AN INTERACTIVE FRAMEWORK FOR COMPONENT-BASED MORPHING

ZHAO YONGHONG

NATIONAL UNIVERSITY OF SINGAPORE

2003

AN INTERACTIVE FRAMEWORK FOR COMPONENT-BASED MORPHING

ZHAO YONGHONG
(M.Eng., Zhejiang University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2003

Acknowledgements

First and foremost, I would like to thank my supervisor, A/P Tan Tiow Seng, for his invaluable guidance in my research. What I have learnt from him is not only how to actively tackle research difficulties and explore original solutions, but also the research philosophy, which will benefit me forever.

I would also like to acknowledge my gratitude to A/P Teh Hung Chuan and Dr. Huang Zhiyong for giving me much encouragement and many suggestions.

I am deeply grateful for the assistance of Mr. Ong Hong Yang, Mr. Xiao Yongguan and Mr. Li Xuetao. I got much motivation and inspiration from the collaboration with them.

Finally, I would like to add personal thanks to my husband Ying Peizhi for his love and to my parents for their endless support.

This thesis is to my lovely son and I wish him a happy and healthy future.

Table of Content

List of Figures

Summary

This thesis presents an interactive framework to empower users to conveniently and effectively control the whole morphing process, which includes both establishing correspondence and calculating interpolation. Although research on 3D mesh morphing has reached a state where most computational problems have been solved in general, the novelty of the 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.

In the correspondence process, this framework enables users to specify only those requirements of interest at either the global level over components or the local level within components, whichever is more intuitive. Firstly, given two polygonal meshes, a user can specify global-level correspondences intuitively by pairing components. To facilitate such specifications, a proposed constraint tree is utilized to process user-specified correspondences, identify candidate components for pairing, support modifications to user specifications, and finally deduce correspondences over all components. Secondly, within two corresponding components, the user can fine-tune a morph by specifying correspondences between local features. The framework automatically derives implied local-level correspondences according to user specifications, and adds assumed ones where appropriate to improve the morph. An

automatic patch-cutting method is then applied to create compatible patch layouts with all local feature pairs aligned.

In the interpolation process, not only can the user modify trajectories of individual vertices at the local level, but also manipulate trajectories of components as a whole at the global level. Firstly, the user can assign an underlying bone for each component and all bones in a mesh form its skeleton. Based on global-level correspondences, the framework can then compute morphing of skeletons. Secondly, a proposed skeleton-guided interpolation method is applied to transform mesh vertices around underlying skeletons and thus produces morphing results that are natural, realistic and rigidity preserving. Thus, the user can predict the final morph from morphing of skeletons at an early stage, and control the interpolation process at both levels.

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 a morph with ease.

A prototype for the component-based morphing framework was implemented and used to produce a number of morphs for meshes with complex structures. In the experiments, we focused on testing the efficiency of user control. Our results show that users can conveniently experience different morphing designs and the overall user time for each morph is only a couple of minutes.

Keywords: mesh morphing, interactive techniques, component decomposition, animation, shape blending, deformation

Chapter 1 Introduction

1.1 Background

The field of interactive computer graphics has continued to experience enormous growth. Among techniques in this field, morphing (or metamorphosis) has been an area of active research in recent years. It has been widely used in many applications such as scientific visualization, education, entertainment and industrial product design. The original use of morphing techniques in the movie industry can be traced back to one century ago. Meliès discovered it by chance and used the cross-dissolving method in several movies he produced (see [BT97]). There have been many impressive examples of morphing in the entertainment industry in recent decades. In most cases, these visual effects were generated using 2D morphing techniques. As 2D representation of an object lacks spatial information, 2D morphing techniques cannot solve some problems such as handling changes in viewing and lighting parameters during a morph. On the contrary, morphing of 3D models directly changes geometry of objects and has attracted much research interest.

3D Morphing involves the creation of a smooth transition from a source object to a target object based on consideration of their geometrical forms such as positions and normals, and possibly other attributes such as colors and textures. For two 3D objects, there are an infinite number of ways to transform one to the other. Algorithms for morphing are mainly evaluated by criteria related to the ease of user control and the

aesthetic 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 good morphing system should enable a user to design a morph efficiently and effectively and the user control should be neither time-consuming nor labor-intensive. In terms of aesthetic quality, it is subjective to judge the visual appearance of morphing sequence. Gomes *et al.* [GDC99] listed some principles for evaluating the visual quality of morphs, such as topology preservation, feature preservation and rigidity preservation.

Generally speaking, a morphing process, whether in 2D or 3D, consists of two steps: establishing a *correspondence* to compute the association between the source object and the target object and calculating the *interpolation* between them to produce intermediate objects. Due to the popularity of polygon meshes in the field of interactive computer graphics, this thesis examines specifically the problem of 3D mesh morphing. Currently, research on 3D mesh morphing has reached a state where most computational problems in these two steps have been solved in general [A02]. However, a similar claim cannot be made for user control.

At the early stage, most research methods on 3D mesh morphing focused either on morphing of a restricted class of objects, or on automatically constructing the correspondence between two original objects. Users had little or no control over morphs. Recent morphing methods allow users to specify correspondences in morphing design. (See the survey papers [LV98, A02].) However, these methods have not paid much attention to issues about user interaction in morphing; they usually concentrate on the computational issues and overlook the interactive process of specifying and modifying user requirements. Thus, there was still no good scheme to

make user interaction intuitive, flexible and efficient. A user still faces a lot of difficulties in controlling a morph, as discussed below.

In the correspondence step, the most common way for users to specify correspondence is to assign vertex pairs. Because original meshes in morphing are often with dense triangulations, many previous algorithms started from a sparse set of user-specified vertex pairs and then used such a set to get the complete vertex correspondence between two meshes. Because users were confined to work with low-level mesh details such as vertices, they had no direct way to specify high-level requirements. For example, to pair a leg of a duck with a leg of a dinosaur, a user had to express such a requirement indirectly by specifying many pairs of mesh vertices. Such a way of specifying correspondence is usually neither intuitive nor convenient. Worse still, when two original meshes are quite different in shape, there will be no obvious, natural way for vertex correspondence. In such cases, user interaction usually becomes rather difficult and cumbersome. Moreover, previous morphing algorithms generally required users to complete a large amount of workload in order to enable the system's computation of morphing sequences; users themselves must be very careful not to specify contradictory requirements because the system cannot provide them any assistance in their specification.

In the interpolation step, users usually faced another difficulty of specifying morphing trajectories. They had to find groups of vertices and adjust morphing trajectories for individual vertices. Such kind of interpolation control needs proficient design skills. Furthermore, it is not suitable for specifying a requirement such as setting a new pose for a human-like object. Thus, users cannot control the interpolation conveniently.

The reason of the above difficulties in previous morphing methods may be the lack of an intensive examination on the interactive morphing design. From a user's point of view, one should be allowed to specify requirements of different levels. For example, in the top-down design approach, which is known as one of the popular design approaches, users can work from global-level conceptual design spaces (e.g. to specify changes in structure) to local-level technical design spaces (e.g. to modify mesh vertices). In addition, these methods often ignored the fact that morphing requirements of users are evolving and much trial and error is needed in the process of morphing design. They put aside the important process of user control and simply assumed that all user specifications were already ready for later computation. How to facilitate user interaction is one of the most challenging issues in 3D mesh morphing research.

1.2 Objectives

The morphing framework proposed in this thesis aims to empower users to conveniently and effectively control the whole morphing process.

Firstly, we realize that in addition to the steps of correspondence and interpolation, user interaction over those two steps is also a vital part in a morphing process. This framework seeks to facilitate user control by not only providing assistance during user interaction, but also exploring implied and potential user requirements through deduction. All assistance and deduction of the system honor user requirements and do not impose any restriction on users. Thus, users can choose to specify any number of requirements and the framework completes all other necessary computations to produce final morphs.

Secondly, being a kind of product design, interactive morphing design also includes both high-level conceptual design and low-level detailed design. This framework attempts to enable users to control the whole morphing process at multiple levels. It makes use of components of objects to support different kinds of user control, and thus users can interact with their morphing design in a natural and intuitive way.

Thirdly, this framework makes an effort to integrate skeletons as an intuitive and effective tool in morphing design. By operating on skeletons, not only can users specify vertex trajectories, but also manipulate objects by modifying their underlying skeletons. Thus, the users can control the interpolation step conveniently and effectively.

1.3 Organization and Contribution

The morphing framework proposed in this thesis is termed a component-based framework as it utilizes *components* of objects to enhance user interaction in morphing design. Consequently, morphing of polygon meshes is decomposed into morphing of components in the framework. Part of the work in this thesis is published in [ZOT03].

The reset of this thesis is organized in the following manner. We begin by reviewing related works in Chapter 2. In Chapter 3, a component-based object representation is introduced and then an overview of the component-based morphing framework is given. Chapter 4 discusses interactive correspondence control. First, users can directly specify global-level correspondences by pairing components. Using a proposed constraint tree, this framework provides users great flexibility and assistance in such specifications. Next, users are able to specify local-level correspondences by assigning and pairing local features; the framework deduces

implied local feature pairs and adds assumed ones where appropriate. The complete vertex correspondences are achieved through an automatic patch partitioning method. Chapter 5 describes interactive interpolation control. First, it discusses morphing between the skeleton of a source object and that of a target object. Next, a skeleton-guided interpolation method is proposed to transform vertices around their underlying bones. Thus, users can edit morphing trajectories at the global level by operating on skeletons as well as at the local level by operating on vertices. Chapter 6 reports the experimental results. Finally, Chapter 7 concludes the thesis and discusses the future work.

Recall that the framework proposed in this thesis aims to address issues about user interaction in mesh morphing. Specifically, main contributions of the framework are as follows:

- **Multi-level correspondence control**

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

- **Effective and flexible user control**

Because of the incorporation of multi-level user control, the framework can automatically deduce correspondences from one level to the other. Moreover, several techniques are presented to provide assistance and deduction in user interaction. At the global level, we utilize a novel constraint tree to process user specification, 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. In addition, it adds assumed correspondences where appropriate to improve morphs. In general, using these techniques, this framework frees users from the tedious workload of specifying detailed correspondences in morphing control. At the mean time, it respects all user specifications and never imposes any system restriction on users.

- **Multi-level interpolation control**

Users can choose to attach skeletons to original meshes. In such a case, morphing of polygon meshes can be abstracted into morphing of their underlying skeletons. Before specifying any correspondence over mesh details, users can predict final morphs from morphing of skeletons, and thus modify their specifications if necessary. This results in short turnaround time in experimenting with different morphing designs. Moreover, with a proposed skeleton-guided interpolation method, mesh vertices are transformed with the guidance of skeleton morphing. Thus, users can edit trajectories by operating on components as a whole and the framework can deduce trajectories of individual vertices accordingly. When local-level control is desired, users can also edit trajectories of individual vertices.

Before Proceedings with those technical details in this framework, the next chapter reviews related works.

Chapter 2 Related Work

Morphing of graphical objects has been investigated for more than a decade. Most early works studied morphing of 2D objects. In this chapter, 2D morphing approaches are first discussed in Section 2.1 as they are related and someway are possible to be extended to 3D morphing. Overview of different 3D morphing approaches is then provided in Section 2.2, where we focus our discussion on the efficiency of user control.

2.1 2D Morphing

The problem of constructing a smooth transformation from a 2D object to another has been extensively studied. Algorithms about 2D morphing can be classified into those for digital images and those for polygonal shapes. A survey can be found in [W98].

A digital image is represented as an array of pixel values. To obtain semantic correspondences in a morph, an image morphing system often requires its user to identify and pair features at a set of pixels of two original images. Beier and Neely [BN92] presented a feature-based method based upon fields of influence surrounding user-specified features. Lee *et al.* [LCS95] applied a computer vision technique called snakes to reduce user workload in feature specification.

For two polygonal shapes, Sederberg *et al.* [SGW93] tried to avoid the shrinkage

or kink effects, which normally occur in the linear interpolation, by interpolating edge lengths and angles between edges rather than vertex positions. Goldstein and Gotsman [GG95] utilized multi-resolution techniques to effectively capture geometric properties for establishing feature correspondence. Surazhsky *et al.* [SSB01] morphed two polygons by constructing a *xy*-monotone surface whose cross-sections at two ends represented two given polygons respectively. Shapira and Rappoport [SR95] presented a star-skeleton method for polygon morphing. First, they decomposed two polygons into the same number of star pieces and constructed a connecting skeleton for each polygon. Then, the interpolation between skeletons was calculated and then star pieces were unfolded from the skeletons. This is the 2D work closest to our proposed framework in partitioning complex objects into simpler forms for morphing. However, the extension from 2D to 3D is not trivial due to the complexity of mesh connectivity. In addition, unlike their work, our method does not have the requirement that objects must be compatibly decomposed. Thus, a user can design a morph more conveniently and flexibly.

When dealing with 3D objects, 2D morphing algorithms do not suffer from the complexity of 3D objects. 2D images are generated from those objects and these algorithms can then produce intermediate images. However, 2D morphing cannot handle changes of viewpoints or lighting parameters. Besides, users lose the flexibility of editing 3D objects represented by intermediate frames.

2.2 3D Morphing

Morphing of 3D objects has its own characteristics. Yet, several 2D morphing approaches can still be extended to 3D morphing research. For example, digital image morphing algorithms can be directly extended to morphing of voxel-based objects.

According to the way of object representation, 3D morphing algorithms are generally classified into two categories: volume-based morphing and boundary-based morphing, as proposed in the survey paper [LV98]. It is also noted that there exist some other kinds of 3D morphing methods. For example, morphing of image-based 3D objects can be done by transforming their light fields [ZWG02].

2.2.1 3D Volume-based Morphing

Morphing algorithms in this category describe 3D objects as volumetric models. Generally, there are voxel-based approaches that sample the 3D space on regular grids, and implicit surface approaches that work on implicit functions.

Voxel-based morphing works represent a 3D object as a set of voxels. Hughes [H92] proposed a method that worked in the Fourier domain and treated individual frequency bands with different functions of time. Lerios *et al.* [LGL95] extended the 2D morphing work of Beier and Neely [BN92] by using fields of influence of 3D primitives to warp volumes.

Implicit surface morphing works focus on morphing of 3D objects represented as implicit functions. Kaul and Rossignac [KR91] provided an interpolation algorithm based on Minkowski Sums. He *et al.* [HWK94] decomposed distance functions with a wavelet transform. Wyvill *et al.* [WGG98] presented a technique for morphing implicit surfaces built from convex skeletal elements, also known as blobs or soft objects. Galin and his coworkers [GA96a, GA96b, GL99] addressed the problem of soft object morphing by interpolating skeletal elements with Minkowski Sums and then extended such interpolation to whole objects. Breen and Whitaker [BW01] employed a deformable surface to smoothly transform the implicit surface model of a source shape

to that of a target shape. After converting the deformable surface into a volume data set, a set of procedures were applied to transform voxels to create a sequence of volumes. Blanding *et al.* [BTS00] computed trimmed skeletons from the symmetric difference between two original solid models and then utilized them as intermediate shapes. Such shape generation procedure can be recursively applied to produce a sequence of shapes in a final morph.

Volume-based morphing works have no restriction on the topological structures of original objects because they are not burdened with surface parameterization. The ultimate advantage of volumetric methods is that they support changing of genus well, for example, transforming a sphere into a donut. On the other hand, algorithms in this category have several limitations. As intermediate shapes are represented as volumes, extracting them to boundary-based models may produce topologically complex objects. In addition, it seems that it is not simple and intuitive for a user to identify vertices, edges, faces or contours of original objects as features in a user interface. In particular, since grids in voxel-based approaches are three dimensional, the memory and computation costs can be prohibitive.

Our proposed framework has some similarities to Galin's work [GA96a, GA96b, GL99]. In both, for example, users can directly pair components of two original objects. However, there exist significant differences between their volume-based and our boundary-based morphing methods in terms of object and skeleton representations. For example, in volumetric methods, the problem of binding surfaces to skeletons can be easily solved as their objects are defined to be constructed from skeletons, while in mesh morphing, this is complicated due to the different mesh connectivity of two original meshes. The skeleton-based algorithm in [BTS00] allowed its user to align

global-level features such as extruding parts. In contrast, a user of our framework can conveniently specify correspondences at both the global and the local levels.

2.2.2 3D Boundary-based Morphing

Algorithms falling into this category focus on morphing of objects represented by their boundaries. The most popular boundary-based object representation is polygonal meshes. Morphing approaches for 2D polygonal shapes can be extended to morphing of 3D polygonal meshes to some extent. 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.

2.2.2.1 Correspondence Approaches

Most of research works in mesh morphing focus on the problem of establishing vertex correspondences and many early works focused on automatically establishing vertex correspondences for original meshes (see the survey papers [A01a, A02]). As user control is essential in morphing of general meshes, recent works usually allow their users to assigns sparse sets of feature vertex pairs and the key problem in these works is to effectively extend such sparse sets to the whole meshes.

Patch-Partitioning Approach

Given two original meshes, this approach partitions each mesh into a collection of patches based on user-specified feature vertex pairs. Patch layouts of the two meshes must be compatible such that patches can be paired and morphed one by one to form the overall mesh morphing. How to establish vertex correspondence over the whole meshes based on user specification is a key problem that must be solved in this

approach. Parent [P92] presented a recursive method to build a common mesh subdivision. In this work, correspondences between mesh vertices were automatically established by using several sheets to cover two original meshes. Some degree of user control was also supported at the step of sheet subdivision. DeCarlo and Gallier [DG96] allowed users to divide mesh surfaces into triangular and quadrilateral patches for morphing with genus change. Bao and Peng [BP98] constructed feature polyhedra based on user-specified patch partitions and established correspondence between patches by using a cluster scheme. Gregory *et al.* [GSL98] presented a feature-based method where users were asked to specify feature nets on original meshes. Meshes were then partitioned into patches according to the nets. This method also supported user control over vertex trajectories by representing them as Bezier curves. Kanai *et al.* [KSK00] used harmonic mapping in morphing of arbitrary triangular meshes. Based on user-specified feature vertex pairs and the connectivity among those vertices at original meshes, they constructed a common control mesh which was then used to define compatible patch partitions of the meshes. Zöckler *et al.* [ZSH00] improved user interaction in morphing by allowing users to specify corresponding regions as well as corresponding points. Being able to specify feature vertices inside individual patches, users need not partition original meshes into dense patches. Recently, Shlafman *et al.* [STK02] proposed a method that can automatically partition a mesh into several patches. However, their resulting patch partitions for two original meshes were not guaranteed to be compatible and thus cannot be used for general morphing.

When dealing with high-genus meshes, users of this approach are required to specify proper patch partitions, of which each patch is homeomorphic to a disk. Although being always possible, such kind of specifications demands proficiency and skill of users. Generally speaking, a user of this approach must specify a feature net for

each original mesh by first assigning feature vertices and then identifying connectivity among those vertices. Moreover, the user still has to consider how to create compatible partitions by specifying enough vertex pairs, without any direct help from the morphing system. So user interaction of controlling the correspondence step is usually difficult and cumbersome.

Global Topological Merging Approach

Algorithms in this approach establish vertex correspondences by using global topological merging techniques. Kent *et al.* [KPC91, KCP92] automatically morphed genus-0 polyhedra by projecting each of them onto a sphere. Kanai *et al.* [KSK98] made use of harmonic mapping to establish vertex correspondence for meshes with boundaries. Users can specify vertex correspondences at mesh boundaries and the system took the main responsibility to associate vertex pairs. Alexa [A00] allowed users to specify scattered features and then aligned them by using a spherical mapping and warping method. This work enhanced user interaction in that users were freed from the workload of specifying connectivity among specified feature vertices. However, because this work did not examine the shapes of original meshes and perform deduction of correspondences accordingly, users may need to spend much time on locally adjusting vertex correspondences.

Multi-resolution Approach

Multi-resolution techniques tackle the problem of convenient manipulation of meshes with complex structure and tremendous size. In recent years, several algorithms were proposed to employ it for the purpose of morphing.

In the work of Lee *et al.* [LDS99], user-specified feature pairs were retained

during the process of multi-resolution parameterization. With all user-specified feature pairs aligned, this method merged the mesh connectivity of the source mesh and that of the target mesh by using their coarse models. When system-established vertex correspondences were unsatisfactory, a user can also perform local adjustment on coarse models.

Alternatively, some other methods made use of the multi-resolution remeshing technique to solve the problem of connectivity difference between original meshes. Michikawa *et al.* [MKF01] represented each 3D object as a series of semi-regular meshes, which were organized in a hierarchical way. Then they applied re-meshing to convert original meshes into common mesh connectivity. Praun *et al.* [PSS01] presented a consistent mesh parameterization algorithm that established parameterization for a set of meshes sharing a base domain. Users of both these methods were required to specify corresponding vertices in original meshes for each vertex in the base domain.

Shape Dissection Approach

Algorithms in this approach establish vertex correspondences by dissecting original meshes into tetrahedra. In the work of Shapiro and Tal [ST98], during a process called tetrahedralization or realization, two original meshes were transformed into convex polyhedra. By merging the realized polyhedra of two original meshes, an isomorphic vertex neighborhood graph was obtained and the complete vertex correspondence was then established accordingly. Alexa *et al.* [ACL00] blended the interiors of original meshes as well as their boundaries by dissecting the meshes into isomorphic complexes (triangles in 2D and tetrahedra in 3D). Vertex correspondence in this approach is highly dependent on the process of shape dissection and this may

result in difficulties in user control, especially for complex 3D objects.

Shape Re-sampling Approach

An ultimate difficulty in the correspondence process is the connectivity difference of two original meshes. This approach obtains common mesh connectivity by re-sampling two original meshes at the same sampling rate.

A mesh can be represented as a collection of 2D cross-sections. In that case, vertex correspondences can then be established within each pair of corresponding cross-sections. Chen and Parent [CP89] proposed an algorithm for morphing 3D objects represented by planar contours. They first morphed corresponding contours and then constructed cylindrical volume using intermediate contours. Korfiatis and Paker [KP97] allowed users to specify vertex pairs for cross-section pairs and established vertex correspondences accordingly. In the work of Chang *et al.* [CLK98], a generalized cylinder was interpreted as the sweep surface of a planar cross-sectional B-spline under B-spline motion. By editing cross-sections, users can then conveniently deform 3D objects.

Lazarus and Verroust [LV97] studied morphing of star-shaped meshes around their underlying skeletal curves. Two original meshes were re-sampled and reconstructed as cylindrical meshes. A user can control a morph by manipulating the skeletal curves of meshes. By using different re-sampling rates, this method also provided different levels of user control. This method only dealt with cylinder-like meshes and their underlying skeletons were only single 3D curves.

User control in this approach is simple and intuitive. However, this approach establishes vertex correspondences at the stage of re-sampling, and thus users cannot

control the correspondence step flexibly by specifying desired vertex pairs.

Shape Space Approach

This approach treats each original mesh in a morph as a base shape and the dimension of the shape space depends on the number of base shapes. Edelsbrunner [E99] first mentioned the term "shape space". It was then used by Cheng *et al.* [CEF01] for canonical deformation among a set of shapes each of which is represented by a simplicial complex and a smooth surface. In this work, a complete matching was performed to avoid the difficulty of determining feature correspondences — such correspondences were automatically established by removing those redundant pairs in the matching. Alexa and Müller [AM99] extended this term to "morphing space". They declared that morphing could be used to describe objects as a composite of other objects. They further discussed synthesizing and analyzing of objects in a morphing space. In this work, the efficiency of user control depended on the employed morphing technique.

This approach is attractive in that it allows morphing among multiple shapes and supports a broad range of shape manipulation mechanisms such as shape searching. However, it does not provide specific solutions for enhancing user control in morphing.

Parametric Space Approach

This approach represents 3D meshes in 2D parametric spaces and makes use of 2D image morphing techniques to morph between their 2D representations. Intermediate 3D objects are reconstructed based on the resulting 2D morphs. Chen *et al.* [CSB95] utilized cylindrical projections to parameterize certain types of objects,

and warped 2D parametric spaces according to user-specified feature pairs. Ramasubramanian and Mittal [RM99] extended it to support general topologies. They represented each 3D object as multiple 2D images and then performed interpolation for corresponding images. Algorithms in this approach support user specification of correspondences and avoid the complexity of morphing in 3D spaces. However, as there is no intrinsic projection method in the planar representation for general 3D objects, this approach still cannot be applied to arbitrary complex meshes.

2.2.2.2 Interpolation Approaches

Linear interpolation is frequently used in morphing due to its simplicity. However, it is well known that this method causes problems such as self-intersection and shape degeneration. Delingette *et al.* [DWS93] presented a physical-based algorithm that represented 3D surfaces as simplex meshes. Morphing between two simplex meshes was obtained by first transforming one mesh connectivity to the other using some defined mesh operators, and then expressing their geometry as shape parameters that were well adapted to simplex meshes. In the work of Sun *et al.* [SWC97], based on assumed correspondences between two isomorphic meshes, some intrinsic geometric parameters and a propagation paradigm were utilized to interpolate vertex positions. Gregory *et al.* [GSL98] made use of weight factors in the interpolation process. They allowed users to interactively define the trajectories for some mesh vertices, and the transformation of those vertices was then propagated to the meshes. Alexa *et al.* [ACL00] presented an interpolation method to transform both boundaries and interiors of original meshes in a morph. This method worked well in 2D cases and provided some simple 3D examples.

In terms of interpolation control in mesh morphing, the most common way is to

set trajectories for mesh vertices. However, because there is no intrinsic association among trajectories of different vertices, designing trajectories for a sample of vertices seems to be inconvenient and thus inadequate for interactive interpolation control [LV98, A02]. Although some improvements have been proposed by using physical simulation [DWS93], intrinsic parameters [SWC97] or weights factor [GSL98], users still cannot control the transformation of 3D shapes directly at a high level. For example, it is difficult to set a new pose for a human-like object as setting weights or finding intrinsic parameters to reflect the desired position is hard and tricky. Hence, an intuitive and convenient way of specifying such a user requirement is still needed.

From the literature review in this chapter, it can be seen that although user interaction in morphing is essential for producing aesthetic morphs, it was still far from convenience and efficiency at both the correspondence and the interpolation steps. The following chapters introduce a framework that employs components to empower users to control the whole morphing process with ease.

Chapter 3 Component-based Morphing Framework

This chapter introduces the proposed interactive framework for component-based morphing. We first introduce polygon meshes and component decomposition in Section 3.1. Section 3.2 discusses the component-based object representation where a polygon mesh is composed of a collection of components. Section 3.3 reviews the framework by briefly introducing its main steps. Using components, this framework enables efficient and effective user interaction at both the global and the local levels.

3.1 Meshes and Components

3.1.1 Polygon Mesh

Polygon mesh representation is the most common method for representing 3D objects. In particular, advances in 3D scanning and acquisition technology have made it a popular representation. In this boundary-based representation method, a polygon mesh comprises a collection of vertices, edges and convex polygons to form the whole 3D shape.

In this framework, we deal with orientable, manifold and topological equivalent polygon meshes. A *mesh* refers to a piecewise linear surface which is made up of a set of polygons. Each polygon of a mesh is triangular in this framework. Otherwise, the mesh is triangulated by the system. The *topology* of a mesh refers to its vertex/edge/triangle connectivity and its *geometry* refers to a set of world coordinates for all its vertices. *A boundary of a mesh* is a closed loop of mesh edges, each of which

has only one incident triangle. Two meshes are said to be *topological equivalent* or *homeomorphic* if one mesh can be continuously deformed into the other. In an *orientable* mesh, vertices of all its triangles are organized in the same order (counterclockwise in this framework). A *manifold* mesh has the property that the neighborhood of every vertex of the mesh is homeomorphic to a disc or a half disc.

The mesh representation has many advantages. All surfaces in a mesh are described by linear equations with low computational complexity. Therefore, polygons are independent entities and can be treated as such by object manipulation and transformation algorithms. In addition, various algorithms for hiding surface and shading are easily implemented on polygons since they have well-defined orientations.

However, dense polygonization, arbitrary topology and irregular connectivity of polygon meshes make it difficult to manipulate them efficiently. Because there is no high-level information explicitly defined on a polygon mesh, previous morphing methods asked their users to represent global-level user requirements indirectly through operations over mesh vertices. Such kind of user control obviously conflicts with the normal working habit of an artist.

3.1.2 Component Decomposition

To impose high-level information of a mesh, one way is to decompose the mesh into a collection of primitives, each of which comprises a group of polygons. Such a primitive is termed a *component*. Obviously a component is simpler in shape and smaller in size than the whole mesh.

Component organization is an important technique in the modeling community. An artist usually constructs a complicated model by composing several simple

components. For example, there are techniques in modeling volumetric models such as soft objects [MLP01] and CAD (computer aided design) solid models. In CAD techniques, components of a model are organized according to a CSG (constructed solid geometry) tree. The component decomposition of a complex 3D object suits the work habit of an artist and eases the modeling task as well. In addition, component-based approaches enable processing of individual components, without requiring a complete new calculation over the whole object. For the above reasons, objects with components have been widely used in many applications such as animation and virtual simulation.

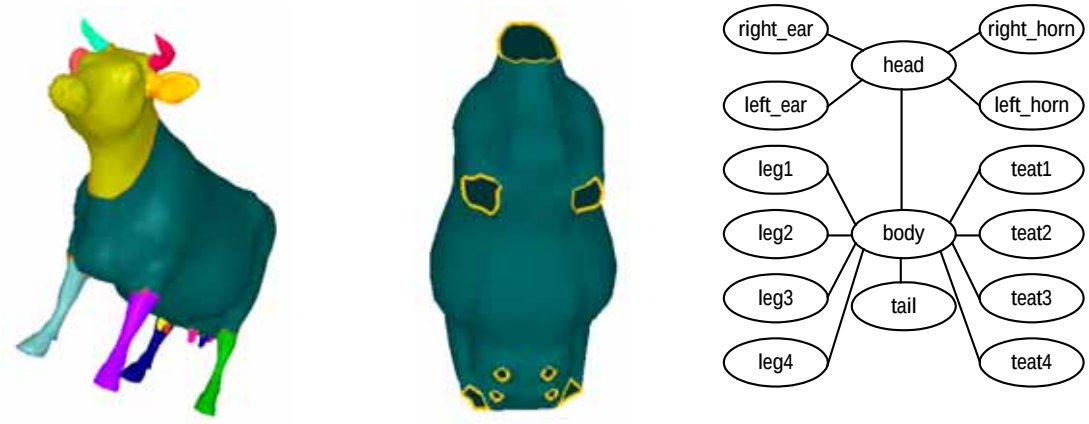
In the field of computer graphics, however, many applications including mesh morphing usually deal with meshes having no components inside. In such applications, decomposing a mesh into components can definitely empower a user to manage its data more effectively. Despite that there have been many publications about mesh decomposition [FS92, GSL98, MW99, LWT01, STK02], results of decomposition have not been utilized in mesh morphing for the purpose of enhancing high-level user control. To some extent, morphing methods using patch partitions of meshes [P92, DG96, BP98, GSL98, KSK00, STK02] can be considered to be similar to methods using components. However, these patch-based methods decompose mesh surfaces on the basis of user-specified vertex correspondences, instead of utilizing decompositions to support interactive morphing control.

The proposed framework attempts to capitalize on the use of components to facilitate user interaction in morphing design. This framework does not require the component decomposition of two original meshes are compatible in the sense of having the same number of components and the same connectivity among the

components. As such, users can freely decompose meshes based on their morphing requirements.

3.2 Component-based Object Representation

A mesh $\boldsymbol{M}$ can be decomposed into a collection of components. Thus, we have $\boldsymbol{M} = \bigcup \boldsymbol{C_i}$, where $\boldsymbol{C_i}$ is a component of $\boldsymbol{M}$ and i = 1, 2, …, n. Users can create or modify mesh decompositions according to their requirements in morphing design. 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. Figure 3.1(a) shows the component decomposition of a cow model. This model contains a set of components such as its head, body, ears and tail. Using such components, a user can easily specify high-level morphing requirements in morphing process.



(a) Decomposition **(b) Component** **(c) Connectivity Graph**

Figure 3.1 Component representation of a cow model

A component connects some other components in the mesh, each of which is defined to be an *adjacent component* of the component. Two *connected components* connect each other at some common mesh edges and vertices, which are termed a *boundary of the components*. A component contains a list of boundaries, each of which

represents a *connection* with an adjacent component. Figure 3.1(b) shows the body component of a cow model and highlights its boundaries by coloring them in yellow.

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 each connection between two components as an edge connecting the two nodes of the components. This abstract graph effectively encapsulates structural information of the mesh. Note that because each mesh edge has only two incident triangles, a connection is only between two components and an edge of a connectivity graph connects only two nodes. Figure 3.1(c) shows the connectivity graph for the cow model. To achieve the component decomposition, users can directly make use of pre-defined components in meshes, such as from groups in OBJ or VRML files. Alternatively, automatic methods [MW99, LWT01, STK02] can be used to compute initial decomposition for users' further modification. In addition, the framework provides several interactive tools to assist users in specifying components, as discussed in the Appendix.

3.3 Framework Overview

Given two homeomorphic polygon meshes, a user of the component-based morphing framework can design a smooth, desirable transition from one mesh to the other with ease. Instead of simply starting from given user inputs, as in other works, this framework empowers its users to interactively specify and modify their requirements during the whole morphing process. Specifically, in this interactive framework, users are able to perform their morphing design in a flexible way—at any step in the overall process, they can (1) choose to specify any number, inclusive of none, of requirements and (2) re-visit any previous step to modify their specifications. Figure 3.2 shows main steps in the propose framework. For clarity, these steps are

connected according to a typical workflow in this figure. The whole morphing process consists of user interaction and system calculation, which are shown in different colors.

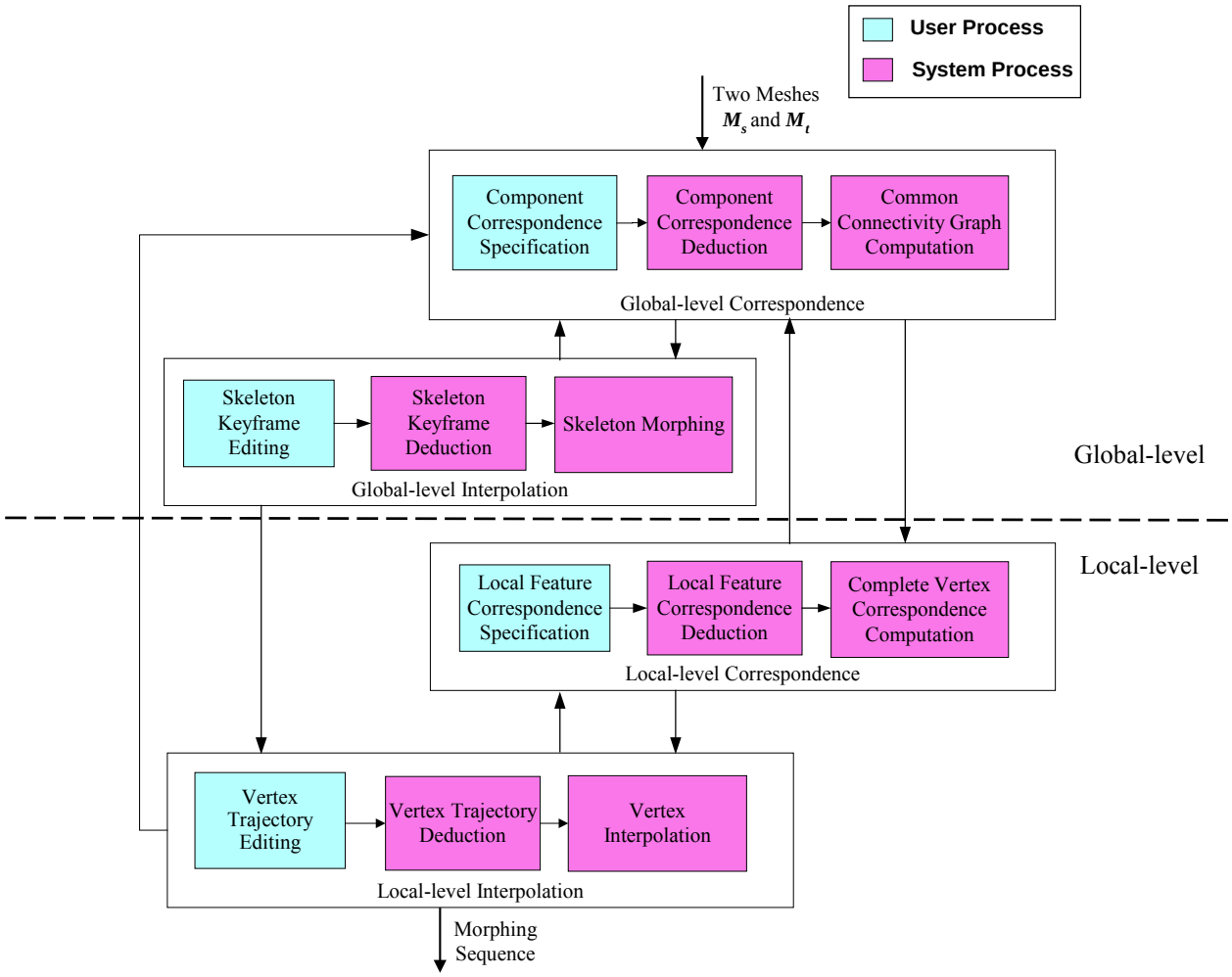


Figure 3.2 A typical workflow in the framework

It can be seen from this figure that a morphing process in this framework comprises a global-level process (including global-level correspondence and global-level interpolation) and a local-level process (including local-level correspondence and local-level interpolation). Thus, a user can conveniently specify morphing requirements at either level. In addition, he only needs to specify those requirements of interest, and the system can complete the remaining work through deduction.

Global-level Correspondence: Given a source mesh M_s and a target mesh M_t, a user can decompose each of them into a collection of components and specify global-

level correspondences over their components. By utilizing a proposed constraint tree, this framework provides several kinds of assistance in user specification in this step: When a user wants to specify correspondence for selected components, it is able to highlight possible counterparts; after the user's modification to component decomposition or component correspondences, it is able to maintain other unaffected user specifications. Moreover, the framework deduces implied correspondences in the process of user specification; at the end of this step, it works out the common connectivity graph which associates individual components of one mesh with those of the other.

Global-level Interpolation: For each original mesh, the user can choose to attach an underlying bone to each component and all these bones form the skeleton of the mesh. In such a case, before referring to mesh details, the user is able to get a draft version of the final morph from the morph between $\boldsymbol{T_s}$ to $\boldsymbol{T_t}$, where $\boldsymbol{T_s}$ is the skeleton of $\boldsymbol{M_s}$ and $\boldsymbol{T_t}$ is the skeleton of $\boldsymbol{M_t}$. Due to the structural difference between $\boldsymbol{T_s}$ and $\boldsymbol{T_t}$, a common skeleton of $\boldsymbol{T_s}$ and $\boldsymbol{T_t}$ is calculated from the common connectivity graph and utilized to morph $\boldsymbol{T_s}$ to $\boldsymbol{T_t}$. Skeleton morphing serves as an indication of the final morph and can be obtained soon after user specification of global-level correspondences. Thus, the user can modify those specifications at an early stage, and the turnaround time in the global-level morphing process is very short. In addition, the user can add/modify keyframes of the skeleton morphing to set global-level component trajectories.

Local-level Correspondence: Within a component pair containing one component from $\boldsymbol{M_s}$ and one from $\boldsymbol{M_t}$, the user can specify and pair several kinds of local features, including feature vertices, feature lines and feature loops. The

framework deduces implied local features according to user specifications and adds assumed ones where appropriate. To establish the complete vertex correspondence for the component pair, an automatic patch partitioning method is then applied to partition the components into compatible patch layouts. For the case where one component of a mesh has no counterpart in the other mesh, the framework applies an automatic method to establish vertex correspondence.

Local-level Interpolation: Besides the linear vertex interpolation method, the user can choose to employ our skeleton-guided interpolation method in this step. Using the latter method, mesh vertices follow the movement of their underlying bones and conform to user-specified feature correspondences at the same time. Therefore, not only can the user modify trajectories of individual vertices at the local level, but also manipulate trajectories of components as a whole at the global level. From user-specified component trajectories, the framework can deduce vertex trajectories accordingly.

In summary, the component-based morphing framework utilizes components to make user control in the whole morphing process easy. Given two meshes, a user can intuitively specify correspondences either at the global level by pairing components, or at the local level by pairing local features. In addition, the user can also control the interpolation process at both levels. The following chapters provide more details about user interaction in main steps of the framework.

Chapter 4 Component-based Correspondence Control

At an early stage of a morphing design, a user mainly concerns the structures of two original meshes. In the component-based morphing framework, users can specify such global-level correspondences by paring components. Section 4.1 discusses user control in the step of global-level correspondence. A proposed constraint tree is utilized to effectively organize user specifications, provide candidate counterparts for user-selected components and maintain user specifications upon the user's modification. The *common connectivity graph*, which associates individual components of one mesh with individual components of the other mesh, is finally constructed. The user is also allowed to fine-tune a morph at the local level by specifying local feature pairs within component pairs. Section 4.2 discusses user control in the step of local-level correspondence. The framework is able to deduce implied local feature pairs and add assumed local feature pairs according to user-specifications. Then the *complete vertex correspondence*, which associates individual vertices of one component with individual vertices of its corresponding component, is established through an automatic patch partitioning method.

4.1 Global-level Correspondence

Global-level correspondence refers to correspondence over components. It is a convenient way for a user to specify a requirement over the structures of original meshes. First, we introduce and analyze several user requirements in Section 4.1.1.

Then, several terms are defined in Section 4.1.2. We discuss the processing of user-specified global-level correspondence in Section 4.1.3. After that, a proposed constraint tree is introduced in Section 4.1.4. By using the constraint tree, the framework provides assistance in user interaction and effectively deduces global-level correspondences according to user input, as discussed in Section 4.1.5 and Section 4.1.6 respectively.

4.1.1 Requirement Analysis

By pairing components of the source mesh $\boldsymbol{M_s}$ and components of the target mesh $\boldsymbol{M_t}$, a user can intuitively indicate requirements of global-level correspondences. The framework seeks to provide as much assistance to the user as possible, without compromising user freedom in specification. Specifically, the following issues for the step of global-level correspondence are addressed in the framework.

First, the user is allowed not only to specify correspondence between one component from $\boldsymbol{M_s}$ and one from $\boldsymbol{M_t}$, but also to specify correspondence between groups of components in one step. Using component groups, the user can start with vague requirements and then iteratively refine the requirements. This is especially useful at an early stage of morphing design. To effectively encapsulate and maintain user-specified correspondences over component groups, the framework proposes a constraint tree, which implicitly records all possible component pairs and keeps track of the evolution of user specifications: Section 4.1.3 introduces the method for recording user-specified correspondences and Section 4.1.4 further organizes all recorded correspondences to keep the history of user specifications. Thus, the user is provided with great flexibility, from undoing any specification to modifying component decompositions.

Next, when the user selects a group of components, the framework assists user specification of component correspondences by identifying the probable counterparts. Such counterparts are those components having similar connectivity to the components in the selected group. We note that the method of analyzing the connectivity at every individual component is not feasible when correspondences between groups of components are enabled. Section 4.1.5 introduces the measure of similarity in connectivity between groups of components and the process of identifying probable counterparts for selected components.

Finally, the user can choose to specify any number of component correspondences, and the framework then automatically works out the complete component correspondence with all user-specified correspondences respected. Achieving this, the framework allows a range of automations: from totally manual (the user specifies detailed correspondence for every component) to semi-automated (the user specifies only important correspondences and the framework computes the others) to fully-automated (the framework computes all correspondences without any user input). Details about this are provided in Section 4.1.6.

Some definitions are first introduced in Section 4.1.2.

4.1.2 Terminology

As defined in Section 3.2, the connectivity among the components of a mesh can be represented as a connectivity graph. In such a graph, each component is represented as a node, and each connection between two components as an edge connecting the two nodes of the components. The connectivity graph of $\boldsymbol{M_s}$ and that of $\boldsymbol{M_t}$ are represented as the two graphs $\boldsymbol{G_s} = G(V_s, E_s)$ and $\boldsymbol{G_t} = G(V_t, E_t)$ respectively, where V_s

and V_t are sets of nodes representing components and E_s and E_t are sets of edges representing connections. An example of $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$ is shown in Figure 4.1, where $V_s = \{a,b,c,d,e,f\}$, $E_s = \{1,2,3,4,5\}$ and $V_t = \{p,q,r,s,t,u\}$, $E_t = \{6,7,8,9,10\}$. In this figure and subsequent figures in this chapter, connections of connectivity graphs are shown explicitly as white nodes for ease of illustration. Note that when a user modifies the component decomposition of a mesh, for example, by merging two connected components, its connectivity graph is changed and updated automatically by the framework.

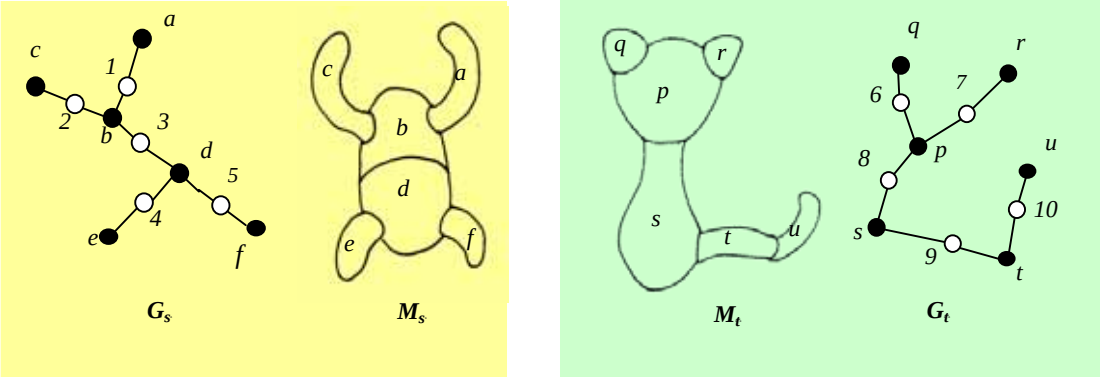


Figure 4.1 The source connectivity graph G_s and the target connectivity graph G_t

A user can specify a global-level correspondence by associating a group of components of one mesh with a group of components of the other mesh. A correspondence between two groups of components is denoted by $\langle X, Y \rangle$ where $X \subseteq V_s$ and $Y \subseteq V_t$. A *constraint* is defined to be a user-specified correspondence between one component group of $\boldsymbol{G_s}$ and one component group of $\boldsymbol{G_t}$.

In the global-level correspondence process, we need to establish correspondences over individual components and over individual connections. The *complete component correspondence* is defined to be a set of pairs of corresponding components, in which each component of $\boldsymbol{G_s}$ (or $\boldsymbol{G_t}$) either is paired with one component of $\boldsymbol{G_t}$ (or $\boldsymbol{G_s}$) or has no counterpart. Similarly, the *complete connection correspondence* can be defined as a

set of pairs of corresponding connections. Two corresponding components form a *component pair* and two corresponding connections form a *connection pair*.

The final product of the global-level correspondence step is a common connectivity graph $\boldsymbol{G_{st}} = G(V_{st}, E_{st})$, which is defined as a graph encapsulating both the complete component correspondence and the complete connection correspondence. V_{st} is the set of *correspondence nodes* each of which represents a component pair and E_{st} is the set of *correspondence edges* each of which represents a connection pair. A *null-component* ζ_V is defined to be an abstract component in the common connectivity graph and is designated as the counterpart of a component having no counterpart. Similarly, a *null-connection* ζ_E is defined to be an abstract connection in the common connectivity graph and is designated as the counterpart of a connection having no counterpart. Therefore, A correspondence node in $\boldsymbol{G_{st}}$ has one of these forms: (c_s, c_t), (c_s, ζ_V) or (ζ_V, c_t) where $c_s \in V_s$, $c_t \in V_t$, and every c_s or c_t appears in exactly one correspondence node of $\boldsymbol{G_{st}}$. Similarly, a correspondence edge has one of these forms: (e_s, f), (e_s, ζ_E) or (ζ_E, e_t) where $e_s \in E_s$, $e_t \in E_t$ and every e_s or e_t appears in exactly one correspondence edge of $\boldsymbol{G_{st}}$. For the two connectivity graphs in Figure 4.1, a possible $\boldsymbol{G_{st}}$ constructed is as shown in Figure 4.2.

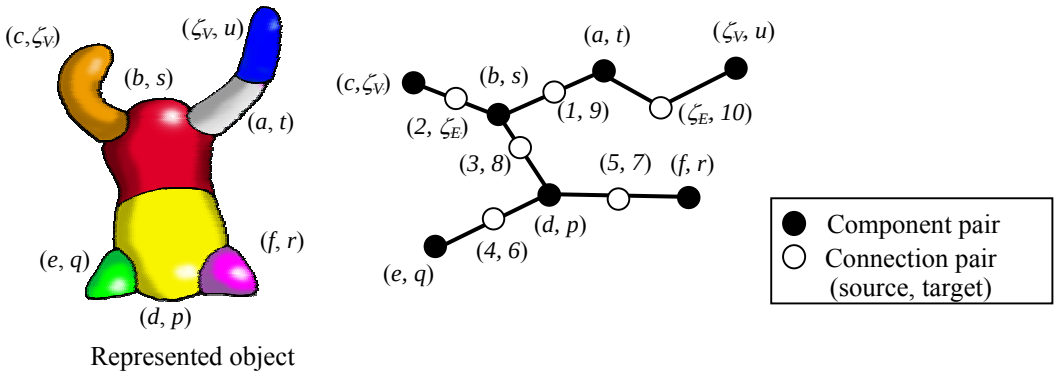


Figure 4.2 Common connectivity graph G_{st}

4.1.3 Correspondence between component groups

To effectively capture all constraints without imposing any system-caused restriction, we need to keep track of all possible correspondences over components and all possible correspondences over connections. Unfortunately, when correspondence specification over groups of components is enabled, 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.

4.1.3.1 Permissibility and Completeness

A *permissible component pair* is defined to be a component pair that possibly appears in $\boldsymbol{G_{st}}$. The set of all permissible component pairs is denoted by R_V. A *permissible connection pair* is defined to be a connection pair that possibly appears in $\boldsymbol{G_{st}}$. The set of all permissible connection pairs is denoted by R_E. Therefore, we can see that to be permissible, a component/connection pair must not contradict with any constraint. Note that $\{(c_s, \zeta_V) \mid c_s \in V_s\}$ and $\{(\zeta_V, c_t) \mid c_t \in V_t\}$ are always subsets of R_V, as it is always possible that a component in a mesh has no counterpart in the other mesh. A correspondence over components $\langle X, Y \rangle$, where $X \subseteq V_s$ and $Y \subseteq V_t$, is defined to be *complete* if and only if $\forall x \in X$ and $\forall y \in Y$, $(x, y) \in R_V$. Similarly, a correspondence over connections $\langle E, F \rangle$ is defined to be *complete* if and only if $\forall e \in E$ and $\forall f \in F$, $(e, f) \in R_E$. From a constraint $\langle X, Y \rangle$, we can deduce that (x, y) is a permissible component pair while $(x, \overline{y})$ or $(\overline{x}, y)$ is not, where $x \in X$, $y \in Y$, $\overline{x} \in V_s - X$ and $\overline{y} \in V_t - Y$.

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 edges in E

are incident to. Therefore, $\boldsymbol{P}$ may not be a usual graph and likewise for $\boldsymbol{Q}$. They are represented as graphs here for the convenience of description. Then, completeness of correspondences can be defined as follow. A global-level correspondence $\langle \boldsymbol{P}, \boldsymbol{Q} \rangle$, where $\boldsymbol{P}= G(X, E)$ and $\boldsymbol{Q}= G(Y, F)$, is said to be *complete* if and only if $\langle X, Y \rangle$ is complete and $\langle E, F \rangle$ is complete.

Throughout the process of specifying global-level correspondences, all current correspondences for $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$ are encapsulated in a *correspondence set*, which is defined as $\mathbb{C} = \left\{ \left\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \right\rangle \middle| i = 1, 2, ..., n \right\}$, where $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ is complete and $\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_n$ is a partition of $\boldsymbol{G_s}$ and $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_n$ is a partition of $\boldsymbol{G_t}$.

Because all constraints are honored in the correspondence set, for $\boldsymbol{P}_i = G(X_i, E_i)$ and $\boldsymbol{Q}_i = G(Y_i, F_i)$, we have R_V and R_E implicitly recorded as:

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

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

Thus, a component pair is permissible if both its components can be found within a $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle \in \mathbb{C}$ for some i. Similarly a connection pair is permissible if both its connections can be found within a $\langle \boldsymbol{P}_j, \boldsymbol{Q}_j \rangle \in \mathbb{C}$ for some j.

4.1.3.2 Constraint Processing

In the process of specifying global-level correspondence, it is more intuitive for a user to pair components than to pair connections. Therefore, a constraint in this framework is a correspondence over components $\langle X, Y \rangle$, where $X \subseteq V_s$, $Y \subseteq V_t$. (The user can also specify a correspondence between connections by pairing two boundaries. This is a kind of local-level correspondence control and will be addressed

in Section 4.2.) Given a constraint $\langle X, Y\rangle$, we can introduce another correspondence $\langle E, F\rangle$, where E denotes all connections that are incident to components in X, and F denotes all connections that are incident to components in Y. Both correspondences are jointly represented as $\langle \boldsymbol{P}, \boldsymbol{Q}\rangle$ where $\boldsymbol{P}=G(X, E)$ and $\boldsymbol{Q}=G(Y, F)$. Note that for a given constraint, its corresponding $\langle \boldsymbol{P}, \boldsymbol{Q}\rangle$ may not be complete.

Initially we have $\mathbb{C} = \{ \langle \boldsymbol{G_s}, \boldsymbol{G_t}\rangle \}$ where every component (connection, respectively) of $\boldsymbol{G_s}$ can be possibly paired with every component (connection, respectively) of $\boldsymbol{G_t}$. 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 $\boldsymbol{Q}_i'' = \boldsymbol{Q}_i - \boldsymbol{Q}_i'$. If either $\langle \boldsymbol{P}_i', \boldsymbol{Q}_i'\rangle$ or $\langle \boldsymbol{P}_i'', \boldsymbol{Q}_i''\rangle$ results in the trivial case of $\langle G(\phi,\phi), G(\phi,\phi)\rangle$, it can be removed from $\mathbb{C}$.

Claim: *Using the above partitioning rule, all correspondences in the new $\mathbb{C}$ still retain the properties of completeness.*

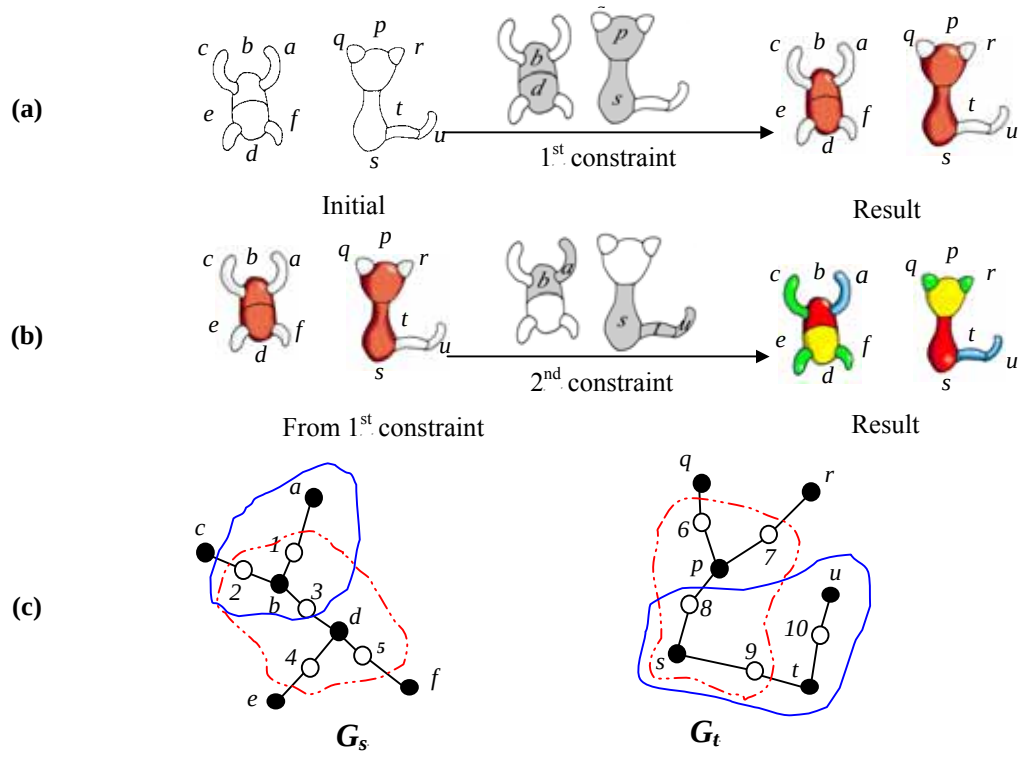
Proof of completeness

Now that the initial $\mathbb{C}$ is always complete, to prove the above claim, we only need to show that neither a permissible component pair nor a permissible connection pair is lost during each partitioning. Without loss of generality, here we only prove that given $\langle \boldsymbol{P}, \boldsymbol{Q}\rangle$, no permissible component pair is lost after partitioning $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i\rangle \in \mathbb{C}$ into $\langle \boldsymbol{P}_i', \boldsymbol{Q}_i'\rangle$ and $\langle \boldsymbol{P}_i'', \boldsymbol{Q}_i''\rangle$. This can be proved by contradiction as follows.

Assume a permissible component pair (x, y), where $x \in \boldsymbol{P}_i$ and $y \in \boldsymbol{Q}_i$, is lost after partitioning $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i\rangle$ to $\langle \boldsymbol{P}_i', \boldsymbol{Q}_i'\rangle$ and $\langle \boldsymbol{P}_i'', \boldsymbol{Q}_i''\rangle$. Then there must be two possibilities: 1) $x \in \boldsymbol{P}_i'$ while $y \in \boldsymbol{Q}_i''$ or 2) $x \in \boldsymbol{P}_i''$ while $y \in \boldsymbol{Q}_i'$. For the first case, from $x \in \boldsymbol{P}_i'$, we have

$x \in \boldsymbol{P}$ as $\boldsymbol{P}_i' = (\boldsymbol{P} \cap \boldsymbol{P}_i) \subseteq \boldsymbol{P}$; from $y \in \boldsymbol{Q}_i''$, we have $y \notin \boldsymbol{Q}$ as $\boldsymbol{Q}_i'' = \boldsymbol{Q}_i - \boldsymbol{Q}_i' = \boldsymbol{Q}_i - (\boldsymbol{Q} \cap \boldsymbol{Q}_i) \not\subset \boldsymbol{Q}$. However, the existence of $\langle \boldsymbol{P}, \boldsymbol{Q} \rangle$ indicates that every component in $\boldsymbol{P}$ can only be paired with some components in $\boldsymbol{Q}$. Hence, it is impossible that a component pair (x, y), where $x \in \boldsymbol{P}$ and $y \notin \boldsymbol{Q}$, exists in the final $\boldsymbol{G}_{st}$. Similarly, for the other case where $x \in \boldsymbol{P}_i''$ and $y \in \boldsymbol{Q}_i'$, we have $x \notin \boldsymbol{P}$ and $y \in \boldsymbol{Q}$. Also, it is impossible that such a component pair (x, y) exists in the final $\boldsymbol{G}_{st}$. All these contradict with the assumption that (x, y) is a permissible component pair. Therefore, the claim that the partitioning rule preserves the completeness of $\mathbb{C}$ is proven.◘

From the above claim, we know that R_V and R_E implicitly recorded in the new $\mathbb{C}$ still contain all permissible component pairs and all permissible connection pairs respectively. Also, because $\mathbb{C}$ is updated upon every new constraint, it respects all user-specified requirements.



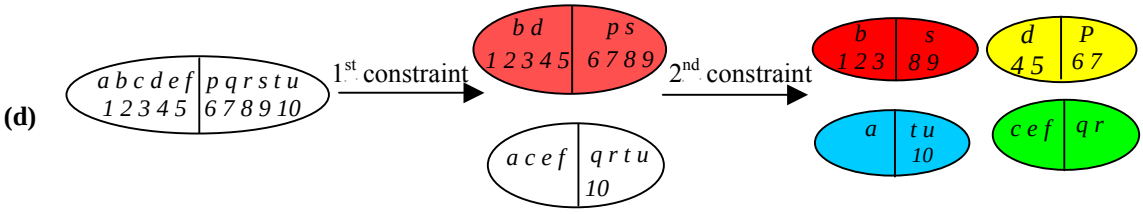


Figure 4.3 Processing constraint and updating $\mathbb{C}$

For $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$ in Figure 4.1, given a first constraint $\langle$ {*b*,*d*},{*p*,*s*} $\rangle$, $\mathbb{C}$ is refined and the resulting component correspondences are as shown in the rightmost picture in Figure 4.3(a). Next, given a second constraint $\langle${*a*,*b*},{*s*,*t*,*u*} $\rangle$, $\mathbb{C}$ is refined again and the resulting component correspondences are as shown in the rightmost picture in Figure 4.3(b). The initial $\mathbb{C}$ for this example is $\langle$(*G*({*a*,*b*,*c*,*d*,*e*,*f*},{*1*,*2*,*3*,*4*,*5*}), *G*({*p*,*q*,*r*,*s*,*t*,*u*},{*6*,*7*,*8*,*9*,*10*})$\rangle$. Upon the first constraint, the connection correspondence $\langle$ {1,2,3,4,5}, {6,7,8,9} $\rangle$ is deduced and upon the second constraint, $\langle${*1*,*2*,*3*}, {*8*,*9*,*10*}$\rangle$ is deduced. Figure 4.3(c) shows the partitioning of $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$ upon the first constraint, which is circled by the red dashed line and upon the second constraint, which is circled by the blue solid lines. Figure 4.3(d) shows contents of $\mathbb{C}$ after each of these two constraints is applied. Corresponding components are shown with the same colors in this figure and subsequent figures in this thesis.

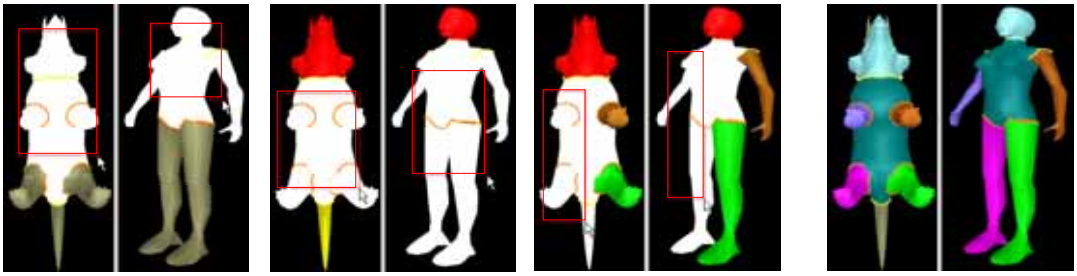


Figure 4.4 Specifying component correspondences using component groups

Figure 4.4 illustrates the efficiency of correspondences over component groups by using a triceratops and a woman. In this figure, user-specified component correspondences are shown in the three left pictures. The first one is to pair the front

part of the triceratops with the upper part of the woman; the second pairs their middles; the third pairs their left halves. Having these specifications, the framework is then able to calculate the complete component correspondence as shown in the rightmost picture, in which their heads, bodies and limbs are corresponding while the tail and the horns of the triceratops have no counterpart.

4.1.4 Constraint Tree

To record the history of user specification of constraints, we make use of a binary tree termed *constraint tree*. Each $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ in $\mathbb{C}$ is represented as a leaf of the constraint tree. 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 corresponds to the new correspondence $\langle \boldsymbol{P}_i', \boldsymbol{Q}_i' \rangle$ and $\langle \boldsymbol{P}_i'', \boldsymbol{Q}_i'' \rangle$ respectively. Obviously the current correspondence set $\mathbb{C}$ is actually the set containing all the leaves of the constraint tree. The content of a parent node is always equal to the union of the contents of its children; thus, it is not necessary to explicitly record the content of all internal nodes in the constraint tree.

The constraint tree for the example in Figure 4.3 is shown in Figure 4.5. Upon each new constraint, the framework extends the constraint tree with one more level by partitioning each leaf into a left child and a right child. Thus in Figure 4.5, there are three levels of the constraint tree upon the two constraints. Note that the i^{th} constraint is encapsulated in those nodes at the i^{th} level of the constraint tree (we say the root of the constraint tree is at level 0), and nodes at the same level in the constraint tree are organized into pairs. In this figure, the contents of the internal nodes of the constraint trees are labeled only for clarity. With the constraint tree, it is easy for the framework to support constraint undoing, as stated next.

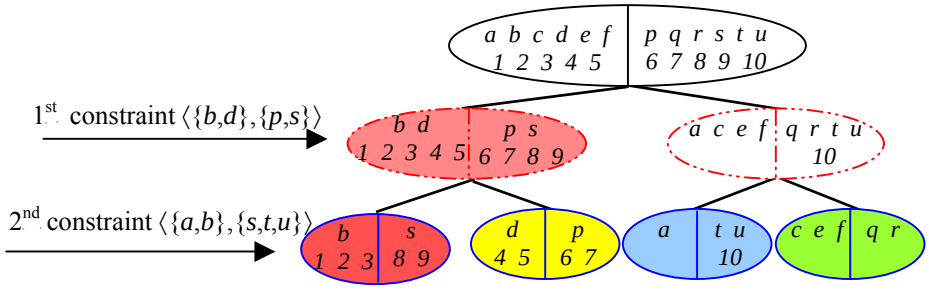


Figure 4.5 Constraint Tree

4.1.4.1 Flexible Undoing

To undo a specific i^{th} constraint, a naïve approach is to remove constraints in the reverse order from the most recent constraint to the i^{th} constraint. However, in this way, all those constraints specified after the i^{th} constraint will be lost. Using the constraint tree, this framework is able to remove solely the influences of the specific i^{th} constraint by performing subtree merging at the level i. For each pair of nodes at this level, we merge the subtrees of the two nodes by superimposing them. The contents of each pair of superimposed nodes in the subtrees are combined and put into a new node. The result for the subtree merging is a new subtree whose structure is the same structure as that of each subtree to be merged. The algorithm for undoing a constraint is shown as below.

[Algorithm 4.1] *Undo_Constraint*
Input: A constraint tree with m constraints (i.e. m+1 levels) and the i^{th} constraint to be removed ($1 \le i \le m$)
Output: Updated constraint tree after the given i^{th} constraint is removed
Step 1: Locate nodes of the constraint tree at level i;
Step 2: Locate a pair of nodes n_l and n_r at level i, which the same parent n_p at level i-1;
Step 3: Merge(n_l, n_r) {
  Locate the left and the right children of n_l , n_{ll} and n_{lr} and those of n_r, n_{rl} and n_{rr};
  Combine contents of n_{ll} and n_{rl} and save the result into a new node n_l';
  Combine contents of n_{lr} and n_{rr} and save the result into a new node n_r';
  Replace n_l with n_l' and n_r with n_r';
  If n_l and n_r are not leaves of the constraint tree {
    Merge(n_{ll}, n_{rl});
    Merge(n_{lr}, n_{rr});
  };
};
Set n_l' and n_r' as the new pair of children of n_p.
Step 4: Repeat Step 2 and Step 3 until all pairs of nodes at level i are updated by the merging.

This algorithm successfully removes the nodes at the tree level representing the unwanted constraint from the constraint tree. Using the constraint tree in Figure 4.5 as an example, if we remove the first constraint, which is represented by level one below the root, we merge the subtrees of the two nodes at level one by merging nodes at the second level, and then get the updated constraint tree as shown in Figure 4.6. It can be seen that the resulting constraint tree is exactly the same as a constraint tree with only the second constraint.

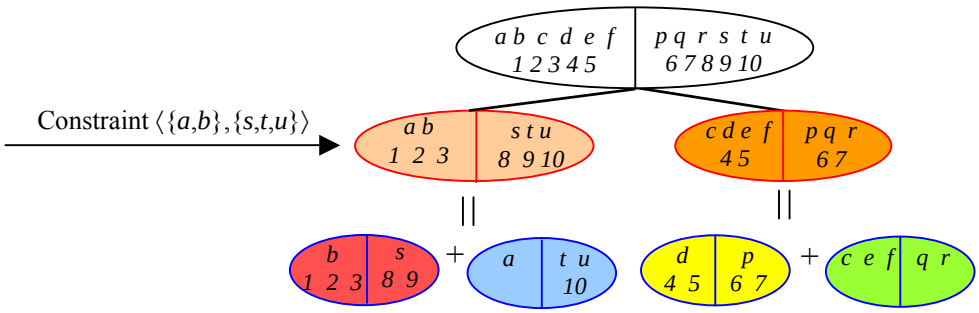


Figure 4.6 Flexible undoing

For the example in Figure 4.4, if we undo the first constraint shown in the leftmost figure, the resulting component correspondences are as shown in Figure 4.7(a). If we directly apply only the second and the third constraints, as shown in the two left pictures in Figure 4.7(b), the resulting correspondences are then as shown in the rightmost picture in Figure 4.7(b). Obviously, the resulting correspondences shown in Figure 4.7(a) and shown in Figure 4.7(b) are exactly the same.

(a) Undoing the first constraint

(b)Specifying only the second and the third constraints

Figure 4.7 Undoing the first constraint in Figure 4.4

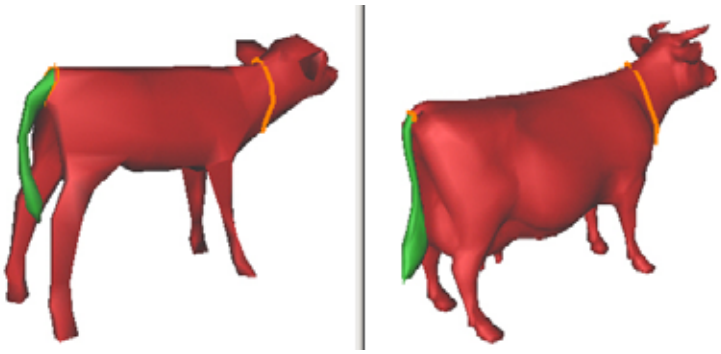
4.1.4.2 Modifying Component Decomposition

Users should be allowed to modify the component decompositions of $\boldsymbol{M_s}$ and $\boldsymbol{M_t}$ in the process of specifying component correspondence. In such a case, we need to update not only $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$, but also the constraint tree. In the updated constraint tree, all the constraints that do not contradict with this modification should be retained. A modification to component decomposition can be always simplified to, or represented as, one of the following operations: splitting a component c into two new components c_1 and c_2, and merging two connected components c_1 and c_2 into a new component c.

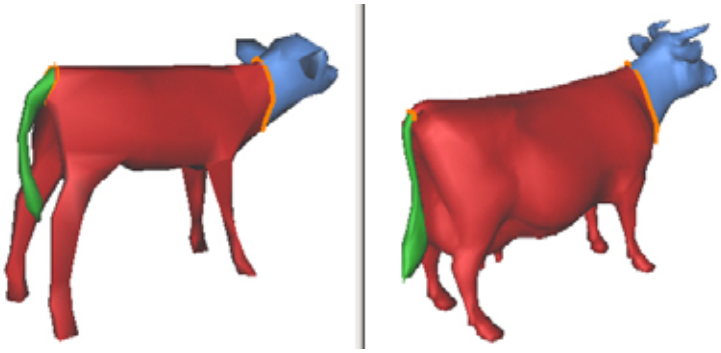
Splitting can be easily handled. This is because we only need to find the leaf containing c, replace c with c_1 and c_2 in the leaf, and update component connections accordingly. The structure of the constraint tree is unaffected here.

Merging is more complicated when the components to be merged are not within the same leaf. In such a circumstance, the structure of the constraint tree must be updated. The framework removes all the constraints that cause the separation of c_1 and c_2 by using the following algorithm. The basic idea of this algorithm is that we search the nearest common ancestor of these two components and recursively remove all the constraints causing the separation of c_1 and c_2.

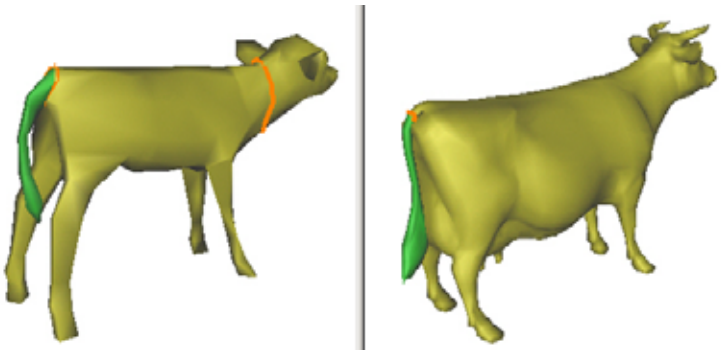
[Algorithm 4.2] *Update_Constraint_Tree_for_Merging*
Input: A constraint tree, two components c_1, c_2 and a new component c merged by c_1 and c_2.
Output: Updated constraint tree
Step 1: Locate the leaves where c_1 and c_2 are located. If they are within the same leaf, go to Step 5;
Step 2: Search upward from the leaf level to find the nearest common ancestor of the two leaves.
Step 3: Assume the found ancestor is at the level i of the constraint tree; remove the i+1th constraint by applying [Algorithm 4.1] for flexible undoing;
Step 4: Repeat Step 1 to Step 3;
Step 5: replace c_1 and $c_2$with c in that leaf where both of them are located;
Step 6: Update affected connections within the leaf.



(a) Pairing tails and then cutting heads



(b) Pairing heads



(c) Merging the body and the head of the cow

Figure 4.8 Correspondence maintenance after modifying component decomposition

Figure 4.8 provides a example for illustrating constraint maintenance upon modification to component decomposition. In Figure 4.8(a), the user pairs the tail of a calf with the tail of a cow (colored in green). The remaining components in both objects are then naturally paired (colored in red). Next, the user specifies a head component for each object by cutting its body, and as shown in the figure, the component correspondences still contain two correspondences represented by blue and red components respectively. In Figure 4.8(b), the user further pairs the calf's head with the cow's head (colored in blue) and the framework updates the component correspondences into three, which are shown in green, red, and blue respectively. Then in Figure 4.8(c), the user merges the head and the body of the cow into one and the component correspondences after the merging is as shown. We can see that the new component correspondences are almost the same as that in Figure 4.8(a) except that the component decompositions of the cow are different. This example indicates that when a user modifies component decomposition at the step of specifying component correspondence, the framework can maintain user-specified correspondences effectively.

4.1.5 Candidate Identification

The constraint tree keeps track of all permissible component pairs and all permissible 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. However, a user generally expects a morph has no unnecessary change in topology. It is often desired that the connectivity among components is kept as much as possible in a morph. For example, after pairing the head and the body of a cow with the head and the body of a triceratops respectively, the user generally does not regard

an ear of the cow as a good counterpart for the tail of the triceratops, although they are in the same leaf of our constraint tree. This is because the ear connects the body whereas the tail connects the head. In general, when pairing selected components, a user usually expects that good counterparts of these components are *similar in connectivity* to them. We call a counterpart that can meet such kind of user expectation a *candidate*.

Hilaga *et al.* [HSK01] proposed an automatic method for matching topology of 3D shapes. In terms of similarity measurement, it analyzed both topology and geometry of Reeb graphs to define similarity between two whole shapes. In our framework, however, because components and connections are already available and organized in connectivity graphs, we define similarity of components based on connectivity graphs. To identify candidates from the set of possible counterparts, we note that with group-to-group component correspondences, analyzing the connections for each component is not feasible. Therefore, we perform an analysis on entire connected groups within each leaf of the constraint tree as follows.

Within a leaf, we first organize its components into several groups of maximally connected components. See Figure 4.9 for an example. For clarity, the contents of $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$ are shown separately and leaves of the constraint tree are colored the same as in Figure 4.5. In this figure, leaf ***II*** contains one group of $\boldsymbol{G_t}$ that consists of components *t* and *u* connected by *10*. Similarly, leaf ***IV*** contains two groups of $\boldsymbol{G_t}$, which contains *q* and *r* respectively. Subsequently, we first define the *neighboring leaf of a group* within a leaf to be a different leaf that contains a connection incident to any component within the group, or one that contains a component incident to any connection within the group. Then, a group of maximally connected components in $\boldsymbol{G_s}$ is said to be *similar in*

connectivity to another group in $\boldsymbol{G_t}$ if and only if they are from the same leaf of the constraint tree and have the same set of neighboring leaves. See Figure 4.9. In leaf ***IV***, the group $\{e\}$ in $\boldsymbol{G_s}$ is similar in connectivity to both groups $\{q\}$ and $\{r\}$ in $\boldsymbol{G_t}$ as they have the same set of neighboring leaves, $\{\boldsymbol{III}\}$. On the other hand, there is no group in $\boldsymbol{G_t}$ is similar in connectivity to the group $\{c\}$ in $\boldsymbol{G_s}$.

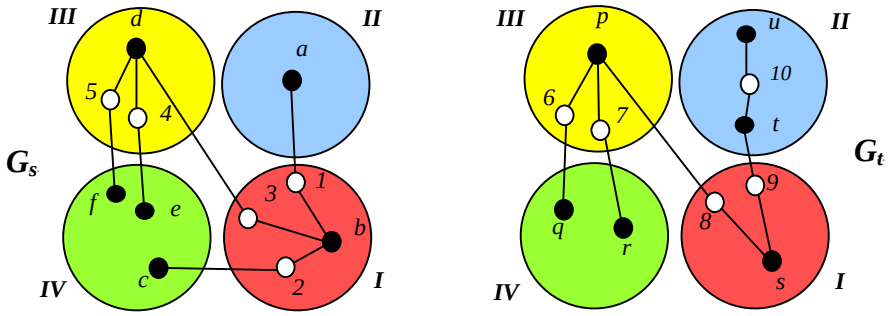
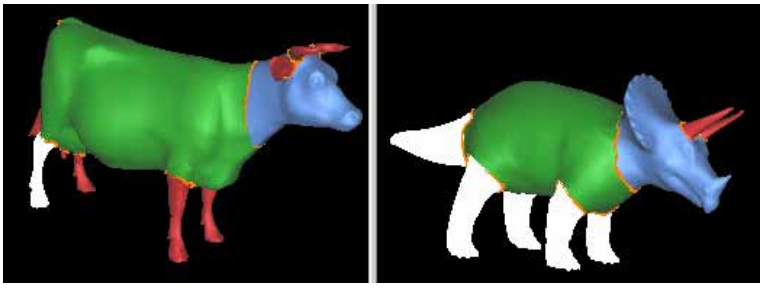
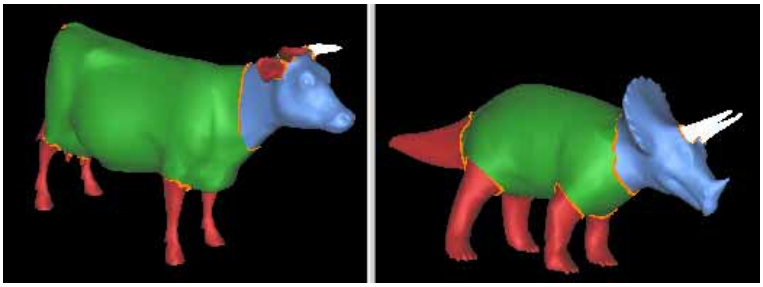


Figure 4.9 Analysis of similarity in connectivity

Using this approach, when a user selects a group of components for pairing, the framework first find those groups of maximally connected components containing user-selected components, then locate all the groups similar in connectivity to those found groups. By highlighting all components in the located groups, the system is able to provide assistance to the user in specifying component correspondence. Moreover, by identifying candidates during user specification, the framework performs a validity check to help the user to avoid unnecessary topological changes in the morph.



(a) Only the legs and the tail of the triceratops are found to be the candidates for the cow's leg



(b) Only the two horns of the triceratops are found to be the candidates of the cow's horn

Figure 4.10 Identifying candidates for user-selected components

Figure 4.10 illustrates the use of candidate identification in a morphing design. Given a cow and a triceratops, the user pairs first their bodies and then their heads. Subsequently, when the user picks a leg of the cow, the system indicates (by blinking) that only the legs and the tail of the triceratops are its candidates, as shown in Figure 4.10(a). When the user picks a horn of the cow, the system indicates that only the horns of the triceratops are its candidates, as shown in Figure 4.10(b).

4.1.6 Common Connectivity Graph Construction

When users finish their specifications in this step, before going into the next step, the framework employs heuristics to refine $\mathbb{C}$ further in order to calculate $\boldsymbol{G_{st}}$. Recall that user specification of component correspondences, i.e. constraints, is assisted by the candidate identification from the framework, and all user constraints have been encapsulated in the constraint tree. Therefore, we only need to deduce assumed component/connection pairs in each leaf of the constraint tree; the constructed common connectivity graph $\boldsymbol{G_{st}}$ must be consistent with all constraints and unnecessary topological changes can be avoided.

Graph matching, which measures the similarity of graphs, is generally a difficult problem and has been studied for more than two decades [V76, M82, B99, BJK00]. Graph is such a versatile and flexible representation that graph matching has been

utilized in a lot of applications (see the survey paper [B00]). In our framework, instead of attempting to establish the best matching of components between $\boldsymbol{G_s}$ and $\boldsymbol{G_t}$ through complicated comparison methods, we seek to work out a set of feasible component/connection pairs within each leaf in an efficient way.

Note that in each leaf of the constraint tree, the number of components of $\boldsymbol{G_s}$ and that of components of $\boldsymbol{G_t}$ are generally different. Therefore, it is obvious that we need to add null-components where appropriate during the computation of component pairs. Thus, the method of simply applying again the similarity in connectivity criteria for components to assign component pairs usually cannot produce satisfactory results. For example, in the leaf *I* shown in Figure 4.9, there are one component of $\boldsymbol{G_s}$, *b*, and one component of $\boldsymbol{G_t}$, *s*, in leaf ***I***, see the circled part in Figure 4.11. The component *b* has three adjacent components (*a*, *c* and *d*) and correspondingly, it has three neighboring leaves ***II***, ***III*** and ***IV***. As for the component *s*, it has only two adjacent components, *t* and *p*, and thus it has two neighboring leaves, ***II*** and ***III***. Hence, if we perform an equality check of similarity in connectivity for *b* and *s*, the result will be *b* and *s* are each paired with a null-component, rather than being paired with each other. Similarly, if each component itself is treated as a group, we note that in the leaf ***II***, *a* has one adjacent component which is in leaf ***I***, *t* has two adjacent components which are in leaf ***I*** and leaf ***II*** respectively, whereas *u* has only one adjacent component that is in leaf ***II***. Thus, the result of pairing components in the leaf ***II*** will be that *a*, *t* and *u* are each paired with a null-component. To solve such problems, we relax our measurement of similarity in connectivity and recursively make use of several heuristics in order in establishing component pairs within each leaf.

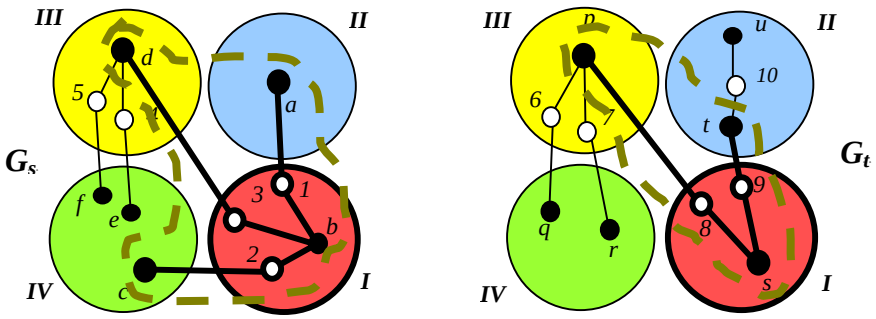


Figure 4.11 Similarity measurement of components

Before applying the heuristics, we perform two tasks in the preprocessing. First, we backup the current constraint tree. This constraint tree encapsulates all user-specified component correspondences, together with the implied component correspondences obtained during processing all constraints. Those component pairs computed by the heuristics will be treated as assumed constraints and they work only during the process of deducing $\boldsymbol{G_{st}}$. Once having $\boldsymbol{G_{st}}$, the framework will again make use of the backup constraint tree. In other words, *what we maintain in the step of global-level correspondence is the constraint tree, not* $\boldsymbol{G_{st}}$. Second, we analyze connections of each component as follows. For a component within a specific leaf, we construct its set of neighboring leaves from its incident connections with components outside the leaf, and its set of neighboring components from its incident connections with components inside the leaf. The *neighboring leaf of a component* within a leaf is defined to be a different leaf that contains a connection incident to this component. The *neighboring component* of a component is defined to be a component adjacent to this component within the same leaf. In Figure 4.11, for example, components of the leaves ***I*** and ***II*** are analyzed as shown in Table 4.1.

	Leaf ***I***		Leaf ***II***		
	b	*s*	*a*	*t*	*u*
Neighboring leaves	{***II***, ***III***, ***IV***}	{***II***, ***III***}	{***I***}	{***I***}	N/A
Neighboring components	N/A	N/A	N/A	{*u*}	{*t*}

Table 4.1 Neighboring leaves and neighboring components of components

[Heuristics 1] If there is only one component in a leaf, this component is then paired with the null-component ζ_V.

[Heuristics 2] If there are some connected components and all of them are paired with ζ_V, they are merged into one component corresponding to ζ_V.

[Heuristics 3] Within a leaf, if a component of $\boldsymbol{G_s}$ and a component of $\boldsymbol{G_t}$ have the same set of neighboring leaves, and the same number of neighboring components, they form a component pair.

[Heuristics 4] Within a leaf, for a component of $\boldsymbol{G_s}$, denoted by c_s, if there are some components of $\boldsymbol{G_t}$ whose sets of neighboring leaves are all the same as that of c_s, pair c_s with the one whose number of neighboring components is closest to that of c_s.

[Heuristics 5] Within a leaf, for a component of $\boldsymbol{G_s}$, denoted by c_s, and a component of $\boldsymbol{G_t}$, denoted by c_t, if they have at least one common neighboring leaf and their other neighboring leaves each contains only one component, these two components are then paired with each other.

The algorithm of constructing component pairs within leaves of the constraint tree is described as below. By recursively applying the above five heuristics one by one over all unpaired components, this algorithm refines the constraint tree with new created component pairs treated as constraints. Among these heuristics, Heuristics 2 means that in the complete component correspondence, if a component of one mesh is paired with ζ_V, its adjacent components must each have a counterpart in the other mesh. As for Heuristics 5, because it is based on neighboring leaves and their contents, it is used after the other heuristics are applied in every leaf. When there is more than one pair of components meeting the condition of a heuristic, we arbitrarily select one pair to be the new component pair. Thus, the result of this algorithm is not unique.

[Algorithm 4.3] *Compute_Complete_Component_Correspondence*
Input: Source connectivity graph $\boldsymbol{G_s}$, target connectivity graph $\boldsymbol{G_t}$, a constraint tree
Output: The complete component correspondence
Step 1: Apply Heuristics 1 to every leaf of the constraint tree;
Step 2: Apply Heuristics 2 to every leaf of the constraint tree;
Step 3: For each leaf {
Apply Heuristics 3;
If a new component pair is created, go to Step 6;
Apply Heuristics 4;
If a new component pair is created, go to Step 6;}
Step 4: For each leaf {
Apply Heuristics 5;
If a new component pair is created, go to Step 6;}
Step 5: Find a leaf that contains unpaired components, randomly select a component in the leaf and pair it with the null-component ζ_V;
Step 6: According to the new created component pair, update neighboring leaves and neighboring components wherever necessary and apply this pair as a constraint to the constraint tree;
Step 7: Repeat Step1-6 until there are at most one component from $\boldsymbol{G_s}$ and one from $\boldsymbol{G_t}$ within every leaf of the constraint tree.
Step 8: For each leaf of the refined constraint tree {
If it contains only one component, pair it with ζ_V;
Otherwise pair the component from $\boldsymbol{G_s}$ with the one from $\boldsymbol{G_t}$;}

For the example in Figure 4.11, possible results of component pairs obtained by the above algorithm are shown in Table 4.2. Note that the five heuristics are applied recursively and thus the results shown in this table do not result from a sequential visit from Step 1 to Step 4 in the algorithm. A similar procedure can be employed on the connections and ζ_E is paired with every connection that has no counterpart. In the end result, for each $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ represented by a leaf of the constraint tree, there is at most one component and at most one connection in $\boldsymbol{P}_i$, and likewise in $\boldsymbol{Q}_i$.

		I	II	III	IV
Step 1	(Heuristics 1)	NONE	(ζ_V, u)	NONE	(c, ζ_V)
Step 2	(Heuristics 2)	NONE	NONE	NONE	NONE
Step 3	(Heuristics 3)	NONE	NONE	(d, p)	(e, q) (f, r)
	(Heuristics 4)	NONE	(a, t)	NONE	NONE
Step 4	(Heuristics 5)	(b, s)	NONE	NONE	NONE

Table 4.2 Added component pair in computation of complete component correspondence

Obviously, the complete component correspondence and the complete connection correspondence are successfully recorded in leaves of the refined constraint tree. The framework then applies the following algorithm to calculate the common connectivity

graph $\boldsymbol{G_{st}}$. From the results shown in Table 4.2, computed $\boldsymbol{G_{st}}$ is as shown in Figure 4.2.

[Algorithm 4.4] *Calculate_Common_Connectivity_Graph*
Input: The refined constraint tree
Output: The common connectivity graph
Step 1: For every $\langle \boldsymbol{P}_i, \boldsymbol{Q}_i \rangle$ of the constraint tree {
For every component pair inside, construct a correspondence node;
For every connection pair inside, construct a correspondence edge;}
Step 2: For every constructed connection edge {
For every connection from this correspondence edge {
If the connection is ζ_E, go to next connection;
For each component incident to this edge {
Find the correspondence node containing the component;
Set this correspondence edge to be incident to the found correspondence node;}
}
}

4.2 Local-level Correspondence

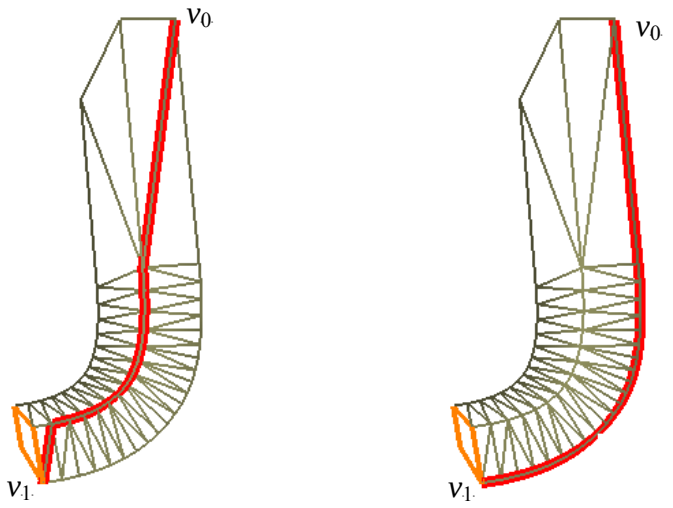
Local-level correspondence refers to correspondence over mesh vertices/edges/ triangles of components. It is a convenient way for a user to specify a requirement over local mesh details of original meshes. For each component pair containing no null-component, a user can specify and pair local features to control the morph (Section 4.2.1). In addition to *user-specified* local-level correspondences, the framework deduces *implied* local-level correspondences according to user specifications (Section 4.2.2) and adds *assumed* ones where appropriate to improve the morph (Section 4.2.3). An automatic patch partitioning method (Section 4.2.4) is proposed here to create compatible patch layouts subsequently. These patches are then used to establish the complete vertex correspondence for the component pair through parameterization (Section 4.2.5). For any component pair containing a null-component, the framework automatically constructs its complete vertex correspondence (Section 4.2.6).

4.2.1 User-specified Local-level Correspondence

For two corresponding components $C_s \in \boldsymbol{M_s}$ and $C_t \in \boldsymbol{M_t}$, a user can specify local-level correspondences by pairing local features on them. The framework provides three kinds of local features and the user can even pair local features of different types when necessary. A *feature vertex* is defined to be a mesh vertex and can be used to specify a feature such as a cow's nose tip. A *feature line* is defined to be a sequence of connected mesh edges and can be used to specify a feature such as a cow's mouth. Two end-vertices of a feature line are treated as feature vertices. A *feature loop* is defined to be a closed loop of mesh edges. It can be either the contour of a group of connected triangles, such as an eye in a cow's head, or a boundary of a component, such as edges where the cow's body connects the cow's head.

To specify a feature line or a feature loop, the user need not pick every mesh vertices on the local feature. When the user sequentially picks two mesh vertices, a path over mesh edges are automatically computed to connect these two vertices.

Among methods of path finding, the shortest path computation is most straightforward and popular. Finding the exact shortest path on a polyhedral surface [CH90, M98] is generally difficult. Instead, there have been several methods for approximating shortest paths [LMS97, KS00]. These methods refine original meshes by adding vertices and edges and then apply Dijkstra's algorithm [AHU83] to compute approximate shortest paths. See the example in Figure 4.12. When a user specifies a feature line connecting two vertices v_0 and v_1 in a component, the computed shortest path over mesh edges between these two vertices is shown in red in (a). In contrast with this path, the path shown in (b) is generally more desirable because it is smoother and passes through sharp edges in the component.



(a) The shortest path **(b) A smoother path**

Figure 4.12 Two kinds of paths connecting two vertices

Computing a desirable path between two vertices in user interaction is an important issue in automatic feature detection over 3D meshes. A general approach for such a computation is to employ energy functions to represent user expectation. Milroy *et al*. [MBV97] identified curvature extrema as possible edge points on segmentation contour. Garland and Heckbert [GH97] proposed a quadric error metric, and defined the feature energy at a vertex as the negation of the maximum of the edge collapse errors for the edges adjacent to this vertex. Y. Lee and S. Lee [LL02] extended 2D image snakes to 3D geometrical snakes, and computed both derivatives at vertices and normal changes at faces in their energy function.

The above methods are helpful in locating protuberant shapes or peaked corners in meshes. We note that such regions are generally observed as local features in morphing. For simplicity, we locate mesh edges surrounding such regions by modifying the cost function of mesh edges in Dijkstra's algorithm [AHU83] as:

$$Cost(e) = len(e) \times \frac{dihedral(e)}{\pi}$$

where $len(e)$ is the length of a mesh edge e, $dihedral(e)$ is the dihedral angle between

the two triangles incident to e and this angle is within $[0,\pi]$ by definition.

Utilizing the new cost function, the adapted Dijkstra's algorithm will more likely find paths passing sharp edges, as dihedral angles of such edges are much smaller than π. Besides, by using the new cost function, we also extend this algorithm in finding the approximated shortest path between two sets of vertices, for example, between two feature loops. Such a computation is more efficient than the naïve method of computing and comparing the shortest paths from every vertex of one set to every vertex of the other set.

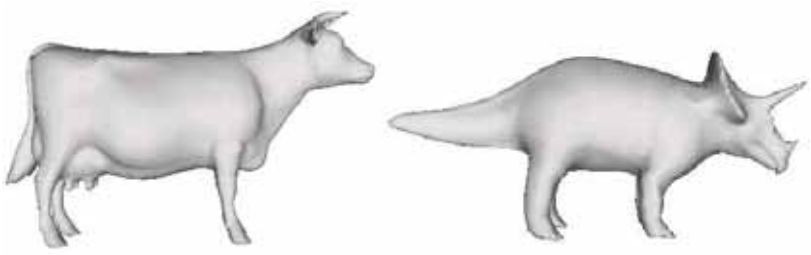
4.2.2 Implied Local-level Correspondence

Well-defined local feature pairs definitely prevent amorphous transformation. However, a user usually has to invest a lot of effort to identify proper and sufficient local feature pairs in a morphing design, especially when dealing with dissimilar objects of complex structures. This causes difficulties in morphing control and thus previous morphing systems used to be only for expert users. This framework solves this problem by deducing implied user requirements from user specifications. By capitalizing on the connectivity among components, the framework works out many implied (thus not explicitly stated) local feature pairs.

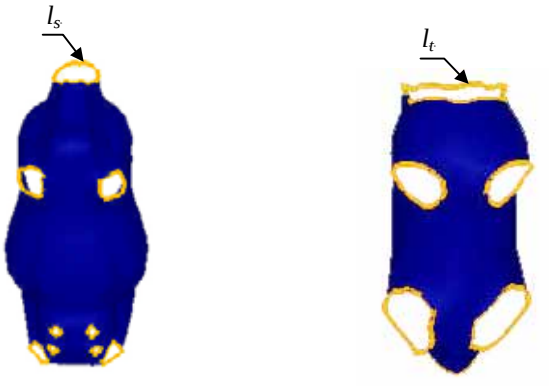
Recall that a boundary between two connected components is shared by both of them and can be easily achieved from component decomposition. In addition, we realize that boundaries of components are important local features in many morphing cases. Therefore, if a user-specified correspondence involves component boundaries, our component-based framework can deduce implied local-level correspondences at boundaries accordingly.

First, from given global-level correspondences, the framework is able to deduce correspondences over component boundaries. Recall that the common connectivity graph $\boldsymbol{G_{st}}$ encapsulates the complete connection correspondence. Thus, from connection pairs in $\boldsymbol{G_{st}}$, we can easily deduce a set of corresponding boundaries (each boundary is then a feature loop). For the example of $\boldsymbol{G_{st}}$ in Figure 4.2, the boundary of component *d* connecting component *f* and the boundary of component *p* connecting component *r* form an implied local feature pair, as deduced from the correspondence edge (*5*,*7*). At the same time, this correspondence edge implies another local feature pair at boundaries in the component pair (*f*, *r*). Successful assignment for such kind of features can definitely help to reduce user workload and improve the visual quality of the morph.

See the example in Figure 4.13, a user may be ignorant about how to start with local-feature specification when being asked to directly specify vertex correspondences for a cow and a triceratops shown in (a). With the help of components, however, the framework can automatically deduce a set of local-level correspondences at component boundaries. See Figure 4.13(b); if the head and the body of a cow are paired with the head and the body of a triceratops respectively, the two boundaries l_s and l_t, each of which represents a connection between a head and a body, must be corresponding. Similarly, all the boundaries of the cow's body, except for the four connecting the teats, can be automatically paired with their respective corresponding boundaries of the triceratops' body.



(a) User difficulty in directly locating local features for two meshes



(b) Corresponding local features at boundaries

Figure 4.13 Deduced correspondences over boundaries

Second, the framework can deduce correspondences at component boundaries from user-specified local-level correspondences. Within a pair of components, if two corresponding local features are both specified at boundaries and the adjacent components at these two boundaries are also corresponding, the framework automatically records these two local features as a local-level correspondence for the two adjacent components. Thus, once the user specifies a pair of local features at component boundaries for two corresponding components, he need not specify them again at the adjacent components of the two components.

Third, for the case where one component having more than one adjacent component is paired with a null-component, its connections with those adjacent components are paired with null-connections in the computation of $\boldsymbol{G_{st}}$. However, when we examine the local-level correspondences for two corresponding components,

we realize that counterparts of boundaries of such a component can be deduced in some cases. See the example in Figure 4.14. left-T and right-T each have one boundary connecting central-T, while left-U and right-U each have one boundary connecting each other. In this example, when the user specifies the component pairs (left-T, left-U) and (right-T, right-U), he actually expects the two component boundaries in T be paired with the boundaries shared by the two components of U.

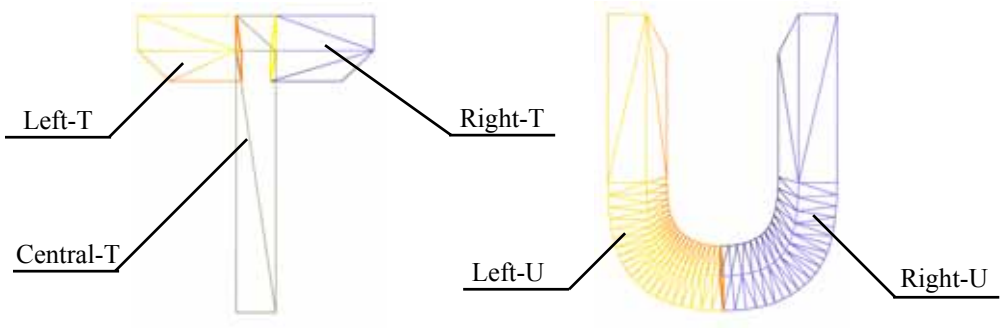


Figure 4.14 Deduced correspondences for the null-component case

The general idea about solving such a problem is as follows. If a component is paired with a null-component and it lies between several adjacent components, the boundaries between this component and its adjacent components must correspond to the boundaries between the counterparts of those adjacent components. First, after deducing local-level correspondences at boundaries from $\boldsymbol{G_{st}}$, the framework locates every unpaired boundary l for every component pair (C_s, C_t), where $C_s \in \boldsymbol{M_s}$ and $C_t \in \boldsymbol{M_t}$. Assume the adjacent component of C_s at l is C_s'. Then, different deductions are applied according to the counterpart of C_s'. If C_s' is paired with ζ_V, the framework analyzes the adjacent components of C_s' to find C_s'', whose counterpart C_t'' is an adjacent component of C_t. Then the framework assigns the boundary between C_t and C_t'' to be the counterpart of l. For the other case, we have C_s' is paired with C_t', where $C_t' \neq \zeta_V$, and C_t' is not an adjacent component of C_t. Similarly, the framework analyzes the adjacent components of C_t' to find whether there is one whose counterpart is incident

to C_s. If such an adjacent component is found, assume this component and its counterpart are denoted by C_t'' and C_s'' respectively. Then the framework assigns l and the boundary between C_t and C_t'' to be a pair of local features.

Implied local feature pairs are actually also what users desire in their specifications. Being able to find such local features, the framework saves a lot of user effort in correspondence control. In addition, through the above deductions, the framework aligns two corresponding components better and thus the final morph will be with higher visual quality. More importantly, if we examine the way of these deductions, we can realize that these advantages result from the utilization of components in the framework.

In addition, we note that when null-components exist in the complete component correspondences, not all component boundaries can be included in implied local-level correspondences. This is because in some cases, there are no unique correspondences over boundaries from user specifications. As such, the framework records those unpaired boundaries as potential local features and prompt the user to specify correspondences for them. The user can then choose to assign their counterparts, or just leave them as unpaired. For the latter case, the framework will then establish the vertex correspondences automatically before performing interpolation, see details in Section 4.2.6.

4.2.3 Assumed Local-level Correspondence

A user can choose to specify only those local feature pairs of interest. Besides being able to deduce implied local-level correspondences during user specification, the framework is also able to add assumed local feature pairs when the user has finished

his specification so far. Assumed local feature pairs supplement but never restrict user specification in local-level correspondence. They are only added when user-specified local feature pairs are not sufficient for aligning two corresponding components. Deduction of assumed local feature pairs is based on user specifications and component shapes, and carried out only in some specific conditions. In the event that the user subsequently specifies new feature pairs after the addition of assumed feature pairs, the existing assumed features are removed and new ones are calculated where appropriate. In the following, we define the *distance between two mesh vertices* to be the length of the shortest path between them along mesh edges. The *distance between two feature loops* is defined to be the minimum distance between their feature vertices, and in case there is no feature vertex in a feature loop, all its vertices are used in the distance comparison. The framework deduces assumed local feature pairs in the following circumstances.

First, for two corresponding feature loops, there should be at least two feature vertex pairs on them so that the system knows how to align them. Otherwise, unnatural twisting will occur in the final morph. The user does not have to specify feature vertex pairs for every corresponding feature loops. When the user does not provide this, the framework adds assumed feature vertex pairs on the feature loops as follows.

If there is only one pair of feature vertices for two corresponding feature loops, the framework simply find the two feature vertices opposite to the two vertices and assign them as the second pair of feature vertices.

For the case that in a component pair (C_s, C_t), no feature vertex pair exists in two corresponding feature loops l_s and l_t, the general idea of our treatment is to first examine the relative orientation between l_s and other local features in C_s, and then

establish feature vertex pairs at l_s and l_t based on given local-level correspondences. In C_s, we locate the local feature l_s' that is nearest to l_s by comparing the distances from l_s and other local features. Assume the shortest path from l_s to l_s' starts from l_s at v_s and ends at l_s' at v_s', l_t' is the corresponding local feature of l_s' in C_t, and the shortest path from l_t to l_t' starts from l_t at v_t and ends at l_t' at v_t'. We then assign (v_s, v_t) and (v_s', v_t') as two assumed feature vertex pairs.

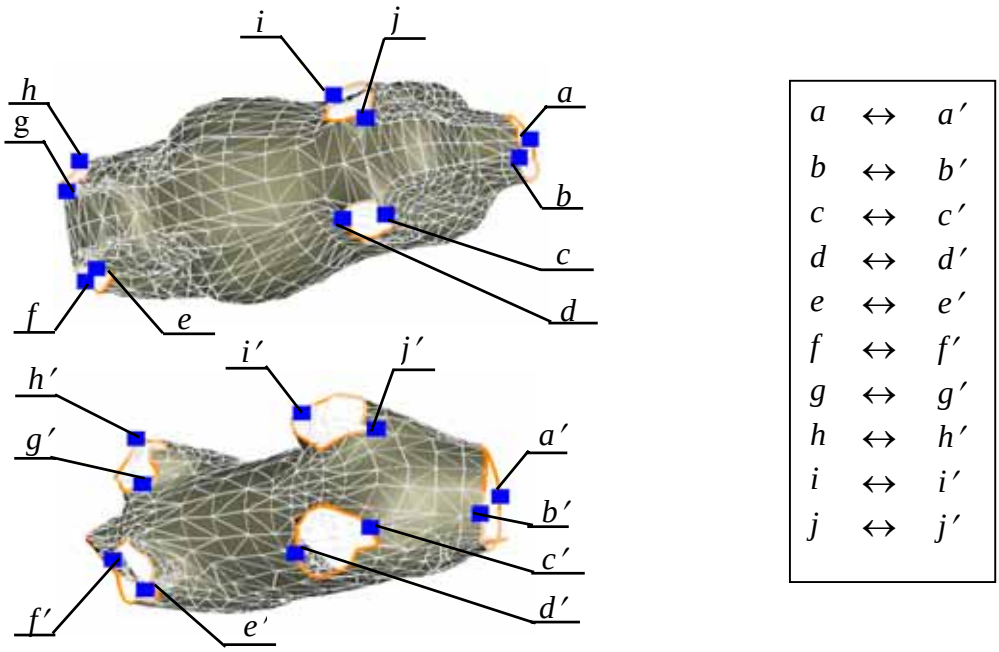


Figure 4.15 Assumed feature vertex pairs at feature loops

Note that the treatment for l_s and l_t is asymmetric in this approach. To make best use of such deduction, we sort all component pairs according to the number of adjacent components of the two components in the pair. The component pair with maximum connections is processed first. For example, in the morph between a cow and a triceratops, deduction of implied feature vertex pairs is performed first in the two corresponding bodies. Our experiment has shown reasonably good outcomes in most cases, especially when components with multiple adjacent components exist in a morph. See the example for the body of a cow and that of a triceratops in Figure 4.15. There are several pairs of feature loops (shown in orange), which are implied local

feature pairs at component boundaries. The assumed feature vertex pairs (shown in blue) at these feature loops are listed in the right. It should be noted that these assumed local feature pairs may not be reasonable choices in case that the shapes of two corresponding components are drastically different (because the relative orientation among the local features in one component and that in the other component differs greatly). In our framework, if users find in a component pair, assumed feature vertex pairs are not satisfactory, they can add new local feature pairs to improve the morph of the component pair.

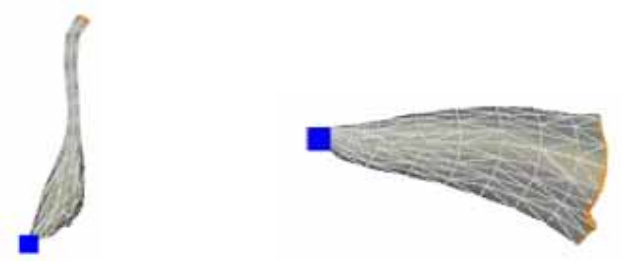


Figure 4.16 Assumed feature vertex pair at tips

Second, consider two corresponding components each of which has only one local feature. Such kind of components often exists in morphing design. For example, the tail of the cow and that of the triceratops each have only one boundary, as shown in orange in Figure 4.16. To better align the two components, the framework adds a pair of feature vertex at their tips, as shown in blue. In 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 be helpful to avoid the "tip-shrinkage" problem mentioned in [GSL98].

Third, in high-genus morphing, we often have such a case where two components in a mesh have more than one connection in between and the same for their corresponding components in the other mesh. The correspondences between their boundaries are initially assigned in the computation of $\boldsymbol{G_{st}}$. When the user specifies one

correspondence between some boundaries, the framework assumes the correspondences for the remaining boundaries by re-computing G_{st}. Figure 4.17 illustrates this using a mug and a donut. The user pairs a boundary in the mug with a boundary in the donut (shown in green); the framework then pairs the remaining boundaries (shown in blue).

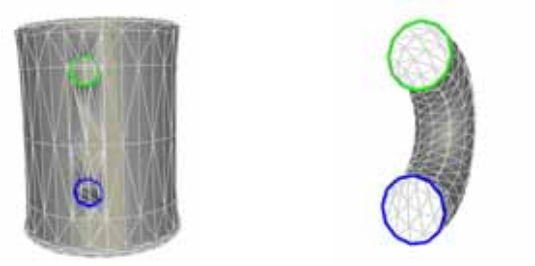


Figure 4.17Assumed local feature pair at boundaries

With all these kinds of local-level correspondences, an automatic patch partitioning method is next applied to generate compatible patch layouts for component pairs.

4.2.4 Automatic Patch Partitioning

Given two corresponding components with their local feature pairs, there are various ways to establish the complete vertex correspondence with all the local feature pairs aligned. A common approach is to first partition them into pairs of compatible patches and then perform a topological merging for each pair of patches. Most morphing methods using the patch partitioning approach establish the patch layout based on a user-specified feature net, or a control mesh [DG96, GSL98, KSK00]. That is, a user is responsible to specify connectivity for all local features. Such specification requires that the user be very clear about the formation of patch layouts, and thus demands proficient skills from the user. Praun *et al.* [PSS01] presented an excellent algorithm for establishing compatible partitions for meshes that share a common

coarse model. In this work, a user first picked a proper coarse model, and then in each mesh, specified corresponding vertices for every vertex of the coarse model. We note that a picked coarse model actually implies the connectivity among feature vertices and it may not always be available in a morphing system. Moreover, using such a coarse model, the user had to finish the assignment of a certain amount of feature vertex pairs. In contrast, user specification of local feature pairs is easier and more flexible in our framework. A proposed automatic patch-partitioning method adapted from [PSS01] is applied here to produce compatible patch layouts for component pairs.

Each *patch* in a patch layout is a group of triangles within a closed loop of mesh edges and such a loop is called the *boundary of the patch*. For two corresponding components, their patch layouts are said to be *compatible* if: 1) both patch layouts have the same number of patches, that is, they can be represented as $\{p_i \mid i = 1, 2, .., k\}$ and $\{q_j \mid j = 1, 2, .., k\}$ respectively; and 2) the connectivity among patches in one component is topologically equivalent to that in the other component. Thus, we know that compatible layouts for a component pair result in a set of corresponding patches.

The general idea of the automatic patch-partitioning method is first constructing two spanning trees for one component and then treating the net formed by the two spanning trees as a coarse model to guide the partitioning of the other component. A local feature together with those feature vertices located at this local feature are put into a *feature group*. A *link* is defined as a path over mesh edges. There are two kinds of links to be constructed. An *inter-link* between two feature groups is defined to be a link that starts at a feature vertex of one feature group and ends at a feature vertex of the other. An *intra-link* within a feature group is defined to be a link that connects two feature vertices of the feature group through mesh edges of the feature group. Let (u, u')

and (v, v') be pairs of feature vertices. For a link that connects u and v in one component, its *corresponding link* is a link that connects u' and v' in the other component. Note that edges in a link are not necessarily all from the original meshes. When a link cannot be computed by using the existing mesh edges, we cut triangles where appropriate and insert new edges.

After organizing all local features in one component into a set of feature groups, the framework needs to connect all feature groups through the computation of links. It is noted that we should not miss any local features in this computation; otherwise patch layouts for a component pair cannot be guaranteed to be compatible as the numbers of linked feature groups in two components are not identical. In addition, it is desirable that patches are as planar as possible and with as few as possible "swirling" in their shapes. To achieve this, all features are linked up sequentially in a minimum spanning tree (MST) fashion similar to the usage in [PSS01]. During the computation of links, the framework again makes use of the cost function in Section 4.2.1.

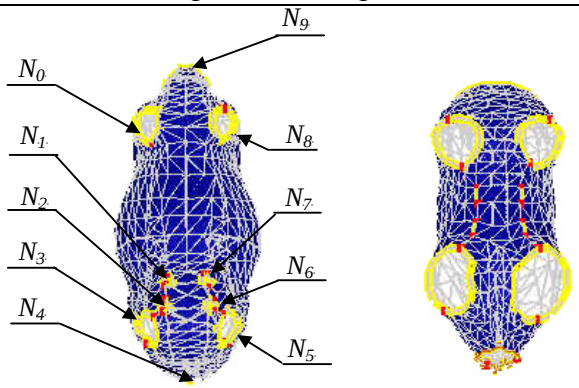
After constructing a MST to connect all the feature groups in one component, we constructs another MST to connect those feature groups each of which has only one connected feature group in the first MST. Obviously, such feature groups are at the root and leaves of the first MST. During the construction of the two MSTs, a link connecting two feature groups is computed in a way that it does not intersect other links except at its their end vertices. At each feature vertex, all the links connecting it are sorted as follows. Suppose at a feature vertex v, there are n links L_1, L_2, …, L_n whose mesh edges connecting v are e_1, e_2, …, e_n respectively. In a counter-clockwise order, e_1, e_2, …, e_n are sorted and L_1, L_2, …, L_n are then sorted accordingly.

Using the two spanning trees, the framework then partitions one component into a

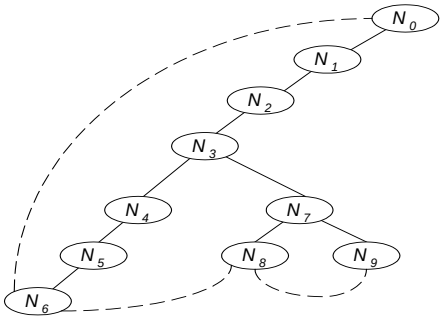
number of patches. Then, for each link in the component, its corresponding link is computed in its corresponding component. Subsequently, a compatible patch layout can be constructed in the other component. The automatic patch partitioning algorithm) is described as follows.

[Algorithm 4.5] *Automatic_Patch_Partitioning*
Input: Local feature pairs in two corresponding components C_s and C_t,
Output: Compatible patch layouts of C_s and C_t,
Step 1: For each local feature l in C_s and C_t {
If (it is not a feature vertex) {
Collect feature vertices on it and put l and these feature vertices into a feature group;
Sort its feature vertices into a counter-clockwise list;
Construct intra-links between every two consecutive feature vertices in the feature group;
}
else if (it does not belong to any other local feature)
Set l as a feature group;
}
}
Step 2: Construct_First_MST for C_s;
Step 3: Construct another spanning tree to connect the feature groups at the root and the leaves of the first MST of C_s;
Step 4: For every link in the two spanning trees, construct its corresponding link in C_t;
Step 5: In C_s, according to its two spanning trees, connect links to form several loops of links;
Step 6: For each loop of links p in C_s {
Construct a corresponding loop of links q in C_t using the corresponding links of all links in p;
Collect triangles at the left side of p to form a patch;
Construct its corresponding patch by collecting triangles at the left side of q;}

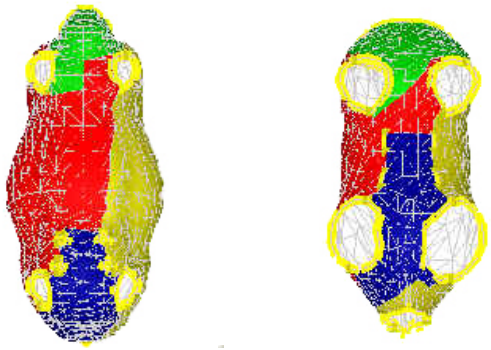
[Algorithm 4.6] *Construct_First_MST*
Input: A given component C_s and all its feature groups
Output: A MST connecting all the feature groups
Step 1:Take an arbitrary feature group in C_s and mark it as connected;
Step 2:While (there exists an un-connected feature group in C_s){
Compute the shortest path from a feature vertex in the connected group(s) to a feature vertex in the un-connected group(s);
Locate the un-connected feature group N' and a connected feature group N at the two ends of the found path and set the path as an inter-link of the spanning tree;
Mark N' as connected and add it to be a child of N in the spanning tree;
}



(a) Two corresponding components with feature groups



(b) Two spanning trees



(c) Compatible patch layouts

Figure 4.18 Automatic patch partitioning

Figure 4.18 shows an example of automatic partitioning a cow's body and a triceratops's body. In Figure 4.18(a), local features in these two components are highlighted in red (feature vertices) and yellow (feature lines and feature loops); feature groups in the cow's body are labeled. In Figure 4.18(b), the links of the first and the second spanning trees for the cow's body are represented by solid lines and dashed lines respectively. According to loops of links in the two spanning trees, each

component is partitioned into four patches. For example, in the cow's body, the four patches correspond to the loops $N_0N_8N_9N_0$, $N_9N_8N_7N_6N_5N_4N_9$, $N_4N_5N_6N_7N_1N_2N_3N_4$ and $N_4N_3N_2N_1N_7N_8N_0N_9N_4$ respectively. The resulting partitions are shown in Figure 4.18(c).

The automatic patch partitioning method produces pairs of corresponding patches, and thus compatible patch layouts, for two corresponding components. Each patch produced is homeomorphic to a disk. In addition, the framework also provides various tools for users to adjust the patch layouts, such as to modify the position of a non-feature vertex along a patch boundary or to specify a link between two feature groups.

4.2.5 Patch Parameterization

Because the source component C_s and the target component C_t are usually different in mesh connectivity, we need to construct the common mesh connectivity by parameterizing them. In other words, we need to convert C_s and C_t into C_s' and C_t' respectively, where C_s' and C_t' are with the same mesh connectivity. Using the common mesh connectivity, the framework then establishes the complete vertex correspondence for C_s and C_t and constructs their *meta-component* C_{st}, which represents C_s' at the first frame and C_t' at the last frame. Mesh parameterization is a technique that constructs a mapping between a mesh surface and an isomorphic simpler form. This technique has been extensively studied in the literature. Some important methods are listed as follows. Note that a morphing algorithm usually needs to perform a topological merging after constructing the mapping for original meshes — the mapped meshes are overlaid and the common mesh connectivity is obtained through calculating the intersections between the maps.

1) Spherical parameterization

For a mesh homeomorphic to a sphere, this method maps its vertices to a sphere surface. Kent *et al*. [KPC91, KCP92] discussed mapping from star-shaped to a wide class of genus-0 polyhedra. Carmel and Cohen-Or [CC98] presented a curve evolution algorithm for this task. Alexa [MA00] studied on how to extend this method to arbitrary genus-0 polyhedra through relaxation of mesh vertices.

2) Planar disk parameterization

There have been many different methods for mapping a mesh with boundary into a planar disk. The barycentric mapping method used in [ZSH00] guarantees the validity of topology of the parameterized mesh. Harmonic mapping used in [KSK98, KSK00] has the property of minimize metric dispersion during the embedding of a topological disk to a planar graph. An area-preserving mapping was introduced in [GSL98] to avoid area compression. Recently, Desbrun *et al.* [DMA02] proposed an intrinsic parameterization method to minimize the distortion of some different intrinsic measures of original meshes.

3) Cylindrical parameterization

Lazarus and Verroust [LV97] developed a re-sampling algorithm to establish vertex correspondences for original meshes. They re-sampled and rebuilt a star-shaped mesh into two hemispherical parts and one cylindrical sheet. An underlying axis was utilized in this algorithm to assist the shape parameterization.

4) Polyhedron realization

Shaprio and Tal [ST98] merged the vertex-neighborhood graphs of two original meshes to a common one by removing and re-attaching vertices on the process of polyhedron realization.

5) Multi-resolution parameterization

This method maps original meshes onto their coarse models. Lee. *et al*. [LSS98] presented a MAPS (multi-resolution adaptive parameterization of surfaces) method. User-specified local feature pairs were retained in the process of mesh simplification. They later employed this method in morphing [LDS99]. Praun *et al*. [PSS01] presented an algorithm for building compatible parameterization for meshes sharing a coarse model. This algorithm avoided the tremendous size of the common mesh in topological merging.

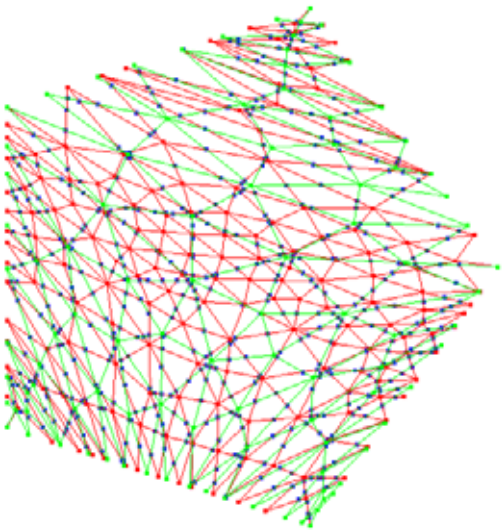


Figure 4.19 Mapping and Merging of corresponding patches

For the two corresponding patches shown in green in Figure 4.18, Figure 4.19 shows the result of barycentric mapping and topological merging. According to the five pairs of feature vertices at the boundaries of the two patches, two mapped patches are aligned in a planar disk. Vertices and lines shown in red are from the source patch, i.e., from the cow's body, and those shown in green are from the target patch, i.e., from the triceratops's body. Blue vertices in this figure are intersection points of the two maps. For clarity, the re-triangulation results with blue vertices are not shown in this figure.

The obtained common mesh topology is then mapped back to both patch surfaces. Thus, vertices in the source patch are bijectively associated with vertices in the target patch. Putting all patches in each component together, we then have the merged version of C_s, say C_s' and that of C_t, say C_t'. The complete vertex correspondence for the two components, which can be represented as a set of *vertex pairs* $\{(v_s, v_t) \mid v_s \in C_s', v_t \in C_t'\}$, is naturally formed.

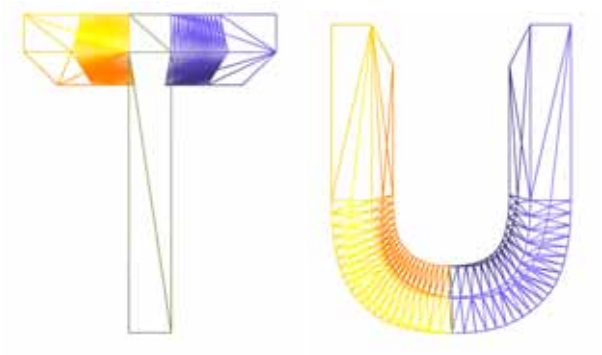


Figure 4.20 Topological Merging for meshes T and U

Figure 4.20 shows another example of topological merging by using the meshes T and U. Initially, the mesh connectivity of T is much simpler than that of U. Component correspondences between these two meshes are (left-T, left-U), (right-T, right-U) and (central-T, ζ_V). After the mapping and topological merging, two resulting meshes are with the same mesh connectivity, as shown in Figure 4.20. Obviously, after the merging, new vertices and triangles are added to the components left-T, left-U, right-T and right-U. For example, see the area where the left-T component joins the central-T component. How to handle the component pair (central-T, ζ_V) is discussed in the next sub-section.

Note that different parameterization methods can be plugged into this framework for patch parameterization. Obtaining the common mesh connectivity through merging the mapped patches is known to be costly in computation. By working on components

instead of the whole meshes, such computation in our framework is speeded up. On the other hand, the framework can also use other efficient methods such as multi-resolution remeshing in the place of topological merging.

4.2.6 Handling Null-components

Two original meshes in a morph are usually different in structure. For example, a cow has the salient features of two ears and four teats while a triceratops does not have. For such kind of exclusive high-level features in one mesh, some morphing works [GSL98, KSK00, ZSH00] relied on users to manually indicate their corresponding local entities in the other mesh. In such a circumstance, a user had to partition an exclusive feature in one mesh into patches and then assign corresponding patches in the other mesh. In [LDS99], a user was required to perform additional control by altering coarse models to make two original objects structurally similar. In contrast, our component-based framework conveniently represents the problem of an exclusive high-level feature as a component pair containing a null-component. For such a component pair, the framework automatically constructs the complete vertex correspondence in the following way.

For a component pair (C_s, C_t) where $C_s \in \boldsymbol{M_s}$ and $C_t = \zeta_V$ (or $C_s = \zeta_V$ and $C_t \in \boldsymbol{M_t}$), there will be a component disappearing (or growing) in the morphing sequence. Without loss of generality, only the case of component disappearing ($C_t = \zeta_V$) is discussed here. From Heuristics 2 in the computation of $\boldsymbol{G_{st}}$, we know that if a component from $\boldsymbol{M_s}$ is paired with ζ_V, its adjacent component must be corresponding to a component from $\boldsymbol{M_t}$. Thus assume C_1 is an adjacent component of C_s at boundary l_s and the corresponding component of C_1 is C_2, we have $C_1 \in \boldsymbol{M_s}$ and $C_2 \in \boldsymbol{M_t}$. After deducing local feature pairs from $\boldsymbol{G_{st}}$ for all component pairs containing no null-

component, the framework checks the boundaries of C_1 and C_2. If l_s is still unpaired, i.e. it is not involved in any local-level correspondence, the framework then prompts the user to assign its correspondence when the user specifies local-level correspondences for (C_1, C_2). Then the user can choose to assign a local feature l_t in C_2 to be the counterpart of l_s or to leave it unpaired. The framework then handles the component pair (C_s, C_t) in two different ways according to user input.

In the case that the correspondences of all boundaries of C_s are all known, the framework handles the component pair after the step of patch parameterization for all component pairs containing no null-component. Specifically, the following two steps are applied to automatically construct a new component C_t' at counterparts of the boundaries of C_s. C_t' is then used to be the counterpart of C_s.

In the first step, the framework converts C_s into a new component C_s' by updating its triangles near its boundaries. After the step of patch parameterization, C_1 is converted into C_1' and C_2 into C_2', where C_1' and C_2' both have the same mesh connectivity as that of the meta-component for (C_1, C_2). Note that after that step, local features of C_1 and C_2 (including l_s and l_t) are usually also changed, that is, they often have some newly inserted vertices. Assume l_s is changed into l_s'. In C_s, we need to update those triangles incident to the edges of l_s accordingly so that the resulting new component C_s' can connect C_1' seamlessly. Such a modification is illustrated in Figure 4.21. For every triangle Δabc on l_s, where ab is an edge of l_s, assume there are a sequence of new vertices $\{v_1, v_2, \ldots, v_{n-1}, v_n\}$ from a to b. We then replace Δabc with a sequence of new triangles Δav_1c, Δv_1v_2c, …, $\Delta v_{n-1}v_nc$, Δv_nbc. Vertex order in triangles is considered here to make sure that new triangles face outside of the mesh surface. Subsequently, we use C_s' instead of C_s for the construction of the new component C_t'.

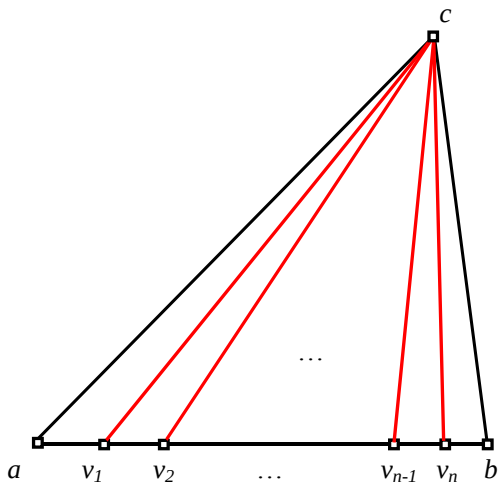
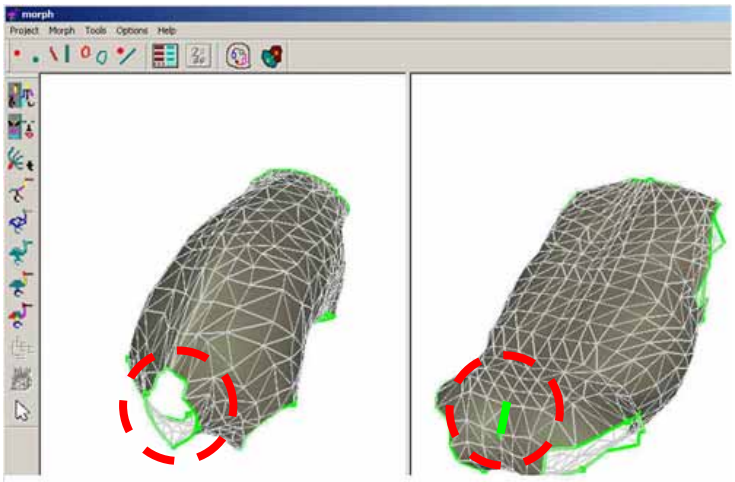


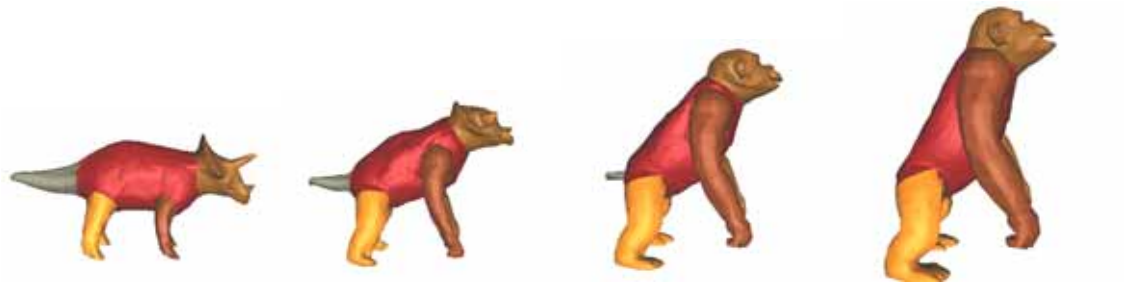
Figure 4.21 Updating boundary triangles

Then in the second step, the framework constructs C_t' by establishing its mesh connectivity and assigning positions for all its vertices. Its mesh connectivity can be easily obtained by copying that of C_s'. In this way, the complete vertex correspondence for C_s' and C_t' is naturally established. The framework next locates all vertices of C_t' at its boundaries. For a vertex pair (v_s, v_t) where $v_s \in C_s'$ and $v_t \in C_t'$, if v_s belongs to a boundary, the position of v_t was already determined by the complete vertex correspondence for the adjacent components of C_s' and C_t'; otherwise, among all vertices at boundaries of C_s', the one which is nearest to v_s is identified by comparing the distances from v_s and all the vertices at boundaries. Let the identified vertex be u. Then v_t is located at the position of v, where (u, v) is a vertex pair.

For the other case that counterparts of some boundaries of C_s are still unknown, the framework handles such cases before the step of patch parameterization. Because the user indicated that a component was paired with a null-component but did not specify how this component to be morphed (implicitly or explicitly), the framework simply merges C_s back to an adjacent component and leaves the step of patch parameterization to determine the counterparts for vertices of C_s.



(a) A user-specified feature line to be paired with the boundary between the body and the tail of the triceratops



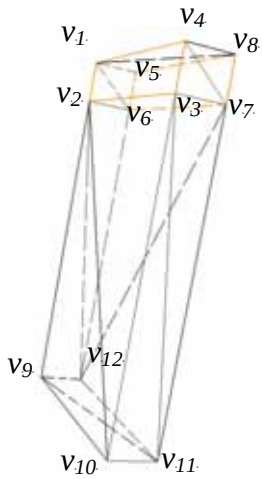
(b) Disappearing tails in the morph

Figure 4.22 Automatic handling the disappearing of the tail

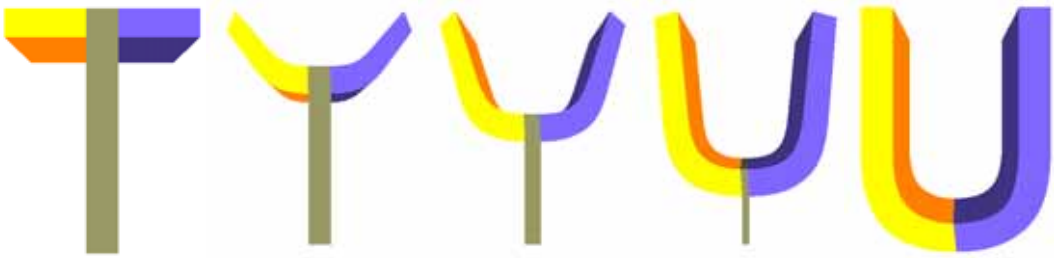
The above method is illustrated in Figure 4.22. The triceratops has a tail while the chimpanzee does not. The user specifies a feature line in the chimpanzee's body to be the counterpart of the boundary between the body and the tail of the triceratops. The framework then automatically generates a component at the feature line in the chimpanzee and produced a morph where the tail gradually disappears.

Figure 4.23 shows another example where a component corresponding to the null-component has more than one adjacent component. In this morph from T to U again, the component correspondences are (left-T, left-U), (right-T, right-U) and (central-T, ζ_V). See Figure 4.23(a), $v_1v_2v_3v_4$ forms the boundary between left-T and central-T while $v_5v_6v_7v_8$ forms the boundary between right-T and central-T. Both of these boundaries are paired with the boundary between left-U and right-U, as deduced by the framework. For the other vertices, v_9, v_{10}, v_{11} and v_{12} are nearest to boundary vertices v_2,

v_3, v_7 and v_6 respectively. To produce a morph with the effect of disappearing central part, the framework automatically constructs a new component, denoted by central-U and adds it between left-U and right-U. According to our method described above, in the constructed central-U, the vertices corresponding to v_9, v_{10}, v_{11} and v_{12} are located at the vertices corresponding to v_2, v_3, v_7 and v_6 respectively. The morphing result with the linear interpolation can be seen in Figure 4.23(b).



(a) Analysis of vertices in central-T



(b) Disappearing central part in the morph

Figure 4.23 Handling null-component in T-U morph

For $\boldsymbol{M_s}$ and $\boldsymbol{M_t}$, after establishing the meta-component for every component pair (with or without a null-component) from the complete vertex correspondence, we obtain a *meta-mesh* that has the common mesh connectivity of $\boldsymbol{M_s}$ and $\boldsymbol{M_t}$. This meta-mesh, denoted by $\boldsymbol{M_{st}}$, represents $\boldsymbol{M_s}$ at the first frame and $\boldsymbol{M_t}$ at the last frame. In each meta-component C_{st} of the meta-mesh, a vertex moves from its corresponding vertex in C_s' to its corresponding vertex in C_t'.

In this chapter, establishing correspondences between two original meshes is discussed. A user can choose to work either at the global level or at the local level, whichever is intuitive and convenient, to specify a requirement about correspondence. The framework does not require the user to complete certain workload. Effective assistance and deduction from the framework enables the user to specify any number of requirements. The end result for the correspondence process is a meta-mesh. Subsequently, the framework performs vertex interpolation for the meta-mesh to produce meshes at intermediate frames, as discussed in the next chapter.

Chapter 5 Component-based Interpolation Control

Given the complete vertex correspondence for every component pair, various vertex interpolation methods can be applied for vertex interpolation, for example, linear interpolation or as-rigid-as-possible interpolation [ACL00]. In this framework, morphing of mesh is decomposed into morphing of components. This approach eases and simplifies the mesh-morphing problem and makes it possible for users to manipulate individual components to control the interpolation process. In this chapter, we make use of the abstract form of meshes – skeletons and propose a skeleton-guided interpolation method. This enables users of our framework to either specify trajectories for components as a whole at the global level or specify trajectories for individual vertices at the local level.

Techniques in the field of skeleton-based animation/deformation are first discussed in Section 5.1. Then we introduce the representation of skeletons in our framework in Section 5.2. Section 5.3 describes our methods for calculating skeleton morphing from the computed common connectivity graph. Section 5.4 introduces the skeleton-guided interpolation method, which associates mesh vertices with underlying bones. Thus, mesh vertices are interpolated according to both the complete vertex correspondence and the guidance of the skeleton morphing. Section 5.5 discusses the way of controlling the interpolation process at the global level as well as the local level.

5.1 Skeleton-based Animation/Deformation

A skeleton of a 3D object is an effective tool for shape manipulation because it abstracts the essence of the object's structure with a low computation cost. Consequently, a change in shape can be well interpreted as a change in structure. In the field of computer graphics, skeleton-based methods have been employed in many applications such as modeling of implicit surfaces [BW90, B95, GKS98, AJC02] and motion capture for animated models [HFP00, MG01]. Only those skeleton-based techniques for animators are discussed here.

Skeletons can be used to deform meshes. To deform a 3D object, a user attaches a skeletal curve on a part of the object. By editing the curve, the user deforms its associated part. This technique is called axial deformation. For example, Lazarus *et al.* [LCJ94] focused on how to use an axis to naturally deform a part of a mesh. For a complex 3D polygonal mesh, its underlying skeleton usually consists of a collection of bones and the structure of the skeleton is thus complex in structure. Skeleton-driven animation/deformation techniques are widely used in the animation community. In these techniques, skeletons play an important role in creating natural, rigid and high-level controllable transformations. These techniques generally can be classified into two categories: geometrical approaches and physically based approaches. Techniques in the second category first examine appropriate physical models and then realize particular animation effects by using the physical models. Though generally producing animation with good visual quality, they involve high computation expenses. For example, Teichmann and Teller [TT98] generated a spring network to bind the movement of mesh vertices with their underlying skeletons. This method needs the complicated computation of stabilizing the network. On the contrary, the first category

is predominately used in the industry because it is more general and provides better user control. The following discussions mainly focus on the first category for this reason.

In geometrical approaches, each mesh vertex is bound with several bones of an underlying skeleton. The transformation of a vertex is obtained by blending the transformations resulting from the associations between the vertex and those bones. Such a problem is usually called skinning. Among all skinning methods, the weighted-vertex method is the simplest and most popular one in many commercial systems [Maya]. In this method, each vertex is assigned several weights for transformation blending. Interactive skeleton techniques [BW76] enhanced user control in motion dynamics. In addition, there are several methods [SK00, CGC02] that made use of FFD (Free form deformation) techniques and bound FFD lattices with skeletons. In these methods, transformation of skeletons affects mesh vertices indirectly through FFD lattices.

The skinning technique using transformation blending has some characteristic defects such as the "elbow shrinkage" and is notoriously difficult to control. Therefore, Lewis *et al.* [LCF00] presented a pose space deformation method to unify this technique and the shape interpolation technique. Sloan *et al.* [SRC01] proposed a similar method that combined these two techniques by "unbending" given hand-sculpted objects. In both of the two methods, vertex correspondences among given objects were assumed. Recently, Allen and his co-workers [ACP02] improved this technique by constructing displacement maps for objects with no obvious vertex correspondences.

Skeletons can assist an animator in tracking one recognizable shape to another

recognizable shape throughout a transition. However, it should be noted that the use of skeletons in mesh morphing is different from that in animation. Although the use of skeletons is not novel, there are several challenges in using skeletons in our framework. For example, the structure of a skeleton usually changes in a morph and user-specified correspondence makes the binding of mesh vertices to underlying bones much more complicated. Research difficulties of using skeletons in mesh morphing and our respective solutions will be discussed in the following sections.

5.2 Skeleton Representation

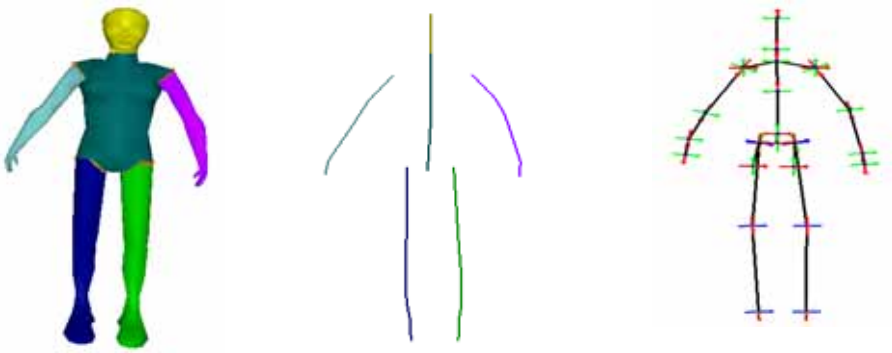
As we know, 3D meshes can be very complicated, involving thousands or even millions of polygons. The use of skeletons makes it possible for users to conveniently and efficiently manage such meshes. For a given mesh, its skeleton is an intuitive and simple tool to abstract its geometrical form and manifest its structural function.

There have been several different skeleton representation methods in the field of computer graphics. Typical ones include (1) medial axis, which is defined as the locus of points that are minimally equidistant from at lest two surface points, (2) geometric primitives, which can be of any dimensions from a point, a line segment, a polygon to a polyhedron and usually used for modeling soft objects (implicit surfaces built around skeletons) and (3) stick figure, in which a bone can be represented as a curve or a sequence of line segments.

Skeletons of the first two representations have complicated structures and thus may not be always intuitive for users, especially for non-experts. For example, some complicated forms such as parabolic curves are generally involved in a medial axis. As for geometrical primitives in skeletons of soft objects, due to different dimensions of

elements in skeletons, some special operations such as Minkowski sums are needed. To understand the function of skeletons of these two representations, users are expected to have sufficient technical knowledge.

In comparison with them, stick-figure skeletons are simpler and more indicative of object structures. Consequently, they are commonly used in building polygonal meshes in animation. One example is a generalized cylinder having a curve as its skeleton together with a set of cross sections [CLK98]. In addition, IK (Inverse Kinematics) skeletons, which are popular in animation systems, can also be treated as stick figures among their joints. Besides, there have been several methods [CH01, AJC02] for modeling implicit surfaces from skeletons containing interconnected curve-segments. Incurring low computation cost, stick-figure skeletons are natural abstractions for shapes [N82] and suitable for interactive modeling or animation systems. For this reason, we utilize stick-figure skeletons in this framework. For a woman model with its components shown in Figure 5.1(a), its skeleton in our framework and a corresponding IK skeleton in an animation system are shown in Figure 5.1(b) and Figure 5.1(c) respectively. Each set of colored arrows in Figure 5.1(c) indicates the position of a joint and the local coordinate system at that joint (with the colors R, G, B representing the axes X, Y, Z respectively).



(a) Components (b) Skeleton in the framework (c) Skeleton in animation

Figure 5.1 Skeleton representation of a woman model

To make the use of skeletons be consistent with our use of components, we define the skeleton of a mesh as follows: each component of the mesh can be abstracted into a *bone*, and all bones of a mesh form its *skeleton*. A user can choose to either assign the bone of a component manually, or obtain it through some automatic methods [VL99, LWT01, WP02]. See the example in Figure 5.1 again. In this figure, the components of the woman are shown with different colors in Figure 5.1(a) and the bones are colored the same as their respective components in Figure 5.1(b). The following discussion details our representation of skeletons.

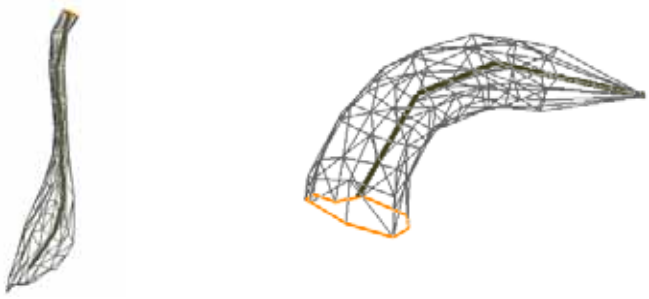
Bone Organization

Each component in a mesh has one underlying bone in the skeleton. With such one-to-one relationship, we organize bones of all components into a skeleton based on the connectivity among the components. That is, if two components connect each other in a mesh, their corresponding bones are said to be *adjacent*. For the example of the woman model, as its body component has five adjacent components of four limbs and a head, the bone of the body must have five adjacent bones representing those adjacent components respectively. We can see that the organization of bones in a skeleton here is different from that in animation techniques, in which bones in a skeleton is dependent on joints in the object.

Bone shape

A user can design the bone of a component according to his requirements in interpolation control, and the bone is not required to represent the shape of the component. To meet different user needs in controlling components via their bones, a bone in the framework can comprise several consecutive line segments, each of which

is called a *bone segment*. See the example in Figure 5.2, the bones for a tail and a horn of a cow model are shown in Figure 5.2(a) and Figure 5.2(b) respectively. Each bone segment has two endpoints and all such endpoints in a bone form a sequence of *skeletal vertices* of the bone. A point on a bone segment is then called a *skeletal point*. A bone has its direction along its skeletal vertices, starting from the first skeletal vertex and ending at the last one, and every bone segment has a direction from its starting skeletal vertex to its ending skeletal vertex. Along the direction of a bone, the i^{th} bone segment is called the *preceding bone segment* of the $i+1^{th}$ bone segment. Two consecutive bone segments are said to be *adjacent*. We can see that the shape of a bone in our framework is different from that in animation techniques, which usually represents a connection between two joints.



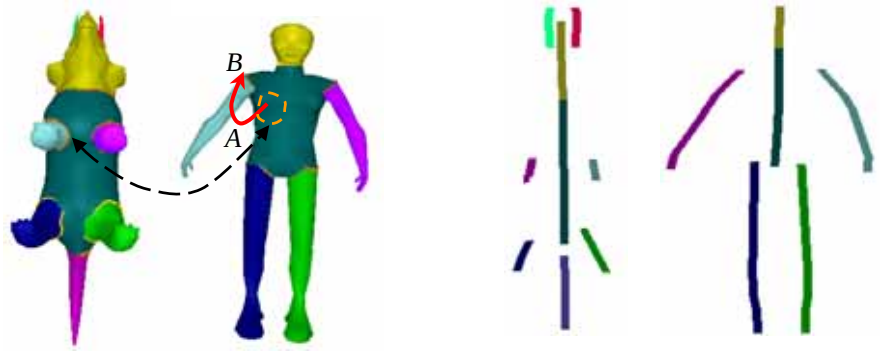
(a) a cow's tail **(b) a cow's horn**

Figure 5.2 Bone shape

Bone Connection

The use of skeletons in our framework should be able to support transformation of meshes. During the transformation form one mesh to the other, a boundary where two components connect each other often has its location changing with respect to these components, while this is almost fixed in an animation. For the example in Figure 5.3(a), when a triceratops is morphed into a woman, its upper legs need to be slid over its body from its initial position *A* to reach its final position *B,* being an arm connecting to the body of the woman. To support this, bones of two connected components are not

necessarily connected to each other. Figure 5.3(b) shows the skeletons of the triceratops and the woman models.

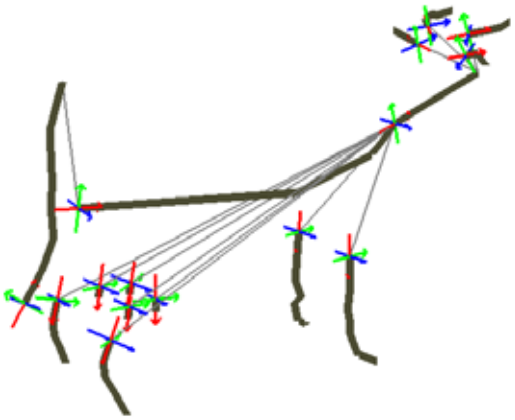


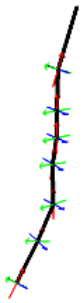
(a) Component sliding **(b) Unconnected bones**

Figure 5.3 Bone connection

Bone Parameterization

To represent the skeleton of a mesh, we make use of a moving local frame here to traverse among its bones and bone segments according to the connectivity among the bones. Take a bone as the *anchor bone* b_1 and its first skeletal vertex as the *anchor point*. Then from the anchor point in the anchor bone, the local frame moves from one bone to its adjacent bones, and within each bone, from one bone segment to another. During the traversal, all bones and their skeletal vertices are ordered and thus directions of the bones are determined. We can choose either the depth-first or the breadth-first traversal order and along the traversal path, each bone b_i (i=2, 3, …, n) in the skeleton has a *reference bone*. The position of the anchor point is measured in the world coordinate system while other skeletal vertices are measured in the moving local frame. In Figure 5.4, each set of red, green and blue arrows represents a local frame with the axes X, Y, Z. Figure 5.4(a) shows the local frames at bones of the skeleton of a cow model. For ease of illustration, we illustrate adjacency and the traversal order among bones by using gray lines between adjacent bones. Figure 5.4(b) shows the local frames at bone segments within the bone of the tail component of the cow model.





(a) Local frames of bones **(b) Local frames of bone segments in a bone**

Figure 5.4 A moving local frame

The transformation parameters of a bone are measured according to the movement of the local frame from the last skeletal vertex of its reference bone to its first skeletal vertex. These parameters include translation vectors $[T_x, T_y, T_z]^T$ and Euler angles (α, β, γ). Along the direction of a bone, the movement from a bone segment with respect to its preceding bone segment in the bone is recorded in a similar way. The directions of three axes in the moving local frame are determined as follows. As shown in Figure 5.4, the X-axis of the local frame at the bone segment is along the direction from its starting skeletal vertex to its ending skeletal vertex. To align the X-axis during the traversal, we make use of quaternions to calculate the minimum rotation for an alignment. This specific rotation naturally determines the direction of the Y-axis and Z-axis for the local frame. Therefore, during the traversal among a skeleton, rotation and translation are applied to the moving local frame and these parameters are used to represent the skeleton.

5.3 Skeleton Morphing

As stated in [BL99], the use of skeletons in morphing is potential because the interpolation between two skeletons permits the interpolation of two different objects

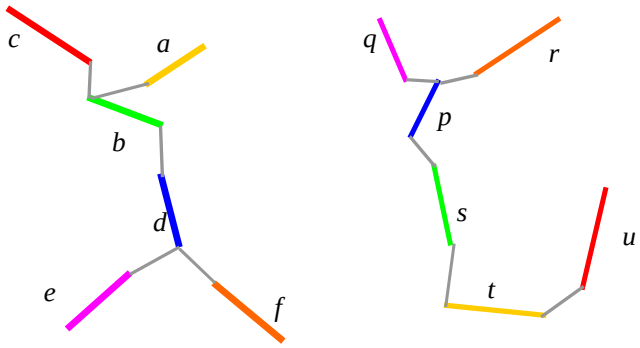
and makes the final morph more convincingly than in classical morphing methods. In volume-based morphing, Galin *et al.* [GA96a, GA96b, GL99] studied on morphing of soft objects. They tackled the problem of structural difference between two original soft objects by decomposing a component into a set of sub-components sharing the same skeletal primitives. Thus, these methods avoid the problem of different skeletal structures — even when the numbers of skeletal primitives in two soft objects are different, the two objects always have the same number of components after the decomposition. In boundary-based morphing, there have been several algorithms making use of skeletons. Shapira and Rappoport [SR95] morphed 2D polygons by using their star-skeletons. In this work, it was required that two skeletons in a morph must be compatible in structure. Lazarus and Verroust [LV97] morphed cylinder-like objects each of which had an underlying skeletal curve. Surazhsky and Gotsman [SG01] morphed stick figures with the same topological structure by improving 2D compatible triangulation methods.

In our framework, however, because a user can specify incompatible component decompositions for two original meshes, skeletons of the two meshes are generally different in structure. Therefore, given the skeletons for two original meshes $\boldsymbol{M_s}$ and $\boldsymbol{M_t}$, say the source skeleton $\boldsymbol{K_s}$ and the target skeleton $\boldsymbol{K_t}$ respectively, we need to construct a skeleton of the common structure, which is called meta-skeleton in the framework. Section 5.3.1 discusses establishing the meta-skeleton for $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$ based on the correspondences between their bones. Section 5.3.2 discusses the transformation of the meta-skeleton.

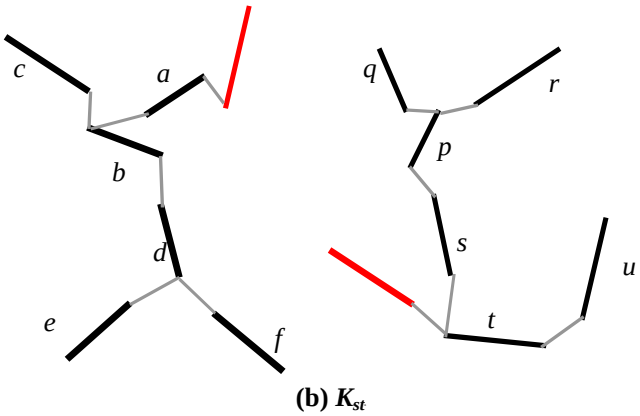
5.3.1 Common Skeleton Construction

Due to the one-to-one relationship between components and bones, a component

pair in each correspondence node of the common connectivity graph $\boldsymbol{G_{st}}$ corresponds to a *bone pair* (b_s, b_t), where $b_s \in \boldsymbol{K_s} \cup \{\zeta_k\}$, $b_t \in \boldsymbol{K_t} \cup \{\zeta_k\}$, and ζ_k is a *null-bone* that corresponds to a null-component ζ_V. Thus, all user-specified component correspondences are respected in the set of bone pairs deduced from $\boldsymbol{G_{st}}$. To create a morph between $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$, the key task is to establish the common structure for them. The *meta-skeleton* $\boldsymbol{K_{st}}$ of $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$, is defined to be a super-skeleton comprising a set of *meta-bones*, each of which represent a bone pair from $\boldsymbol{G_{st}}$. The meta-skeleton of $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$ represents $\boldsymbol{K_s}$ at the first frame and $\boldsymbol{K_t}$ at the last frame in a morph. Note that each bone in $\boldsymbol{K_s}$ or in $\boldsymbol{K_t}$ is mapped to one and only one meta-bone in $\boldsymbol{K_{st}}$. All meta-bones in $\boldsymbol{K_{st}}$ are organized as follows. For each correspondence edge in $\boldsymbol{G_{st}}$, we say the two meta-bones, which are obtained from its two incident correspondence nodes in $\boldsymbol{G_{st}}$ respectively, are *adjacent*.



(a) K_s and K_t



(b) K_{st}

Figure 5.5 Meta-skeleton

The meta-skeleton $\boldsymbol{K_{st}}$ can be regarded as the union of $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$. Figure 5.5 and

Figure 5.6 provide two examples to illustrate meta-skeleton. For the ease of comparison between both figures, the meta-bones containing null-bones are colored in red and the shape of a null-bone inserted to original skeletons is shown to be the same as that of its counterpart (in fact, the length of a null-bone inserted into an original skeleton is zero.) For the example of $\boldsymbol{G_s}$ and $\mathbf{G}_t$ in Figure 4.1, their corresponding skeletons $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$ are assumed to be as shown in Figure 5.5(a). For consistency, their bones are labeled the same as their corresponding components in Figure 4.1. Assume the complete component correspondence between $\boldsymbol{G_s}$ and $\mathbf{G}_t$ is as shown in the example of $\boldsymbol{G_{st}}$ in Figure 4.2. In Figure 5.5(a), those bones having no counterpart in the other skeleton are colored in red whereas corresponding bones among the other bones are shown with the same colors. The first and the last frames of the meta-skeleton deduced from $\boldsymbol{G_{st}}$ are shown in Figure 5.5(b). Gray lines in this figure are used to represent relationship among adjacent bones in both skeletons. In Figure 5.6,a skeleton of an animal and a skeleton of a plant are shown in the first row. For the ease of illustrating skeleton structure, bones in the two skeletons are shown connected without added gray lines in this figure. Corresponding bones are labeled with the same numbers and those bones having no counterparts in the other skeleton are highlighted by red. The first and the last frames of the meta-skeleton are shown in the second row. From these two examples, it can be clearly seen that the common structure of two original skeletons is established in the meta-skeleton, by properly adding null-bones into them. Note that the geometrical position of $\boldsymbol{K_{st}}$ is only meaningful at a certain frame in the morph. Due to null-bones in $\boldsymbol{K_{st}}$, transformation parameters of $\boldsymbol{K_{st}}$ at the first and the last frames are not the same as those of $\boldsymbol{K_s}$ or $\boldsymbol{K_t}$.

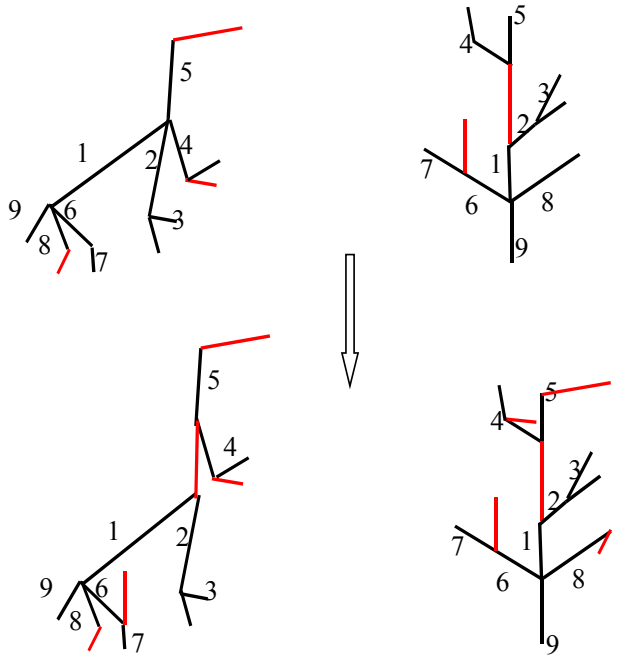


Figure 5.6 Another example of meta-skeleton

To morph $\boldsymbol{K_s}$ to $\boldsymbol{K_t}$, it is also necessary to align their bone segments. The result of such alignments is represented as bone segments of the meta-bones of $\boldsymbol{K_{st}}$. Consider a meta-bone b_{st} representing a bone pair (b_s, b_t). If $b_s \in \boldsymbol{K_s}$ and $b_t \in \boldsymbol{K_t}$, we align their bone segments by inserting new vertices to them as follows. We define the relative length of a bone segment s_i in a bone b as:

$$ratio(s_i) = \frac{\sum_{j=1}^{i} length(s_j)}{\sum_{k=1}^{n} length(s_k)} = ratio(s_{i-1}) + \frac{length(s_j)}{length(b)}.$$

For a bone segment of one bone, we insert a new skeletal vertex in the corresponding bone such that the new bone segment formed will have the same relative length, if such a skeletal vertex does not exist.

In the case that one bone in a bone pair is ζ_k, we count the skeletal vertices for the other bone in the bone pair, create a new bone containing the same number of skeletal

vertices, and use it to replace ζ_k in the bone pair. Thus, the two corresponding bones have the same number of bone segments. The lengths of bone segments of the new created bone segment are all set to zero.

Having the same number of bone segments in every two corresponding bones, we align $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$ and compute the parameter of $\boldsymbol{K_{st}}$ at the first and the last frames. Then by transforming the meta-skeleton between the two frames, we get the morph between $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$.

5.3.2 Skeleton Transformation

To compute the morph from $\boldsymbol{K_s}$ to $\boldsymbol{K_t}$, the framework needs to interpolate $\boldsymbol{K_{st}}$ from the first frame to the last frame. The simplest way is to record the Cartesian coordinates of all skeletal vertices in each meta-bone, and then to compute the morph of skeletons by interpolating coordinates of skeletal vertices. However, direct interpolation of vertex positions may result in unnatural transformation. An obvious example is that for a human skeleton, if we interpolate between two poses in the movement of a circling arm, intermediate arms will turn to be shorter than its original length. For more examples and discussions about the linear transformation method, see [SWC97].

To produce natural movement of skeletons, we again make use of a moving local frame here to traverse among all meta-bones and their bone segments of the meta-skeleton, according to connectivity of the meta-skeleton. The Cartesian coordinates of skeletal vertices of $\boldsymbol{K_{st}}$ at the first and the last frames are known from $\boldsymbol{K_s}$ and $\boldsymbol{K_t}$ respectively. Thus, we can compute parameters of $\boldsymbol{K_{st}}$, including translation vectors $[T_x, T_y, T_z]^T$ and Euler angles (α, β, γ), for all meta-bones and their bone segments at

these two frames. By interpolating the transformation parameters of meta-bones and their bone segments between the first frame and the last frame, the framework transforms $\boldsymbol{K}_{st}$ to produce the morph of skeletons.

Different interpolation methods can be used for interpolation parameters of skeletons. In addition to the linear interpolation of these transformation parameters, Spline interpolation [PTV92, U99] can be also employed to compute a smooth trajectory passing through all control points at intermediate frames. This method first calculates the control points of a spline curve from a given parameter set, and then obtains interpolation coefficients accordingly. As for the interpolation of orientation, it is known that direct interpolation of Euler angles might result in non-orthogonal matrix in general and it has the well-known problem of "Gimbal lock". In addition, Euler angles are dependent on coordinate axes and thus not unique. For example, one well-known setting for them is yaw, pitch and roll [HFK94]. To solve the above problems, Quaternions [S85] can be used to represent rotations in computer graphics. Interpolation of quaternions creates smoother transformation of orientations than interpolation of Euler angles. Given two quaternions q_1 and q_2, we use the spherical linear interpolation (SLERP) [B98] and when within a small region, apply the simple linear interpolation (LERP) for the interpolation at $t \in [0,1]$:

$$\text{SLERP: } q(t) = \frac{\sin[(1-t)\theta]}{\sin\theta} q_1 + \frac{\sin(t\theta)}{\sin\theta} q_2$$

$$\text{LERP: } q(t) = tq_1 + (1-t)q_2$$

After specifying requirements about component decomposition and component correspondence, the user need not wait till the last step of computing morphing sequences to see how those global-level specifications affect the final morph. Instead, the user can obtain the morph of skeletons, which can be regarded as a global-level

morph, at this early stage of a morphing design. Thus, the user can make decisions accordingly about whether to modify those global-level specifications. It is already known that morphing of skeletons incurs low computational cost. After the user revisits the step of global-level correspondences and modifies his specifications, the framework updates skeleton morphing swiftly. This results in short turn-around time in the global-level morphing process, and thus the user can perform the morphing design at the global level conveniently through a trial-and-error process.

A user can modify the trajectory of skeleton morphing by manipulate the meta-skeleton at an intermediate frame. Such a modification is saved as a keyframe of the meta-skeleton. Detailed discussions about updating the morphing sequence according to such keyframe editing will be provided in Section 5.5.

5.4 Skeleton-guided Interpolation

Morphing between two original meshes is abstracted into morphing between their underlying skeletons. In this section, we bind vertices of the meta-mesh to its underlying meta-skeleton so that skeleton morphing can be used to guide morphing of components. Specifically, a skeleton-guided interpolation method is presented here to make transformation of components follow the movement of their underlying meta-bone in the meta-skeleton.

5.4.1 Vertex Binding Technique

Although both the meta-skeleton and the meta-mesh are already available till now, the conventional skeleton-driven vertex interpolation technique in animation works cannot be applied to produce the final morph directly. This is due to the shape difference between two original meshes and the existence of user-specified local-

feature correspondences. First, in skeleton-driven animation, each mesh vertex generally has fixed relationship with bones in a skeleton. In morphing, however, the relationship between a bone and a vertex is varying in terms of both influence of the bone to the vertex and the relative position of the vertex with respect to the bone. Next, it usually happens in morphing that the vertex is required to move from one end of the bone to the other end of the bone, crossing several bone segments. Thus, to produce a smooth movement in such a case, solely making use of local frames at skeletal vertices is definitely insufficient. Moreover, local feature pairs in meta-components make the binding of vertices to bones more complicated. In general, there is a conflict between vertex positions determined by bones and those determined by the complete vertex correspondence.

To solve the above difficulties, the framework adapts the weighted-vertex method for our purpose. Given an initial state, this method calculates the current position of a vertex according to a weighted transformation blending function as shown below.

$$v = \frac{\sum_{i=1}^{n} \omega_i T_i (T_i^0)^{-1} v_0}{\sum_{i=1}^{n} \omega_i}$$

where v_0 is the initial vertex position of the vertex v and ω_i is the weight for the influence of i^{th} bone to the vertex. The transformations from the world coordinate system to the local frame at the i^{th} bone in the initial and the current state are represented as T_i^0 and T_i respectively. Note that in this function, $(T_i^0)^{-1} v_0$ represents the local coordinates of the vertex in the local frame at the i^{th} bone in the initial state; with the assumption that the local coordinates are always unchanged, $T_i (T_i^0)^{-1} v_0$ represents the world coordinates of the vertex when the bone is transformed with its current

transformation matrix.

Analyzing this function, we can see that there are several new problems when we try to employ the weighted-vertex method in morphing: the local coordinates are varying; the influence of a bone to a vertex, represented by ω_i, is not fixed and local frames solely located at bones (and bone segments) are insufficient. Next, we introduce our method for binding each mesh vertex with a meta-bone.

5.4.2 Single Binding

A meta-mesh contains a collection of meta-components and each meta-component has an underlying meta-bone. To transform vertices of meta-components around their underlying meta-bones, we parameterize these vertices as follows.

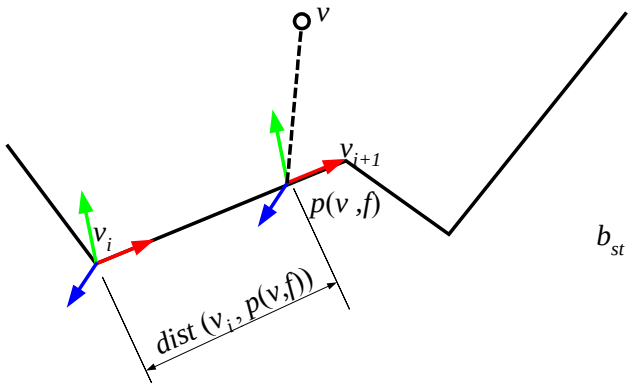


Figure 5.7 Parameters for binding a vertex to a bone

See the illustration in Figure 5.7. For the binding of a vertex v to a meta-bone b_{st} at a certain frame $f \in [0,1]$, the shortest distance between v and b_{st} is computed by comparing the distance between v and each bone segment of b_{st}. Use $p(v, f)$ to denote the skeletal point that is nearest to v on b_{st}. Assume $p(v, f)$ is from the i^{th} bone segment connecting two skeletal vertices v_i and v_{i+1}, and the distance between v_i and $p(v, f)$ is $d = dist(v_i, p(v, f))$. By translating the local frame at this bone segment with $[d,0,0]^T$, this method locates a local frame at $p(v, f)$. Then, similar to the definition of relative

lengths of bone segments in Section 5.3.1, the relative length of the skeletal point $p(v, f)$ along b_{st} is defined as

$$ratio(v, f) = \frac{dist(v_i, p(v, f)) + \sum_{j=1}^{i} length(s_j)}{(\sum_{k=1}^{n} length(s_k))}$$
$$= \frac{d}{length(b_{st})} + ratio(s_i)$$

Consequently, let the transformation matrix for the local frame at $p(v, f)$ be $MAT(p(v, f))$ and the world coordinates of v be x, y and z; its local coordinates with respect to the local frame at $p(v, f)$ can then be calculated by $[x_l(f), y_l(f), z_l(f)]^{\mathrm{T}} = MAT(p(v, f)) \cdot [x(f), y(f), z(f)]^{\mathrm{T}}$.

The framework first computes the relative lengths and local coordinates of each vertex of meta-components at both the first and the last frames. These parameters are saved in the *vertex keyframe list* of the vertex. Then, given an intermediate frame, the framework computes the location of each vertex at that frame by interpolating the relative length and local coordinates between the first frame and the last frame. With such an interpolation method, a vertex of a meta-component can be successfully transformed around its underlying meta-bone at the same time of respecting given vertex correspondences. Consequently, this method is called single binding.

See the example in Figure 5.8. For the two components colored by brown, there is a pair of corresponding vertices (v_s, v_t), and both vertices are located at component boundaries. Obviously, during the interpolation between v_s and v_t, the vertex should move from one end of the meta-bone to the other end. Therefore, the bone segment closest to the vertex varies during the interpolation. Using the single binding method, the framework can produce a smooth transformation from v_s to v_t. The algorithms of

recording vertex keyframe list and calculating the vertex interpolation are described as follows.

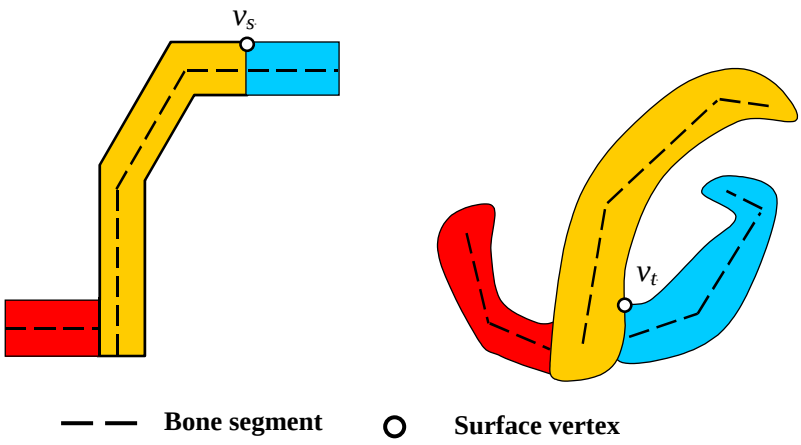


Figure 5.8 Binding a vertex to a meta-bone

[Algorithm 5.1] *Vertex_Keyframes_for_Single_Binding*
Input: A vertex v in a meta-component C_{st} with a meta-bone b_{st}
Output: Vertex keyframes at the first and the last frames
Step 1: At the first frame $f = 0$, find the skeletal point nearest to v, i.e. $p(v,0)$;
Step 2: Calculate $ratio(v,0)$ and the transformation matrix for the local frame at $p(v,0)$;
Step 3: Calculate local coordinates of v, i.e. $x_l(0)$, $y_l(0)$ and $z_l(0)$;
Step 4: Save $ratio(v,0)$, $x_l(0)$, $y_l(0)$ and $z_l(0)$ to the vertex keyframe at the first frame;
Step 5: At the last frame $f = 1$, find the skeletal point nearest to v, i.e. $p(v,1)$;
Step 6: Calculate $ratio(v,1)$ and the transformation matrix for the local frame at $p(v,1)$;
Step 7: Calculate local coordinates of v, i.e. $x_l(1)$, $y_l(1)$ and $z_l(1)$;
Step 4: Save $ratio(v,1)$, $x_l(1)$, $y_l(1)$ and $z_l(1)$ to the vertex keyframe at the last frame.

[Algorithm 5.2] *Interpolate_By_Single_Binding*
Input: A vertex v with its two keyframes in a meta-component C_{st} with a meta-bone b_{st}, an intermediate frame $f \in (0,1)$
Output: The world coordinates of v at f
Step 1: Read $ratio(v,0)$, $x_l(0)$, $y_l(0)$, $z_l(0)$, $ratio(v,1)$, $x_l(1)$, $y_l(1)$ and $z_l(1)$ from the vertex keyframes;
Step 2: At frame f, calculate relative length and local coordinates using linear interpolation.

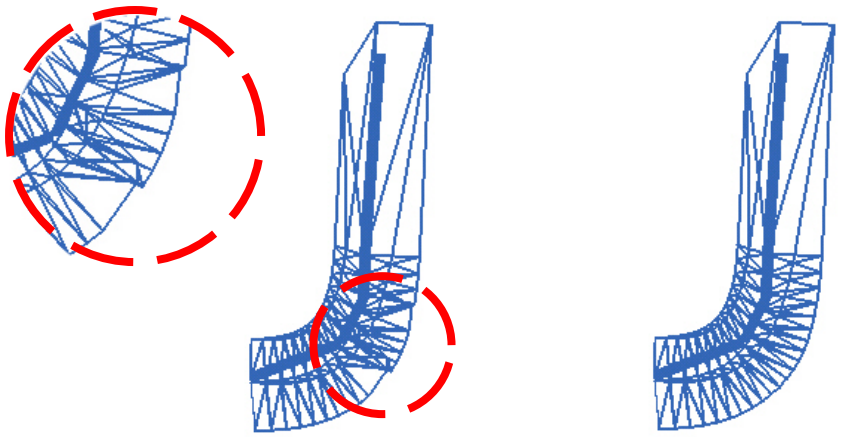
$$ratio(v, f) = (1 - f) \cdot ratio(v,0) + f \cdot ratio(v,1)$$

$$\begin{bmatrix} x_l(f) \\ y_l(f) \\ z_l(f) \end{bmatrix} = (1-f) \cdot \begin{bmatrix} x_l(0) \\ y_l(0) \\ z_l(0) \end{bmatrix} + f \cdot \begin{bmatrix} x_l(1) \\ y_l(1) \\ z_l(1) \end{bmatrix};$$

Step 3: According to $ratio(v,f)$ and $x_l(f)$, $y_l(f)$ and $z_l(f)$, find the skeletal point $p(v,f)$ on b_{st};
Step 4: Calculate the transformation matrix $MAT(p(v,f))$;
Step 5: Calculate the world coordinate of v at f by

$$[x(f), y(f), z(f)]^{\mathrm{T}} = MAT(p(v, f)) \cdot [x_l(f), y_l(f), z_l(f)]^{\mathrm{T}}$$

5.4.3 Double Binding



(a) Single binding **(b) Double binding**

Figure 5.9 Fold-over in the interpolation

In the single binding method, each vertex of a meta-component is bound to a corresponding skeletal point along the meta-bone of this meta-component. When it is applied to the T-U morph, the meta-component for the component pair (right-T, right-U) at frame $f = 0.5$ is shown in Figure 5.9(a). It can be seen that fold-over exists at the circled area where two bone segments connect each other, see also the picture with enlarged details at the corner of Figure 5.9(a). Because the three bone segments of the meta-bone at this frame are not co-linear, the orientation of the local frame needs to be changed twice when it traverses along these bone segments. Thus, the position of a vertex is suddenly changed when its corresponding skeletal point moves from one bone segment to another. The visual quality of morphs in such cases can be improved by binding a vertex to two adjacent bone segments — the one nearest to the vertex is called the *primary bone segment* of the vertex and the other is called its *secondary bone segment*. To distinguish this method from the previous single binding method, we call it double binding. For the same example in Figure 5.9(a), the meta-component at $f = 0.5$ computed by the double binding method is as shown in Figure 5.9(b).

When we parameterize vertices of a meta-component with the double binding method at a certain frame $f \in [0,1]$, the secondary bone segment $s'(f)$ of a vertex v is determined as follows. Generally, along the direction of its primary bone segment $s(f)$ starting from one skeletal vertex v_i and ending at another skeletal vertex v_{i+1}, if its corresponding skeletal point $p(v, f)$ is on the first half of $s(f)$, that is, $fraction(p(v, f)) = \frac{dist(v_i, p(v, f))}{dist(v_i, v_{i+1})} \leq 0.5$, $s'(f)$ is the bone segment preceding to $s(f)$; otherwise, $s'(f)$ is the bone segment next to $s(f)$. In addition, a weight w is assigned to v and used to balance the influences from the primary bone segment and from the secondary bone segment. Because the former is always more important than the latter, w increases from 0.5, when p is at the common skeletal vertex of two adjacent bone segments, to 1, when p is at the middle of the primary bone segment. With the double binding method, the algorithms of recording vertex keyframe list at the first and the last frames and calculating the vertex interpolation at intermediate frames are described as below. The double binding method generally produces smooth vertex transformation for meta-components.

[Algorithm 5.3] *Vertex_Keyframes_for _Double_Binding*

Input: A vertex v in a meta-component C_{st} with a meta-bone b_{st}

Output: Vertex keyframes at the first and the last frames

Step 1: Apply [Algorithm 5.1] to compute $ratio(v,0)$, $x_l(0)$, $y_l(0)$, $z_l(0)$ at $f = 0$ and $ratio(v,0)$, $x_l(0)$, $y_l(0)$, $z_l(0)$ at $f = 1$;

Step 2: For the corresponding skeletal point of v at $f = 0$, say $p(v,0)$, assume it is from the bone segment $s(0)$ which is the i^{th} bone segment of b_{st} ($i = 1, 2, .., m$), calculate $fraction(p(v, f))$;

Step 3: Determine the secondary bone segment $s'(0)$ as follows. If $s(0)$ is the first bone segment (i=1) or the last bone segment (i=m), $s'(0)$ is always the second or the last second bone segment; otherwise, $s'(0)$ is the i-1^{th} if $fraction(p(v, f)) < 0.5$ and the i+1^{th} if $fraction(p(v, f)) \geq 0.5$;

Step 4: Find the skeletal point $p'(v,0)$ on $s'(0)$ that is nearest to v and compute its relative length $ratio'(v,0)$;

Step 5: Compute the local coordinates of v with respect to the local frame at $p'(v,0)$, i.e. $x_l'(0)$, $y_l'(0)$ and $z_l'(0)$;

Step 6: Save $ratio(v,0)$, $x_l(0)$, $y_l(0)$, $z_l(0)$, $ratio'(v,0)$, $x_l'(0)$, $y_l'(0)$ and $z_l'(0)$ to the vertex keyframe at the first frame;

Step 7: Similarly at $f = 1$, compute $ratio'(v,1)$, $x_l'(1)$, $y_l'(1)$, $z_l'(1)$;

Step 8: Save $ratio(v,1)$, $x_l(1)$, $y_l(1)$, $z_l(1)$, $ratio'(v,1)$, $x_l'(1)$, $y_l'(1)$ and $z_l'(1)$ to the vertex keyframe at the last frame.

[Algorithm 5.4] *Interpolate_By_Double_Binding*

Input: A vertex v with its two keyframes in a meta-component C_{st} with a meta-bone b_{st}, an intermediate frame $f \in (0,1)$

Output: The world coordinates of v at f

Step 1: Read $ratio(v,0)$, $x_l(0)$, $y_l(0)$, $z_l(0)$, $ratio(v,1)$, $x_l(1)$, $y_l(1)$ and $z_l(1)$ from the vertex keyframes;

Step 2: At frame f, apply [Algorithm 5.2] to compute the skeletal point $p(v, f)$, locate the primary bone segment $s(f)$ on b_{st} by calculating $ratio(v,f)$, and then compute the world coordinates $x(f), y(f), z(f)$ by calculating $x_l(f)$, $y_l(f)$, $z_l(f)$;

Step 3: Read $ratio'(v,0)$, $x_l'(0)$, $y_l'(0)$, $z_l'(0)$, $ratio'(v,1)$, $x_l'(1)$, $y_l'(1)$ and $z_l'(1)$ from the vertex keyframes;

Step 4: Linearly interpolate between $ratio'(v,0)$ and $ratio'(v,1)$ to get $ratio'(v,f)$ and between $[x_l'(0), y_l'(0), z_l'(0)]^T$ and $[x_l'(1), y_l'(1), z_l'(1)]^T$ to get $[x_l'(f), y_l'(f), z_l'(f)]^T$;

Step 5: Calculate $fraction(p(v, f))$ and determine the secondary bone segment $s'(f)$ accordingly;

Step 6: The corresponding skeletal point $p'(v, f)$ determined by $ratio'(v,f)$ is not necessarily on $s'(f)$. Construct a local frame at $p'(v, f)$ by translating the local frame from the starting vertex of $s'(f)$;

Step 7: Calculate the world coordinates $x'(f), y'(f), z'(f)$ by using $x_l'(f)$, $y_l'(f)$, $z_l'(f)$ and the local frame at $p'(v, f)$;

Step 8: Calculate the weight of v at f as follow:

$$w(v, f) = \begin{cases} 0.5 + fraction(p(v, f)) & \text{if } fraction(p(v, f)) \le 0.5 \text{ and } s(f) \text{ is not the first bone segment} \\ 1.5 - fraction(p(v, f)) & \text{if } fraction(p(v, f)) > 0.5 \text{ and } s(f) \text{ is not the last bone segment} \\ 1 & \text{otherwise} \end{cases}$$

Step 9: Calculate the final world coordinates of v at frame f as:

$$w(v, f) \cdot \begin{bmatrix} x(f) \\ y(f) \\ z(f) \end{bmatrix} + (1 - w(v, f)) \cdot \begin{bmatrix} x'(f) \\ y'(f) \\ z'(f) \end{bmatrix}$$

5.4.4 Boundary Blending

After applying the single/double binding method, each meta-component moves around its underlying meta-bone in a morph. As such, if we put all meta-components together, the resulting meta-mesh may not be seamless throughout the morphing sequence. To connect meta-components at intermediate frames, it is apparently insufficient if only vertices of component boundaries are glued. There have been several methods for establish smooth connections for disconnected components in a polygon mesh. For example, Kanai *et al.* [KSM99] attached a part of one mesh to a part of the other by using morphing techniques. Given several user-specified vertex pairs, they first established vertex correspondences for two original meshes. Then they made use of three kinds of geometrical operations, including rigid transformation,

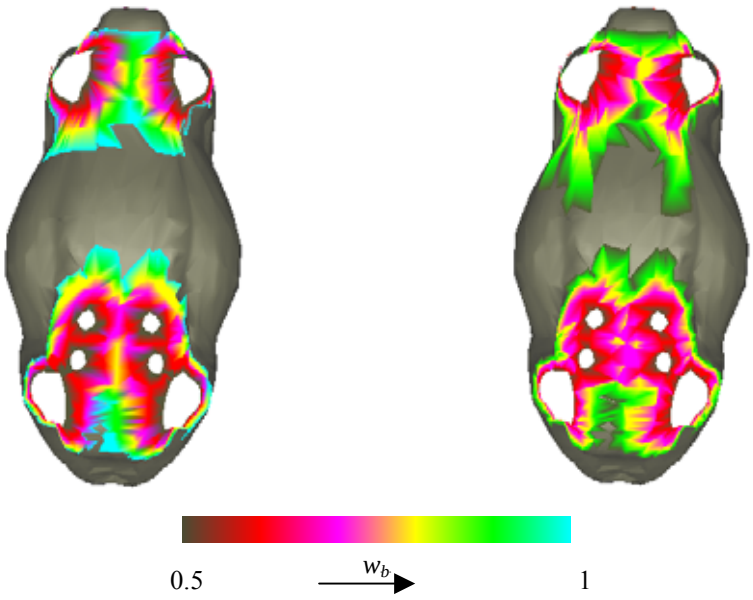
scaling and deformations, to smoothly align two parts at their boundaries. Alexa [A01b] allowed users to specify a region of interest by drawing boundaries at a mesh. Linear interpolation of Laplacian coordinates was then applied to produce morphs with local deformation. The component disconnection in this framework results from our use of skeleton-guided interpolation. Consequently, we propose a skeleton-based method to blend adjacent components at their common boundaries. Specifically, we adapt the weighted-vertex method further to automatically generate smooth connections among all meta-components computed by the single/double binding method at intermediate frames.

For two adjacent meta-components C_{st}^i and C_{st}^j sharing a boundary, their vertices near the boundary are influenced by both of their respective meta-bones, b_{st}^i and b_{st}^j. A vertex of a meta-component is always ultimately bound to the meta-bone of the meta-component. Hence, each vertex of a meta-component is assigned a blending weight w_b in the following way. In a meta-component C_{st}^i, its vertices that are sufficiently far away from the boundary have $w_b = 1$, those on the boundary have $w_b = 0.5$, and others in between have their weights between 0.5 and 1. Correspondingly, the weight for binding a vertex of C_{st}^i with b_{st}^j is $1 - w_b$. Thus, for a vertex v of C_{st}^i at an intermediate frame f, assume its position driven by the meta-bone b_{st}^i is $v^i(f)$ and that driven by b_{st}^j is $v^j(f)$, its final position is then calculated by $v(f) = w_b \cdot v^i(f) + (1 - w_b) \cdot v^j(f)$. In the case that a vertex is near to multiple component boundaries, its weight is assigned to be the largest one among all its weights resulting from those boundaries nearby, and the adjacent meta-component for the boundary blending is determined accordingly.

The distance from a vertex to a boundary in a meta-component can be calculated

using several methods. A common way is to define the distance as the length of the approximate shortest path over mesh edges, as stated in Section 4.2.1. Alternatively, one simple yet effective method is to measure the distance based on the topology of the meta-component. That is, the distance is increasing during the propagation from the component boundary to inside, according to the mesh connectivity. If a vertex has an adjacent vertex on the boundary, it has a large weight; if a vertex has to across several vertices to be on the boundary, its weight is small. It is known that a meta-component has a fixed topology, i.e. the common mesh connectivity, during a morph. Therefore, this method has an advantage that vertex weights only need to be computed once for a meta-component. Among the above two methods, we can choose to use the second in the process of morphing design and the first when the final morphing sequence is produced.

Figure 5.10 illustrates these two methods for weight calculation. In the morph from a cow to a triceratops, two corresponding components: the cow's body and the triceratops' body result in a body meta-component. At the first frame, the weight distribution of this meta-component using the first method is as shown in Figure 5.10(a) and that using the second method in Figure 5.10(b). Those vertices not affected by any boundary (and also any other meta-component) are un-weighted and they are not highlighted here. It can be seen that with the first method, the weight distribution is tighter around the boundaries whereas with the second method, the distribution is highly dependent on the mesh connectivity.



(a) Weights based on shortest paths **(b) Weights based on mesh topology**

Figure 5.10 Distribution of blending weights

A polygon mesh only has its surface information and simply transforming individual vertices ignores the existence of its interiors. Employing skeletons, the presented skeleton-guided interpolation method enables the blending of interiors and successfully preserves shape rigidity in final morphs. In addition, the problem of shape distortion that arises in the linear interpolation method can be generally avoided. A user also does not have to align original meshes in a morphing design. See the example in Figure 5.11 for morphing between a calf and a cow with different orientations. The morphing sequence produced by the linear interpolation method is shown in Figure 5.11(a). This morph involves serious distortion, especially at the third frame shown. In contrast, the morphing sequence produced by our skeleton-guided interpolation method is shown in Figure 5.11(b). It can be seen the morphed object gradually turns from its initial orientation of the calf to its final orientation of the cow, just as what a user usually expects.

(a) Linear interpolation

(b) Skeleton-guided interpolation

Figure 5.11 Morphing two objects with different orientations

5.5 Trajectory Editing

To support user control over intermediate objects, the most common way for interpolation control in previous morphing works is to set trajectories for individual mesh vertices. However, explicit handling of vertex trajectories is very tedious and thus inconvenient for interactive control [LV98, A02]. Although some alternative methods have been proposed by using physical simulation [DWS93], intrinsic parameters [SWC97] or weights [GSL98], it is difficult for users to express their requirements about adjusting the transformation at the high level. For example, to set a particular pose for a human-like object at an intermediate frame, a user must possess good design skills to set appropriate weights or intrinsic parameters and such user interaction is very labor-intensive.

In our framework, however, a user can easily control the interpolation process at the global level by inserting/editing keyframes of the meta-skeleton. With the proposed skeleton-guided interpolation method, the user can modify meta-components

intuitively by moving their underlying meta-bones. Keyframe editing technique is ubiquitous in animation works. In the framework, a meta-skeleton has two default keyframes at the first and the last frames and these two keyframes represent the source and the target skeletons respectively. The user can insert/edit keyframes by modifying the positions of meta-bones at intermediate frames. The final morph is then automatically updated through the skeleton-guided interpolation. Note that all correspondences at both the global and the local levels are retained when a keyframe of the meta-skeleton is inserted. Therefore, additional computations only include the interpolations of the meta-skeleton and mesh vertices. By inserting a few keyframes, the user can conveniently incorporate additional motions in a morph, for example, to make a morphed object walk.

Besides, the user can also control the interpolation process at the local level by modifying vertex positions at an intermediate frame. A modified position of a vertex is saved into the vertex keyframe list. Within the list, the new position is converted into vertex parameters for single/double binding. Subsequently, the skeleton-guided interpolation of this vertex is performed between relevant keyframes of the vertex during the vertex interpolation.

The speed of morphing along the trajectory can also be determined by user specification. Different kinds of mapping between the frame number f and time t can be defined. Suppose the trajectory of a vertex (or a meta-bone) can be represented as a function of time $p(t)$, where $t \in [0,1]$. We can modify it to $q(t) = p(f(t))$ where $f(t)$ is a function from [0,1] to [0,1]. By defining $f(t)$ properly, we can obtain the effects of speeding up or speeding down.

In this chapter, calculating the interpolation between two original meshes is discussed. Besides supporting the conventional linear interpolation method, the framework employs skeletons to enable multi-level interpolations control. A meta-skeleton of two original skeletons is introduced and used to compute the morph between them. The use of skeleton morphing results in short turnaround time when a user experiments with global-level morphing design. To transform mesh vertices around skeletons, an effective skeleton-guided interpolation method is proposed. This method not only preserves shape rigidity in morphing sequences, but also facilitates user control in the interpolation process at both the global and the local levels. In the next chapter, our experimental results are reported.

Chapter 6 Experimental Results

A prototype for the component-based morphing framework has been implemented on a Pentium IV 2GHz PC in C/C++ windows environment. As the framework aims to address issues about interactive morphing control, the main focus of our experiments is to test the efficiency and effectiveness of user interaction in the framework. Our graphics user interface (GUI) is first introduced in Section 6.1. Section 6.2 then describes several demos of the whole morphing process. Section 6.3 provides morphing sequences and statistics for several morphs. The results reported in this chapter can be also found at our morphing webpage (see [ZOT03]).

6.1 Graphical User Interface

The implemented system provides a friendly and easy-to-use GUI to assist user interaction. There are three kinds of views for users to conveniently specify their requirements. In all views, the GUI provides tools for object selection, viewing options setting, object properties (such as component names and materials) configuration and object transformation (translation, rotation and zooming).

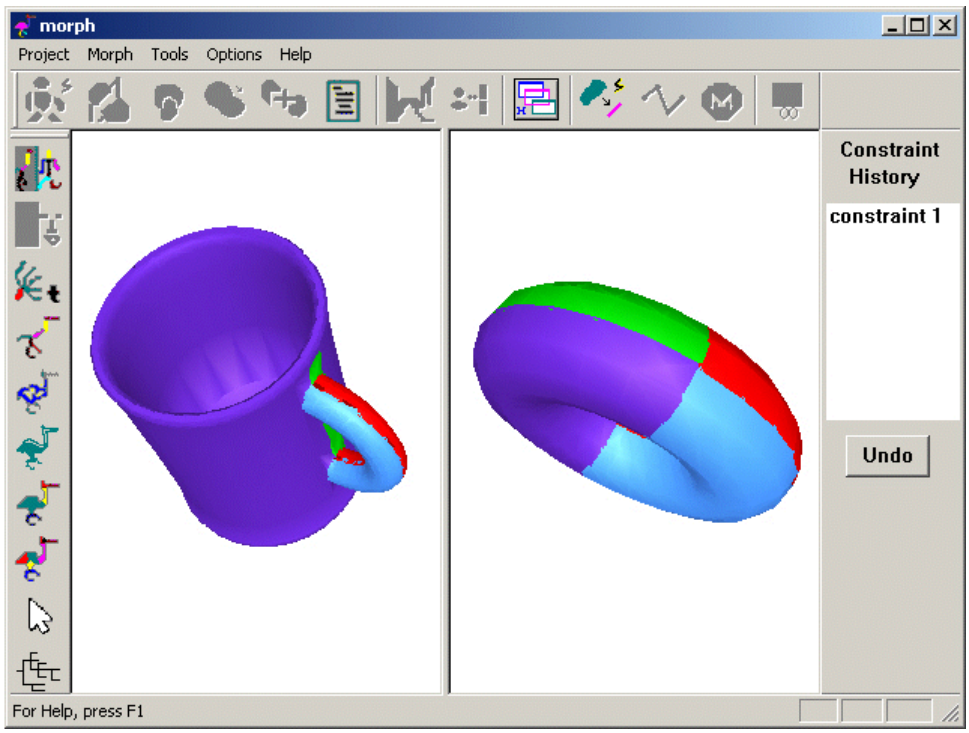


Figure 6.1 Object view

The first one is an *object view* for user control over global-level correspondence. In this view two original meshes are displayed in a side-by-side window. A user can operate on the meshes by performing many kinds of tasks in this view, for example, decomposing meshes into components, specifying/modifying component correspondences, specifying/adjusting skeletons and automatically cutting meshes into compatible patches. Figure 6.1 shows a screenshot of the object view. The window displays patch layouts of two meshes and the dialog bar docked at the right shows the history of user-specified component correspondences.

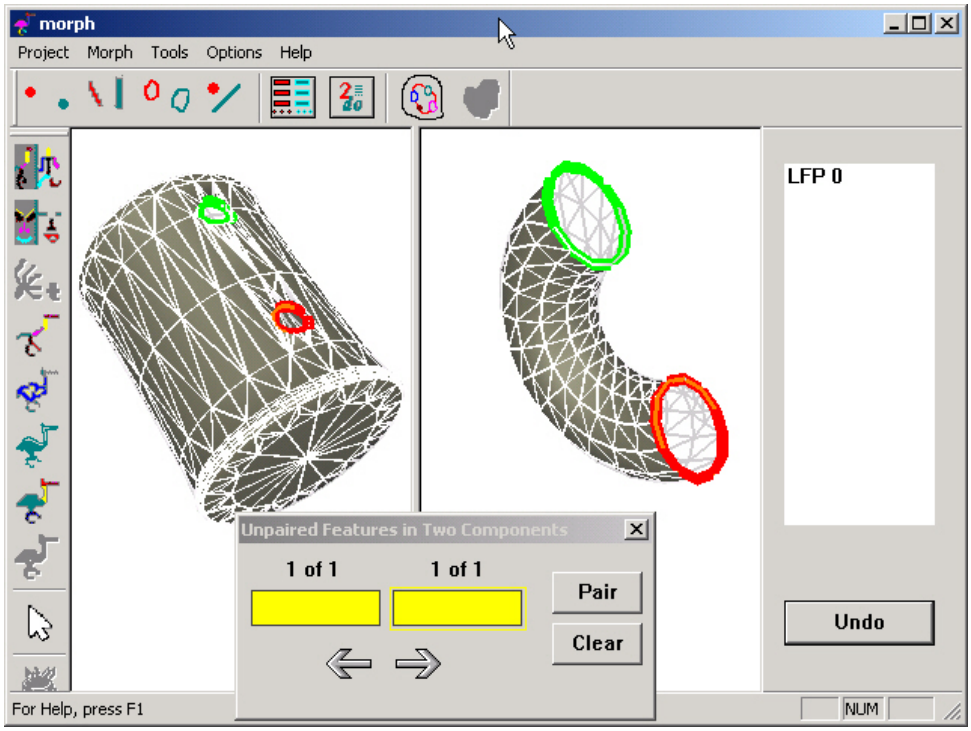


Figure 6.2 Component view

The second view is a *component view* for user control over local-level correspondence. In this view, two corresponding components are displayed in a side-by-side window. Users can either specify/modify local feature correspondences or patch layouts in this view. Figure 6.2 shows a screenshot of the component view. The window displays local feature pairs of two components. The pop-up dialog is for user specification over unpaired component boundaries and the dialog bar docked at the right shows the history of user-specified local feature pairs.

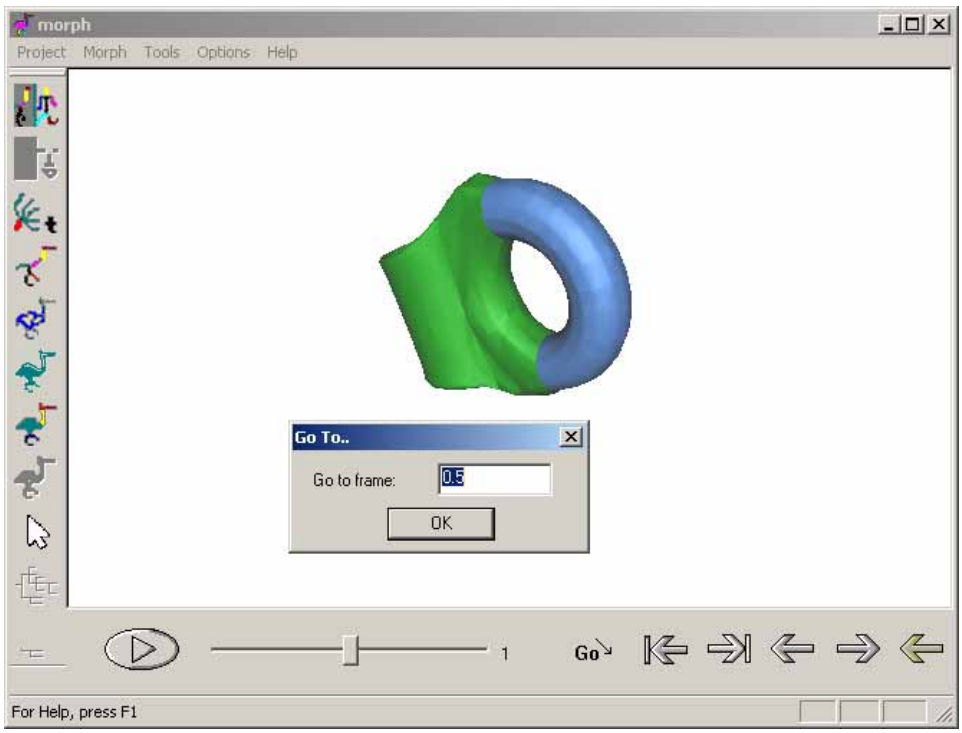
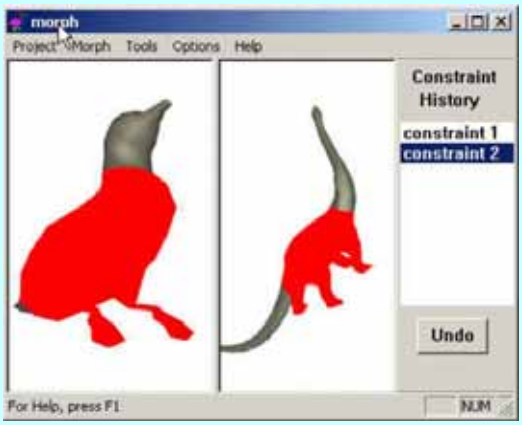


Figure 6.3 Frame view

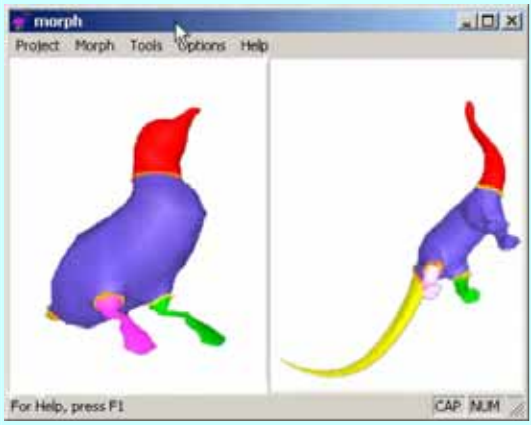
The third view is an *interpolation view* for multi-level user control over interpolation. In this view the result of skeleton morphing and final morphing sequences are displayed, together with a control panel for playing. A user can switch into an intermediate frame to see morphed objects/skeletons, adjust morphing speed and specify keyframes for components or vertices in this view. Figure 6.3 shows a screenshot of the frame view. The window displays an object at an intermediate frame and the pop-up dialog bar is for a user to switch to a specific frame.

6.2 Demo of Whole Morphing Process



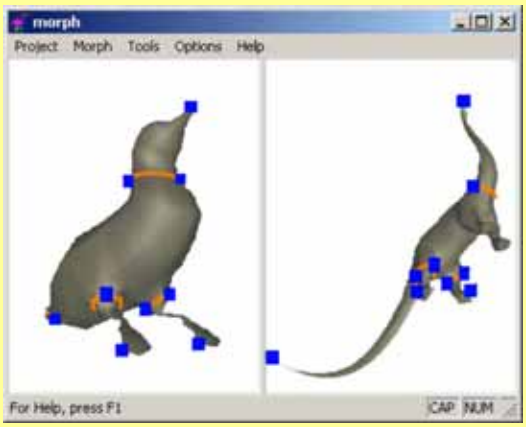
(a) Component pairing

The user only paired components of interest.



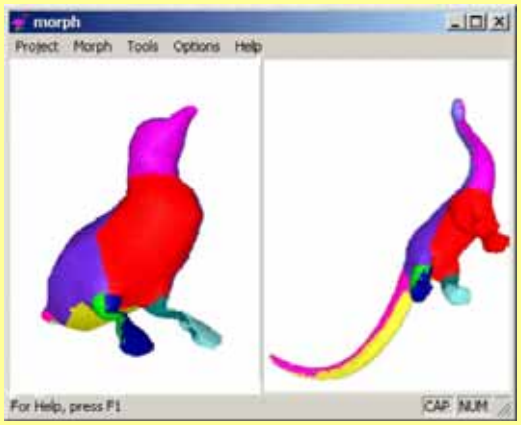
(b) Complete component correspondence

The system deduced the complete component correspondence.



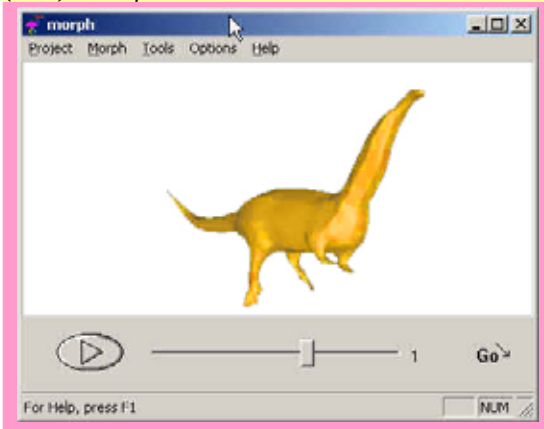
(c) Deduced local feature pairs

The user specified no local feature pair; the system deduced implied (orange) and assumed (blue) local features.



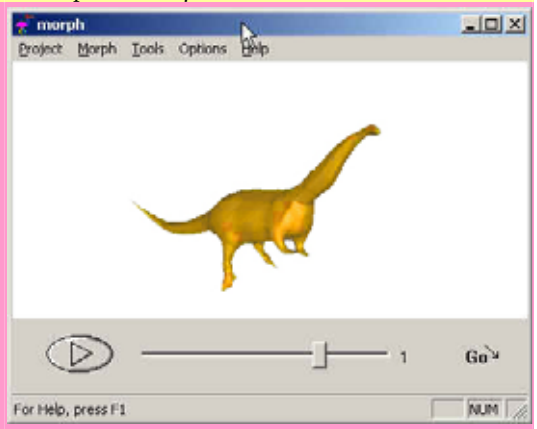
(d) Compatible patch layouts

Through automatic patch partitioning, the system computed the complete vertex correspondence for two meshes.



(e) An intermediate object

The system produced the morphing sequence through vertex interpolation.



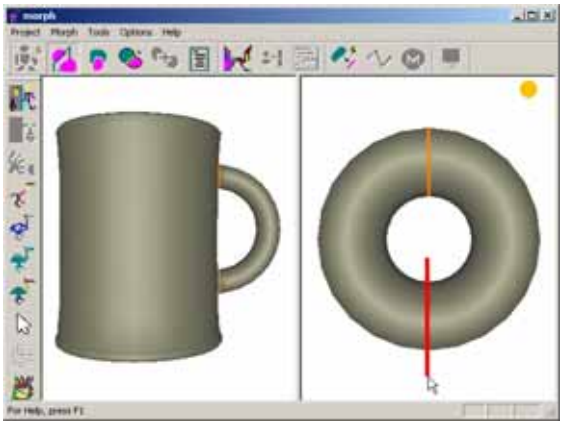
(f) Component keyframe editing

The user manipulated the head component as a whole to edit morphing trajectories.

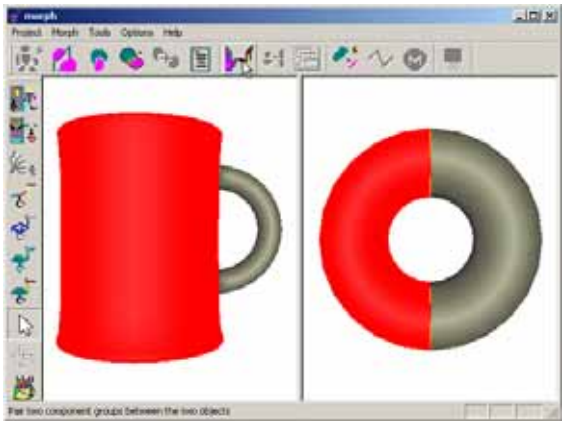
Figure 6.4 A demo of duck-dinosaur morph

For the morph from a duck to a dinosaur, Figure 6.4 demonstrates a complete morphing process with a series of screenshots from the GUI. The user began by decomposing the meshes into components and specifying correspondences over those components of his interest (see (a)). When he finished the specifications of component correspondences, he invoked a system tool and the system deduced the complete component correspondence (see (b)). Then, the user did not specify any local feature pairs and the system deduced implied and assumed local feature pairs (see (c)), generated compatible patch layouts (see (d)), and produced the morphing sequence (see (e)). Then the user adjusted the position of the head at an intermediate frame and the system then updated the morphing sequence accordingly (see (f)).

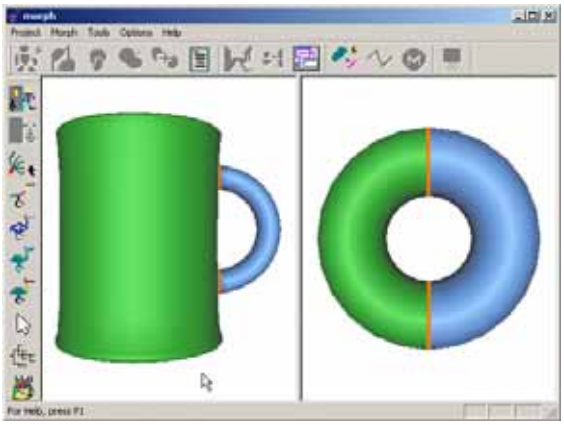
Figure 6.5 demonstrates that a user can easily design a high-genus morph in our framework. Given a mug and a donut, the user decomposed the mug into a body and a handle, and the donut into its left and right (see (a)). At the step of global-level correspondence, the user paired the mug's body with the donut's left (see (b)) and the system automatically paired the other two components (see (c)). At the step of local-level correspondence, the user paired one boundary of the body with one boundary of the left ((see (d)), and the system deduced a set of local feature pairs, as shown in blue in (e). Then, the system automated all other computations and produced the morph successfully (see (f) for an intermediate object).



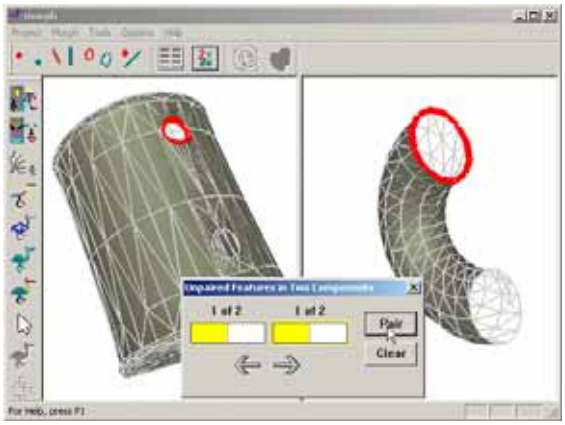
(a) The user cut meshes into components



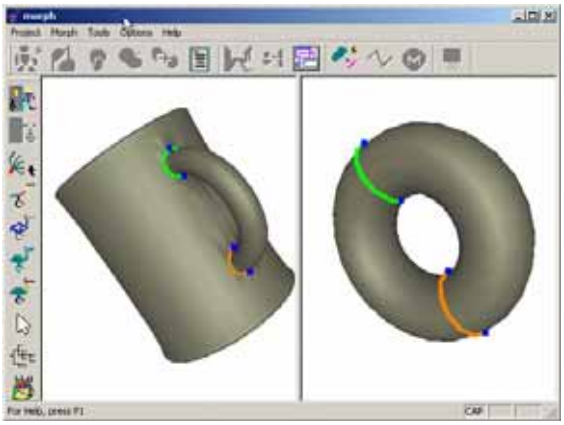
(b) The user paired two components



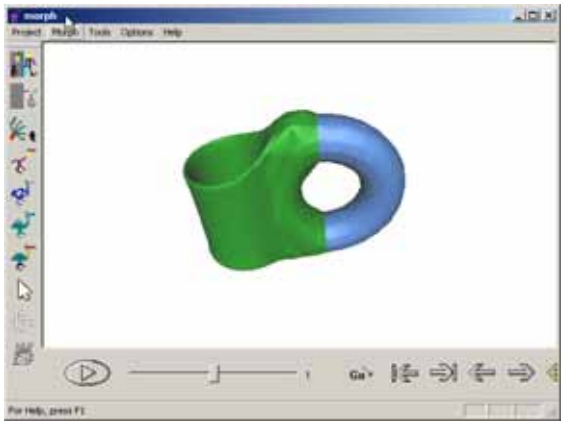
(c) The system paired all components



(d) The user paired two boundaries



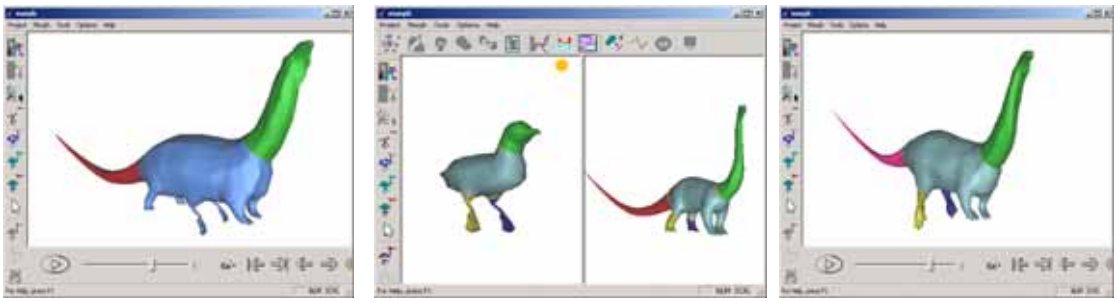
(e) The system deduced local feature pairs



(f) The system produced the morphing sequence

Figure 6.5 A demo of mug-donut morph

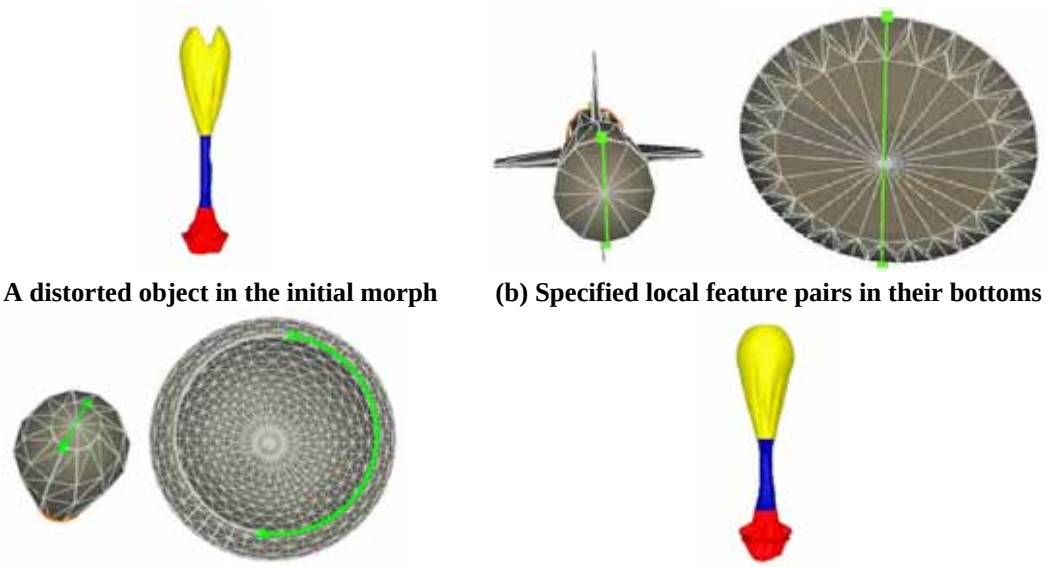
To test the ability of the system to support trial-and-error procedure in interactive morphing design, the user experimented with two morphs after obtaining initial morphs, as shown in the following two figures. In the first example, the user modified global-level correspondence, and in the second, the user added local feature pairs.



(a) An intermediate object having six legs **(b) Added leg components** **(c) A new intermediate object at the same frame**

Figure 6.6 Global-level trial and error morphing design

Given the duck and the dinosaur, after cutting each object into its tail, body and head, the user paired the two heads. The system produced a morph accordingly. Realizing that there were six legs in the intermediate objects, as shown in Figure 6.6(a), the user then went back to specify two legs for each objects and pair their right legs, as shown in Figure 6.6(b). Maintaining all previous user specifications, the system updated the morph correspondingly and a better morph was produced. See Figure 6.6(c) for an intermediate object in the updated morph. The whole process took just a few minutes. This example indicates that in the framework, a user can design a morph at the global level without considering any mesh detail.



(a) A distorted object in the initial morph **(b) Specified local feature pairs in their bottoms**

(c) Specified local feature pairs in their tops **(d) An object in the improved morph**

Figure 6.7 Local-level trial and error morphing design

Figure 6.7 illustrates trial-and-error morphing design at the local level by using the example of morphing a rocket to a glass. After cutting each object into three components, the user specified one pair of components and then the system produced a morph accordingly. Realizing that intermediate shapes were distorted, as shown in Figure 6.7(a), the user then revisited the local-level correspondence step to add two pairs of feature lines and one pair of feature vertices, as shown in green in Figure 6.7(b) and Figure 6.7(c). Subsequently, the system respected all user specifications and produced a better morph. See Figure 6.7(d) for an intermediate object.

From the above two examples, we can see that in our framework, 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. Therefore, the user does not have to complete a large number of specifications to obtain a satisfactory morph.

6.3 Morphing Sequences and Statistics

In each morph reported in this section, two original meshes are different in structure. For example, a cow has two horns while a calf does not have and a triceratops has a tail while a chimpanzee does not have. In this section, we show several morphs and introduce the way of designing them in our framework. In Figure 6.8 to Figure 6.10, corresponding morphs of skeletons are also provided. Components and their corresponding bones are shown in the same colors.

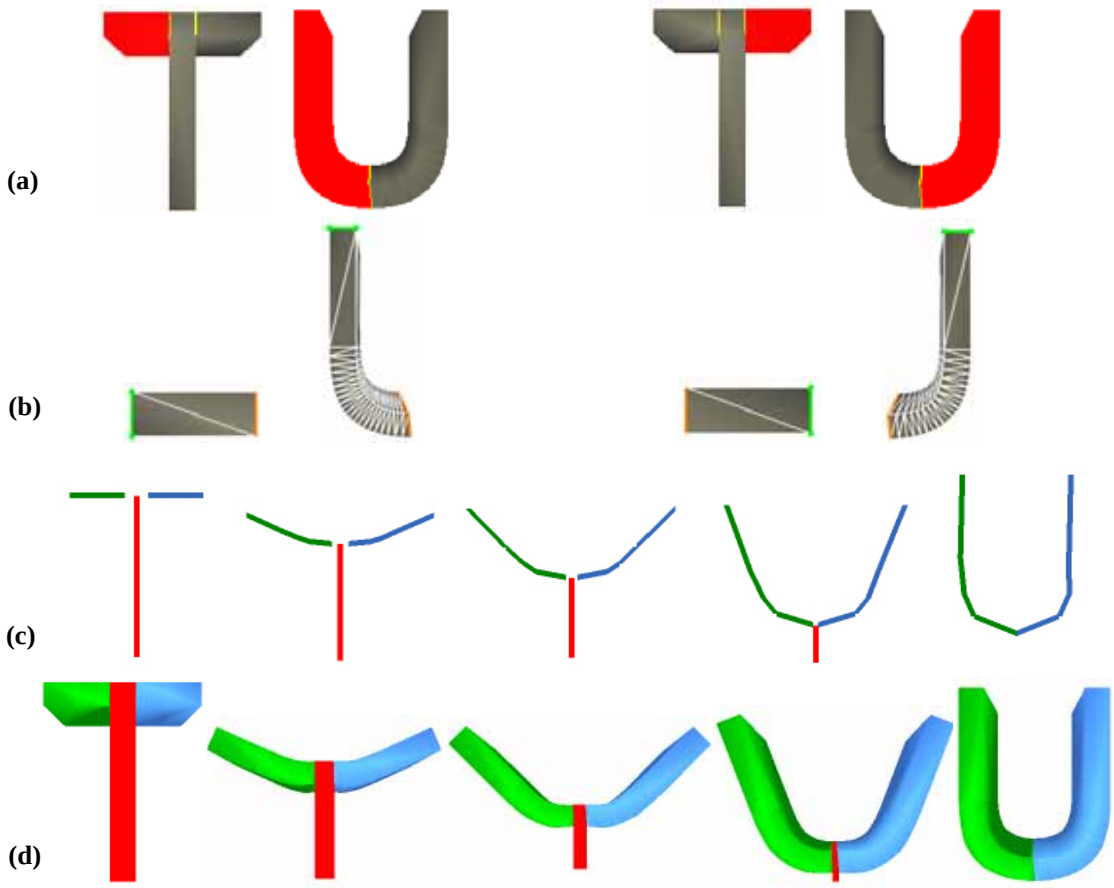


Figure 6.8 T-U morph

For the morph from T to U, the user specified two pairs of components as shown in Figure 6.8(a) and the central-T has no counterpart in U. Then the user specified a pair of feature lines for each specified component pair, as shown in Figure 6.8(b). This example has the special problem of the meta-component for the component pair (central-T, ζ_V) has two adjacent meta-components in the meta-mesh. From the final morph shown in Figure 6.8(d), we can see that the framework can successfully handle such a case to produce a morph where the central component gradually shrinks between the other two components till it completely disappears.

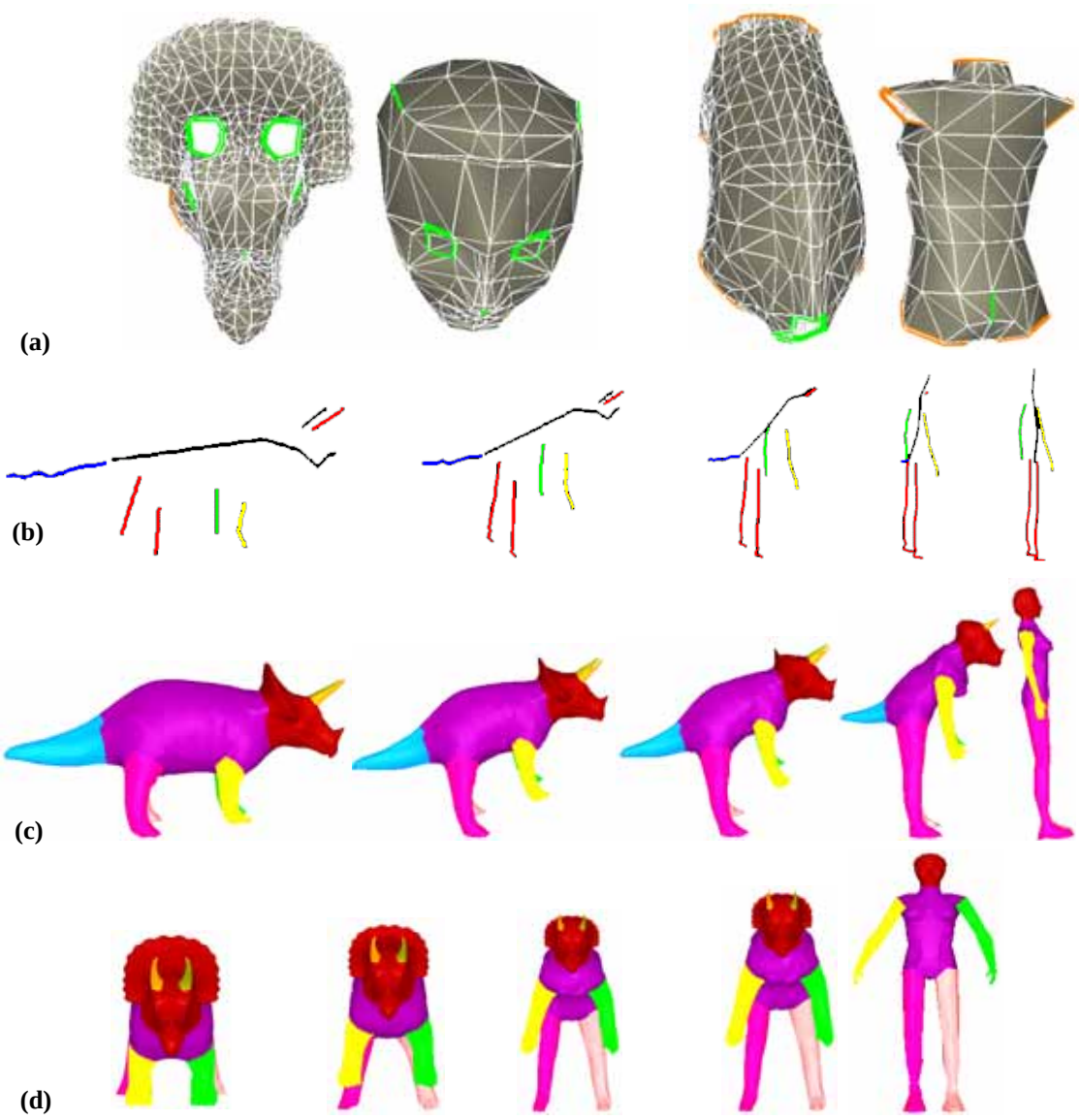


Figure 6.9 Triceratops-woman morph

For the morph between a triceratops and a woman shown in Figure 6.9, user-specified component correspondences were as shown in Figure 4.4. As two horns and a tail of the triceratops have no counterpart in the woman, the user specified three feature lines in the head and the body of the woman. These feature lines are then the corresponding locations of these components. Besides, the user added three local feature pairs to align their eyes and noses. All these user-specified local feature pairs are shown in Figure 6.9(a). In Figure 6.9(c), we can see that the tail and the horns gradually disappear in this morph. Moreover, see how semantic features on the heads

are well aligned at intermediate frames in Figure 6.9(d). It can be seen that our system is able to produce morphs of good visual quality at the same time of providing flexible user control. In other words, the user of our framework can design a satisfactory morph by investing little effort on correspondence specification.

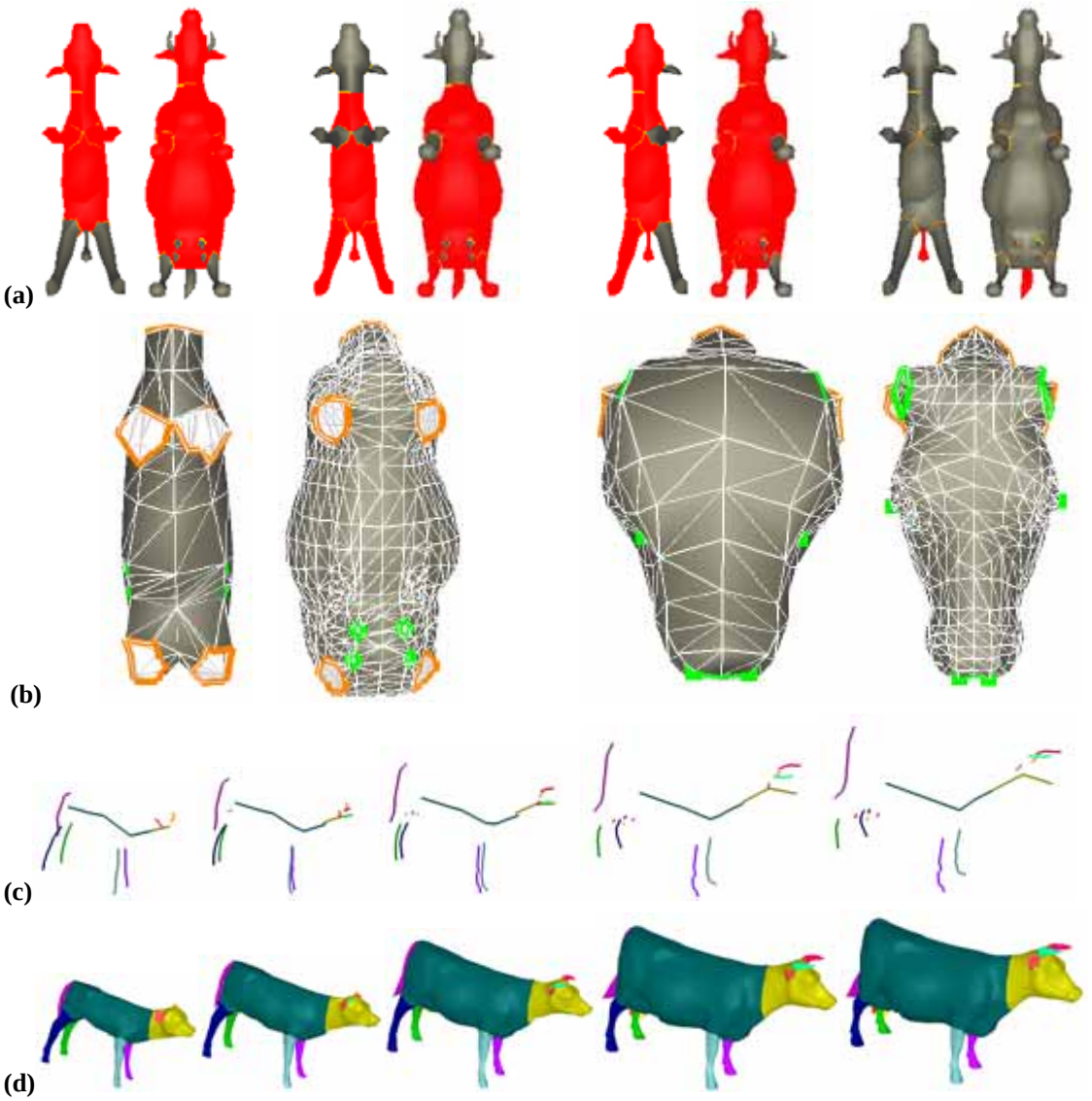


Figure 6.10 Calf-cow morph

In the calf-cow morph shown in Figure 6.10, user-specified component correspondences are shown in Figure 6.10(a). The first is to pair their front parts; the second pairs their rear parts; the third pairs left parts; and the last one pairs their tails. Given these four component correspondences, the framework deduced the complete component correspondence for the nine components of the calf and the fifteen

components of the cow. There are six components in the cow that have no counterpart in the calf — four teats and two horns. Accordingly, the user specified six local feature pairs for the calf to indicate the corresponding locations in the cow for the six components. To align semantic local features on the two heads, the user specified three pairs of local features — two pairs of feature vertices for their eyes and one pair of feature lines for their mouths. Given all these user-specified local feature pairs shown in Figure 6.10(b), the system then deduces eight implied feature loop pairs at component boundaries, seven assumed feature vertex pairs at tails, legs and ears, and twenty-eight assumed feature vertex pairs at component boundaries. The final morph is shown in Figure 6.10(d).

In addition, comparing the morphs of skeletons shown in Figure 6.8(c), Figure 6.9(b) and Figure 6.10(c) with their corresponding final morphs, we can see that the former are good indications of the latter. In cases that two original objects, and thus their underlying skeletons, are different in structure, the morph of skeletons also has the effect of bone disappearing/growing. For the example of T-U morph in Figure 6.8, the bone of the central component gradually shrinks, as shown in Figure 6.8(c). In the calf-cow morph shown in Figure 6.10, the bones of two horns and four teats gradually grow up in the morph of skeletons shown in Figure 6.10(c). All these examples of skeleton morphing indicates that morphs of skeletons serve as good indication of final morphs, and they can be used to effectively guide the final morphs.

Figure 6.11 to Figure 6.13 show more morphs for genus-0 meshes. In the morph from a triceratops to a chimpanzee as shown in Figure 6.11(c), user-specified global-level correspondences are shown in the Figure 6.11(a). In addition, the user specified three local feature pairs to pair their eyes and mouths, as shown in Figure 6.11(b). Note

how the eyes and the mouth of the triceratops are morphed to those of the chimpanzee. Because the tail of the triceratops has no counterpart in the chimpanzee, the user also added another local feature pair to handle such a case, as shown in Figure 4.22.

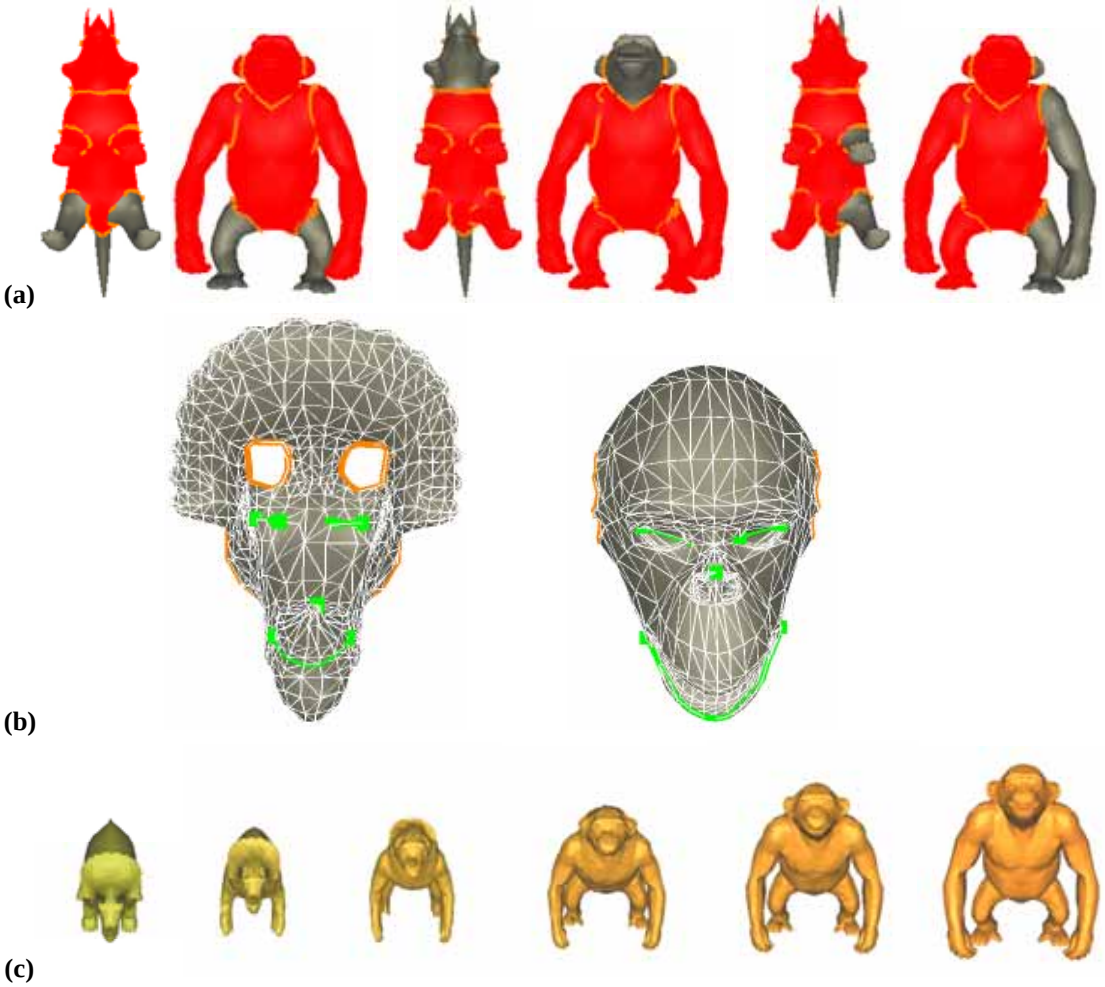


Figure 6.11 Triceratops-chimpanzee morph

For the duck-dinosaur morph shown in Figure 6.12, user-specified global-level correspondences are as reported in Section 6.2. The user did not specify any local feature pairs in this morph. Besides, the user made use of another way to handle the growing effect of two forelegs. Instead of assigning two forelegs of the dinosaur as individual components and pairing each of them with a null-component, the user assigned the body and the two forelegs as one component. Thus, the vertex correspondences for the two forelegs are determined by the step of patch parameterization.

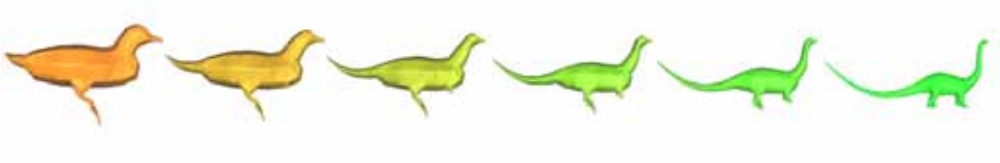


Figure 6.12 Duck-dinosaur morph

In the rocket-glass morph shown in Figure 6.13, user-specified correspondences at the global and the local levels are as reported in Section 6.2. Note how the top and the bottom parts of the rocket are transformed into the body and the base of the glass respectively.

Figure 6.13 Rocket-glass morph

Figure 6.14 shows a morph between two high-genus meshes: a mug and a donut. A demo of the whole morphing process for this morph is as reported in Section 6.2. Note how the mug's body has its inner surface turned out to be a part of the donut, and how the hole in the mug gradually changes into that in the donut.

Figure 6.14 Mug-donut morph

All the above experimental results indicate that our component-based morphing framework can produce smooth transformations with feature preservation, for both genus-0 and high-genus cases. Moreover, due to the effective deduction in the framework, users can perform their morphing design flexibly and conveniently.

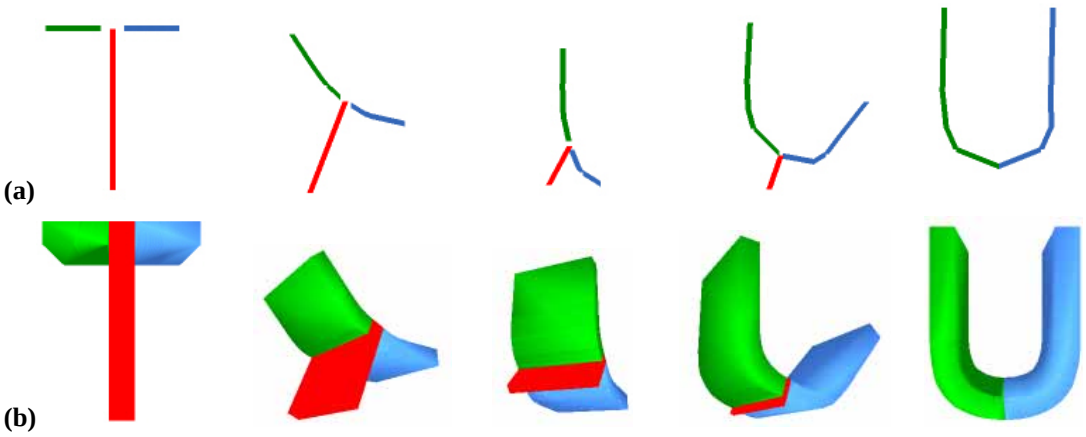


Figure 6.15 T-U morph with a keyframe at f = 0.5

Figure 6.16 A morph with walking effects

Figure 6.15 and Figure 6.16 demonstrate convenient interpolation control in the framework. For the T-U morph shown in Figure 6.15, the user simply specified one keyframe of the meta-skeleton at f = 0.5, as shown in Figure 6.15(a), and the final morph turned to be different, as shown in Figure 6.15(b). Instead of always facing front, the object turned backward to the assigned keyframe in the first half of morphing sequence and then went back toward the frontal position in the second half. The technique of keyframe editing of meta-skeletons can also be used to effectively incorporate additional motions in a morph. See Figure 6.16 for a morph with walking effects, where we combine the morph from a calf to a cow with the morph from the cow to a triceratops. In each morph, the user added two component keyframes for the legs to achieve the walking effects from the calf to the cow then to the triceratops. These two results show that a user can conveniently and easily achieve sophisticated morphing trajectories by specifying a small number of keyframes in skeleton morphing.

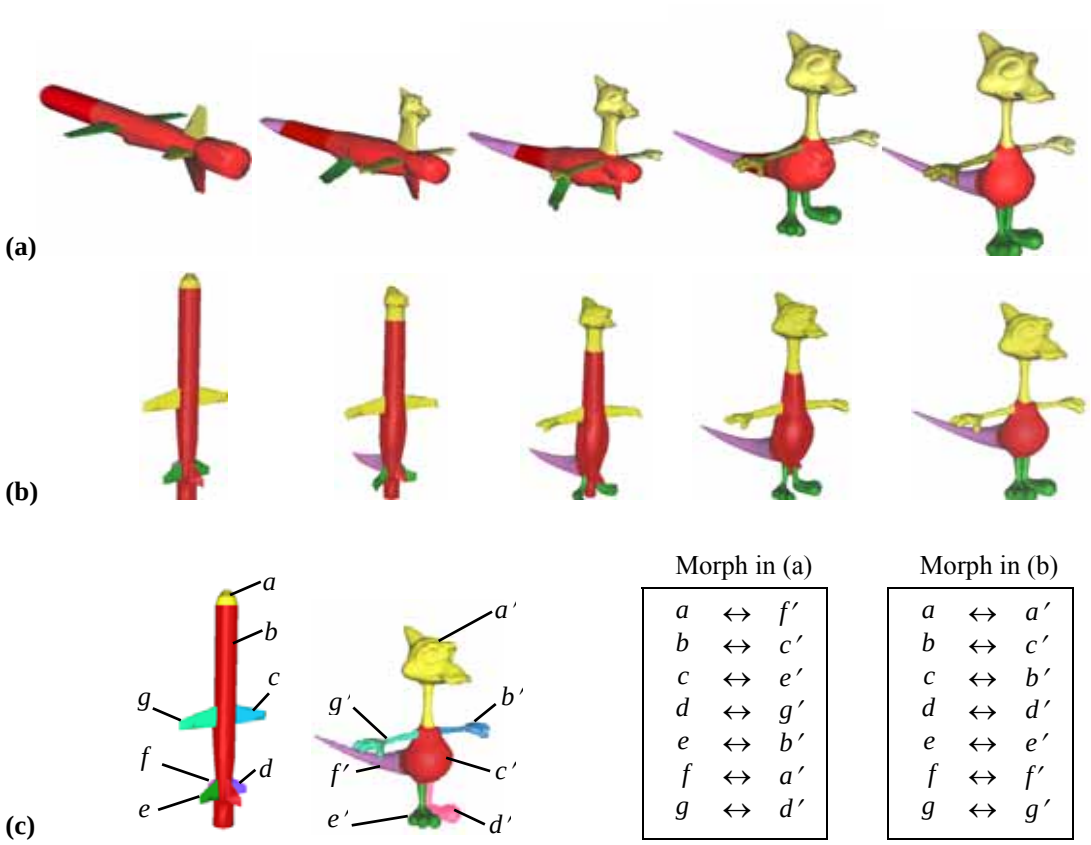


Figure 6.17 Two different morphs using different component correspondences

Figure 6.17 demonstrates the ease of experiencing different morphs by using a rocket and a duckling. Given the same meshes, the user conveniently achieved two interesting morphs shown in Figure 6.17(a) and (b). The component decompositions in these two morphs are the same, and the user assigned different component correspondences for these components, as shown in Figure 6.17(c).

Table 6.1 summarizes statistics of those morphs reported in this section. Besides the model complexity, it reports numbers of user-specified and system-deduced correspondences at both the global level and the local level in these morphs. The user only specified a small number of component correspondences and local feature pairs. The system successfully produced morphs through deduction at both the global level and the local level. In addition, the system's assistance, such as providing candidate components, automatically construction of compatible patch layouts, also makes it

possible for a user to obtain a satisfactory morph with ease. Because the user made no adjustment of morphing trajectories in these morphs, user time on interpolation control is not reported in this table. All these morphs were obtained within a few minutes

Morphs	T-U	Triceratops-Woman	Calf-Cow	Triceratops-Chimpanzee	Duck-Dinosaur	Rocket-Glass	Mug-Donut	Rocket-Duckling	
Triangles in source / target	36 / 328	5660 / 1266	1023/ 5806	5660 / 4912	550 / 5076	330 / 2642	1640 / 576	330 / 3836	
Components in source / target	3 / 2	9 / 6	9 / 15	9 / 8	5 / 5	3 / 3	2 / 2	7 / 7	
Figure number	6.8	6.9	6.10	6.11	6.12	6.13	6.14	6.17(a)	6.17(b)
USER-SPECIFIED component correspondences	2	3	4	3	2	1	1	3	3
USER-SPECIFIED local feature pairs	2	6	9	4	0	3	1	2	3
SYSTEM-DECUCED local feature pairs	6	29	43	27	16	5	5	24	24
Estimated user time on specifying component correspondences	7sec	15sec	20sec	15sec	10sec	5sec	5sec	10sec	10sec
Estimated user time on specifying local feature pairs	10sec	30sec	50sec	20sec	0sec	20sec	5sec	15sec	20sec

Table 6.1: Statistics of examples

We can see from this table that user interaction in our framework is efficient and effective. We invited several non-expert students in our university for testing, and they all reported that it is easy and convenient for them to design morphs in our system.

In this chapter, we introduce our experimental results in the implemented system. From these results, it can be clearly seen that a user can obtain a satisfactory morph by making little effort. In the next chapter, the component-based framework is summarized and our conclusion is given.

Chapter 7 Conclusion

In this chapter, advantages of the component-based morphing framework are first summarized in Section 7.1. Then Section 7.2 discusses key methods in this framework. Several directions for future research are given at the end.

7.1 Summary of Framework

Plenty of methods have been proposed to solve the correspondence and the interpolation problems in mesh morphing. However, user interaction reported in these methods is still cumbersome and far from flexibility. This thesis formulates an interactive framework for component-based morphing to empower users to experiment with morphing design with ease.

In our experiments in the implemented prototype, the user carried out morphing design by specifying only those requirements of his interest and interacting with the morphing design through a trial-and-error process. Besides having practical potential in supporting amateurs in flexible morphing control, the proposed framework has several technical novelties. Specifically, it can be concluded that this component-based morphing framework has the following advantages regarding to the morphing criteria of the ease of user control and the visual quality of morphing sequence.

- **Enable multi-level user control**

In this framework, a user can control the whole morphing process at either the global or the local level, whichever is convenient. At the global level, the user can

specify component decomposition according to his requirements and the decompositions of two original meshes need not be compatible. In addition, the user can pair component groups and modify components in the process of specifying component correspondences. At the local level, the user can specify several kinds of local features to fine-tune a morph and the correspondences over component boundaries can be automatically located by the framework.

- **Facilitate user control through assistance and deduction**

The use of components makes it possible for the framework to easily deduce correspondences from one level to the other. Moreover, several methods are proposed to make user interaction easy. At the global level, the framework makes use of the constraint tree to process user-specified correspondences, deduce probable counterparts for user-selected components, support user modification to decomposition and correspondences, and work out the complete component correspondence. At the local level, the framework deduces implied and assumed local feature pairs based on user specifications, and constructs the complete vertex correspondences through automatic creation of compatible patch layouts for component pairs. It is clear that in this framework, user control is not simply separated into two levels. Instead, this framework frees users from the tedious tasks of specifying detailed vertex pairs in a morphing design. Hence, user control is greatly facilitated and even an amateur can design a morph with ease.

- **Provide effective interpolation control**

Through the use of skeletons, the framework supports effective user control over the interpolation process. Skeleton morphing is achieved soon after the step of global-level correspondence and provides a good indication of the final morph. Thus, user can design the morph at the global level in short turn-around time. Furthermore, by using

the skeleton-guided interpolation method, this framework enables users to control intermediate shapes at both the global level by operation on skeleton morphing and the local level by adjusting vertex trajectories. The above skeleton-based methods also make it possible for the framework to be incorporated into animation systems.

- **Produce natural and rigid morphs**

By performing effective deduction of correspondences, this framework produces morphing results where semantic features are well aligned even when the user only specifies a small number of local feature pairs. By employing skeletons in interpolation, the framework considers both the boundaries and the interiors of objects so that intermediate shapes are rigidly transformed around underlying skeletons. Furthermore, by making use of keyframes, the framework can easily incorporate additional motions to a morphing sequence.

7.2 Discussion of Methods

It is known that user control is essential to achieve good morphing results. Typically, desirable properties for user interaction in a morphing system are as follows.

- **Intuitive**

When users want to specify their morphing requirements, the system should provide them with intuitive ways of specifications. Specifically, requirements of different levels should be directly specified.

- **System-assisted**

A morphing system should not be solely for expert users. It should assist users in their morphing design instead of requiring sufficient working experience. During their specifications, the users should also be informed of potential input mistakes

immediately to avoid painful backtracking at later stages.

- **Intelligent**

Users only need to specify those requirements of interest. The system should be able to derive from user inputs and find implied user requirements. In addition, the system should be able to add reasonable assumed choices where appropriate in order to produce satisfactory morphs.

- **User-preferred**

A morphing system should respect all user specifications, instead of imposing extra restrictions on users due to the limitation of its own. In other words, the assistance and deduction of the system should not contradict with user specifications. Also, assumed choices should be updated when users add/modify their specifications.

- **Flexible**

Users should be allowed to design morphs through a trial-and-error process. When they feel current morphs are unsatisfactory, they should be able to improve the results by simply adding specifications to those unsatisfactory parts, without having to restart from sketch. In addition, after such a modification, the system should let them see the influence of this modification as soon as possible. Thus, users are able to experience different morphing designs conveniently and effectively.

A morphing system having the above properties frees users from the tedious tasks of specifying detailed requirements. Hence, users can focus on important requirements and achieve morphs swiftly. Previous algorithms for mesh morphing only allowed users to operate on vertices for morphing control, and the maintenance and assistance of user specifications was generally ignored. This resulted in heavy user workload in a

morphing design. The component-based morphing framework makes use of the following mechanisms to empower users to conveniently and effectively control the whole morphing process.

First, with the decomposition of a mesh into components, its vertices can be perceived and manipulated in groups. The utilization of components in the framework supports the top-down design approach, which is known as one of the most popular design approaches. Users can carry out a morphing design from high-level conceptual design spaces to low-level technical design spaces. Based on correspondences over components, correspondences over mesh vertices are effectively organized. Moreover, connectivity among components, which is much simpler than that among vertices, is capitalized on in our framework to facilitate user interaction. For example, the system can conveniently deduce local feature pairs at component boundaries from user-specified component correspondences. This makes user interaction in both the correspondence and interpolation steps intuitive and efficient, especially when dealing with meshes of complex structures. In addition, users can still fine-tune morphs by working directly on individual vertices within components.

Second, the framework is designed with the same philosophy of helping users as much as possible and not imposing on users any system-caused restriction. Specifically, in every step of the whole morphing process, the system first gets user specifications, then deduces implied user requirements based on these specifications, and finally adds assumed but reasonable choices. Besides, if a user revisits this step to modify his specifications, the system replaces assumed choices with updated ones, respecting all user specifications. The constraint tree and the deduction of implied and assumed local-level correspondences are examples of realizing this philosophy.

Moreover, there are still some important questions to be answered. The first one is "now that a component and a patch are both a collection of polygons, what is exactly the difference between them?" There are two fundamental differences. First, generally speaking, a patch is homeomorphic to a disk and thus has only one boundary that encloses triangles inside, while a component has a set of boundaries each of which is for an adjacent component. Thus, unlike a patch, a component is not homeomorphic to a disk and its shape is relatively complex. Second, the connectivity among patches is much more complex than that among components. This is because in most cases, a patch connects several patches at its boundary while a component connects only one adjacent component at each boundary.

From the above differences between patches and components, we can find the answer of the second question, "What is exactly the difference between our component-based morphing framework and previous patch-based morphing approach?" Because a patch must be homeomorphic to a disk while a component need not, it is usually difficult and tedious for a user to manually specify the patch layout of a mesh while it is easy and intuitive for the user to specify the component decomposition. More importantly, connectivity among components can be utilized to facilitate user control, whereas patch layouts are too detailed to be a tool of assisting user interaction. Instead, patches usually are results of user control in morphing. Hence, in previous patch-based morphing approach, users must specify enough vertex pairs in order to assist the system to produce morphs through construction of compatible patch layouts. In our framework, however, users first construct the component decompositions of two original meshes, which are not necessarily

compatible; the framework then capitalizes on connectivity among components to deduce the complete correspondence at both the global and the local levels.

Then there is the third question, “For what kind of objects can our framework work well, and for what kind of objects cannot?” There is no special requirement for original meshes in the framework. Just like other works in mesh morphing, original meshes must be orientable and manifold. As this thesis does not discuss the case of genus change, two meshes in a morph should be topologically equivalent. However, because the ease of user control in our framework mainly results form our use of components, advantages of the framework are more obvious when original meshes are with complex structures. For example, for a morph between two heads, the efficiency of user control is not so greatly improved than for a morph between a triceratops and a chimpanzee.

7.3 Future Work

There are several potential extensions in the framework.

- **Support of Complex Component Connectivity**

Currently, we only deal with the simple case where each component only connects one adjacent component at a boundary. To solve the general case of multiple components sharing mesh vertices or edges, the fundamental mechanism for encapsulating user specifications in partitioning constraint tree is still working. However, the representation of a connectivity graph needs to be updated accordingly and how to use them to facilitate user interaction is to be investigated.

- **Handling Topological Change**

Topological change, which includes change of genus, is a challenging issue in morphing research, and the efficiency of user control in such cases is significantly

important. Extension to this problem requires modification to framework mechanisms. For example, a morph with topological changes involves the appearing/disappearing of a connection between two components, while currently a null-connection only appears together with null-components. Consequently, we should develop techniques to handle such changes in connection, and to deduce implied and assumed correspondences over such connections.

- **Integration of Animation Data**

Skeletons in this framework are different from those in animation systems and explicit components do not exist in the latter. The concept of components is usually represented by weights of vertices or other similar attributes in animation systems. Thus, if the framework can be integrated with an animation system, users will be able to morph objects more conveniently.

- **Improvement of Complex Sequence Design**

There are some possible improvements on interpolation issues. First, as components are morphed around their underlying skeletons, it is possible to develop an interpolation method that allows users to define spatial constraints in a transition so that a spatially non-uniform morph can be obtained. Secondly, different morphing rates in interpolation can be explored by using methods such as wavelet transformation. Interpolation of texture coordinates is also very important to create an aesthetic morph. Simply applying the cross-dissolving technique does not always produce pleasing morphs.

- **Combination of IK engine**

Inverse kinematics (IK) is a simple but effective tool in animation systems for motion control. In IK, motion is inherited bottom up in the hierarchy so that a bone at the leaf level can be precisely aligned with a specified target position. The system is

able to automatically adjust other bones in the hierarchy accordingly. Currently, this framework has been able to animate morphed objects by using the keyframe-editing technique. If equipped with an IK engine, this framework will be able to allow users to modify skeletons at intermediate frames more effectively.

References

[A00] M. Alexa, “Merging Polyhedral Shapes with Scattered Features”, The Visual Computer, v16(1), 26-37, 2000.

[A01a] M. Alexa, “Mesh Morphing”, Proceedings of Eurographics, State of the Art Report, 2001.

[A01b] M. Alexa, “Local Control for Mesh Morphing”, Proceedings of Shape Modeling International, 209-215, 2001.

[A02] M. Alexa, “Recent Advances in Mesh Morphing”, Computer Graphics Forum, v21(2), 173-196, 2002.

[ACL00] M. Alexa, D. Cohen-Or and D. Levin, “As-rigid-as-possible Shape Interpolation”, Proceedings of ACM SIGGRAPH, 157-164, 2000.

[ACP02] B. Allen, B. Curless and Z. Popovic, “Articulated Body Deformation from Range Scan Data”, Proceedings of ACM SIGGRAPH, 612-619, 2002.

[AHU83] A.V. Aho, J.E. Hopcroft and J.D. Ullman, “Data Structures and Algorithms”, Addison-Wesley, 1983.

[AJC02] A. Angelidis, P. Jepp and M. Cani, “Implicit Modeling with Skeleton Curves: Controlled Blending in Contact Situations”, Proceedings of Shape Modeling International, 137-144, 2002.

[AM99] M. Alexa and W. Müller, “The Morphing Space”, Proceedings of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, 329-336, 1999.

[B95] J. Bloomenthal, “Skeletal Design of Natural Forms”, Ph.D. Dissertation, University of Calgary, 1995.

[B98] N. Bobick, “Rotating Objects Using Quaternions”, Game Developer Magazine, July 3, 38-39, 1998.

[B99] B. Messmer and H. Bunke, “A Decision Tree Approach to Graph and Subgraph Isomorphism Detection”, Pattern Recognition, v32, 1979-1998, 1999.

[B00] H. Bunke, “Recent Developments in Graph Matching”, Pattern Recognition, v2, 117-124, 2000.

[BJK00] H. Bunke, X. Jiang and A. Kandel, “On the Minimum Common Supergraph of Two Graphs”, Computing, v65(1), 13-25, 2000.

[BL99] J. Bloomenthal and C. Lim, “Skeletal Methods of Shape Manipulation”, Proceedings of Shape Modeling International, 44-47, 1999.

[BN92] T. Beier and S. Neely, “Feature-based Image Metamorphosis”, Proceedings of ACM SIGGRAPH, 35-42, 1992.

[BP98] H. Bao and Q. Peng, “Interactive 3D Morphing”, Proceedings of Eurographics, 23-30, 1998.

[BT97] D. Bordwell and K. Thompson, “The Power of Mise-en-scene”, Film Art, an Introduction, McGraw Hill, 1997.

[BTS00] R.L. Blanding, G.M. Turkiyyah, D.W. Storti and M.A. Ganter, “Skeleton-based Three-dimensional Geometric Morphing”, Computational Geometry Theory and Applications, v15, 129-148, 2000.

[BW76] N. Burtnyk and M. Wein, “Interactive Skeleton Technique for Enhancing Motion Dynamics in Keyframe Animation”, Communications of the ACM, v19, 564-569, 1976.

[BW90] J. Bloomenthal and B. Wyvill, “Interactive Techniques for Implicit Modeling”, Proceedings of ACM Symposium on Interactive 3D Graphics, 109-116, 1990.

[BW01] D.E. Breen and R.T. Whitaker, “A Level-set Approach for the Metamorphosis of Solid Models”, IEEE Transactions on Visualization and Computer Graphics, v7(2), 173-192, 2001.

[CC98] E. Carmel and D. Cohen-Or, “Warp-guided Object-space Morphing”, The Visual Computer, v13(9/10), 465-478, 1998.

[CEF01] H.L. Cheng, H. Edelsbrunner and P. Fu, “Shape Space from Deformation”, Computational Geometry Theory and Applications, v19, 191-204, 2001.

[CGC02] S. Capell, S. Green, B. Curless, T. Duchamp and Z. Popovic, “Interactive Skeleton-driven Dynamic Deformation”, Proceedings of ACM SIGGRAPH, 586-593, 2002.

[CH90] J. Chen and Y. Han, “Shortest Paths on a Polyhedron”, Proceedings of ACM Symposium on Computational Geometry, 360-369, 1990.

[CH01] M.P. Cani and S. Hornus, “Subdivision Curve Primitives: A New Solution for Interactive Implicit Modeling”, Proceedings of Shape Modeling International, 82-88, 2001.

[CLK98] T. Chang, J. Lee, M. Kim and S.J. Hong, "Directed Manipulation of Generalized Cylinders Based on B-spline Motion", The Visual Computer, v14(5/6), 228-239, 1998.

[CP89] S.E. Chen and R.E. Parent, "Shape Averaging and Its Application to Industrial Design", IEEE Computer Graphics and Application, v9(11), 47-54, 1989.

[CSB95] D.T. Chen, A. State and D. Banks, "Interactive Shape Metamorphosis", Proceedings of ACM Symposium on Interactive 3D Graphics, 43-44, 1995.

[DG96] D. DeCarlo and J. Gallier, "Topological Evolution of Surfaces", Proceedings of Graphics Interface, 194-203, 1996.

[DWS93] H. Delingette, Y. Watanabe and Y. Suenaga, "Simplex Based Animation", Proceedings of Computer Animation (Models and Techniques in Computer Animation), 13-28, 1993.

[E99] H. Edelsbrunner, "Deformable Smooth Surface Design", Discrete Computational Geometry, v13, 87-115, 1999.

[FS92] B. Falcidieno and M. Spagnuolo, "Polyhedral Surface Decomposition Based on Curvature Analysis", Proceedings of Workshop on Modern Geometric Computing for Visualization, 57-72, 1992.

[GA96a] E. Galin and S. Akkouche, "Blob Metamorphosis Based on Minkowski Sums", Proceedings of Eurographics, 143-153, 1996.

[GA96b] E. Galin and S. Akkouche, "Shape Constrained Blob Metamorphosis", Proceedings of Implicit Surfaces, 9-23, 1996.

[GDC99] J. Gomes, L. Darse, B. Costa and L. Velho, "Warping and Morphing of Graphical Objects", Morgan Kaufmann, 1999.

[GG95] E. Goldstein and C. Gotsman, "Polygon Morphing Using a Multiresolution Representation", Proceedings of Graphics Interface, 247-54, 1995.

[GH97] M. Garland and P.S. Heckbert, "Surface Simplification Using Quadric Error Metrics", Proceedings of ACM SIGGRAPH, 209-216, 1997.

[GKS98] N.Gagvani, D.R. Kenchammana-Hosekote and D. Silver, "Volume Animation Using the Skeleton Tree", IEEE Symposium on Volume Visualization, 47-53, 1998.

[GL99] E. Galin, A. Leclercq and S. Akkouche, "Blob-Tree Metamorphosis", Proceedings of Implicit Surfaces, 9-16, 1999.

[GSL98] A. Gregory, A. State, M.C. Lin, D. Manocha and M.A. Livingston, “Feature-based Surface Decomposition for Correspondence and Morphing between Polyhedra”, Proceedings of Computer Animation, 64-71, 1998.

[H92] J.F. Hughes, “Scheduled Fourier Volume Morphing”, Proceedings of ACM SIGGRAPH, 43-46, 1992.

[KFK94] J.C. Hart, G.K. Francis and L.H. Kauffman, “Visualizing Quaternion Rotation”, ACM Transaction On Graphics, v13(3), 256-276, 1994.

[HSK01] M. Hilaga, Y. Shinagawa, T. Kohmura and T.L. Kunii, “Topology Matching for Fully Automatic Similarity Estimation of 3D shape”, Proceedings of ACM SIGGRAPH, 203-212, 2001.

[HWK94] T. He, S. Wang, and A. Kaufman, “Wavelet-based Volume Morphing”, Proceedings of IEEE Visualization, 85-91, 1994.

[KCP92] J.R. Kent, W.E, Carlson and R.E. Parent, “Shape Transformation for Polyhedral Objects”, Proceedings of ACM SIGGRAPH, 47-54, 1992.

[KP97] I. Korfiatis and Y. Paker, “A Simple Approach to 3D Object Metamorphosis”, Proceedings of IEEE Conference on Information Visualisation, 32-39, 1997.

[KPC91] J.R. Kent, R.E. Parent and W.E. Arlson, “Establishing Correspondences by Topological Merging: A New Approach to 3-D Shape Transformation”, Proceedings of Graphics Interface, 271-278, 1991.

[KR91] A. Kaul and J. Rossignac, “Solid-interpolating Deformations: Construction and Animation of PIPS”, Proceedings of Eurographics, 493-505, 1991.

[KSK98] T. Kanai, H. Suzuki, and F. Kimura, “Three-Dimensional Geometric Metamorphosis Based on Harmonic Maps”, The Visual Computer, v14(4), 166-176, 1998.

[KSK00] T. Kanai, H. Suzuki, and F. Kimura, “Metamorphosis of Arbitrary Triangular Meshes”, IEEE Computer Graphics and Applications, v20(2), 62-75, 2000.

[KSM99] T. Kanai, H. Suzuki, J. Mitani and F. Kimura, “Interactive Mesh Fusion Based on Local 3D Metamorphosis”, Proceedings of Graphics Interface, 148-156, 1999.

[LCF00] J. P. Lewis, M. Cordner and N. Fong, “Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation”, Proceedings of ACM SIGGRAPH, 65-172, 2000.

[LCS95] S.Y. Lee, K.Y. Chwa, S.Y. Shin, and G. Wolberg, "Image Metamorphosis Using Snakes and Free-Form Deformations", Proceedings of ACM SIGGRAPH, 439-448, 1995.

[LCJ94] F. Lazarus, S. Coquillart and P. Jancène, "Axial Deformations: an Intuitive Deformation Technique", Computer-Aided Design, v26(8), 607-613, 1994.

[LDS99] A.W.F. Lee, D. Dobkin, W. Sweldens, and P. Schroder, "Multiresolution Mesh Morphing", Proceedings of ACM SIGGRAPH, 343-350, 1999.

[LGL95] A. Lerios, C.D. Garfinkle and M. Levoy, "Feature-based Volume Metamorphosis", Proceedings of ACM SIGGRAPH, 449-456, 1998.

[LL02] Y. Lee and S. Lee, "Geometric Snakes for Triangular Meshes", Proceedings of Eurographics, 229-238, 2002.

[LSS98] A.W.F. Lee, W. Sweldens, P. Schroder, L. Cowsar and D. Dobkin, "MAPS: Multiresolution Adaptive Parameterization of Surfaces", Proceedings of ACM SIGGRAPH, 95-104, 1998.

[LV97] F. Lazarus and A. Verroust, "Metamorphosis of Cylinder-like Objects", Journal of Visualization and Computer Animation, v8(3), 131-146, 1997.

[LV98] F. Lazarus and A. Verroust, "Three Dimensional Metamorphosis: A Survey", The Visual Computer, v14(8/9), 373-389, 1998.

[LWT01] X.T. Li, T.W. Woon, T.S. Tan and Z.Y. Huang, "Decomposing Polygon Meshes for Interactive Applications", Proceedings of ACM Symposium on Interactive 3D Graphics, 35-42, 2001.

[M82] J. McGregor, "Backtrack Search Algorithms and the Maximal Common Subgraph Problem", Software Practice and Experience, v12, 23–34, 1982.

[M98] J.S.B. Mitchell, "Geometric Shortest Paths and Network Optimization", The Handbook of Computational Geometry, Elsevier Science, 1998.

[Maya] "Maya Alias|Wavefront", http://www.aliaswavefront.com/en/products/maya/index.shtml.

[MBV97] M.J. Milroy, C. Bradley, and G.W. Vickers, "Segmentation of a Wrap-around Model Using an Active Contour", Computer-Aided Design, v29(4), 299–320, 1997.

[MG01] T.B. Moeslund and E. Granum, "A Survey of Computer Vision-Based Human Motion Capture", Computer Vision and Image Understanding, v81(3), 231-268, 2001.

[MKF01] T. Michikawa, T. Kanai, M. Fujita and H. Chiyokura, "Multiresolution Interpolation Meshes", Proceedings of Pacific Graphics, 60-69, 2001.

[MLP01] K.H. Min, I.K. Lee and C.M. Park, “Component-based Polygonal Approximation of Soft Objects”, Computers and Graphics, v25, 245-257, 2001.

[MW99] A.P. Mangan and R.T. Whitaker, “Partitioning 3D Surface Meshes Using Watershed Segmentation”, IEEE Transactions on Visualization and Computer Graphics, v5(4), 308-321, 1999.

[N82] R. Nevatia, “Machine Perception”, Prentice Hall, 1982.

[PSS01] E. Praun, W. Sweldens and P. Schröder, “Consistent Mesh Parameterizations”, Proceedings of ACM SIGGRAPH, 179-184, 2001

[PTV92] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, “Numerical Recipes in C”, Cambridge University Press, 1992.

[RM99] M. Ramasubramanian and A. Mittal, “Three-Dimensional Metamorphosis Using Multiplanar Representation”, IEEE Multimedia Computing and Systems, v1, 270-275, 1999.

[S85] K. Shoemake, “Animating Rotation with Quaternion Curves”, Proceedings of ACM SIGGRAPH, 245-254, 1985.

[SG01] V. Surazhsky, and C. Gotsman, “Morphing Stick Figures Using Optimized Compatible Triangulations”, Proceedings of Pacific Graphics, 40-49, 2001.

[SGW93] T. Sederberg, P. Gao, G. Wang and H. Mu, “2-D Shape Blending: An Intrinsic Solution to the Vertex Path Problem”, Proceedings of ACM SIGGRAPH, 15-18, 1993.

[SK00] K. Singh and E. Kokkevis, “Skinning Characters Using Surface-oriented Free-form Deformations”, Proceedings of Graphics Interface, 35-42, 2000.

[SR95] M. Shapira and A. Rappoport, “Shape Blending Using the Star-skeleton Representation”, IEEE Computer Graphics and Application, v15(2), 44-50, 1995.

[SRC01] P. J. Sloan, C. F. Rose and M. F. Cohen, “Shape by Example”, Proceedings of ACM Symposium on Interactive 3D Graphics, 135-144, 2001.

[SSB01] T. Surazhsky, V. Surazhsky, G. Barequet, and A. Tal, “Metamorphosis of Polygonal Shapes with Different Topologies”, Computers and Graphics, v25(1), 29-39, 2001.

[ST98] A. Shapiro and A. Tal, “Polyhedron Realization for Shape Transformation”, The Visual Computer, v14(8/9), 429-444, 1998.

[STK02] S. Shlafman, A. Tal and S. Katz, “Metamorphosis of Polyhedral Surfaces Using Decomposition”, Proceedings of Eurographics, 219-228, 2002.

[SWC97] Y.M. Sun, W. Wang and F.Y.L. Chin, “Interpolating Polyhedral Models Using Intrinsic Shape Parameters”, Visualization and Computer Animation, v8(2), 81-96, 1997.

[TT98] M. Teichmann and S. Teller, “Assisted Articulation of Closed Polygonal Models”, Proceedings of Eurographics Workshop on Animation and Simulation, 87-101, 1998.

[U99] M. Unser, “Splines: A Perfect Fit for Signal and Image Processing”, IEEE Signal Processing Magazine, v16(6), 22-38, 1999.

[VL99] A. Verroust and F. Lazarus, “Extracting Skeletal Curves from 3D Scattered Data”, Proceedings of Shape Modeling International, 194-201, 1999.

[W98] G. Wolberg, “Image Morphing: A Survey”, The Visual Computer, v14(8/9), 360-372, 1998.

[WGG98] B. Wyvill, A. Guy and E. Galin, “Extending the CSG Tree (Warping, Blending and Boolean Operations in an Implicit Surface Modeling System)”, Proceedings of Implicit Surfaces, 113-121, 1998.

[WP02] L. Wade and R.E. Parent, “Automated Generation of Control Skeletons for Use in Animation”, The Visual Computer, v18(2), 97-110, 2002.

[ZOT03] Y.H. Zhao, H.Y. Ong, T.S. Tan and Y.G. Xiao, “Interactive Control of Component-based Morphing”, Proceedings of Symposium on Computer Animation, 2003. http://www.comp.nus.edu.sg/~tants/morphing.html

[ZSH00] M. Zöckler, D. Stalling and H. Hege, “Fast and Intuitive Generation of Geometric Shape Transitions”, The Visual Computer, v16(5), 241-253, 2000.

[ZWG02] Z. Zhang, L. Wang, B. Guo and H. Shum, “Feature-based Light Field Morphing”, Proceedings of ACM SIGGRAPH, 457-464, 2002.

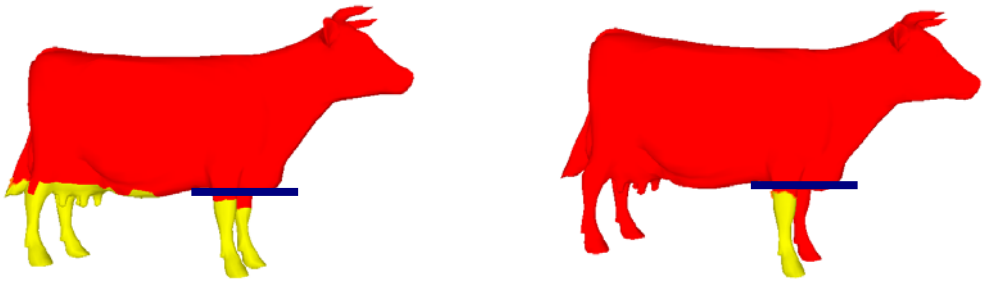
Appendix

Interactive Decomposition

Automatic decomposition methods cannot always meet user requirements. To assist users in specifying desired component decomposition, several interactive tools are provided in our user interface. With these tools, users can modify components in several ways.

- **Cutting a component using a cutting plane**

A user can draw a 2D line segment on the screen and use it to cut a selected component. A line segment defined by the clicking and dragging of a mouse represents a 3D cutting plane that is perpendicular to the *xy* plane in current orientation. A user can adjust the orientation of a 3D model using a mouse, with the help of a trackball-simulation program provided in the user interface. Therefore, the user can cut a selected component by using a specific 3D cutting plane. Note that the 2D line segment drawn by the user also determines the normal of the 3D cutting plane. Consequently, the cutting plane partitions triangles of C into three groups: triangles intersected with the plane, triangles above the plane and triangles below the plane. For the ease of user control, our cutting method puts both intersected triangles and triangles above the plane into one new component C_1 and puts the rest of its triangles into the other, C_2.



(a) Grouping Triangles **(b) Collecting Triangles**

Figure A.1 Cutting a component

[Algorithm A.1] *Cut_Component_By_Collecting_Triangles*

Input: A 2D line segment L which a user draws on the screen, its endpoints a and b and their respective screen coordinates $[x_0, y_0, 0]^T$ and $[x_1, y_1, 0]^T$, current transformation matrix for a specific orientation, a user-selected component C to be cut

Output: New components C_1 and C_2

Step 1: $y_0 = WinHeight - y_0$; $y_1 = WinHeight - y_1$; (*WinHeight* is the height of the screen window[①])
Construct two empty triangle lists *intList* and *abvList*;

Step 2: Calculate the 3D cutting plane P determined by $p_0p_1p_2$, where p_0, p_1 and p_2 are the 3D points that can be projected to a, b and $(\frac{x_0 + x_1}{2}, \frac{y_0 + y_1}{2}, 1)$ respectively[②]. The normal of P is $\frac{(p_1 - p_0) \times (p_2 - p_0)}{|(p_1 - p_0) \times (p_2 - p_0)|}$;

Step 3: Among all the triangles of C, find the triangle T whose projection intersects L and is nearest to a;

Step 4: Mark T and add it into *intList*;

Step 5: For each of the triangles incident to T and lying above P, add it into *abvList*;

Step 6: For each of the triangles incident to T and intersecting P, denote it T' and repeat Step 4 to 5 with $T = T'$ till no more triangles are added to *intList*;

Step 7: From each triangle in *abvList*, collect a set of triangles in C by flooding over all unmarked triangles and add all collected triangles into *abvList*;

Step 8: Put the triangles in *intList* and *abvList* into C_1 and put the remaining triangles of C into C_2.

A simple way for this component cutting is to group triangles of C by computing the distance from every vertex of C to P. That is, if all the three vertices of a triangle are below P, this triangle belongs to C_2. Otherwise it belongs to C_1. However, this usually produces unexpected results where C is cut into more than one component. See the example in Figure A.1(a). The cow model is cut into four disconnected components by a cutting plane: a left foreleg, a right foreleg, a part of its tail and one comprising four teats, two hind legs and a part of its belly. In such cases, users cannot

① Because screen coordinates of mouse positions originate from the upper-left corner of the screen, these coordinates need to be changed so that the new origin is the lower-left corner.

② p_0, p_1 and p_2 can be calculated by using OpenGL function: gluUnProject( ).

control which triangles are needed to form new components. Instead, we apply [Algorithm A.1] to solve such a problem. In this algorithm, we mark a list of intersected triangles near the cutting plane and collect triangles based on mesh connectivity. For comparison, Figure A.1(b) shows the result of this algorithm. It can be seen that the component cutting only happens at those triangles near the cutting plane and results in exactly two components.

- **Assigning a new component by drawing its boundary**

A user can also create new components by sequentially picking mesh vertices over the surface of a selected component. Based on all picked vertices, we form a loop of mesh edges and triangles at different sides of the loop are put into two different components. Obviously, such a loop is then the boundary between the two new components.

To facilitate user interaction, the framework computes the shortest path over mesh edges between every two consecutive user-selected mesh vertices. All calculated paths connecting user-selected vertices form the boundary B between two new components. Triangles of the new components are collected recursively as follows.

[Algorithm A.2] *Assign_Component_By_Collecting_Triangles*
Input: A list of mesh vertices *vList* picked by a user and a selected component C
Output: New components C_1 and C_2
Step 1: Compute the boundary B by calculating the shortest path between every two consecutive vertices in *vList*;
Step 2: Take an arbitrary edge E from B;
Step 3: Suppose E connects two vertices v_0 and v_1, where v_0 is the predecessor of v_1 along B. Find T which is the incident triangle of E and in whose vertex list v_0 is the predecessor of v_1;
Step 4: Add T into C_1;
Step 5: For each edge of T that is different from E, set it as E and repeat Step 3 to 4 till no new triangle is added to C_1;
Step 6: Put the remaining triangles of C into C_2.

- **Merging two connected components**

Using this tool, a user can pick two connected components C_1 and C_2, and merge

them into a new component C. In such a case, all the triangles of C_1 and C_2 are put into C and the boundary between C_1 and C_2 is removed.

- **Passing triangles to an adjacent component**

Using this tool, a user can move a set of triangles at a component boundary from one component to another. Within a selected component C, a user clicks a set of mesh vertices one by one in a counterclockwise order and all these vertices form a closed loop L. It is required that the first and the last user-selected vertices must be on the same boundary of C. The adjacent component C' of the component C can be found as follows. If an edge of L has one of its incident triangles belonging to a component different from C, this component is then be the component C' for this operation. Subsequently, by using an algorithm similar to [Algorithm A.2], we can identify all triangles at the left side of L as triangles to be moved and put them into C'. Note that unlike other tools in this section, passing triangles between components in a mesh does not affect the connectivity graph of the mesh.