spent more than half an hour.

• Trial-and-error morphing design at the local level (a rocket and a glass)
Figure 13 illustrates trial-and-error morphing design at the local level. After cutting each object into three components, the user specified one pair of components and then the system produced a morphing accordingly. Realizing that intermediate shapes were distorted at the top, as shown in (a), the user then revisited the local-level correspondence step to add one pair of feature lines as in (b). Subsequently, the system respected all the specifications and produced a better morph as in (c).

• Different morphing designs using components (a rocket and a duckling)
Figure 14 demonstrates the effectiveness and efficiency of utilizing components in multi-level morphing control. Given the same models, a rocket and a duckling, the user assigned different component correspondences and conveniently achieved two interesting morphs shown in (a) and (b).

Table 1 reports the numbers of user-specified and system-deduced correspondences for our examples. From the results, we can see that user interaction in our framework is efficient and effective, and all our morphs were obtained within a few minutes. We also invited several non-expert students in our university to try our system, and they all reported that it is easy and convenient for them to design a morph in our system.

8. Discussion

We have demonstrated a system that implements our proposed framework. The novelty of the component-based morphing framework lies in that it reduces user workload and provides flexibility in user control such that even an amateur can design a morph with ease. Specifically, it utilizes components to address major issues about interactive control, and its main advantages are summarized as follows.

First, with the component decomposition, mesh vertices can be perceived and manipulated in groups. Connectivity among components, which is much simpler than that among vertices, is capitalized in our framework. This makes user interaction in both the correspondence and the interpolation steps intuitive and efficient, especially when dealing with meshes of complex shapes. In addition, users can still fine-tune morphs by working directly on individual vertices.

Second, the whole framework is designed with the philosophy of helping users as much as possible and not imposing on users any system-caused restriction. In every step of the whole morphing process, the system first gets user specifications, then deduces implied user

Morphs		Cow-Triceratops	Duck-Dino	Mug-Donut	Rocket-Glass	Rocket-Duckling	
Triangles in source / target		5806 / 5660	550 / 5076	1640 / 576	330 / 2642	330 / 3836	
Components in source / target		9 / 11	5 / 5	2 / 2	3 / 3	7 / 7	
Figure number		9(a)-(b)	11	12	13	14 (a)	14 (b)
USER-SPECIFIED component correspondences		4	2	1	1	3	3
USER-SPECIFIED local feature pairs		2	0	1	3	2	3
SYSTEM-DEDUCED local feature pairs		23	12	4	10	16	15
Estimated user time	specifying component correspondences	15 sec	10 sec	5 sec	5 sec	10 sec	10 sec
	specifying local feature pairs	10 sec	0 sec	5 sec	20 sec	15 sec	20 sec

Table 1: *Statistics of our morphing examples*

requirements based on these specifications, and finally adds assumed but reasonable choices. Moreover, if a user revisits this step to modify the specifications, the system updates assumed choices, respecting all user specifications. The use of the constraint tree and the deduction of implied and assumed local-level correspondences are means of realizing this philosophy.

Our framework has several limitations and potential extensions. Currently, we do not deal with the case where a boundary is shared by more than two components. In this case, a connectivity graph will be replaced by a connectivity hyper-graph. Another challenging extension is to handle topological changes, which includes changes in genus, during morphing design. Such a morph involves the appearing or disappearing of a connection between two components. Consequently, we should develop techniques to handle such changes in connection, and to deduce implied and assumed correspondences over such connections.

Acknowledgements
We would like to thank Dr. Huang Zhiyong for many helpful discussions. This research is supported by the National University of Singapore under grant R-252-000-107-112.

References

1. M. Alexa. Merging Polyhedral Shapes with Scattered Features. *Visual Computer*, 16:26-37, 2000.
2. M. Alexa. Recent Advances in Mesh Morphing. *Computer Graphics Forum*, 21(2):173-196, 2002.
3. M. Alexa, D. Cohen-Or and D. Levin. As-rigid-as-possible Shape Interpolation. *SIGGRAPH'00*, 157-164, 2000.
4. J. Bloomenthal and C. Lim. Skeletal Methods of Shape Manipulation. *International Conference on Shape Modeling and Applications*, 44-47, 1999.
5. R. Blanding, G. Turkiyyah, D. Storti and M. Ganter. Skeleton-based Three-dimensional Geometric Morphing. *Computational Geometry – Theory and Applications*, 15:129-148, 2000.
6. D. DeCarlo and J. Gallier. Topological Evolution of Surfaces. *Graphics Interface'96*, 194-203, 1996.
7. E. Galin and S. Akkouche. Shape Constrained Blob Metamorphosis. *Implicit Surfaces'96*, 2:9-23, 1996.
8. E. Galin, A. Leclercq and S. Akkouche. Blob-Tree Metamorphosis. *Implicit Surfaces'99*, 4:9-16, 1999.
9. A. Gregory, A. State, M. Lin, D. Manocha and M. Livingston. Feature-based Surface Decomposition for Correspondence and Morphing between Polyhedra. *Computer Animation'98*, 64-71, 1998.
10. T. Kanai, H. Suzuki and E. Kimura. Three-Dimensional Geometric Metamorphosis Based on Harmonic Maps. *Visual Computer*, 14(4):166-176, 1998.
11. T. Kanai, H. Suzuki and E. Kimura. Metamorphosis of Arbitrary Triangular Meshes. *IEEE CG&A*, 62-75, 2000.
12. A. Lee, D. Dobkin, W. Sweldens and P. Schroder. Multiresolution Mesh Morphing. *SIGGRAPH'99*, 343-350, 1999.
13. F. Lazarus and A. Verroust. Metamorphosis of Cylinder-like Objects. *Journal of Visualization and Computer Animation*, 8:131-146, 1997.
14. F. Lazarus and A. Verroust. Three-dimensional Metamorphosis: a Survey. *Visual Computer*, 14: 373-389, 1998.
15. X. Li, T. Woon, T. Tan and Z. Huang. Decomposing Polygon Meshes for Interactive Applications. *I3DG'01*, 35-42, 2001.
16. A. Mangan and R. Whitaker. Partitioning 3D surface Meshes using Watershed Segmentation. *IEEE Transaction on Visualization and Computer Graphics*, 5(4):308-321, 1999.
17. E. Praun, W. Sweldens and P. Schröder. Consistent Mesh Parameterizations. *SIGGRAPH'01*, 179-184, 2001.
18. M. Shapira and A. Rappoport. Shape Blending using the Star-skeleton Representation. *IEEE CG&A*, 15:44-50, 1995.
19. S. Shlafman, A. Tal and S. Katz. Metamorphosis of Polyhedral Surfaces using Decomposition. *Eurographics'02*, 21(3):219-228, 2002.
20. M. Zöckler, D. Stalling and H. Hege. Fast and Intuitive Generation of Geometric Shape Transitions. *Visual Computer*, 16:241-253, 2000.

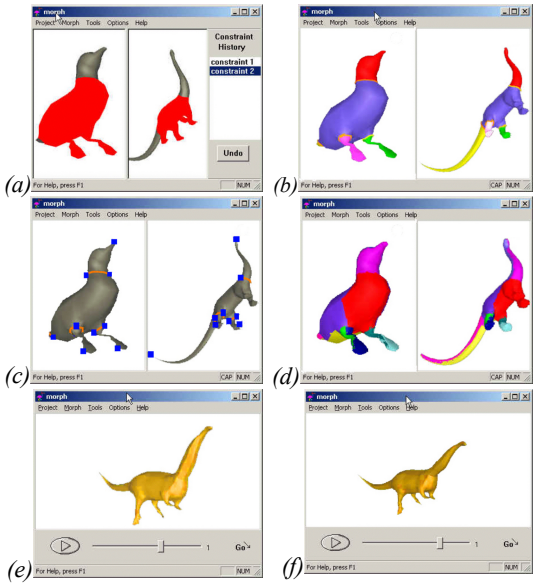

Figure 1: *An example of user control in the framework. (a) The user only pairs components of interest. (b) The system deduces the complete component correspondence. (c) The user only specifies local feature pairs of interest (but none here); the system deduces implied (orange) and assumed (blue) local features. (d) Through automatic patch partitioning, the system computes the complete vertex correspondence for the two meshes. (e) The system produces the morphing sequence through vertex interpolation. (f) The user manipulates a component as a whole (such as the head here) to edit morphing trajectories.*

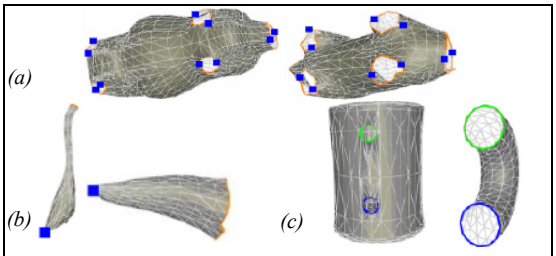

Figure 9: *(a) Assumed feature vertex pairs at boundaries. With correspondences at boundaries (shown in orange) for two body components, the system adds two pairs of feature vertices (shown in blue) for every two corresponding boundaries. (b) Assumed feature vertex pair at tips. Each tail has only one boundary (shown in orange). Based on their shapes, the system adds a pair of feature vertex at their tips (shown in blue). (c) Assumed local feature pair at boundaries. The user specifies one pair of boundaries for the two components, (shown in green); the system then automatically pairs the remaining boundaries (shown in blue).*

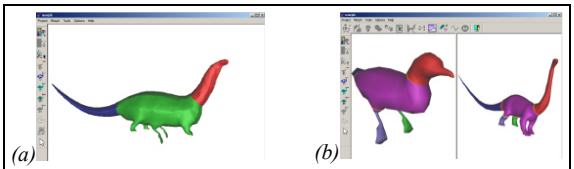

Figure 11: *Global level trial-and-error morph design. (a) A intermediate object which has six legs. (b) Specify two legs and pair the right legs to improve the morphing.*

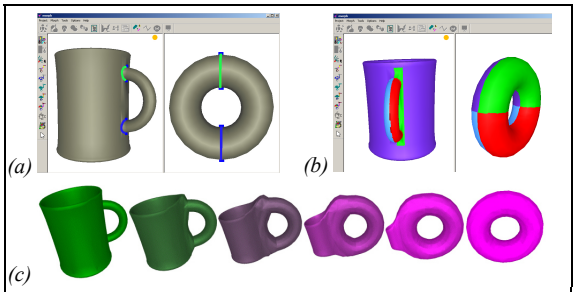

Figure 12: *Demonstration of morphing of genus-1 objects. (a) One user-specified feature loop pair (in green), one system-deduced feature loop pair and four system-deduced feature vertex pairs (in blue). (b) Computed compatible patch layouts based on the six pairs of local features. (c) Morphing from a mug to a donut.*

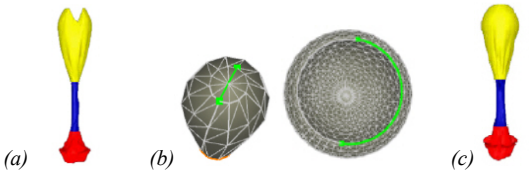

Figure 13: *Local level trial-and-error morph design. (a) A distorted intermediate object in the initial morph. (b) User-specified local feature pairs in their tops. (c) An intermediate object in the improved morph.*

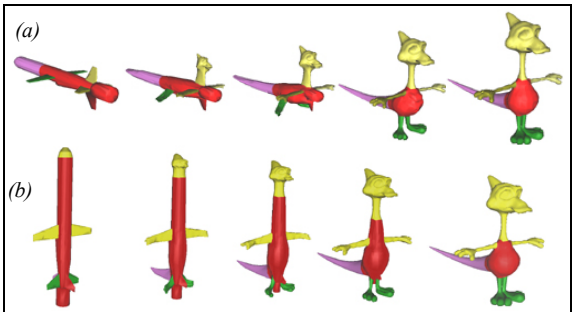

Figure 14: *Two different morphs with same components but different component correspondences.*

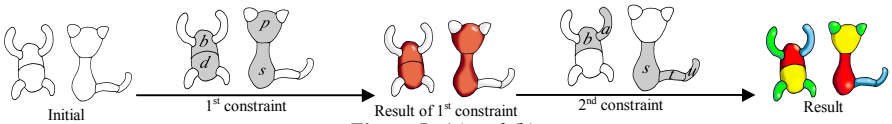

Figure 5: *(a) and (b).*