Západočeská univerzita v Plzni
Fakulta aplikovaných věd

MORFOVÁNÍ GEOMETRICKÝCH OBJEKTŮ V HRANIČNÍ REPREZENTACI

Ing. Jindřich Parus

disertační práce
k získání akademického titulu doktor
v oboru Informatika a výpočetní technika

Školitel: Doc. Dr. Ing. Ivana Kolingerová

Katedra: Katedra informatiky a výpočetní techniky

Plzeň 2009

University of West Bohemia
Faculty of Applied Sciences

MORPHING OF GEOMETRICAL OBJECTS IN BOUNDARY REPRESENTATION

Ing. Jindřich Parus

doctoral thesis
submitted in partial fulfillment of the requirements for a degree of Doctor of Philosophy in Computer Science and Engineering

Supervisor: Doc. Dr. Ing. Ivana Kolingerová

Department: Department of Computer Science and Engineering

Pilsen 2009

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni. Prohlašuji, že tuto práci jsem vypracoval samostatně s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této práce.

V Plzni dne 24. února 2009 Jindřich Parus

Abstract

Morphing is a technique which transforms one object into another object. It can be used to simulate natural phenomena which involve some kind of shape transformation, or it can be used to produce completely artificial shape transformations used in computer games and movie industry. Alternatively, it can be viewed as a modeling technique which combines some existing shapes to obtain new shapes.

This thesis is focused on morphing of objects given in boundary representation, namely triangular meshes. Triangular mesh is a "native" representation of graphical hardware, it is widely used and due to its piecewise linear nature it is easy to store, modify and render.

The morphing technique is partially covered in professional animation tools. Particularly, the morphing of images has been successfully used in the movie industry to produce special effects. Also the 3d morphing of boundary representation is supported in some 3d animation tools. However, the technique is limited because it allows to morph between objects with the same connectivity. Therefore, it is mainly used with objects which are specially prepared for morphing, e.g., objects deformed by skeletal deformation or free-form deformation. Thus, a technique which handles meshes with arbitrary connectivities is needed.

We approached morphing at several different levels. First, we focus on morphing between meshes with different connectivities. Here, we improved some aspects of a well established technique called topology merging. Next, we focus on a generalization of classical morphing – the multimorphing. We introduce an abstract space of shapes which is motivated by the affine space. We propose a new method for synthesis of new shapes and animations and analysis of existing shapes. Next, we focus on a post-processing stage of deforming meshes animations. At this stage we show our achievements in normal vector computation and collision detection. Last but not least, we introduce core-increment morphing – a new 2d polygon morphing technique which is motivated by the process of growing.

This work has been supported by the following projects:

- Microsoft Research project 2003-187,
- the project FRVŠ 1349/2004/G1, 2004,
- the project FRVŠ 1509/2005/G1, 2005,
- Ministry of Education project LC06008,
- Ministry of Education project MSM 235200005.

Copies of this thesis are available on http://herakles.zcu.cz/publications or by surface mail on request sent to the following address:

University of West Bohemia
Department of Computer Science and Engineering
Univerzitní 8
306 14 Pilsen, Czech Republic

Abstrakt

Morfing je technika používaná pro transformaci tvaru jednoho objektu v druhý. Lze ji použít pro simulaci různých přírodních jevů, jejichž součástí je tvarová změna, nebo pro vytvoření zcela umělých animací pro počítačové hry a filmový průmysl. Morfing lze také chápat jako modelovací techniku, která umožňuje kombinovat existující tvary pro tvorbu nových tvarů.

Tato práce je zaměřena na morfing objektů daných v hraniční reprezentaci, konkrétně na trojúhelníkové sítě. Trojúhelníková síť je často používaná po částech lineární aproximace povrchu, poměrně snadno se ukládá, upravuje a zobrazuje. Jedná se také o „nativní“ reprezentaci grafických karet.

Morfing je podporován i v profesionálních animačních nástrojích. Zejména morfing obrázků je úspěšně používán ve filmovém průmyslu pro tvorbu speciálních efektů. 3d morfing objektů v hraniční reprezentaci je částečně podporován i v 3d animačních nástrojích, avšak omezuje se pouze na objekty se stejnou konektivitou. Je tedy zejména používán na speciálně předpřipravené objekty, např. objekty deformované skeletálními nebo free-form deformacemi. Je tedy zapotřebí metoda, která by dokázala morfovat objekty s libovolnou konektivitou.

K problematice morfingu přistupujeme na několika různých úrovní. Nejdříve se zaměříme na morfing trojúhelníkových sítí s různou konektivitou. Zde jsme se snažili o zlepšení některých aspektů zavedené techniky topologického slučování. Dále se zaměříme na zobecnění klasického morfingu, tzv. multimorfing. Zde ukážeme prostor tvarů jakožto analogii afinního prostoru. Také ukážeme jak vytvářet nové tvary a analyzovat existující tvary. Dále se zaměříme na fázi post-procesingu animace deformujících se trojúhelníkových sítí. V této fázi se zabýváme výpočtem normálových vektorů a detekcí kolizí. V neposlední řadě se zaměříme na core-increment morfing – novou metodu 2d polygonálního morfingu, která je inspirovaná procesem růstu.

Contents

Acknowledgement

At this place I would like to thank to all people who have helped me with my PhD studies. At the first place, I would like to thank to my supervisor Doc. Dr. Ing. Ivana Kolingerová who not only guided me through the PhD studies but also gave me an academic insight which I value the most. Thanks also belong to prof. Václav Skala who provided great conditions under which we could carry out our research. Many thanks also go to Anders Hast with whom we found common interests which resulted in a lot of interesting work. Thanks also goes to Martina Málková, who helped a lot with an experimental verification of proposed ideas and who got interested in morphing in such a way that she carries out her own research in this field now.

Thanks also go to my colleagues from the lab for their willingness to discuss anything anytime. An important feedback, I am very grateful for, was given to me during stays abroad by Borut Žalik, University of Maribor, Slovenia. Last but not least, thanks belong to my family and yet another Martina for their patience, support and understanding.

1. Introduction

Almost every thing in the world changes its shape. Erosion, growth of plants or animals, metal forging, sculpting – all these processes and many others involve some kind of shape transformation. One of the techniques which helps to simulate the shape transformation is morphing. Morphing is a shape transformation which transforms one shape into another. It is essentially an interpolation between two shapes. Generally, the interpolation can be viewed as a tool which fills gaps between some discrete samples. Thus, morphing between an initial shape and a final shape is a continuous sequence of shapes which starts in the initial shape and ends in the final shape. Clearly, the gap between the samples can be filled in many different ways. Therefore, additional interpolation constrains are usually defined. The interpolation constrain usually depends on the application domain.

There are two main areas where the shape interpolation can be used – an animation and a modeling. In the animation, the morphing is used to produce a sequence of shapes. Animations are used, e.g., in scientific visualization, education, entertainment industry, etc. Especially large field, where the computer animation is used, is the movie industry. In the modeling, the morphing is used to produce new shapes by combining some existing shapes. The advantage is that the new shapes do not have to be modeled from scratch; instead the user specifies shares of input shapes, which express how much the given shape contributes to the desired final shape.

The morphing technique has been extensively studied in morphing of 2d raster images which has been successfully used in the movie industry. The problem of 2d morphing is that it is not possible to change the camera position, environment properties (e.g., lighting, shadows) or material properties during the morphing. A 3d morphing goes one step further – instead of interpolation of images of objects, it interpolates the objects and the images are produced (if needed) by rendering the intermediate 3d objects. The 2d techniques are unable to handle correctly visibility, shadows or highlights, because they do not work with a 3d representation of objects. On the other hand, the 3d techniques allow capturing changes of viewing and lighting parameters during morphing.

The 3d morphing is partially covered in professional animation tools. It appears under different names in different products (e.g., Blend shapes in Maya, Morph targets in 3ds max, Posemixer in Cinema4D, Morph Mixer in Lightwave). A common limitation of these techniques is that the morphed objects are required to have the same connectivity, i.e., the same number of vertices and the same number of faces. This limitation avoids

use of morphing on arbitrary objects. Clearly, morphing among arbitrary objects is a hot topic in the research.

In this thesis we focused on objects given in boundary representation. The reason is that the boundary representation is very widespread; it is a "native" representation of GPU and it is easy to store, modify and render. However, some of our techniques (multimorphing) are general enough so that they can be used with another representation as well.

1.1. Contribution of the thesis

Morphing can be viewed as a part of an animation production system. Such system generally consists of three main blocks:

- objects acquisition (e.g., by scanning or modeling) and preprocessing
- animation generation
- animation rendering and post-processing

The contribution of this thesis touches all three blocks of the animation production system. The preprocessing of the input data is done by a topology merging technique (Chapter 4). It is a well established technique used by several authors [Ale00b, Ken92, Kan97]. It is used to convert meshes with different connectivities into meshes with a shared connectivity. It is also a key step to overcome the limitation of commercial applications which are not able to morph between meshes with different connectivities. We describe algorithmic aspects of this technique and we present our original modification.

The result of the topology merging can be imported into a professional animation tool and the rest of the animation production can be done there. Or, our second contribution – the multimorphing (Chapter 5) – can be used. The multimorphing operates in the animation generation stage. The inputs of multimorphing are meshes with the same connectivity. It offers some new ways how to generate shapes and animations. It also shows how to "invert" morphing so that it can be used for shape analysis instead of the usual shape synthesis. While in the morphing an intermediate shape is computed by specifying weights of the input shapes, in the "inverted" morphing the weights of an unknown shape are computed.

The last block of the animation system is touched in Chapter 6 and Chapter 7. Chapter 6 is focused on fast computation of normal vectors of deforming meshes. The normal vectors are important, e.g., in rendering, point containment test, collision tests, etc. Therefore a fast method for normal vector computation is needed. Chapter 7 deals with continuous collision detection of deforming meshes. We focused on fast and robust elementary collision detection tests which include point/plane intersection test and line/line intersection test.

Another contribution of this thesis is a new morphing technique called core-increment morphing. This new technique is focused on morphing of very complicated shapes. The reason why we paid attention to a new morphing technique is that a lot of methods are able to morph quite similar shapes. However, morphing between highly dissimilar shapes is required as well, especially in the entertainment industry or in the art. The

core-increment technique is motivated by a process of growing; therefore it is suitable for growing like morphing animations.

1.2. Organization of the thesis

In Chapter 2 we will introduce a notation which will be used throughout the thesis. It also contains a description of common terms and some definitions. The definitions are presented to fit the context of this thesis, for more general definitions the reader must refer to a corresponding textbook (e.g., [Ebe04]).

Chapter 3 reviews a related work in the area of morphing in general. It presents taxonomy of morphing techniques and it describes common techniques which are shared among different morphing approaches. It briefly reviews the most important approaches in morphing of various object representations. It focuses on morphing of boundary representation in more detail since it is the main topic of this thesis.

The following chapters (4-8) contain a description of our contribution. Since the contribution cover different areas of morphing the chapters are organized so that they are self-contained. Each section contains an introduction, description of related work, description of our contribution and directions and analysis for a future research.

Chapter 4 contains a description of algorithmic technique called topology merging. Chapter 5 describes multimorphing – a tool for shape synthesis and analysis. Chapter 6 discusses fast computation of normal vectors and Chapter 7 shows elementary predicates for continuous collision detection of deformable meshes. Chapter 8 introduces core-increment morphing – a new technique for morphing of highly dissimilar shapes.

Appendixes A – C contain some additional description of a mathematical apparatus used in the thesis. It is in a separate section so that a reader is not overloaded with details when reading the main part of the thesis. However, the equations presented there can be useful when implementing techniques described in the thesis.

2. Notation, terms and definitions

In this section we will describe basic terms and definitions which will be repeatedly used throughout the thesis. If a term appears in a limited scope only it is defined in the text to facilitate the reading.

2.1. Mathematical expressions

In the mathematical expressions we will use following notation:

Example	Notation	Use
$\mathbf{V}$	small boldfaced letters	vectors
$\mathbf{A}$	capital boldfaced letters	points, sets
R	small italic letters	scalar values
Φ	capital Greek letters	mappings
$\lvert\mathbf{A}\rvert$		cardinality of the set $\mathbf{A}$
N	small normal font letters	integer values, number of elements, loop variables

2.2. Terms

A 3d entity will be denoted as an **object**. The term **shape** will be understood as a property of an object (i.e., an object *has* a shape). In some cases, especially when speaking in general about any 2d or 3d entity, the term shape will be used interchangeably with the term object.

A triangular mesh will be shortly denoted as a **mesh**. The mesh consists of vertices, edges and **faces**, where the faces are the triangles of the mesh. The way how vertices are connected by edges will be denoted as **connectivity**, the set of vertices will be referred to as **geometry**.

We will say that an object is **bounded** if it has a boundary. On the other hand, the term **unbounded** object will refer to an object which is closed and it does not have any holes in its surface.

An animation of a shape transformation between two or multiple shapes will be referred to as a **morphing transition**. In the context of morphing between two shapes we will use terms a **source shape** and a **target shape** to refer to the initial and the final shape of the morphing transition, respectively. Generally, the shapes which we want to morph

will be denoted as **input shapes**. The in-between shapes generated by morphing will be denoted **intermediate shapes**. In the context of mesh representation, we will use terms as source mesh, target mesh, input meshes or intermediate meshes to refer to shapes represented as meshes. Similarly, the term source vertex will refer to a vertex of the source mesh; the term target face will refer to a face of the target mesh, etc. In symbolical expressions, the elements regarding the source shape will be denoted with the number zero in the upper index (e.g., $\mathbf{V}_i^0$ refers to an i-th vertex of the source mesh) and the elements regarding the target shape will be denoted with the number one in the upper index (e.g., $\mathbf{f}_j^1$ refers to a j-th face of the target mesh).

A value which controls an amount of morphing between two shapes will be called a **transition parameter** (TP) and it will be denoted *t*. The values of the transition parameter are usually restricted to the canonical interval <0; 1>, i.e., for t=0 the object has the shape of the source object and for t=1 the object has the shape of the target object.

2.3. Definitions

Vector space consists of a set of vectors with two operations: addition and scalar multiplication. The vector space is closed under these two operations. There is one important element called zero vector **o** with the properties that $a.\mathbf{o} = \mathbf{o}$ for all scalars a and $\mathbf{o} + \mathbf{v} = \mathbf{v}$ for all vectors **v**. In the text we will denote vectors as lower case bold letters.

Affine space consists of a set of points and an associated vector space. It has two operations: subtraction of two points yielding a vector in the associated vector space and a point and a vector addition yielding another point in the affine space. In the text we will denote points as upper-case bold letter.

Linear combination is an expression in the form:

$$\sum_{i=1}^{n} a_i x_i \ ,$$

where a_i are the coefficients of the linear combination, x_i are element which are being combined with weights a_i, n is the number of elements in the linear combination.

Affine combination is a special case of a linear combination where the coefficients a_i sum up to one (so called “sum-up-to-one condition”), i.e.:

$$\sum_{i=1}^{n} a_i = 1 .$$

Convex combination is a special case of an affine combination where all coefficients a_i are positive.

Barycentric coordinates express the position of a point with respect to some simplex, i.e., they express the position of a point **P** as an affine combination of vertices $\mathbf{V}_i$ of a simplex. For example, for a simplex in 2d (a triangle), the point **P** can be expressed as

$\mathbf{P} = t\mathbf{V}_1 + u\mathbf{V}_2 + w\mathbf{V}_3$, where t, u, w are barycentric coordinates and $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$ are vertices of the simplex. More, t, u, w are positive and $t + u + w = 1$ for a point inside the triangle, i.e., the point **P** can be expressed by a convex combination of vertices of the simplex with weights given by barycentric coordinates.

Bijection is a function which maps values from a set **A** to a set **B** so that for each element of **B** there is exactly one element in **A**. Also, the bijection defines a **one-to-one correspondence** between sets **A** and **B**.

Genus is a number of handles of an object. Equivalently, the genus is the maximum number of cuts which does not disconnect an object. For instance, a sphere has the genus 0, a torus has the genus 1.

3. Related Work

In this chapter we will review a related work in the area of morphing between two objects. In the sections 3.1 and 3.2 we present taxonomy of morphing approaches. The sections 3.4 – 3.6 contain a brief overview of areas where the morphing is extensively researched. The Section 3.7 describes the related work in the area of mesh morphing. It is more detailed since the mesh morphing is the main topic of this thesis.

3.1. Data representation

Approaches to morphing can be divided according to a dimension (usually 2d and 3d) or a representation of input data. One rough division of data representations in computer graphics is a volume representation and a boundary representation. Additionally, the data representation can be further divided according to the raster/vector nature of the data. The raster representation represents an object by discrete samples (e.g., pixels or voxels) whereas the vector representation represents an object by lines, curves, patches or analytical surfaces.

The volume representation describes an object by enumerating the volume it occupies. It is usually represented by an analytic function or 3d grid of discrete samples. The advantage of this representation is that it defines also an interior of objects. On the other hand, the boundary representation describes just a boundary of an object. Examples are polygons (2d), triangular meshes (3d) or parametric surfaces (3d). The advantage of the boundary representation is that it is usually more economical (from the storage point of view) than volume representation because it enumerates just the boundary of the object and not the entire volume. On the other hand, since the interior is not defined, the boundary representation may cause problems during an object deformation.

Advantages and disadvantages of the object representation project to the morphing approaches as well. The morphing is essentially a deformation; therefore, the morphing of a volume representation must handle an interior of objects, too. For instance, when morphing between objects represented as a 3d grid of voxels, one has to interpolate all voxels comprising the object volume. On the other hand, in the morphing of a boundary representation it is enough to handle only a boundary (e.g., a triangular mesh) of an object. By a deformation of the boundary we must “pretend” a deformation of a volume enclosed by the boundary. The problem is that a volume in the boundary representation is not properly defined. A typical consequence of this problem is a self-intersection of deformed objects in the boundary representation.

Despite the problems with deformation of boundary representation, the boundary representation is widespread. It is mainly because it is easier to capture just the boundary than an entire volume (scanning, modeling). Therefore, in the morphing more attention has been given to algorithms working with boundary representation.

A comprehensive description of different morphing approaches with respect to the input data representation was given by Lazarus and Verroust in [Laz98]. Overview of the most common approaches was also given [Par05]. In this section we will briefly describe the most important methods. Methods which we have developed are described in more detail in later sections.

3.2. Paradigms

Another possible classification of the morphing approaches is a classification according to an algorithmic technique used to compute the morphing. In computational geometry and algorithmic, well known paradigms are divide-and-conquer, sweep line or sweep plane, incremental construction, brute force solution, etc. In the morphing, we identified the following ideas which are common for various approaches:

- physical model,
- decomposition,
- space-time,
- alternative representation,
- dimension reduction.

The physical model paradigm models the shape transformation as some physical process which constrains the deformation. For instance, in [Sed93a] the input polygons were modeled as a piece of wire and the shape transformation is done so that the work needed to bend and stretch the wires is minimized. Din et al. [Din05] model the morphing problem as heat propagation from the source to the target contour. The decomposition paradigm decouples a possibly complicated task of morphing into several less complicated morphing problems. For instance, Shapira and Rappoport [Sha95] decompose the input polygons into star-shaped polygons which are interpolated independently. We also use the decomposition paradigm in our core increment morphing approach (Chapter 8). We decouple a complicated task of morphing of two polygons into several less complicated tasks of morphing of polylines. The space-time paradigm transforms the dynamic morphing problem into a static problem in a higher dimension. Turk and O'Brien [Tur99] placed 2d polygons in a 3d space and computed a smooth 3d surface which interpolates the input polygons. The 3d surface is then cut by a plane to obtain intermediate shapes. An alternative representation paradigm is used to transform the morphing in some representation (e.g., 3d grid) to another representation (e.g., Fourier domain) where it might be easier to solve the morphing problem. The dimension reduction paradigm was used, e.g., in [Kor98], where the input 3d meshes were sliced to obtain 2d cross-sections. The cross-sections were interpolated using some polygon morphing approach and the interpolated cross-sections were then merged to obtain intermediate 3d shape.

In the subsections 3.4-3.6 we will describe a related work in image morphing (2d raster representation), volume morphing, polygon morphing and boundary representation

morphing. When describing the approaches, when applicable, we will indicate paradigms used to solve the morphing.

3.3. Interpolation constrains

Morphing is essentially an interpolation. In general, the interpolation takes some discrete samples and computes a function which fits those samples. In practice, it is usually used to replace discrete samples by a continuous function. There are many different ways how to interpolate data, e.g., simple nearest-neighbor interpolation, polynomial interpolation, radial basis functions, etc. Clearly, the same holds for morphing.

The transformation between two shapes is not unique. Theoretically there is a big number of possible transformations, e.g., a degeneration of the source object into one single point followed by an evolution of the target object or disintegration of the source object to individual faces and transformation of the individual faces into the shape of the target object. The problem is that such a kind of transformation is usually not very visually plausible and so we are looking for some more attractive shape transformations. In [Gom99], there are given some principles for a good morphing. These include:

- topology preservation,
- feature preservation,
- rigidity preservation,
- smoothness,
- monotonicity.

Topology preservation means to preserve topology of the source and the target object, e.g., no holes should suddenly appear during the morphing transition when the source and the target objects are topologically equivalent. Feature preservation refers to the preservation of important features, which are present in the source as well as in the target object during the morphing transition, e.g., when morphing between two animals, legs, heads, tails, etc. should remain aligned during the transition. Rigidity preservation refers to the fact that sometimes a rigid transformation (rotation, translation) is preferred to a soft-body transformation (scaling, shearing, etc.). Smoothness means that the shape transformation should be smooth, avoiding discontinuities. Monotonicity refers to a monotone change of some parameters, e.g., angles should change monotonically avoiding so a local self-intersection. It is important that these principles are strongly application dependent, e.g., in special-effects industry an artificial shape transformation violating some of theses principles is simply more impressive that some completely physically correct transformation, which on the other side would be required in some technical applications.

3.4. Image morphing

A morphing of 2d raster images is probably the oldest form of morphing in computer graphics. It has been successfully used in the entertainment industry to produce special effects and it is supported by many professional image and video processing tools.

The original method was described by Beier and Neely [Bei92]. Their approach uses the physical model paradigm – they model the raster image as a field which is deformed by corresponding line segments. The line segments are used to warp the source and the

target image to produce two intermediate images. The intermediate images are then blended to produce an in-between image. The blending is usually some interpolation of pixel values of the warped images.

Since the 2d raster image is some projection of a real 3d scene, the intermediate stages of morphing of 2d image need not correspond to morphing of a real 3d scene. For instance, when morphing between two shapes with highly specular material. In the 2d morphing, the specular highlights are somehow interpolated between the source image and the final image. On the other hand in the 3d morphing we compute for each intermediate stage of the morphing a complete 3d representation of a shape. Then, the intermediate shape is rendered using some lighting model and the specular highlights are represented exactly according to the 3d shape, lights and a camera position. Additionally, using 3d representation of the intermediate shape we can change a position of camera during the morphing transition, so that it is possible to observe the morphing from different points of view, which is of course not possible in the case of 2d image morphing, where we have just a fixed view of some 3d scene.

3.5. Volume morphing

The idea of Beier and Neely [Bei92] was generalized for 3d grids by Lerios et al. [Ler95]. In their approach, a user delineates corresponding features using pairs of feature elements (points, lines, rectangles and boxes). During morphing, the corresponding feature elements are interpolated. Together with a feature element a certain neighborhood (i.e., voxels) is warped. Similarly to [Bei92] the warped grids are blended to produce an in-between grid. Other approaches incorporate the alternative representation paradigm – they interpolate 3d grids in Fourier [Hug92] or in wavelet [He94] domain, which allows scheduled interpolation of different frequency bands.

Another interesting approach is the so-called *space-time morphing*. It is usually connected with the implicit representation [Pas04, Tur99] but also with tetrahedral meshes [Din05]. The basic idea is that the space in which the input objects (e.g., a 2d space for polygons or a 3d space for meshes) are defined is extended with one more dimension. The added dimension can be considered a time, thus the new space is called *space-time*. For example a 2d point (x, y) is expressed in space-time as a triple (x, y, t). The basic idea is to interpolate n-dimensional input objects by an (n+1)-dimensional smooth surface. The cross-sections of the interpolating surface define an intermediate shape. The advantage of this method is that is handles the topological transformation, e.g., genus change or morphing between sets of disconnected objects. The topological transformation is "for free" given by the interpolation method. On the other hand, the topological transformation is usually hard to control because it is automatically solved by an underlying mathematical apparatus.

3.6. Polygon morphing

The morphing of polygons is usually divided into two parts – a computation of correspondence between vertices of input polygons and a computation of trajectories of corresponding vertices (the so-called vertex path problem). Note that the source and the target polygon need not have the same number of vertices, so some new vertices have to be added.

The computation of the correspondence was addressed by Sedeberg and Greenwood in [Sed93a]. They incorporated the physical model paradigm. The polygon edges are

modeled as wires with some material properties (modulus of elasticity, stretching stiffness constant). Then, the shape transformation involves some stretching and bending work. The goal is to establish such a correspondence, that the work needed to transform the source shape to the target shape is minimized. This algorithm performs well if the input polygons are similar, it can also handle cases when the first shape is a rotated or translated copy of the other shape. It has problems with highly dissimilar shapes, where intersections usually occur.

The vertex path problem was addressed by Sedeberg et al. in [Sed93b]. They used the alternative representation paradigm; they represent a polygon in terms of edge lengths and angles (so-called *edge-angle representation* [Gom99]) instead of the absolute vertex positions. The advantage of the edge-angle representation is that it is invariant to rigid transformation. The absolute vertex coordinates are extracted from interpolated intrinsic parameters. This interpolation scheme avoids edge collapsing and non-monotonic angle changes. This technique was used for generating in-betweens for the animation based on keyframes. The concept of interpolation of intrinsic parameters was also further used for morphing of planar triangulations in [Sur01, Sur04].

Another interesting approach to 2d morphing was introduced in [Sha95]. It combines the divide and conquer paradigm with alternative representation paradigm. It first decomposes the source and the target polygon into *star-shaped* polygons. Then the skeletons of the decompositions are constructed. The skeleton is a planar graph which joins star-points of neighboring star-shaped polygons, i.e., it is a dual graph to the star-shaped decomposition. Important is that the skeletons of the source and the target polygon have to be isomorphic, which requires an isomorphic star-shaped decomposition. Then, the interior and the boundary of the polygon can be expressed relatively to the skeleton. During the morphing, the skeletons are interpolated and the intermediate shapes are reconstructed from the interpolated skeletons. The difference between this approach and previous approaches [Sed93a, Sed93b] is that this approach takes into consideration also the interior of the polygon and not only the boundary.

Alexa et al. [Ale00c] introduced an approach called as-rigid-as-possible shape interpolation. The basic idea is to compute a compatible triangulation of input polygons. The compatible triangulation is a dissection of the source and the target polygon so that the triangulations are isomorphic, i.e., we have one-to-one correspondence between triangles in the source triangulation and triangles in the target triangulation. Then, for each triangle an affine transformation which transforms a source triangle to the target triangle is computed. By interpolation of the affine transformation a source triangle is transformed to the target triangle. The transformation of one triangle influences the transformation of adjacent triangles as well; therefore the transformations for the whole triangulations are computed in a least square sense. Similar approaches were also described by Surazhsky and Gotsman in [Sur01, Sur04]. A challenging issue of approaches based on compatible triangulation is an extension of this idea in 3d, where it requires computing compatible tetrahedronization of input 3d objects.

Another approach is called 2d merging [Gom99]. It is a “generalization” of an algorithm which was originally developed for 3d meshes, e.g., [Ken92, Ale00b]. Input polygons are mapped to the unit disc. Then both mappings are merged, the vertices of the first polygon are mapped on the second polygon and vice versa using an inverse mapping. This results in polygons with the same number of vertices. A linear

interpolation is used to obtain the resulting morphing transition. This technique is suitable for convex, star-shaped or slightly non-convex polygons. For highly non-convex polygons (spirals etc.) it produces self-intersections during the morphing transition.

Johnstone and Wu [Joh02] described an approach to morph two separate polygons into one. The 2-to-1 morphing is a fundamental case in morphing between different numbers of polygons. The basic idea is to merge the two polygons into one and then use some 1-to-1 polygon morphing technique to morph between the merged polygon and a target polygon. The key step is the merging. During the merging the two polygons are morphed towards each other until they meet in one point. Then a curve evolution technique is used to morph the two polygons connected in some point into a more natural shape which is later morphed towards the target shape.

3.7. Mesh morphing

In this section we will review the core idea of mesh morphing. We will not present method by method, instead, we identified subproblems which are common for many methods. These subproblems are – a correspondence computation, a remeshing and an interpolation. We will describe each subproblem in a separate section. This section will be more detailed than the previous sections since the mesh morphing is the main topic of this thesis.

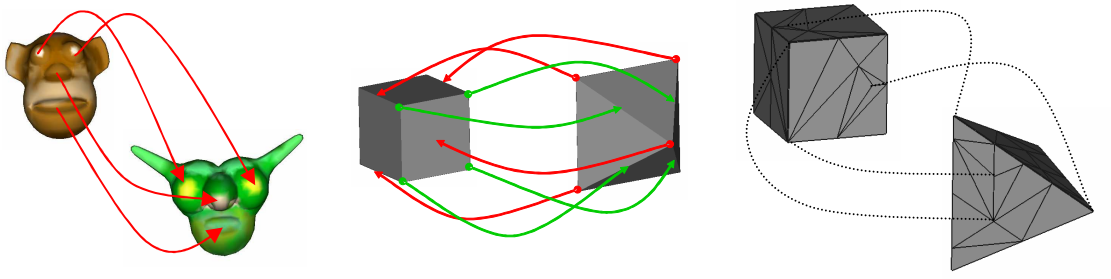
A triangular mesh is probably the most widespread boundary representation. It is supported by a graphical hardware and standard libraries (e.g., OpenGL, DirectX) and it is easy to store and modify. Therefore, a lot of attention has been given to morphing of triangular meshes. In the further text we will denote a triangular mesh shortly as a mesh.

3.7.1. Correspondence computation

Correspondence computation is further divided into two steps – a *feature correspondence* and a *vertex correspondence*. A feature correspondence is usually established by a user. It involves a selection of corresponding features, e.g., when morphing between human faces, a user usually selects eyes, nose, mouth and ears on both input objects. This step ensures that corresponding features remain aligned during a morphing transition, i.e., the eyes of the first object transform to the eyes of the other object, etc. The feature correspondence is usually established by specifying several corresponding vertices (Figure 3.1a). Sometimes, the feature correspondence is not established by a user but it is derived from a mutual position and an orientation of the input objects.

During the mesh morphing we only change the vertex positions, i.e., we do not modify the connectivity. Therefore, it is necessary to compute a correspondence between vertices, i.e., a vertex correspondence, which is guided by the feature correspondence. Formally, we need a map Ψ: $\mathbf{V}^0 \rightarrow \mathbf{V}^1$, where $\mathbf{V}^0$ and $\mathbf{V}^1$ are sets of the vertices of the source mesh and the target mesh respectively. The map Ψ represents a correspondence between vertices $\mathbf{V}_i^0$ and $\mathbf{V}_j^1$, i=1, …, m, j=1,…, n. It is required that the map Ψ is bijective. The main problem here is that the source and the target meshes generally do not have the same connectivity, they may even have a different number of vertices ($m \neq n$), and so it is not possible to establish a bijective map. Instead, two independent maps Ψ_0:$\mathbf{V}^0 \rightarrow \mathbf{P}$ and Ψ_1:$\mathbf{V}^1 \rightarrow \mathbf{Q}$ are computed, which are later merged to compute the map Ψ.

The map Ψ_0 represents a correspondence between vertices $\mathbf{V}_i^0$, i=1, …, m of the source mesh and generally some places $\mathbf{P}_i$ on the surface of the target mesh (i.e., a vertex-place correspondence), analogously, the map Ψ_1 represents a correspondence between vertices $\mathbf{V}_j^1$, j=1, …, n of the target mesh and generally some places $\mathbf{Q}_j$ on the surface of the source mesh (Figure 3.1b). Figure 3.1c) shows an example of vertex-vertex correspondence which is computed by merging Ψ_0 and Ψ_1 (see Section 4).



a) b) c)

Figure 3.1: a) feature correspondence, b) example of a vertex-place correspondence, c) example of a vertex-vertex correspondence.

Technically, the vertex correspondence is established by computing a *parametrization* of input meshes. The parametrization is a mapping Π: $\mathbf{S}\rightarrow\mathbf{D}$ of a 3d surface **S** to a 2d parametric domain **D**. The parametric domain **D** is chosen according to the topology of the input shapes. For shapes topologically equivalent[1] to the unit sphere (e.g., unbounded genus 0 meshes), the surface of the unit sphere is used as a parametric domain, for shapes topologically equivalent to the unit disc (e.g., bounded meshes) the unit disc is used as a parametric domain. A mapping to a planar parametric domain is called planar parametrization; a mapping to spherical parametric domain is called spherical parametrization. Parametric domains for objects with genus higher than one are constructed by adding an appropriate number of handles to the unit sphere, e.g., by adding one handle to the sphere a torus is obtained, which can be used as a parametric domain for genus one objects. In this work we will focus on meshes topologically equivalent to the disc and to the sphere, i.e., we will consider planar and spherical parametrizations. In the case of the planar parametrization edges of the mesh map to 2d line segments and faces map to 2d triangles. In the case of the spherical parametrization the edges of the mesh map to arcs[2], the faces of the mesh map to spherical triangles. In the further text, we will not distinguish between the planar and the spherical parametrization unless necessary. Many operations are common for both types of parametrization[3]; the difference is only in fundamental geometrical computations – e.g., point-in-triangle test, edge-edge intersection, etc. A description of the fundamental spherical geometric operations is in the Appendix A.

[1] In simple terms, topological equivalence of two objects means that one can be deformed to the other only by twisting or stretching but without tearing or cutting.

[2] These arcs are always parts of great circles, i.e., they are the shortest connection between two points on a surface of a sphere.

[3] Note that a point position in 3d can be expressed in spherical coordinates which are essentially three dimensional but since the parametrized points lie on the surface of the unit sphere, they can be represented just by 2 parameters, which makes the spherical parametric domain similar to the planar parametric domain.

A mesh parametrization is computed by mapping its vertices $\mathbf{V}_i$ to the parametric domain. A vertex $\mathbf{V}_i$ mapped to the parametric domain will be denoted $\Pi(\mathbf{V}_i)$. Many methods for parametrization computation exist. Kent et al. [Ken92] used a spherical parametrization. They considered only *star-shaped* meshes. Star-shaped mesh is such a mesh where at least one interior point exists (so-called *star-point*) from which all mesh vertices are *visible*. The term visibility means here that the line segment connecting the star-point and a vertex lies entirely inside the polygon. A subset of star-shaped objects are, e.g., convex objects. A spherical parametrization of star-shaped meshes is computed by projecting the vertices of the mesh to a unit sphere. The vertices of the mesh are projected to the unit sphere using a star-point[4]. Alexa [Ale00b] used the spherical parametrization too but he was able to process general genus 0 meshes. He proposed a *relaxation* scheme which starts with spherical projection and it is further optimized to obtain a valid parametrization. Additionally, he warped the parametrization according to a feature correspondence in order to align corresponding features. In the work of Zockler et al. [Zoc00] the input shapes are dissected into patches which are parametrized independently. The dissection is based on a feature correspondence.

Whereas the mapping Π: $\mathbf{S}\rightarrow\mathbf{D}$ is usually computed using some algorithm, the inverse mapping Π^{-1}: $\mathbf{D}\rightarrow\mathbf{S}$ of points from a parametric domain to points of a 3d surface is computed using barycentric coordinates. Each point $\mathbf{Q}\in\mathbf{D}$ of the parametrization can be expressed by barycentric coordinates u, v, w with respect to some triangle $\Pi(\mathbf{A})$, $\Pi(\mathbf{B})$, $\Pi(\mathbf{C})$ in which the point $\mathbf{Q}$ lies. Note that $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ are vertices of the 3d surface. Then the coordinates of the point $\mathbf{Q}$ on the 3d surface are computed as:

$$\Pi^{-1}(\mathbf{Q}) = u.\Pi(\mathbf{A}) + v.\Pi(\mathbf{B}) + w.\Pi(\mathbf{C}),\ u + v + w = 1 \qquad (3.1)$$

As some steps of the mesh morphing are done in the parametric domain we need the inverse mapping to "project" vertices and edges computed in the parametric domain back to the 3d mesh. The inverse mapping is also the core idea behind remeshing using a parametrization [Mic01].

The parametrization is not only useful in mesh morphing but also in texture mapping, interactive 3d painting, remeshing, geometry processing, etc. In the mesh morphing it is important that the input meshes are mapped to a common parametric domain. This implies a restriction that both meshes have to be topologically equivalent. The key problem of morphing between shapes with different topology (e.g., a sphere to a torus) is a discontinuity during the change of topology.

To compute a vertex-place correspondence, the parametrizations must be overlaid, i.e., the parametrizations are put each over other. If a parametric domain is a unit disc, the discs are moved so that their centers coincide. Analogously, if a parametric domain is a unit sphere, the spherical parametrizations are overlaid by moving spheres so that their center coincide (Figure 3.2a). First, we will describe how to compute the vertex-place correspondence for the vertices of the source mesh. Each vertex $\Pi(\mathbf{V}_i^0)$ of the parametrization of the source mesh lies in some triangle $\Pi(\mathbf{f}_j^1)$ of the parametrization of

[4] If the mesh is translated so that the star-point is in an origin of a coordinate system, then the projection is computed simply by normalization of vertex positions.

the target mesh. In the case of a planar parametrization, it can be checked by a standard point-in-triangle test; in the case of spherical parametrization, a specialized point-in-spherical-triangle test must be used (see Appendix A). The position of the vertex $\Pi(\mathbf{V}_i^0)$ is expressed by barycentric coordinates *u*, *v*, *w* with respect to the triangle $\Pi(\mathbf{f}_j^1)$ [5]. The inverse mapping (Eq. 3.1) is used to compute the corresponding place of the vertex $\mathbf{V}_i^0$ on the surface of the target, i.e.:

$$\mathbf{P}_i^1 = u.\mathbf{A} + v.\mathbf{B} + w.\mathbf{C}\,, \tag{3.2}$$

where the point $\mathbf{P}_i^1$ is a point on the surface of the target mesh which corresponds to the vertex $\mathbf{V}_i^0$ of the source mesh, **A**, **B**, **C** are the vertices of the triangle $\mathbf{f}_j^1$. The situation is also depicted in Figure 3.2. There is a vertex $\Pi(\mathbf{V}_i^0)$ which lies inside the triangle $\Pi(\mathbf{f}_j^1)$ formed by the vertices **A**, **B**, **C**.

In a similar way, corresponding places for vertices of the target mesh are computed. Each vertex $\Pi(\mathbf{V}_k^1)$ of the parametrization of the target mesh lies in some triangle $\Pi(\mathbf{f}_l^0)$ of the parametrization of the source mesh. Again, barycentric coordinates of the vertex $\Pi(\mathbf{V}_k^1)$ are computed with the respect to the triangle $\Pi(\mathbf{f}_l^0)$, the barycentric coordinates computed in the parametric domain are used to compute a corresponding place $\mathbf{P}_k^0$ of the vertex $\mathbf{V}_k^1$ on the surface of the source mesh.

Algorithmically, the vertex-place correspondence computation is a point location problem. For each vertex $\Pi(\mathbf{V})$ of one mesh we have to determine a triangle $\Pi(\mathbf{f})$ of the other mesh in which the vertex $\Pi(\mathbf{V})$ lies. Many algorithms for point location in a planar subdivision exist. Note that even for a spherical parametrization the point location is possible, because vertices mapped to the surface of a sphere always lie in some spherical triangle. Also note that in the case of spherical parametrization almost the same algorithms as in the planar case can be used, it is only necessary to change some elementary predicates – e.g., a point-in-planar-triangle test replace by a point-in-spherical-triangle test, an edge-edge intersection replace by an arc-arc intersection, etc. A detailed description of the point-in-spherical-triangle test and the arc-arc intersection test is in the Appendix A.

[5] The computation of barycentric coordinates of a point with respect to a spherical triangle is described in the Appendix A.

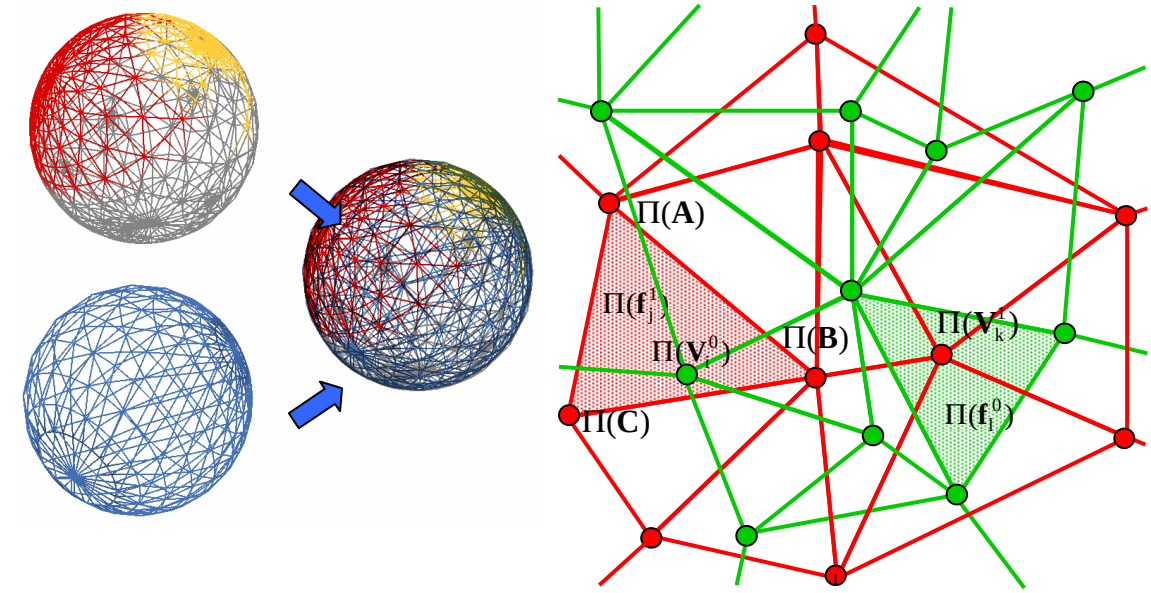


Figure 3.2: a) an overlying of the source parametrization (top) and the target parametrization (bottom), b) a detail of the overlaying planar parametrization, the red triangulation is the parametrization of the source mesh and the green triangulation is the parametrization of the target mesh.

3.7.2. Remeshing

After the vertex-place correspondence is computed, it is not possible to interpolate between input meshes yet. To be able to interpolate between the input meshes, a correspondence between vertices (i.e., a vertex-vertex correspondence) must be computed. Since the input meshes have a different number of vertices, it is not possible to compute one-to-one correspondence between the vertices of the input meshes. Therefore, the input meshes must be refined in order to be able to compute one-to-one correspondence.

Hence, a general idea of the remeshing step is to construct a new mesh (called *supermesh*) by refining one of the input meshes so that it is possible to transform the new mesh to the shape of the source mesh as well as to the shape of the target mesh. Of course, the remeshing takes into consideration the previously established vertex-place correspondence. Several approaches to compute the supermesh exist. Michikawa [Mic01] used a subdivision scheme to remesh both input meshes. Since the subdivision scheme is the same for both input meshes it results in two meshes with the same connectivity. Kraewoy and Scheffer [Kra04] remeshed the target mesh with the connectivity of the source mesh and later they optimized both the source and the target mesh so that the new mesh represents sufficiently the shape of the source mesh as well as the shape of the target mesh. Kent et al. [Ken92] remeshed the target shape by inserting edges of the source mesh. This approach is described in more detail in the Section 4 together with our original modifications.

3.7.3. Interpolation

In the previous step a new mesh which is possible to transform to the shape of the source mesh as well as to the shape of the target mesh was constructed. The morphing transition is done by interpolating the vertex positions. The simplest way how to interpolate between two vertex positions is a linear interpolation, i.e. the vertices travel along a line connecting corresponding vertices. This simple approach is used in the

majority of morphing approaches. As stated in [Ale01a], the linear interpolation works well for morphing of objects which are rather similar and are oriented in a similar way. For objects with different shapes the linear vertex interpolation may introduce a self-intersection or some sort of collapsing, which is usually not a very plausible effect.

An interpolation of higher degree is also possible. It yields a smoother vertex path, but on the other side it requires adding some information, e.g., in the form of tangents for Hermite interpolation, control vertices for Bézier interpolation, etc. For instance, Michikawa et al. [Mic01] suggest using vertex normals as tangents for Hermite interpolation. Gregory et al. [Gre99] suggest specifying tangent vectors for some vertex paths. The modified trajectory is then spread with some falloff to the neighboring vertices.

Besides the methods which interpolate between corresponding vertices, there are so-called *intrinsic interpolation* methods which take into account also intrinsic shape parameters. A basic idea of intrinsic interpolation methods is to represent a mesh in some alternative representation, then interpolate the alternative representation and convert the interpolated form back to a mesh representation to obtain an intermediate shape. The alternative representation must unambiguously represent the original mesh. The interpolation of an alternative representation is usually easier or it has better results than direct interpolation of original representations. Methods based on the intrinsic interpolation require a forward transformation from the mesh representation to an intrinsic representation and a backward transformation which transforms an interpolated intrinsic representation back to the original mesh representation. The backward transformation is usually harder to compute. In 2d, examples of intrinsic representation are the edge-angle representation [Sed93b] or star skeleton representation [Sha95]. In 3d we will briefly describe Laplacian representation and an analogy of edge-angle representation for 3d meshes.

In the Laplacian representation a vertex $\mathbf{V}_i$ is represented as follows. First a center of mass $\mathbf{C}_i$ of one-ring neighborhood[6] of the vertex $\mathbf{V}_i$ is computed. The Laplacian representation[7] $\mathbf{l}_i$ of $\mathbf{V}_i$ is the difference between $\mathbf{C}_i$ and $\mathbf{V}_i$, i.e., $\mathbf{l}_i = \mathbf{C}_i - \mathbf{V}_i$. Thus, the forward transformation is simple; the backward transformation involves computation of a linear system. Alexa [Ale01b, Ale03] used this representation to morph between isomorphic meshes. Instead of interpolating absolute vertex coordinates, he interpolated linearly the Laplacian coordinates. The interpolated Laplacian coordinates are transformed backwards in order to obtain absolute vertex coordinates. The advantage of the Laplacian representation is that it is translation invariant. Therefore they are suitable for morphing of features which are not aligned in space. Laplacian coordinates were also used in the mesh editing [Lip04] where a mesh is deformed by a handle which influences some specified region of interest.

Sun et al. [Sun97] showed an intrinsic representation similar to edge-angle representation [Sed93b]. Their approach considers isomorphic meshes. First, a *vertex adjacency graph* (a graph representing vertices of the mesh connected by edges, denoted as VAG) and its dual, a *face adjacency graph* (a graph representing adjacent faces of the mesh, denoted as FAG) for the source and the target mesh are constructed.

[6] One-ring neighborhood of a vertex **V** is a set of vertices which are connected with the vertex **V** by an edge.

[7] Some authors refer to this representation as a differential representation.

A FAG contains face normals in its nodes and each edge of the graph contains a flag indicating whether two incident faces form a convex or a concave dihedral angle or whether they are coplanar. Note that face adjacency does not represent a mesh uniquely, so an additional geometric representation is needed to make the FAG a complete representation of a mesh. The interpolation then goes as follows. First the FAG is interpolated, which means an interpolation of face normals. It is started with two initial faces; the remaining normals are computed by propagation along edges of the FAG. The result of the FAG interpolation is establishing of an intermediate orientation of faces. Then the VAG is interpolated so that the vertices fit the already oriented faces.

4. Topology merging

In this section we will describe the method of topology merging. First, we will describe the original method – we will show that the underlying geometrical operation is an edge insertion and we will discuss its algorithmic aspects (Section 4.2). Then we will show our contribution where we extended this method in the following directions: merging of meshes with attributes (Section 4.3), improvements of quality of the resulting mesh (Section 4.4) and a generalization for multiple meshes (Section 4.5). The sections 4.4 and 4.5 do not describe a finished research; it rather contains a problem description and a suggestion how to solve it. In some cases we also included some preliminary results which indicate that this direction is worth researching in the future.

4.1. General idea

The basic idea behind the topology merging is to insert edges of one mesh into the other mesh. Without loss of generality, we will insert edges of the target mesh to the source mesh[8]. The result is a mesh which shares connectivity of both input meshes. We will refer to this mesh as a *supermesh*[9]. Edges are inserted so that the shape of the mesh is not altered, but the connectivity allows transforming from one shape to the other shape. To be able to insert an edge of the target mesh into the source, first we have to determine *where* to insert it. An edge is defined by two endpoints; therefore, we have to determine where to insert its endpoints. The endpoints are inserted at places defined by the vertex-place correspondence computed in the correspondence computation step (i.e., the mapping Ψ_1 defined in Section 3.7.1).

After an edge of the target mesh is inserted into the source mesh, it "disappears" in the surface of the source mesh, i.e., it does not disturb the original shape of the source mesh. But it can be moved towards its original position in the target mesh to represent some feature of the target mesh. The basic idea of the topology merging is demonstrated in Figure 4.1. There is a source mesh (Figure 4.1a) and a target mesh (Figure 4.1b). The red edges of the target mesh are inserted into the source mesh. It is demonstrated in Figure 4.1c) – it can be seen that the edges of the target mesh are inserted in the source mesh so that the shape of the source mesh is not changed, i.e., the red edges are "wrapped" along the 3d surface of the source mesh. Figure 4.1d) shows the supermesh transformed to the shape of the target mesh. The transformation is done by moving the

[8] It can be done in the reversed order as well.

[9] In the literature the supermesh appears also under the terms metamesh [Lee99], combination mesh or an interpolation mesh [Kan97, Kan99].

edges of the target mesh (picked out in red) towards its original position while the edges of the source mesh (picked out in blue) “disappear” in the surface of the target mesh. The places where the edges of the source mesh “disappear” are given by the vertex-place correspondence between the vertices of the source mesh and the surface of the target mesh. Figure 4.1e) shows an example of an interpolation of the supermesh.

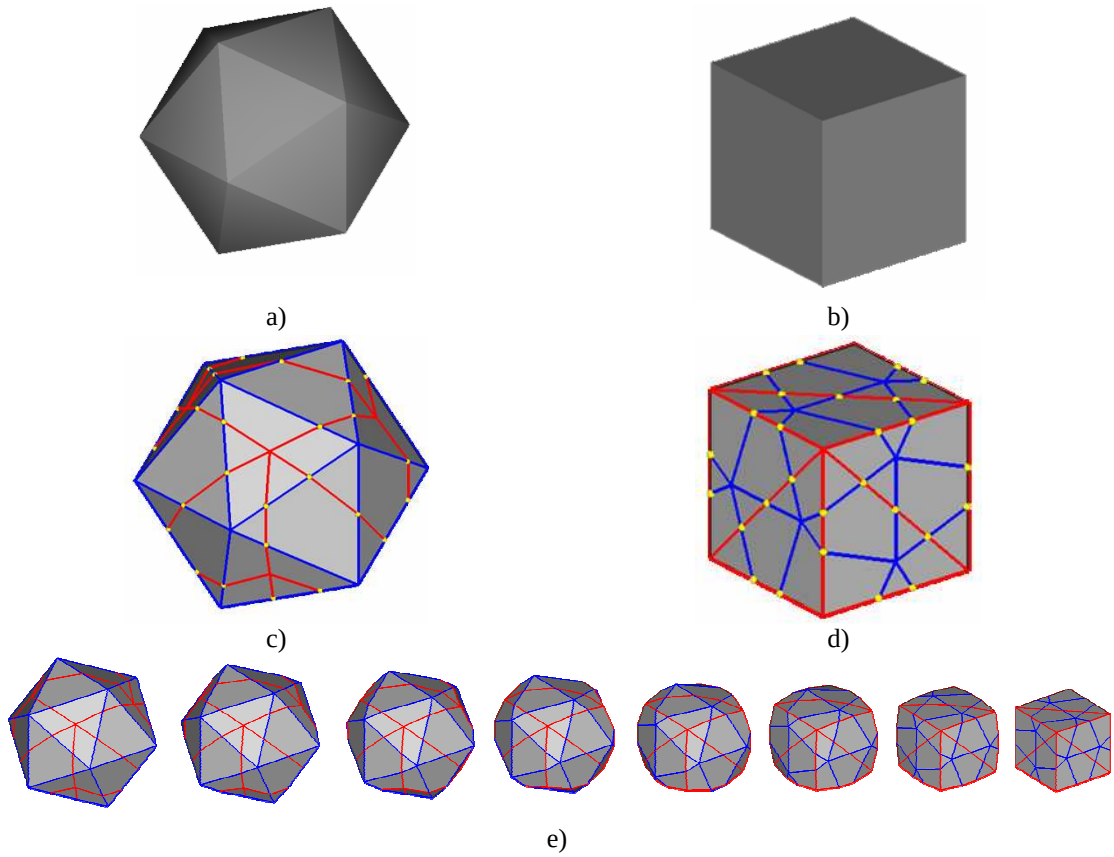


Figure 4.1: a) source mesh, b) target mesh, c) supermesh transformed to the shape of the source mesh, d) supermesh transformed to the shape of the target mesh, e) an example of an interpolation of the supermesh.

In Figure 4.1c), d) it can be seen that an edge may extend across several triangles; therefore, it is necessary to subdivide the triangles which are intersected by the edge and subdivide the edge in the intersection points. Additionally, in Figure 4.1c), d) it can be seen that there are some non-triangular polygons; these polygons have to be triangulated in order to have a valid triangular mesh. Also note that the supermesh contains the vertices of the source mesh, the vertices of the target mesh and the intersection vertices (picked out in yellow in Figure 4.1).

The main problem with the edge insertion is that it is not possible to insert an edge directly in another mesh in the 3d space. Even though we know where to insert endpoints of edges, it is not clear how to “wrap” an edge along a 3d surface, i.e., which triangles to subdivide. The reason is that the inserted edge is generally nonparallel and nonintersecting with edges and triangles of the other mesh. For this reason, we have to insert edges in the parametric domain.

In the parametric domain, all edges are mapped to a plane (a planar parametrization) or to a sphere. In both cases an edge is the shortest connection between its endpoints and it is unambiguous which triangles will be affected by the edge insertion. During the edge

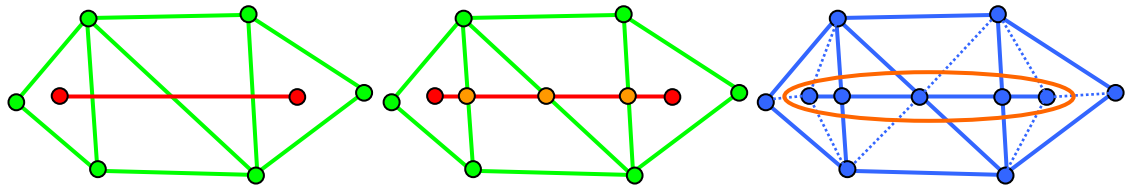
insertion intersections between the inserted edge and the original triangulation must be computed. The intersection vertices computed in the parametric domain are mapped back to the original mesh using an inverse mapping (Section 3.7.1).

4.2. Edge insertion

In this section we will discuss algorithmic aspects of the edge insertion. In the context of the edge insertion we will use the term *original triangulation* to refer to the triangulation in which the edge is being inserted.

4.2.1. Overview

The edge insertion consists of an endpoints insertion and a subdivision of triangles intersected by the inserted edge. The subdivision requires finding all triangles which are intersected by the inserted edge and computing intersections between them. After the edge is inserted, the triangulation must be repaired in order to have a valid triangulation. It is schematically demonstrated in Figure 4.2. Figure 4.2a) shows the original triangulation (green) and the inserted edge (red). In Figure 4.2b) the orange points represent the intersection vertices and finally Figure 4.2c) shows the final triangulation after the edge insertion. It can be seen that some additional edges (dashed lines) had to be inserted in order to have a valid triangulation.



a) b) c)

Figure 4.2: A demonstration of an edge insertion – a) the original triangulation (green) and the inserted edge (red), b) the intersection vertices (orange), c) the resulting triangulation.

In the following subsections we will describe the intersection computation and the re-triangulation in more detail.

4.2.2. Intersection computation

Intersections between the inserted edge and the original triangulation can be computed by brute force checking of each edge of the original triangulation against the inserted edge. In the topology merging process we have to insert all edges of the target mesh to the source mesh, therefore the brute force approach has a quadratic complexity, which is not very suitable for complex meshes with a high number of edges.

In [Ken92] and [Ale00b] better algorithms based on walking were described. Both algorithms work basically in the same way; they differ only in the underlying data structure. The former approach by Kent et al. [Ken92] uses a variation of the winged-edge data structure; the latter approach uses the DCEL[10] data structure. In our description we abstract away from a specific data structure, we only suppose that we are able to extract adjacency of triangles and edges.

The algorithm is based on the idea that for the inserted edge we can construct a *candidate list* (CL) of edges of the original triangulation. The candidate list contains edges which may be possible intersected by the inserted edge. Additionally, we have to

[10] DCEL – Doubly Connected Edge List.

know, in which triangles of the original triangulation the endpoints of the inserted edge lie. Generally, it can be computed by some point location algorithm, but in the case of morphing we already know it from the vertex-place correspondence step[11]. Let us denote **V** the starting vertex of the inserted edge **e**, $\mathbf{f}_0$ the face in which the vertex **V** lies (see Figure 4.3). Edges of the face $\mathbf{f}_0$ are added to CL because one of them is intersected by the inserted edge **e** originating from **V**. When the intersection **I** is encountered, the algorithm "walks" to the face $\mathbf{f}_1$, which is neighboring to $\mathbf{f}_0$ so that $\mathbf{f}_1$ and $\mathbf{f}_0$ share an edge $\mathbf{e}_{0,1}$. Now, two edges of $\mathbf{f}_1$ are inserted into CL because one of them is intersected by **e**. In this way, the algorithm walks until no intersection of **e** and edges in CL is found. The algorithm is schematically depicted in Figure 4.3.

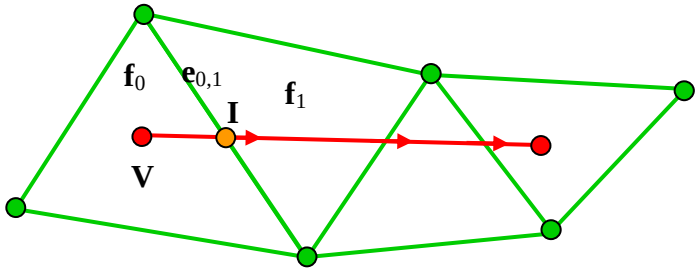


Figure 4.3: A demonstration of the walking algorithm which encounters all intersections of the inserted edge **e** (red) with the original triangulation (green).

4.2.3. Re-triangulation

After the intersections are computed, the inserted edge and the original triangulation are merged in the intersections. Then, new triangles must be created in order to have a valid triangulation.

Let us remind that the topology merging consists of many edge insertions. Therefore, there are basically two approaches how to re-triangulate the modified area – an incremental approach and a global approach. The approaches differ in the aspect *when* is the re-triangulation done. In the case of the incremental approach, the incremental construction concept is used, i.e., the original triangulation is re-triangulated each time a new edge is inserted. In the global approach the re-triangulation is done after all edges are inserted.

The advantage of the local approach is that it is usually easier to implement because a mesh is modified only in a relatively small part. The disadvantage is that in some cases the part which was re-triangulated in the previous step must be subdivided and re-triangulated again because a new edge was inserted close to the previously inserted edge. So, in the topology merging the incremental approach may lead to a repeated re-triangulation of the same area which is time consuming. Additionally, it may lead to a high number of edges of the resulting triangulation. On the other hand, the incremental nature can be useful in some progressive or adaptive processing. For instance, in time critical applications we can insert edges edge by edge while the incremental construction concept guarantees that after each edge insertion we have a valid triangular model.

[11] See Section 3.7.1.

The global approach [Kan97] works in two stages. In the first stage, for each inserted edge the intersection vertices are computed using the procedure from the Section 4.2.2 and the edges are merged with the original triangulation. In the second stage, for each vertex **V** an *edge fan* is computed. The edge fan is an angularly sorted list of edges incident to the vertex **V**. Each edge fan is traversed and locally triangulated (to create a triangle fan around the vertex **V**) by inserting some edges. The edge fan triangulation is done by checking if endpoints of two successive edges in the edge fan are connected by an edge. If they are not connected, a new edge is inserted together with a creation of a new face. The fan triangulation is demonstrated in Figure 4.4, where first a triangle fan around the central vertex is built and then in a greedy way all remaining vertices are processed, which results in a completely triangulated model.

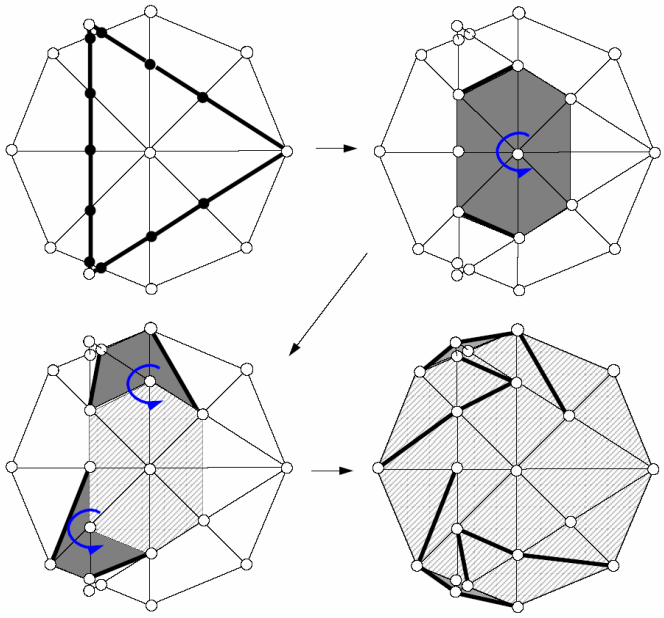
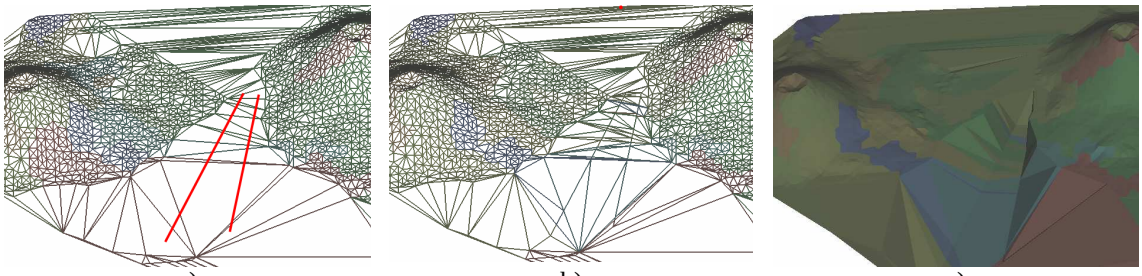


Figure 4.4: Demonstration of the edge fan triangulation, the black thick lines are added during the edge fan triangulation (taken from [Kan97]).

The global approach avoids multiple processing of the same area as the incremental approach. Thus, it is more efficient from the time consumption point of view and it generates fewer edges than the incremental approach. On the other hand, it relies on the angularly sorted edge fans which are more complicated to compute than the local subdivision as in the case of the incremental approach.

4.2.4. Additional use of edge insertion

The edge insertion is not only useful in the topology merging. It can be used in the area of terrain modification. The terrain modification usually involves some raising or lowering of a landscape. The terrain is usually represented as 2.5D triangular mesh. Sometimes, the mesh is not dense enough to represent an intended terrain modification. Usually some edges must be added. Since the terrain is 2.5D, the parametrization is obtained by omitting heights of vertices. Thus, the edge insertion can be done in 2d and only the heights of newly added vertices must be computed. An example of the terrain modification is in Figure 4.5 where a terrain is modified by “digging” a ridge. Note that it is not possible to dig a ridge in the original triangulation (Figure 4.5a) since there is no connectivity to represent the ridge.



a) b) c)

Figure 4.5: An example of terrain modification – a) the original terrain with an intended modification represented by red lines which delineate a bottom of some ridge, b) the terrain mesh is subdivided so that the bottom of the ridge can be lowered, c) the shaded terrain with the resulting ridge.

Another application is a computation of compatible triangulations of polygons [Ale00c]. Two triangulations are compatible if they have the same number of vertices connected in the same way. In other words, if we take the triangulations as graphs[12] then the graphs are isomorphic. If the two triangulations are compatible it is possible to interpolate them, e.g., to interpolate the corresponding vertices.

The computation of compatible triangulations works as follows. The input polygons are triangulated independently using Delaunay triangulation. Then, the perimeter of the triangulated polygons is mapped to a regular n-gon, which results in a triangulated n-gon. Then, n-gons are overlaid and edges of the first n-gon are inserted into the triangulation of the other n-gon. The new interior vertices are mapped back into the original polygons, which yields a compatible triangulation of the input polygons.

4.3. Surface attributes

In the previous sections we described how to modify a connectivity of meshes so that it is possible to transform them to some other shapes. In this section we will show our contribution which deals with surface attributes.

Until now we dealt with a shape transformation only. However, objects are represented not only by the shape but also by surface attributes – e.g., color, texture coordinates, surface normals, opacity, BRDF[13], etc. Since the topology merging modifies a mesh so that new vertices and faces appear, it is necessary to assign attributes to the newly added vertices and faces. Then, during the morphing, the surface attributes are interpolated along with a shape transformation. In the following sections we will describe how to handle surface attributes after the topology merging process. First we will describe attributes classification and then we will introduce an approach called face mapping which computes attributes of the supermesh.

4.3.1. Attributes classification

To be able to generalize the attributes handling for different kinds of attributes we will classify the surface attributes into two groups [Hop96] – *discrete attributes* and *scalar attributes*. Discrete attributes are usually associated with faces. A typical discrete attribute is, e.g., a material identifier, i.e., some property which is constant over the

[12] The triangulation can be viewed as a planar graph by using the triangulation vertices as vertices of the graph and the triangulation edges as edges of the graph.
[13] BRDF – Bidirectional Reflectance Distribution Function

whole face. From a certain point of view a face normal can be considered an independent discrete attribute as well[14].

On the other hand, the scalar attributes represent some local property of a surface. In simple cases the scalar attributes are associated with the vertices of the mesh – e.g., a per vertex color or per vertex texture coordinates. In a simple case a value of a scalar attribute in the vertex **V** is common for all faces adjacent to the vertex **V**. Hence, it is not possible to represent a discontinuity of the attribute field along an edge. Therefore, the scalar attributes are associated rather with *corners* than with vertices. The corner is a tuple (face, vertex) which allows us to assign some attribute to a particular vertex with respect to same face. For example a vertex normal is a typical scalar attribute, for one vertex we can have multiple normals depending on which face we are considering, e.g., a vertex representing a corner of a cube has typically three different normals (one for each face adjacent to the vertex).

4.3.2. Face mapping

The topology merging process builds a supermesh so that it "inherits" a shape of the source mesh and it is possible to transform it to the shape of the target mesh. Naturally, it is expected that the supermesh also "inherits" the surface attributes of the source mesh and it is possible to interpolate these attributes together with the shape. The problem is that during the topology merging a new mesh (i.e., the supermesh) is constructed, therefore, it is necessary to compute values of attributes (i.e., scalar and discrete) for the supermesh in the shape of the source mesh and the values of attributes for the supermesh in the shape of the target mesh. We propose a method called face mapping which computes the values of attributes so that it is possible to interpolate the attributes during the morphing transition.

The basic idea is that the faces and the vertices of the supermesh inherit attributes from faces and vertices of the input meshes. Therefore, for each face and for each vertex of the supermesh we have to establish how the attributes will be inherited. This is done by computing two mappings. A mapping between the faces of the supermesh and the faces of the *source* mesh and a mapping between the faces of the supermesh and the faces of the *target* mesh. The map represents from which face of the source mesh and the target mesh a face of the supermesh originate. A face of the supermesh inherits attributes of the face of the source mesh or the target mesh, respectively, to which it maps. From the topology merging process it is clear that each face of the supermesh maps to (i.e., originates from) exactly one face of the target mesh and to exactly one face of the source mesh (i.e. no face of the supermesh can overlap faces of the source and the target mesh)[15]. Let us denote $\Phi_0 : \mathbf{f}_i \rightarrow \mathbf{f}_j^0$ the mapping of the face $\mathbf{f}_i$ of the supermesh to the face $\mathbf{f}_j^0$ of the source mesh and analogously $\Phi_1 : \mathbf{f}_i \rightarrow \mathbf{f}_j^1$ for the mapping of the face $\mathbf{f}_i$ of the supermesh to the face $\mathbf{f}_j^1$ of the target mesh. Since the topology merging is done in a parametric domain, the mapping is established in the parametric domain as well. Figure 4.6 depicts a principle of the face mapping approach, on the right side there is a supermesh transformed to the shape of the source mesh (top) and to the shape of the

[14] Usually the face normal is computed as a normal of a triangle, thus the face normal is essentially a function of geometry of the mesh. However, in some applications (e.g., shading) the face normal can be modulated; therefore it can be considered as an independent discrete attribute.
[15] So this mapping is "onto".

target mesh (bottom), the arrows demonstrate the mapping of faces of the supermesh to the faces of the source and the target mesh. It can be seen that the highlighted face (the four-sided polygon) of the supermesh in the shape of the source mesh inherited the blue color from the source mesh while the same face inherited the red color from the target mesh when the supermesh is deformed to the shape of the source mesh. During the morphing transition the highlighted face will change its color from blue to red.

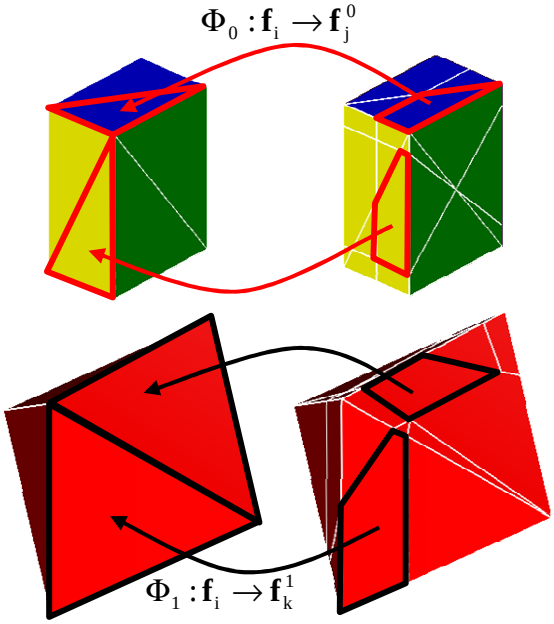


Figure 4.6: Mapping of a face of the supermesh to faces of the source mesh (top) and the target mesh (bottom).

Now we will describe how to compute the face map. In general, it is a location problem of a triangle **f** in some coarse triangulation, where **f** is the triangle of the supermesh and the coarse triangulation is the source or the target mesh. Note that a brute force approach would require checking each face of the supermesh against each face of the source and the target mesh. The complexity of the brute force approach is then $O(|\mathbf{F}|.|\mathbf{F}^0| + |\mathbf{F}|.|\mathbf{F}^1|)$, where $|\mathbf{F}|$ is the number of faces of the supermesh and $|\mathbf{F}^0|$ and $|\mathbf{F}^1|$ is the number of faces of the source mesh, target mesh, respectively. We propose a simple $O(N)$ algorithm which reuses an already established mapping of vertices of one mesh to the surface of the other mesh. The supermesh is traversed vertex by vertex and *triangle fans* of each vertex are processed. A triangle fan is a set of triangles incident to a particular vertex. Each triangle fan is processed depending on the type of the central vertex. Let us remind that there are three types of vertices in the supermesh – source vertices, target vertices and intersection vertices. Also note that each triangle belongs to more than one triangle fan, so once the triangle is processed, it must be marked by some flag to avoid multiple processing in the context of other triangle fans.

Let us first consider the case of the source vertex $\mathbf{V}_s$. From the correspondence computation step we know the mapping of the source vertex $\mathbf{V}_s$ to the target face $\mathbf{f}_t$. So each face of the supermesh adjacent to the vertex $\mathbf{V}_s$ maps to the face $\mathbf{f}_t$, i.e.:

$$\Phi_1(\mathbf{f}_i) = \mathbf{f}_t , \qquad (4.1)$$

where $\mathbf{f}_i$ are faces incident to the vertex $\mathbf{V}_s$, $\mathbf{f}_t$ is the face of the target mesh to which the vertex $\mathbf{V}_s$ maps. Now the mapping Φ_1 is solved. It remains to compute the mapping Φ_0, i.e., the mapping of the faces to the source mesh. It is easy, because faces of the supermesh adjacent to the source vertex map to the faces of the source mesh adjacent to the same vertex. So the faces adjacent to the vertex $\mathbf{V}_s$ in the source mesh are candidates for mapping of faces of the supermesh adjacent to the vertex $\mathbf{V}_s$ in the supermesh.

For the case of the target vertices the mapping is computed analogously. For the target vertex $\mathbf{V}_t$ the mapping to the face $\mathbf{f}_s$ of the source mesh is known. So each face of the supermesh adjacent to the vertex $\mathbf{V}_t$ maps to the face $\mathbf{f}_s$, i.e.:

$$\Phi_0(\mathbf{f}_j) = \mathbf{f}_s, \tag{3.2}$$

where $\mathbf{f}_j$ are incident faces to the vertex $\mathbf{V}_t$, $\mathbf{f}_s$ is the face of the source mesh to which vertex $\mathbf{V}_t$ maps. The establishing of mapping Φ_1 is again easy, because faces of the supermesh adjacent to the target vertex map to the faces of the target mesh adjacent to the same vertex.

For the intersection vertex we know from which two edges of the source and the target mesh it arises. For each edge we also know the indices of adjacent faces. Adjacent faces are candidates for mapping of faces of the supermesh. So it remains to check to which of candidates a particular face of the supermesh maps.

4.3.3. Handling attributes

Once we establish the mapping of the faces of the supermesh, it is necessary to establish the extreme values of attributes, i.e., the values between which we are going to interpolate during the morphing animation.

For discrete attributes it is simple, because the face $\mathbf{f}_i$ of the supermesh gets the value of the discrete attribute of the face to which it maps, i.e. for the transition parameter t=0 the face of the supermesh $\mathbf{f}_i$ gets the value of the discrete attribute of the face of the source mesh $\mathbf{f}_j^0$ to which the face $\mathbf{f}_i$ maps; and for the transition parameter t=1 the face of the supermesh $\mathbf{f}_i$ gets the values of the discrete attribute of the face of the target mesh $\mathbf{f}_k^1$ to which the face $\mathbf{f}_i$ maps. Let us denote $D(\mathbf{f}_i)(t)$ a value of a discrete attribute of the face $\mathbf{f}_i$ for the transition parameter t, then for t=0 we can write:

$$D(\mathbf{f}_i)(0) = D(\Phi_0(\mathbf{f}_i)), \tag{3.3}$$

where $\Phi_0(\mathbf{f}_i)$ is the mapping of the face $\mathbf{f}_i$ of the supermesh to the face of the source mesh. For t=1 we can analogously write:

$$D(\mathbf{f}_i)(1) = D(\Phi_1(\mathbf{f}_i)), \tag{3.4}$$

where $\Phi_1(\mathbf{f}_i)$ is the mapping of the face $\mathbf{f}_i$ of the supermesh to the face of the target mesh. A linear interpolation of a discrete attribute is then:

$$D(\mathbf{f}_i)(t) = (1-t)D(\Phi_0(\mathbf{f}_i)) + tD(\Phi_1(\mathbf{f}_i)). \tag{3.5}$$

The values of scalar attributes are computed with respect to the relative position of the vertex inside the face to which the vertex maps. Let us denote $S(\mathbf{f}_i, \mathbf{v}_j)(t)$ a value of a scalar attribute of the corner $(\mathbf{f}_i, \mathbf{v}_j)$ for the transition parameter t. So for t=0 we can say that the value of scalar attribute $S(\mathbf{f}_i, \mathbf{v}_j)(0)$ is given by some linear combination of the values of the attributes in corners of the face $\mathbf{f}_j^0$, where $\mathbf{f}_j^0$ is the face of the source mesh to which the face $\mathbf{f}_i$ of the supermesh maps. The coefficients of the linear combination are barycentric coordinates of the vertex $\mathbf{v}_j$ with respect to the face $\mathbf{f}_j^0$. This situation is depicted in the Figure 4.7, where the blue triangle $\mathbf{f}_i$ is the face of the supermesh and the green triangle $\mathbf{f}_j^0$ is the face of the source mesh to which the face $\mathbf{f}_i$ maps. The values of the scalar attribute $S(\mathbf{f}_i, \mathbf{v}_k)$ are given by the relative position of the vertex $\mathbf{v}_k$ with respect to vertices of the triangle $\mathbf{f}_j^0$.

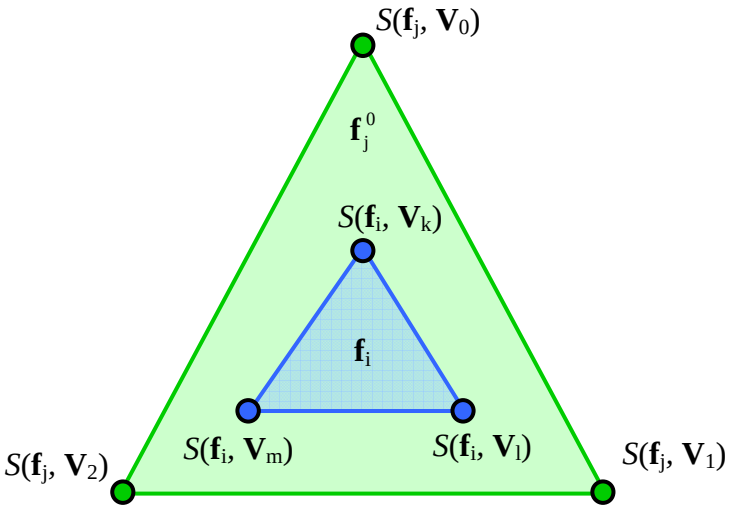
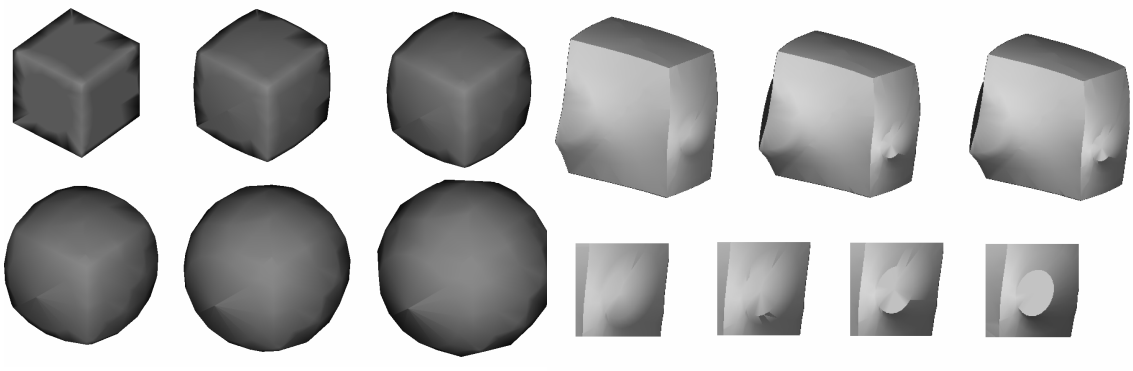


Figure 4.7: Mapping of the face $\mathbf{f}_i$ of the supermesh (blue) to the face $\mathbf{f}_j^0$ of the source mesh (green).

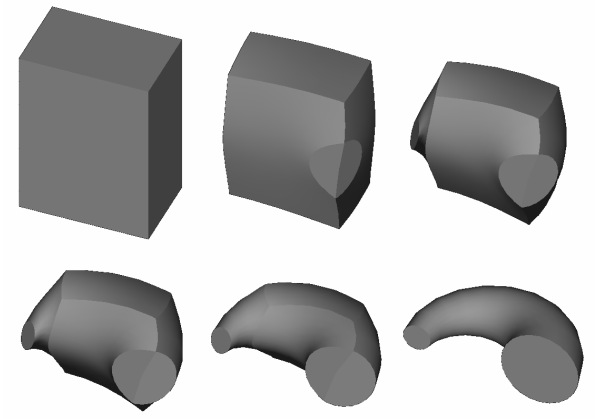
It is similar for the time t=1, with the difference that the mapping of the faces of the supermesh to the faces of the target mesh is considered. During the morphing transition the values of scalar attributes are linearly interpolated.

4.3.4. Examples

In this section we will show some results of the face mapping approach. The first example (Figure 4.8) is focused on computation of normal vectors which are used for shading. Figure 4.8a) shows a morphing animation between a cube and a sphere. The objects are rendered so that sharp edges were not considered. It is especially apparent in the first frame of the animation where the object is in the shape of cube. Since the existence of sharp edges is not considered the edges which should be sharp are smoothed out. Figure 4.8b) shows a sequence of another animation. In this case we recomputed normal vectors in each frame of the animation. It can be seen (the detailed view at the bottom) that it is not correct as well since the sharp edge appears abruptly. Finally, Figure 4.8c) shows an animation where the face mapping approach was used to compute normal vectors for the supermesh in the shape of the source and the target mesh. The normals for the intermediate shape were interpolated linearly. It can be seen that the sharp edges of the box smoothly disappear while the sharp edges of the cylinder smoothly appear.



a) b)



c)

Figure 4.8: A morphing animation rendered so that: a) sharp edges were not considered, b) normals recomputed in each animation frame, c) normals computed using the face mapping approach.

The next example (Figure 4.9) demonstrates how the face mapping works for colored meshes. Figure 4.9a) shows a morphing animation between objects which were painted *per face*, i.e., a color was assigned to a whole face (discrete attribute). On the other hand, Figure 4.9b) shows a morphing animation between objects which were painted *per vertex*, i.e., a color was assigned to vertices. Per vertex painting allows us to make smooth painting of the surface. In fact, the underlying data structure which contains information about color is "per corner" oriented, i.e. color is assigned to corners of triangles (Section 4.3.1). It is a little bit redundant, especially in the case of per face painting, but it allows to combine per face painting and the per vertex painting. It is demonstrated in Figure 4.9c), where a morphing animation between per face painted flower and per vertex painted pig is shown.

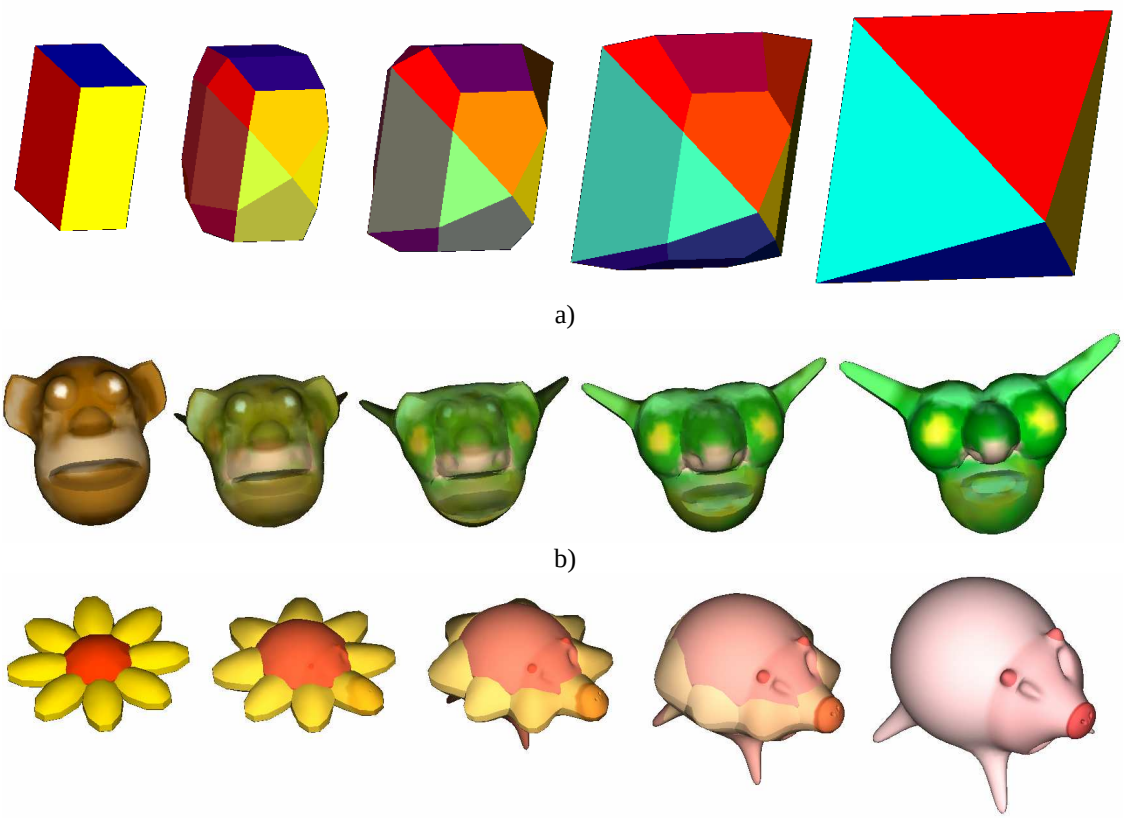


Figure 4.9: A morphing animation between a) per face painted meshes, b) per vertex painted meshes, c) a combination of per face and per vertex painted meshes.

4.4. Mesh improvements

In this section we will deal with an improvement of a quality of a mesh which was constructed by the topological merging procedure. This section contains description of methods which has not been fully tested, yet they have some promising results providing directions for a future research. Additionally, in the subsections 4.4.1 and 4.4.3 we present some preliminary results which indicate that these methods could be used to improve the quality of the mesh and to reduce the complexity of the mesh. However, we are aware that the following methods require additional research and tests.

4.4.1. Point insertion

The topology merging is based on the edge insertion. The edge insertion consists of some vertex insertions (i.e., insertion of endpoints of the edge and insertion of intersection vertices). In the original version of the algorithm [Ken92, Ale00b] the vertices were inserted in the plane of the triangles or on the edges (intersection vertices). However, the meshes are usually a piecewise linear representation of some smooth surface, therefore in some cases it would be better to insert vertices so that the resulting subdivided region is more curved. To accomplish this, we use a Bézier triangular patch [Vla01][16]. It is supposed that the input meshes have vertex normals which reflect the true shape of the original object. Therefore, before a vertex is inserted into a triangle we build a cubic triangular Bézier patch. The patch is constructed using vertex normals and the vertices of the triangle. Since we know the barycentric coordinates of the inserted vertex with respect to a triangle in which the vertex is inserted, we can compute a

[16] In [Vla01] referred to as PN triangle.

position of the vertex on the Bézier patch. It usually results in a position which is slightly above the original triangle. The advantage of the Bézier patches is that the "bulging" of a patch is controlled by normals associated with the vertices of the triangle. For instance, if the normals are parallel, the patch is flat, after inserting some vertices in the flat patch the patch still remains flat. Therefore the concept of Bézier patches can be used for non-smooth shapes (e.g., a cube) as well.

The influence of the type of vertex insertion is demonstrated in Figure 4.10. Figure 4.10a) shows the original mesh, it can be seen that the contour is not smooth because the mesh is not dense enough. During the topology merging some vertices are inserted, these new vertices can be used to improve the contour of the shape. Figure 4.10b) shows the result of topology merging when the vertices are inserted in planes of the triangles (i.e., the standard approach). The detailed picture (Figure 4.10b), bottom) shows that even if many new vertices were added, the contour is the same as the contour of the original mesh. Figure 4.10c) shows the result of topology merging when vertices are inserted in patches, it can be seen that the contour is much smoother.

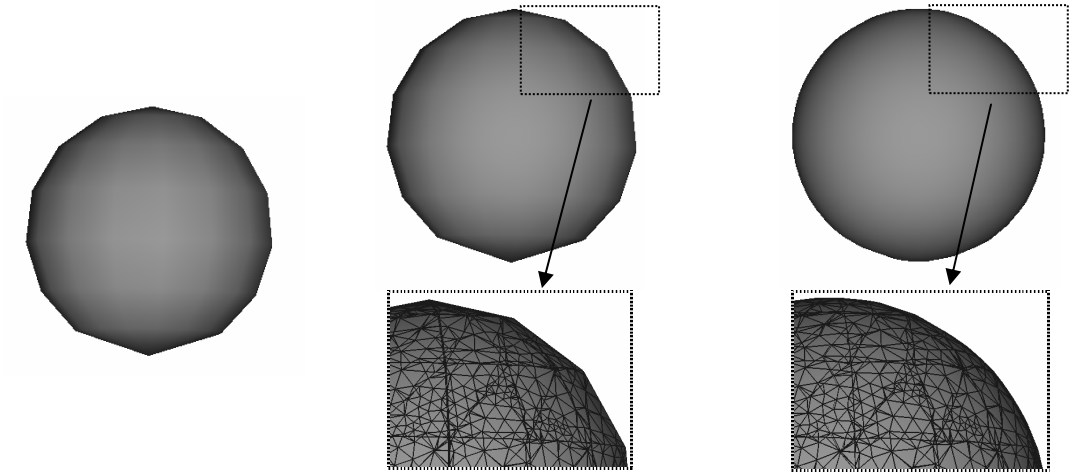


Figure 4.10: a) the original mesh, b) the result of the topology merging process with vertex insertion in a plane of a triangle, c) the result of the topology merging process with vertex insertion in a Bézier patch.

4.4.2. Edge flipping

The advantage of the topology merging is that the shape of the supermesh can capture exactly the shape of the source mesh as well as the shape of the target mesh. This fact is paid by a poor quality of the triangular mesh, i.e., the mesh contains badly shaped triangles. Generally, the quality of a mesh is important for computational analysis, it can influence a quality of a solution and a time needed to obtain it. For instance, vertex normals of triangular meshes are usually computed using some weighting scheme which takes into account the triangles adjacent to the vertex. If the triangles have a bad shape (i.e., long skinny triangles) and we use the computed vertex normals for shading, the shading is bad.

One possibility how to improve the quality of the mesh is a flipping of edges. Each non boundary edge has two adjacent triangles; the two adjacent triangles form a quadrilateral where the edge is its diagonal. In some cases it is possible to flip the diagonal in order to improve the quality of the mesh.

During the topology merging some edges are added, these edges must not be flipped since they might represent some important feature when transforming the supermesh to the shape of the target mesh. However, during the topology merging some regions must be re-triangulated (Section 4.2.3), i.e., some edges must be inserted to have a valid triangular mesh. These edges can be arbitrarily flipped to improve the quality of the mesh.

It must be said that during morphing the quality of the mesh varies as the shape transforms. Using edge flipping it is possible to improve only one frame of the animation. For instance, edge flips done for the supermesh in the shape of the source mesh may be different than edge flips needed to improve the quality of the supermesh in the shape of the target mesh.

4.4.3. Adaptivity

In the original version of the topology merging process, all edges of the target mesh were inserted in the source mesh. We found out that it is not necessary to insert all edges; moreover, the amount of edges can be adaptively controlled in order to reduce the complexity of the supermesh.

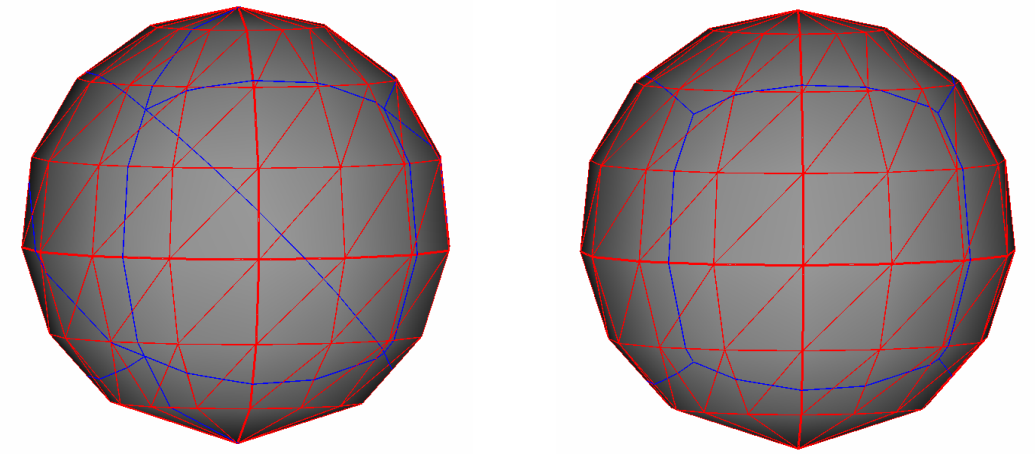
In the mesh morphing, the edges of the target mesh are inserted in the source mesh so that they "respect" the shape of the source mesh, i.e., they are not visible when the supermesh is in the shape of the source mesh. However, the inserted edges can be interpolated to represent some feature of the target mesh, i.e., during the interpolation edges "emerge" from the source shape to form the target shape.

Generally, we distinguish between two types of edges – feature edges and auxiliary edges. The feature edges represent the shape of the mesh; the auxiliary edges do not contribute to the shape of the mesh but they keep the mesh triangular. For instance, a simple triangular mesh representing a cube contains 12 feature edges (i.e., edges of the cube) and 6 auxiliary edges which subdivide each quadrilateral face of the cube so that the mesh consists solely of triangles. When using the topology merging in the context of the mesh morphing, it is enough to insert feature edges only. Since the auxiliary edges do not contribute to the shape of the object, they do not have to be inserted. Clearly, if we do not insert some edges, we save some computation and the resulting supermesh will contain less triangles.

To distinguish between feature edges and auxiliary edges, we use a geometric criterion based on a dihedral angle between faces adjacent to an edge. We set a threshold value of the dihedral angle so that an edge is auxiliary if the dihedral angle of the faces adjacent to the edge is less than the threshold value, an edge is a feature edge if the dihedral angle of the faces adjacent to the edge is greater than the threshold value. For instance, setting the threshold value to zero makes auxiliary edges only those edges whose adjacent faces lie in the same plane.

Let us demonstrate an adaptive edge insertion on supermesh which was computed to morph between a sphere and a cube, i.e., the model of the cube was inserted to the model of sphere. In Figure 4.11a) all edges (including the auxiliary edges) of the cube were inserted into the sphere mesh. It can be seen how the diagonal subdivides the sphere mesh. On the other hand, Figure 4.11b) shows a supermesh where the auxiliary edges were not inserted. It is clear that the supermesh without auxiliary edges contains

fewer triangles. Let us recall that the shape transformation of both supermeshes will be the same even though the supermesh without auxiliary edges contains less elements than the supermesh with auxiliary edges. Note that in the figures, for simplicity, we do not display the edges which need to be inserted during the re-triangulation process.



vertices: 269, # faces: 534, # edges: 801
a)

vertices: 191, # faces: 378, # edges: 567
b)

Figure 4.11: a) the supermesh with all edges inserted, b) the supermesh with only feature edges inserted (the '#' sign indicates the number of elements).

The concept of adaptive edge insertion can be used for progressive refinement of large meshes. For instance, let us have very dense meshes, first we can insert only the most important edges of the target shape to the source shape. Immediately we are able to morph between an original shape and the rough approximation (given by the most important edges) of the target shape. Then we can progressively insert the remaining edges to obtain more detailed morphing between the source and the target mesh.

4.5. Generalization for multiple meshes

The original version of the topology merging was designed for two input meshes only. However, in many applications (e.g., animation, design) it is required to morph multiple meshes at the same time. Therefore, we will show how to generalize the topology merging for multiple meshes. The goal is to compute such a supermesh which can be transformed to the shapes of all input meshes.

The basic idea is that we use the original topology merging technique and we apply it on pairs of meshes (see an example for 8 meshes in Figure 4.12). Let us have n input meshes $\mathbf{M}_{0,0}$, $\mathbf{M}_{1,0}$, …, $\mathbf{M}_{n-1,0}$. For simplicity, suppose that n is a power of 2. In the first stage we will always merge pairs of meshes, i.e., we merge $\mathbf{M}_{0,0}$ and $\mathbf{M}_{1,0}$, $\mathbf{M}_{2,0}$ and $\mathbf{M}_{3,0}$, etc. The result is a set of meshes $\mathbf{M}_{0,1}$, $\mathbf{M}_{1,1}$, …, $\mathbf{M}_{n-1,1}$ where the pairs $\mathbf{M}_{i,1}$, $\mathbf{M}_{i+1,1}$, i=0, 2,…, n-2 can be interpolated because they have the same number of vertices and the same connectivity. Then we merge meshes $\mathbf{M}_{0,1}$ and $\mathbf{M}_{2,1}$, $\mathbf{M}_{1,1}$ and $\mathbf{M}_{3,1}$, etc. The result is a set of meshes $\mathbf{M}_{0,2}$, $\mathbf{M}_{1,2}$, …, $\mathbf{M}_{n-1,2}$ where the quadruples can be interpolated. We continue in this way until all meshes share the same connectivity. $\log_2 n$ merging stages is required, in each stage the merging is computed n/2 times. If the number of the input meshes is not a power of 2 we always merge pairs, quadruples, octets, etc., as long as possible and the remaining meshes are transferred to the next stage. Main advantage of this approach is that it is not necessary to modify the original algorithm; we just

repeatedly apply the merging technique so that the input of j-th merging stage is the result of the (j-1)-th merging stage.

Recall that the merging procedure operates in the parametric domain; the vertices inserted during merging are projected back to the input meshes using an inverse mapping (Section 3.7.1). The key assumption of the described method is that we use a common parametrization for the merged meshes, e.g., the meshes $\mathbf{M_{0,1}}$ and $\mathbf{M_{1,1}}$ have a common parametrization.

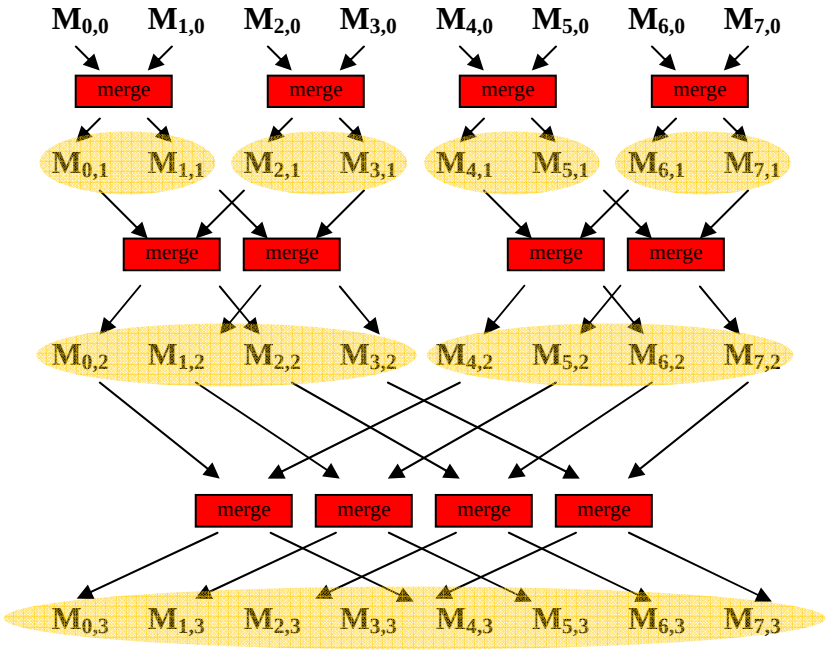


Figure 4.12: An example of topology merging for 8 input meshes, the red box represents the merging operation which always takes two meshes as an input and it produces two isomorphic meshes as an output.

5. Multimorphing

In this chapter we will show a generalization of the classical morphing. The classical morphing operates between two input shapes (a source and a target). We will show how to extend it to multiple input shapes. A morphing between multiple input shapes generates a space of shapes which we approach as an analogy of the affine space. The general idea has been already described in [Ale99], but we concretize their ideas for boundary representation, additionally we introduce an inner product which allows us to compute distances in the space of shapes and to compute an orthogonal projection. The orthogonal projection is used to express a shape as a weighted combination of basis shapes. We also show how to explore a space of shapes spanned by the basis shapes and we discuss some user interaction aspects of the shape generation. We propose a geometrical representation of a morphing space and we show how to easily generate new shapes using the geometrical representation in a similar way as a geometrical representation of a color space (i.e., a color system) is used to choose colors.

In this chapter we will consider so-called isomorphic meshes, i.e., meshes which have the same connectivity but different vertex positions. First, we will describe isomorphic meshes and how to compute them, then we will describe some related work in the area of spaces of shapes and finally we will present our contribution.

5.1. Isomorphic meshes

5.1.1. Definition

Isomorphic meshes are meshes with the same number of vertices and the same connectivity. A formal definition is as follows. Let us have a set of meshes $\mathbf{M}_i$, i=1, …, n. Meshes $\mathbf{M}_i$ are said to be isomorphic if there is a bijective map $\mathbf{f}$: $\mathbf{V}_i \rightarrow \mathbf{V}_j$ between vertices of the meshes $\mathbf{M}_i$, $\mathbf{M}_j$, $i \neq j$ with the property that any two vertices from $\mathbf{V}_i$ are adjacent in $\mathbf{M}_i$ if and only if they are adjacent in $\mathbf{M}_j$. In other words, graphs of triangular meshes are isomorphic. The connectivity which is shared among isomorphic meshes will be denoted as *common connectivity*. Although the isomorphic meshes share a common connectivity, they can differ in vertex positions so that each mesh can have a different shape. Therefore, isomorphic meshes are economical from data storage point of view. For a set of meshes, it is enough to store only one instance of connectivity (i.e., the common connectivity) and vertex positions for each element of the set. It is significantly less than storing connectivity and vertex positions for each mesh.

Additionally, in many applications, besides a bijective map between vertices, a *feature vertex correspondence* is required. The bijective map between vertices requires that vertices of isomorphic meshes are interconnected by edges in the same way in all meshes. The feature vertex correspondence additionally requires that the bijective map relates vertices which represent the same feature in all meshes. For instance, let us have isomorphic meshes representing human faces; vertices of each mesh are stored in an array. The feature vertex correspondence requires that eyes, noses or mouths are represented by vertices with the same index in the array. Clearly, the concept of the feature vertex correspondence is applicable only in cases when the set of isomorphic meshes is homogenous, i.e., the meshes represent some class of shapes which have some common features, e.g., human faces, models of cars, etc.

5.1.2. Use of isomorphic meshes

Besides the efficient storing, which does not require the feature vertex correspondence; isomorphic meshes are useful in many areas. Since there is a one-to-one correspondence between vertices, we can directly interpolate between corresponding vertices to obtain intermediate meshes. The intermediate meshes can be used to produce an animation or it can be viewed as a way to generate a new mesh by combination of some existing meshes.

Isomorphic meshes are well suited for *attributes transplantation*. Usually, a mesh does not only contain information about a shape but also information about an appearance the shape, e.g., color, texture coordinates, opacity, etc. Attribute transplantation is useful in the cases when we have a reference mesh equipped with some attributes and we want to apply these attributes also on another shapes. For instance, let us have a reference mesh representing a human face. The mesh has a texture and texture coordinates. The texture coordinates are assigned to the vertices of the mesh. Then, let us have another face without a texture. It is possible to paint the other mesh from the scratch, however it might be very time consuming. Using the attribute transplantation it is possible to use the texture of the reference mesh to paint the other mesh. It is done by copying the attributes of the reference mesh vertices to the corresponding vertices of the other mesh. Clearly, it requires a feature vertex correspondence; otherwise the painting of features of the reference mesh might be transplanted on different features of the other meshes.

Another important application of isomorphic meshes is the area of mesh analysis. Vertices of the mesh can be organized into a vector. A set of isomorphic meshes forms a matrix. The matrix can be analyzed by means of principal component analysis in order to find a new uncorrelated basis. Then, each original mesh can be expressed in the terms of the new basis. This concept was used by Alexa and Müller [Ale00d] to compress the animation. The basic idea was to choose from the new basis only a subset of basis vectors with the highest importance and represent the remaining meshes with respect to the new basis.

The main problem of isomorphic meshes is that meshes rarely fulfill the conditions described in the definition (Section 5.1.1). Therefore, meshes have to be preprocessed in order to be isomorphic.

5.1.3. Computation of isomorphic meshes – general aspects

Usually, during the mesh generation, the main goal is to represent some shape, the underlying connectivity is not important at this stage. Therefore, if we want to work

with isomorphic meshes, the meshes have to be *remeshed*. The remeshing is an operation which changes the connectivity (i.e., the model) while the shape is maintained. First, let us discuss some general aspects of methods used for a computation of isomorphic meshes. Then, we will describe particular methods which are used to turn a set of meshes into isomorphic meshes.

The input of isomorphic meshes computation algorithms is a set of meshes with generally different connectivity. The goal is to remesh the input meshes so that they are isomorphic. The following aspects of the isomorphic meshes computation algorithms are usually evaluated:

- mesh complexity,
- mesh quality,
- number of input meshes,
- method complexity.

The mesh complexity aspect is important when the input meshes have many small detailed features. Then the common connectivity must be dense enough to be able to represent all features of all input meshes. The mesh quality aspect refers to the shape of triangles; usually long skinny triangles (slivers) are not convenient, because they cause problems for example in shading (Figure 6.8). Note that, even if the input shapes have good quality of triangles, the remeshed models may contain triangles with a bad shape because of the remeshing procedure. Another important aspect is the number of input meshes which it is possible to process. In a morphing between two objects it is enough to remesh just the source and the target mesh, if we consider a morphing between multiple objects, all input objects must be remeshed. Last but not least the method complexity and robustness is also important from the practical point of view.

The simplest way how to generate isomorphic meshes is to modify some reference mesh. This approach is usually used in commercial applications to generate so called *morph targets*. Morph targets are meshes obtained by modification of vertex positions of the reference mesh. Facial animations are usually done in this way. A user usually models a neutral face expression (which is the reference mesh) and then adjusts positions of some vertices to create a specific face expression, e.g., a happy expression is made by adjusting of vertex positions in the corner of the lips. If the user manipulates just vertex positions then all new meshes created from the reference model are isomorphic. Another possibility how to model isomorphic meshes is a free-form deformation (e.g., bending, twisting or tapering [Bar84]). By a free-form deformation we usually change the overall shape but not the connectivity.

5.1.4. Computation of isomorphic meshes – a related work

Hutton et al. [Hut01] described an algorithm to compute isomorphic meshes for models of human faces. They first established a feature correspondence by placing nine landmark points (eyes, mouth, nose and chin). Then, they computed mean landmarks by averaging individual landmark positions. Then, each mesh is warped onto mean landmarks using thin-plate spline (TPS) technique. One of the input meshes is chosen as a reference mesh and the rest of warped meshes are remeshed using the connectivity of the base mesh. Then, the warped meshes are transformed back using the inverse TPS. The result is a set of isomorphic meshes. The resulting meshes were used to compute a principal component analysis.

Kraevoy and Sheffer [Kra04] described an algorithm which produces isomorphic meshes with adequate number of elements and it preserves the original shape of the input meshes. First, a feature correspondence is established by manual selection of corresponding vertices. Based on the feature correspondence, a *common base mesh* is computed. The common base mesh is a coarse approximation of the input meshes and it has the identical connectivity for all input meshes. The common base mesh partitions the input meshes into patches. Triangles of the common base mesh are used as parametrical domains for the corresponding patches. The patches of the input meshes are mapped to the corresponding triangle of the common base mesh which provides a parametrization of input meshes. Using the parametrization, the target mesh is remeshed with the connectivity of the source mesh. It results in meshes with identical connectivity. However, since the target mesh was remeshed with the connectivity of the source mesh the resulting shape is a poor approximation of the original shape, therefore, an additional smoothing and refinement steps are applied which minimize the difference between the original shape and the remeshed shape. The advantage of this method is an automatic coarse mesh generation (in contrast to [Mic01]). Also, the method can be generalized for multiple input objects.

Michikawa et al. [Mic01] proposed to compute isomorphic meshes using multiresolution representation. They first construct[17] a common base mesh. Similarly to Kraevoy and Sheffer [Kra04] they dissect input meshes into patches according to the common base mesh. Then each patch is parametrized over a planar face of the base mesh using Floater's shape preserving parametrization [Flo05]. Using the parametrization the input meshes are remeshed using 4-to-1 splits[18]. The advantage of this approach is a semiregular connectivity. It is also very easy to extend this approach to more than two meshes. The subdivision scheme works adaptively which means that some areas of input meshes can be subdivided more in order to capture some small features. On the other hand, if one mesh contains small detailed features which require denser subdivision, also the other input meshes must be subdivided in the same way even though it is not really necessary.

The approach introduced by Kent et al. [Ken92] was originally designed for two input meshes. Briefly, it works as follows. Edges of the target mesh are inserted into the source mesh so that the shape of source mesh is maintained; only the connectivity is modified. The topology merging method represents the input shapes exactly but it produces meshes with large number of faces and faces might have bad shape. The edge insertion involves computation of intersections which might be a weak point due to numerical stability and robustness. A detailed description together with our original improvements of the topology merging method is described in the Section 4.

Let us summarize the aforementioned approaches in the following table. The complexity of the method will be characterized by the most difficult part in the computation.

[17] In contrast to [Kra04] the common base mesh is constructed manually, i.e., along with feature vertices, edges and faces of the common base mesh must be constructed.

[18] 4-to-1 split is a subdivision of one face into four sub-faces. New vertices are inserted in the midpoints of the original face edges.

Approach	Mesh complexity	Mesh quality	Number of input meshes	The most difficult part
Hutton et al. [Hut01]	given by the base mesh	given by the base mesh	any	thin-plate spline interpolation
Kraevoy and Sheffer [Kra04]	Optimal	good	any	decomposition, remeshing
Michikawa et al. [Mic01]	according to the subdivision	good	any	decomposition, subdivision
Kent et al. [Ken92]	Complex	poor	2	remeshing

Table 5.1: A comparison of different approaches for a computation of isomorphic meshes.

5.2. Multimorphing – a related work

Next we will describe a related work in the area of space of shapes generated by means of morphing. Approaches differ mainly in the object representation. Each representation requires specific techniques for interpolation between multiple objects.

Cheng et al. [Che97] constructed a space of shapes from a collection of base shapes. They considered an implicit shape representation based on spheres and blending patches. The space of shapes is modeled as an n-dimensional manifold, where n is the number of basis shapes. A shape is represented by barycentric coordinates of a point inside an n-dimensional manifold. They sketched an approach for a metric based on the Hausdorff distance and using the metric they envisioned a stochastic process for identification of base shapes.

Similarly to [Che97], Lee et al. [Lee99] approached the space of images as a simplex where an image is represented by barycentric coordinates with respect to the simplex vertices. They extended the traditional formulation of image morphing to morphing among multiple images (denoted as polymorphing). They represent a morphing animation as a path inside the simplex where each point of the path corresponds to some intermediate image.

Alexa and Müller [Ale99] formalized a morphing between two objects as a morphing function. Using the morphing function they derived a morphing space and investigated conditions under which the morphing space is linear. The general concept is independent of shape representation. They approached the morphing space as an affine space. Elements of the morphing space are again represented by barycentric coordinates with respect to basis shapes. They proposed a recursive procedure to synthesize new shapes, i.e., to compute a shape given by its barycentric coordinates within the morphing space. They also propose an algorithm for analysis of existing shapes, i.e., to express some shape as a convex combination of the basis shapes.

Rossignac and Kaul [Ros94]] described an approach for computing polyhedral shapes in the space of all possible polyhedra. For morphing between polyhedra **A** and **B** to obtain intermediate t-variant shape $\mathbf{C}(t)$ they used a linear interpolation $\mathbf{C}(t) = (1-t) * \mathbf{A} + t * \mathbf{B}$ where **A**, **B** are input shapes, t is a transition parameter which controls a morphing between **A** and **B**. This relation combines shapes **A**, **B** and scalar t. The multiplication operator "*" denotes scaling and the addition operator "+" denotes Minkowski sum.

They further extended the interpolation among multiple meshes by introducing parametric curves (Bézier curves) and bi-parametric patches in the space of polyhedra.

Alexa and Müller [Ale00d] analyzed the space of shapes by means of principal component analysis. They focused on static connectivity mesh animations, where each frame of the animation represents one element of the space of shapes. Each frame is represented as a 3*n vector where n is the number of vertices. The vectors for each frame are organized into an m x 3*n matrix, where m is the number of frames. The matrix is analyzed by means of principal component analysis in order to find a new uncorrelated basis. Then, each original frame can be expressed in the terms of the new basis. This concept is used in their paper to compress the animation, because it is possible to choose from the new basis only the subset of basis vectors with the highest importance.

Sloan et al. [Slo01] described an abstract space of shapes which is defined by a set of examples and their adjectives. A set of examples is a homogenous user supplied set of shapes. Examples are characterized by adjectives. For instance, when shapes are human faces then adjectives can be gender or age. Additionally, the user has to annotate the examples with values of adjectives, e.g., specify an age and a gender of a human face. The adjectives form an axis of the abstract space of shapes. Then a smooth interpolation of examples is computed by using radial basis functions. Using a smooth interpolation new shapes can be generated by specifying values of adjectives.

Our approach differs from [Che97] in the data representation, additionally we are more specific about an analysis of a set of shapes and about a metric on the space of shapes. In [Ale99] a very general concept was introduced, as we focused on concrete representation, we can concretize a general idea of the morphing function. Rossignac and Kaul [Ros94] focused mainly on a novel morphing technique (based on the Minkowski sum) which was further extended to consider multiple input shapes. Sloan et al. [Slo01] considers the same representation as we do, however, they focus on synthesis of new shapes based on the values of adjectives, and they also do not consider an analysis of examples.

5.3. Morphing space

In this section we will approach the space of shapes as an affine space and a vector space. We will show how general concepts of the affine space generalize for meshes and we will introduce an inner product on the space of meshes.

5.3.1. Affine morphing space – space of shapes

In this section we will describe the Affine Morphing Space (AMS) which is an analogy of an affine space. Definitions and properties of the affine space suited for the computer graphics community are given in [Mil99]. Elements of the classical affine space are points; elements of AMS are isomorphic meshes. In the affine space we can compute an affine combination of points. In AMS we can compute an affine combination of meshes so that each vertex of the mesh is computed as an affine combination of corresponding vertices, i.e.:

$$\mathbf{R}_j = \sum_{i=1}^{n} w_i \mathbf{V}_j^i , \qquad (5.1)$$

where $\mathbf{R}_j$ is the resulting j-th vertex, w_i are weights of the affine combination and $\mathbf{V}_j^i$ is the j-th vertex of the i-th basis mesh. Symbolically, we will write:

$$\mathbf{S} = \Sigma\, w_i \mathbf{S}_i \text{ , } i=1,2, \ldots, n, \tag{5.2}$$

where $\mathbf{S}_i$ are the basis meshes and **S** is the resulting mesh. An affine combination of n basis meshes requires to specify weights w_i, i=1, …, n, so that $\Sigma w_i = 1$. Since we can choose independently only n-1 weights, all shapes obtained by the affine combinations of n basis shapes form n-1 dimensional AMS. A simple case of the affine combination is a linear interpolation between two shapes, i.e., the classical linear morphing. In this case there are two basis shapes and their combinations form a one dimensional AMS. Vertices of intermediate shapes move along lines defined by the initial and the final positions of corresponding vertices.

Note that the classical morphing usually generates shapes "between" the initial and the final shape. Clearly the affine combination allows shapes which are not only between the initial and final shapes, but also shapes which are extrapolations of the classical morphing, thus the affine combination generates a wider class of shapes than the convex combination (i.e., the classical morphing). Negative weights in the affine combination might cause flipping of orientation which may result in distorted shapes. Therefore, in some cases it is convenient to restrict a general affine combination to a convex combination. In this case the intermediate shapes will lie "between" basis shapes. In the case of classical morphing the vertices will move along line segments defined by the initial and the final position, in the case of multimorphing the vertices will move inside the convex hull of corresponding vertices. The fact that vertices of intermediate shapes move within some fixed region can be used for example in the area of collision detection [Lar03].

5.3.2. Morphing vector space

In the previous section we showed that the basis shapes can be viewed as elements of the affine space and that they can be combined using an affine combination to produce new shapes. In this section we will show an analogy of a vector space. Elements of the vector space are vectors which can be viewed as a relation between two elements of an affine space (i.e., points). We will use the concept of vector space to produce new shapes by a linear combination.

In the affine space the subtraction **P – Q** of two points **P**, **Q** results in a vector from **Q** to **P** in an associated vector space. In AMS two shapes are subtracted by subtracting the corresponding vertices. The result is an n-tuple of vectors which we denote as a *morphing vector*. Its components are vectors of trajectory which we will refer to as *trajectory vectors* (in fact, the morphing vector is an n x 3 matrix, where each row is a 3d vector but we will use the term vector instead of matrix to be consistent with the terminology of the vector space). Morphing vectors are elements of the Morphing Vector Space (MVS) which is an analogy of a vector space. So the MVS is a space of trajectories. A morphing vector is computed as:

$$\mathbf{v}_j = \mathbf{V}_j^1 - \mathbf{V}_j^0, \tag{5.3}$$

where $\mathbf{V}_j^0$, $\mathbf{V}_j^1$ are corresponding vertices of the initial and the final shape, $\mathbf{v}_j$ is the j-th component of the morphing vector. Symbolically we will write $\mathbf{v} = \mathbf{B} - \mathbf{A}$, where **A**, **B** are shapes and **v** is the morphing vector.

Addition of a point **P** and a vector **v** results in another point **P'** which is translated by an amount given by the vector, i.e., $\mathbf{P'} = \mathbf{P} + \mathbf{v}$. Addition of a shape and a morphing vector results in another shape whose vertices are translated by the amount given by the components of the morphing vector. Symbolically we will write $\mathbf{B} = \mathbf{A} + \mathbf{v}$, where **B** is the resulting shape obtained by adding a morphing vector **v** and a shape **A**.

Scalar multiplication of a morphing vector is represented in the MVS as a scalar multiplication of individual components of the morphing vector. Similarly, the vector addition is represented by vector addition of individual components of morphing vectors. Note that MVS is closed under scalar multiplication and vector addition.

In Figure 5.1a) there are two shapes – a square and a triangle. Arrows represent components of a morphing vector, i.e., trajectories of vertices when morphing from the shape of triangle to the shape of the square. Figure 5.1b) shows symbolically basic operations between shapes and morphing vectors. The first symbolical relation represents a computation of a morphing vector. The second relation shows an addition of a shape and a morphing vector which results in another shape. The third relation demonstrates an addition of a shape and a 0.5 multiple of a morphing vector which results in a halfway shape between the triangle and the square.

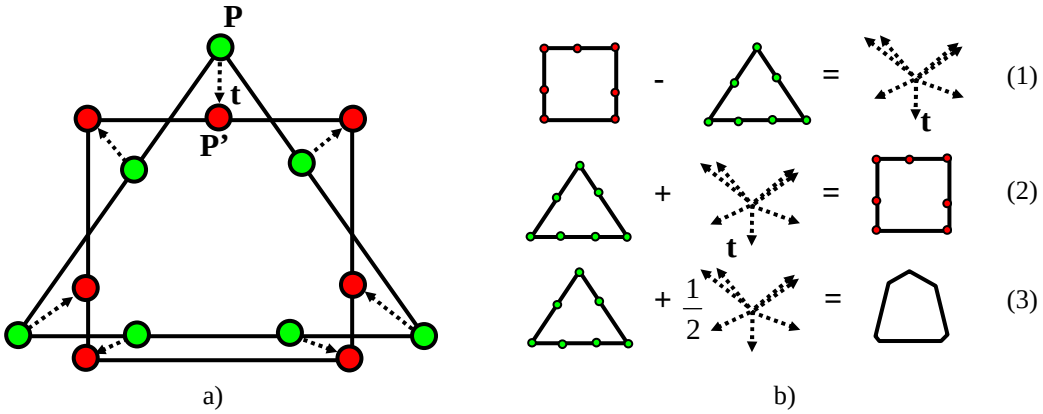


Figure 5.1: a) an example of morphing between a square and a triangle, arrows represent individual components of the morphing vector, b) a symbolical expression of morphing vector computation (1), an addition of a shape and a morphing vector (2) and an addition of a shape and a scalar multiple of a morphing vector (3).

Note that the MVS is a space of morphing vectors, so it does not contain any shapes, but the elements of MVS are used to construct new shapes by adding a shape from the AMS and morphing vector from the associated MVS. Elements of a vector space are usually represented as a linear combination of basis vectors. The number of linearly independent basis vectors gives us the dimension of the vector space. For example, $\mathbf{E}^2$ is a two-dimensional vector space spanned by basis vectors (1, 0) and (0, 1).

In the MVS the basis morphing vectors are constructed as follows. Let us have n basis shapes $\mathbf{M}_i$, i=0,…,n-1, without a loss of generality let us pick the shape $\mathbf{M}_0$ as a *zero*

element shape and compute the basis morphing vectors $\mathbf{v}_j = \mathbf{M}_j - \mathbf{M}_0$, j=1,…, n-1. Note that the element $\mathbf{M}_0$ is an analogy of the zero element vector **o** in the classical vector space.

It is clear that by adding $\mathbf{M}_0$ and $\mathbf{v}_j$ we obtain $\mathbf{M}_j$. The classical linear morphing $\mathbf{M}_j(t)$ between two shapes $\mathbf{M}_0$ and $\mathbf{M}_j$ is computed as $\mathbf{M}_j(t) = \mathbf{M}_0 + \mathbf{v}_j(t)$ where $\mathbf{v}_j(t) = t.\mathbf{v}_j$, $t \in$ <0; 1>, i.e., a scalar multiplication of the vector $\mathbf{v}_j$ by t. It is also demonstrated in Figure 5.1b) in the third symbolical relation. Basis morphing vectors $\mathbf{v}_k$ are combined by means of a linear combination to compute new shapes, i.e.:

$$\mathbf{M}(w_1, ..., w_{n-1}) = \mathbf{M}_0 + \Sigma w_i \mathbf{v}_i, \text{ i=1,…, n-1,} \qquad (5.4)$$

where w_i are coefficients of a linear combination. It is clear that if $w_i = 0$, i=1, …, n-1, then $\mathbf{M}(w_1, \ldots, w_{n-1}) = \mathbf{M}_0$. Note that n shapes generate n-1 dimensional AMS. Also note that if $\Sigma w_i = 1$ then the term $\mathbf{M}_0$ cancels out - it has no meaning - and Eq. 5.4 turns to the affine combination of shapes as defined in Section 5.3.1. If $\Sigma w_i \neq 1$ then the task of $\mathbf{M}_0$ is to "stabilize" the morphing. The linear combination of basis morphing vectors can be rewritten as an affine combination of basis shapes so that the weight of the zero element shape is $1 - \sum w_i$, i = 1, …, n-1.

So, why do we bother with the zero element shape if the same result can be achieved with the affine combination? It is mainly because of a user interaction. First, the zero element shape is usually some "neutral" shape so if the user does not specify any weights in the Eq. 5.4, the resulting shape is just the neutral shape. Second, the concept of the zero element shape allows us to work in an *additive way* as for example in the RGB color system, where by adding color components we obtain brighter colors. For instance, let us have face expressions. A neutral shape is some neutral face expression and the basis shapes contain some simple face expression – e.g., "left eye closed", "right eye closed" or a "smile" expression. The simple face expressions are *added* to the neutral face expression to create a new complex face expression.

Additionally, the morphing vectors express the difference between the zero element shape and some specific shape. If the difference is only in some local area (e.g., one eye closed) then the morphing vector is a sparse vector, which might be useful for instance for some efficient encoding. For example, to represent the morphing vector as a list of tuples (i, **v**), where **v** is a non-zero component of the sparse morphing vector on the i-th position.

With an analogy of vectors we can now define an inner product which is used to introduce a norm on the space of shapes.

5.3.3. An inner product in the AMS

In order to define a norm in the space of shapes, we have to introduce an inner product. We define the inner product as a sum of dot products of components of the morphing vector. We will denote such an inner product as a *morph dot product*. The morph dot product is computed as Frobenius inner product of morph vectors **u**, **v**, i.e.:

$$(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} v_{ij} , \qquad (5.5)$$

where **u**, **v** are n-component morphing vectors and each component is an m-dimensional vector. Note that the inner sum $\sum u_{ij}v_{ij}$ is the classical dot product of i-th components of the morphing vectors **u**, **v**. The components of a morphing vector are 3d (or 2d) vectors from the Euclidean space. In the Euclidean space an inner product is defined (denoted as dot product). Then our definition of the inner product is correct as well because it is a sum of properly defined dot products and it fulfills all properties of the inner product (i.e., it is distributive, commutative and positive). By defining the morph dot product we can introduce a norm l(**x**) on the space of shapes:

$$l(\mathbf{x}) = (\mathbf{x},\mathbf{x}) , \tag{5.6}$$

where **x** is a morphing vector. Using the norm we can compare shapes of the AMS. The distance between the points **A** and **B** is the size of the vector **v** obtained as **v** = **B** – **A**. In the same way we can compute the distance between the shapes $\mathbf{M}_1$, $\mathbf{M}_2$ which is a sum of distances between corresponding vertices. If the distance is zero then the shapes $\mathbf{M}_1$ and $\mathbf{M}_2$ are identical.

Note that the norm takes into account only relative positions of corresponding vertices, so it is not possible to capture scaling or rotation. For instance, let us have two input objects where the other object is the uniformly scaled first object. Even if the overall shape of both objects is the same, our norm will yield non-zero value since corresponding vertices are not coincident. In this case the input shapes must be properly aligned first and the effects of an affine transformation must be eliminated [Ale00d].

5.3.4. Orthogonal projection

We use the concept of the orthogonal projection to express a shape **S** as an affine combination of basis shapes. First, let us briefly describe the concept of the orthogonal projection in general. Denote **L** a vector space spanned by linearly independent basis vectors $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1}$. Denote $\mathbf{L}_0$ a subspace of **L** spanned by basis vectors $\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{m-1}$, m < n. The vector $\mathbf{v}_0$ is an orthogonal projection of $\mathbf{v} \in \mathbf{L}$ iff (i) $\mathbf{v}_0 \in \mathbf{L}_0$ and (ii) $(\mathbf{v}-\mathbf{v}_0)\perp\mathbf{g}_i$, i=0, …, m-1. The condition (i) can be expressed as $\mathbf{v}_0=\lambda_0\mathbf{g}_0 + \lambda_1\mathbf{g}_1 + \ldots + \lambda_{m-1}\mathbf{g}_{m-1}$. By substituting the condition (i) into the condition (ii) and by expanding the dot products a linear system is obtained:

$$\begin{aligned}
&(\mathbf{v},\mathbf{g}_0) = \lambda_0(\mathbf{g}_0,\mathbf{g}_0) + \lambda_1(\mathbf{g}_1,\mathbf{g}_0) + \ldots + \lambda_{m-1}(\mathbf{g}_{m-1},\mathbf{g}_0)\\
&(\mathbf{v},\mathbf{g}_1) = \lambda_0(\mathbf{g}_0,\mathbf{g}_1) + \lambda_1(\mathbf{g}_1,\mathbf{g}_1) + \ldots + \lambda_{m-1}(\mathbf{g}_{m-1},\mathbf{g}_1)\\
&\ldots\\
&(\mathbf{v},\mathbf{g}_{m-1}) = \lambda_0(\mathbf{g}_0,\mathbf{g}_{m-1}) + \lambda_1(\mathbf{g}_1,\mathbf{g}_{m-1}) + \ldots + \lambda_{m-1}(\mathbf{g}_{m-1},\mathbf{g}_{m-1})
\end{aligned} \tag{5.7}$$

Note that the matrix of the linear system contains all possible inner products – it is called a Gram matrix. By solving the system we obtain coefficients $\lambda_0, \lambda_1, \ldots, \lambda_{m-1}$ of a linear combination which expresses the orthogonal projection of **v** to $\mathbf{L}_0$. The system can be solved only if the Gram matrix is regular. The Gram matrix is regular if the basis vectors $\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{m-1}$ are linearly independent. The orthogonal projection guarantees the "closest" approximation of the vector **v** in the subspace $\mathbf{L}_0$. The closeness of the projection is of course expressed with respect to the defined inner product. Note that

$|\mathbf{v} - \mathbf{v}_0|$ expresses the distance between the original vector **v** and its projection $\mathbf{v_0}$. If $|\mathbf{v} - \mathbf{v}_0|$ is zero then $\mathbf{v}_0 \in \mathbf{L}$.

Since we introduced the morph dot product in the Section 5.3.3, we can compute orthogonal projections of morphing vectors. We will show an algorithm which computes a projection of a shape **S** into a subspace spanned by shapes $\mathbf{B}_i$, i=0, …, n-1.

Input: a set of shapes $\mathbf{B} = \{\mathbf{B}_i\}$, i=0, …, n-1, a shape **S** to be projected
Output: **S'** – the projection of **S**

1. Without loss of generality, pick $\mathbf{B}_0$ as the zero element shape.
2. Compute the basis morphing vectors $\mathbf{g}_j=\mathbf{B}_j - \mathbf{B}_0$, j=1,…, n-1.
3. Compute the morphing vector $\mathbf{v} = \mathbf{S} - \mathbf{B}_0$.
4. Compute the Gram matrix (Eq. 5.7).
5. If the Gram matrix is singular eliminate an arbitrary shape from **B** and continue with the step 1.
6. Solve the linear system for $\lambda_0, \lambda_1,\ldots, \lambda_{m-1}$.
7. Compute the projection $\mathbf{S'} = \mathbf{B}_0 + \Sigma\lambda_k\mathbf{g}_k$, k = 0, …, m-1.

Additionally, a distance between the original shape **S** and the projected shape **S'** can be computed using the norm (Section 5.3.3). If the distance is zero, then the shape **S** is an element of $\mathbf{L}_0$ (i.e., it can be obtained as an affine combination of basis shapes $\mathbf{B}_i$). Otherwise, the distance represents an error of the approximation. Note, that in the step 5 the algorithm essentially checks for a linear independence of the basis shapes. In linear algebra, a set of vectors is linearly independent if none of them can be expressed as a linear combination of vectors from the set. In MVS, if morphing vectors are linearly dependent then it means that some of the basis shapes can be obtained by a linear combination of the other basis shapes. Naturally, a set of basis shapes with no redundant shapes is required. Therefore, the steps 1-5 can be used to compute linearly independent set of basis shapes.

Alexa and Müller [Ale99] introduced a concept of a morphing function m(**A**, **B**, *t*) where the parameters **A**, **B** are input shapes and *t* is a transition parameter, which expresses the contribution of an input shape to the final shape. Thus, the orthogonal projection can be viewed as an inverse morphing function m^{-1}(**A**, **B**, **C**) which computes the value of the transition parameter *t* of the shape **C** when morphing between **A** and **B**. Of course, if the shape **C** does not "lie between" **A** and **B**, then *t* is the transition parameter of the projection of **C** to the morphing space spanned by **A** and **B**. Furthermore, the concept of the orthogonal projection is not limited just to two shapes (1D morphing space), so we can project to any-dimensional morphing space.

5.4. An exploration of the space of shapes

In this section we will describe user interaction aspects of the multimorphing. Theoretically, new shapes are generated by specifying weights of a linear combination of basis shapes. In an implementation, it is usually done by a set of sliders where each slider controls a contribution of one basis shape. The number of sliders is given by the dimension of the morphing space. It is clear that manipulation of large number of sliders might be complicated. Moreover, it is even harder when we want to maintain sum-up-to-one condition for computation of affine combinations. Therefore we propose two

alternative ways how to control the synthesis of new shapes – barycentric coordinates and curves in the morphing space.

5.4.1. Barycentric coordinates

Barycentric coordinates allow a coordinate-free expression of a point with respect to a triangle; they are infinitely differentiable, so they provide a good technique to interpolate data given in the vertices of a triangle. The concept of barycentric coordinates can be directly generalized for n-dimensional simplices. Other generalizations were proposed for general n-sided irregular polygons [Mey02], 3d convex and star-shaped polyhedra [Flo05] or for convex sets [War06]. An important property is that they fulfill sum-up-to-one condition so they can be directly used for the affine combinations of shapes.

First, we will describe the use of barycentric coordinates for a triangle and later we will discuss possible generalizations and their limitations with respect to the multimorphing application. First, let us have three basis shapes. The key idea is to associate the basis shapes $\mathbf{B_0}$, $\mathbf{B_1}$, $\mathbf{B_2}$ with the vertices of a triangle $\mathbf{V_0}$, $\mathbf{V_1}$, $\mathbf{V_2}$. By picking a point **P** from inside (or possibly outside) the triangle we can compute the barycentric coordinates *u*, *v*, *w* of **P** with respect to the triangle and use the barycentric coordinates as coefficients when computing an affine combination of the basis shapes. It is clearly easier to pick a point from a triangle than to independently specify three values.

By picking an arbitrary vertex of the triangle we obtain the shape associated with the vertex of the triangle (Figure 5.2a). By picking a point on an edge of the triangles we obtain a shape computed by morphing between the shapes associated with the endpoints of the edge, by picking a point from the interior of the triangle we obtain a mixture of all three basis shapes. Clearly, a sequence of points generates a sequence of shapes, i.e., an animation.

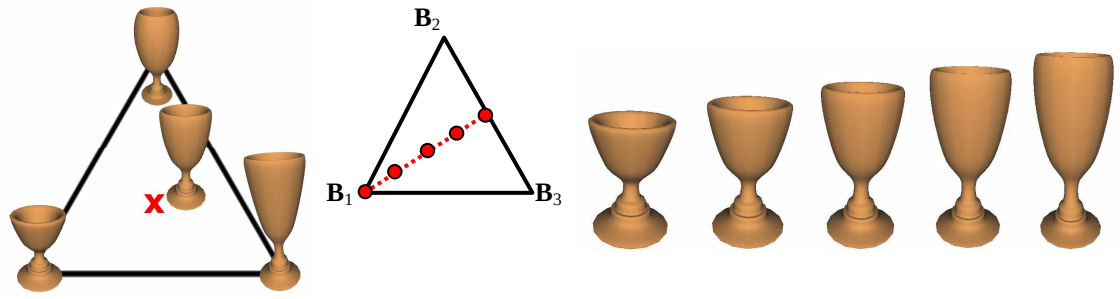


Figure 5.2: a) an association between a point (the red cross) and a shape, b) a sequence of points generates a sequence of shapes, c) the sequence of shapes.

Considering the triangle as a geometrical representation of the morphing space, the user associates shapes with a point. It is considerably easier to reproduce a point position than some complicated shape. Additionally, the user has a notion of distances in the space, i.e., if a point **P** is close to some triangle vertex $\mathbf{V_i}$, the resulting shape will be influenced most by the shape $\mathbf{B_i}$ associated with the triangle vertex $\mathbf{V_i}$. Analogously, when specifying points close to the edge, the resulting shape will be mostly a mixture of shapes associated with the endpoints of the edge. Similarly, in the geometrical representation of the morphing, the morphing animation is represented as a curve. So it is possible to associate the whole morphing transition (possibly very complicated) with

a simple curve. It is demonstrated in Figure 5.2b) where the red line segment represents the glass animation depicted in Figure 5.2c).

Since the computation of barycentric coordinates can be generalized for n-dimensional simplices, we can use the same idea for four basis shapes and a tetrahedral simplex. By picking an inner point of the tetrahedron we can compute affine combinations of four basis shapes.

The problem of higher dimensions (n > 3) is that it is hard to imagine and display an n-dimensional simplex and it is not easy to pick points from higher dimensional simplex using conventional input devices.

A generalization of barycentric coordinates for n-sided polygons [Mey02] can be used for higher dimensional morphing spaces. Again, the basis shapes are associated with vertices of the polygon and by picking points from the interior of the polygon coefficients of a convex combination are obtained. Generally, all coefficients of the convex combination are non-zero (except the cases when a point lies on a polygon vertex or on a polygon edge). It means that, using a polygon as a geometrical representation of a morphing space, it is not possible to generate a shape which is a mixture of a subset of basis shapes. For instance, given 5 basis shapes, it is not possible to generate a mixture of only 3 basis shapes, because the coefficients of the convex combination of a point inside a polygon are non-zero. So it means that using barycentric coordinates for polygons we cannot access the whole space of shapes. Note that in the case of tetrahedra it was possible to compute combinations of 1, 2, 3 or 4 shapes, which corresponds to points on a vertex, on an edge, in a face or generally inside the tetrahedra. In case of four sided polygon only a combination of 1, 2 and 4 shapes can be computed. The same problem will appear in 3d generalization of barycentric coordinates [Flo05, War06].

5.4.2. Curves in the morphing space

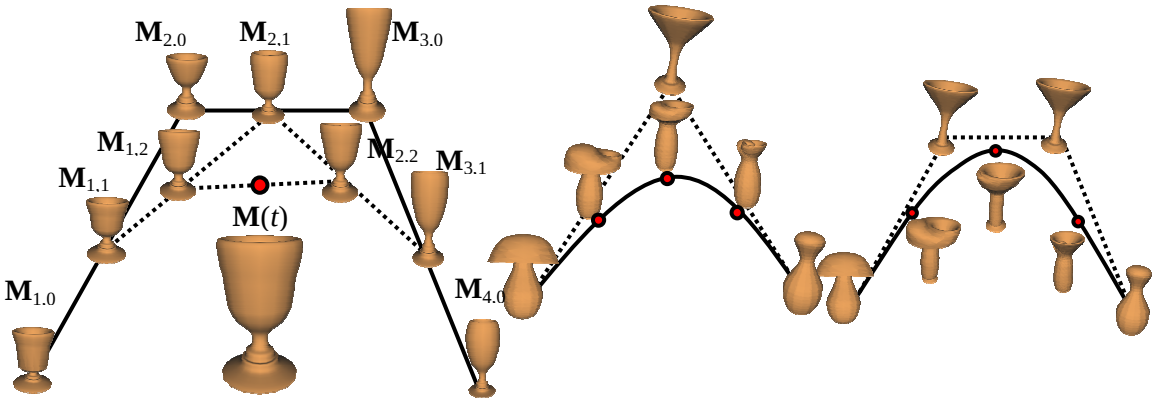
Another interesting way how to easily generate shapes in the space of shapes was outlined by Rossignac and Kaul [Ros94], but they considered a different morphing technique (Section 5.2). We consider a boundary representation and we propose a generalization of de Casteljau algorithm, which can be used for fast generation of shapes. De Casteljau algorithm for morphing can be used for other representations as well. We will briefly review the basic idea and then we will describe our extensions. Rossignac and Kaul proposed the so called Bézier metamorphosis which is motivated by a classical Bézier curve. As the points of a Bézier curve are computed as convex combinations of control points, the Bézier metamorphosis consists of shapes which are convex combinations of the control shapes. Rossignac and Kaul defined the convex combination of objects using the Minkowski sum and scaling.

Our basic idea is the same as Rossignac's and Kaul's [Ros94]. Control points of the Bézier curve are replaced by the control shapes. To compute a point on a Bézier curve, Bernstein polynomials must be evaluated. The difference between Rossignac's and Kaul's approach and our approach is that we use the values of the Bernstein polynomials in the convex combination of the control shapes (Eq. 5.2) while Rossignac and Kaul used them for scaling of control shapes which are subsequently added using the Minkowski sum. By replacing the control points by the control shapes, a Bézier curve in the space of shapes is generated. This curve has analogous properties as the

classical Bézier curve. It interpolates the first and the last control shape and it approximates intermediate control shapes.

Bézier metamorphosis can be viewed as a new way of designing morphing animations. In the classical morphing a user chooses two shapes between which some smooth transition is computed. Then, the transition can be adjusted by introducing other shapes which *bend* the animation so that the initial and the final shapes are preserved but the intermediate shapes are influenced by the additional shapes. The amount of influence is given by Bernstein polynomials. Note that using this idea is extremely easy; the user just adds intermediate shapes to adjust the multimorphing animation and it is still controlled by one parameter.

Interesting and useful property of Bézier curves is that they can be generated using de Casteljau algorithm. De Casteljau algorithm is a recursive subdivision of a control polygon. It is used to evaluate a point on a Bézier curve using a sequence of linear interpolations. In the multimorphing, instead of the linear interpolation, a classical morphing function is used to compute a shape on a Bézier curve in the space of shapes. It is depicted in Figure 5.3a), where the classical morphing between two glasses $\mathbf{M}_{1,0}$ and $\mathbf{M}_{4,0}$ is influenced by additional shapes $\mathbf{M}_{2,0}$, $\mathbf{M}_{3,0}$. The first subdivision results in shapes $\mathbf{M}_{1,1}$, $\mathbf{M}_{2,1}$, $\mathbf{M}_{3,1}$, the second subdivision results in shapes $\mathbf{M}_{1,2}$, $\mathbf{M}_{2,2}$ and the third subdivision results in the final shape $\mathbf{M}(0,5)$.



a) b) c)

Figure 5.3: a) a demonstration of the de Casteljau algorithm in the multimorphing setting, b) a quadratic Bézier curve in the morphing space, c) a cubic Bézier curve in the morphing space.

We propose two approaches for Bézier morphing computation. In the first approach we compute values of Bernstein polynomials for a given t. The values are used as weights in Eq. 5.1. This approach considers isomorphic meshes or polygons. The second approach is based on the de Casteljau algorithm. All it requires is just a classical morphing function. So it can be used in any area where a morphing between two objects (shapes, volumes, images, etc.) is defined. For example, if we are able to morph between two meshes with different connectivities, we can use the Bézier morphing among multiple basis meshes with different connectivities as well, since we just apply repeatedly a morphing function on two meshes. But let us remind that it might be computationally expensive because in each step of the de Casteljau algorithm the whole morphing function must be computed (including the correspondence computation and the remeshing), which is not a problem in the first approach for the Bézier morphing.

The de Casteljau algorithm is a subdivision scheme which converges towards a Bézier curve. In a similar manner as we subdivide the curve we can subdivide a morphing animation. In each subdivision stage we can adaptively control the depth of the subdivision. In the case of curves the curve is subdivided until a small relatively flat segment is generated or until a segment is smaller than a pixel size (in case of generating a curve on a raster device). In each stage of the subdivision process we have two shapes. We can adaptively control the depth of the animation subdivision by comparing these two shapes and possibly stop further subdivision. Of course, some function which compares shapes is required. Generally, we can use some measure based on the Hausdorff distance to compare two shapes. Since we work with isomorphic meshes, we can use the measure introduced in Section 5.3.3. Or, for example, we can compare some scalar quantity of the two shapes (e.g., volume or area) and stop the subdivision if the quantity difference is small. Using this approach we can generate a morphing animation which is parameterized by an amount of the shape transformation (and not by an artificial transition parameter). For instance, it should be possible to reparametrize the animation so that the volume of the shapes changes linearly.

Clearly, the Bézier curve in the morphing space addresses just a small subspace of the entire morphing space. It generates just convex combinations of basis shapes. Simplicity of the use is paid by less control over the whole morphing transition. One possibility how to control an influence of some specific basis shape (except the initial and the final) is an increase of multiplicity of the basis shape. If we want some basis shape to influence more on the morphing animation we just repeat the basis shape in the sequence of control shapes. It is demonstrated in Figure 5.3b), where a quadratic Bézier curve in the morphing space is depicted. In Figure 5.3c) we wanted to emphasize the glass shape, so we increased the multiplicity of the glass shape. It can be seen that the intermediate shape is closer to the glass shape.

The idea of Bézier curves can be generalized for other types of curves as well. For example, rational Bézier curves have similar properties as Bézier curves, but each control shape can be explicitly assigned a weight. It can be useful when we want to change the influence of some basis shapes. It can be done by increasing the multiplicity of the basis shape but still the weights are given by Bernstein polynomials, whereas using the rational specialization of Bézier curves, the weights can be adjusted more precisely. Another well known type of curves is B-spline curve. Besides, it has a local support which means that by changing some of control shapes we do not change an entire animation but just a local part of the animation. B-splines are defined by specifying an order of basis function (which implies an extent of the local support) and by a knot vector. There is also an analogy of de Casteljau algorithm for B-spline generation which is called de Boor algorithm. A rational specialization of B-spline curves are NURBS curves, which in addition to the degree of basis function and the knot vector introduce weights for each control shape.

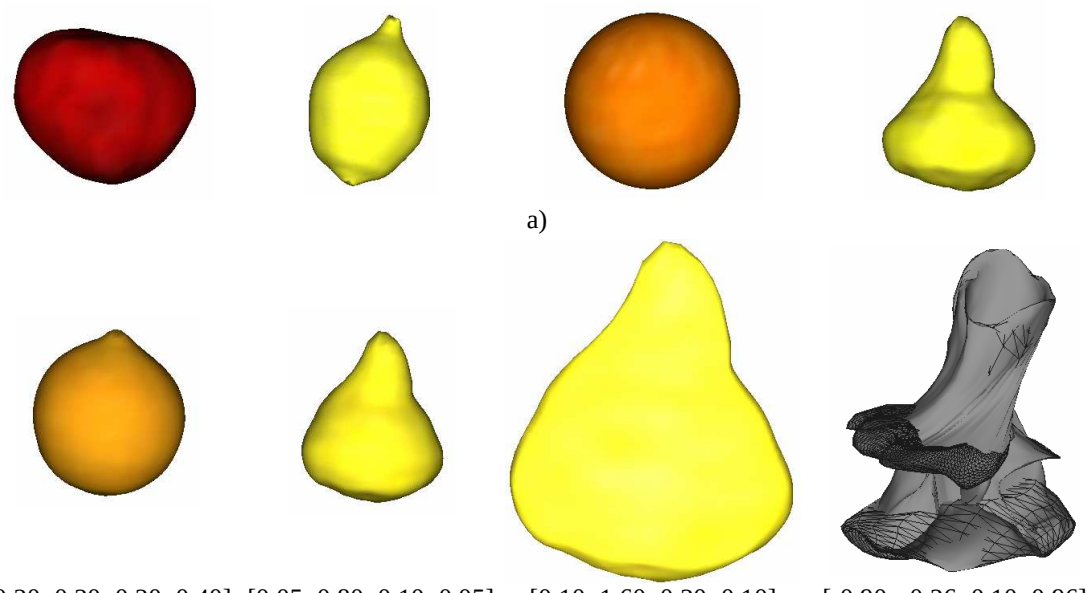
The idea of generating one-parametric curves in the morphing space can be also generalized for multi-parametric objects as well. For example, a bilinear patch can be used to control the morphing among four objects. It can be useful, e.g., for controlling LOD morphing animations where one parameter controls the shape transition whereas the second parameter controls the level of detail.

5.5. Examples of use of our apparatus

Using our apparatus we can construct new shapes in two ways – by a convex combination of shapes or by a linear combination of morphing vectors. In fact both approaches can be expressed by an affine combination. But let us consider both cases separately because we use each apparatus in a slightly different situation.

5.5.1. Shape synthesis – convex combination

The convex combination of shapes is used when no zero element shape is specified, i.e., all basis shapes are on the same level. As an example let us show a multimorphing between an apple, a lemon, an orange and a pear (Figure 5.4a). In this case no shape has a special role, all shapes are equal. Shapes in Figure 5.4b) and c) are examples of convex combinations of basis shapes.



a)

[0.20; 0,20; 0.20; 0.40] b) [0.05; 0,80; 0.10; 0.05] c) [0.10; 1,60; 0.20; 0.10] d) [-0.90; -0.26; 0.10; 0.96] e)

Figure 5.4: a) basis shapes, b), c) an example of a convex combination, d) an example of linear combination, e) an example of linear combination with negative weights.

An affine combination of shapes is theoretically well defined too but it may produce distorted shapes as shown in Figure 5.4e). It is because it accesses an extrapolation area of the classical morphing (in the same way as all the affine combinations of two points fill a line in contrast to the convex combinations which fill just a line segment). One effect of accessing the extrapolation area of the morphing is a flipping inside out of an orientation of triangles and their normals which results in distorted rendering (wireframe triangles in Figure 5.4e).

The shape in Figure 5.4d) shows a result of a linear combination of shapes, i.e., sum-up-to-one condition was violated. Weights of the linear combination can be normalized to obtain convex combination, i.e.:

$$w_i' = \frac{w_i}{\sum_{j=1}^{n} w_j}, \tag{5.8}$$

where w_i are original weights, w_i' are normalized weights. By normalizing weights of the third shape (Figure 5.4d) the second shape (Figure 5.4c) is obtained. It can be also seen that the third shape is just a scaled version of the second shape where the scale factor is 2, i.e., the sum of weighs $\sum w_j$, j=1,…,n [Klu04]. So, the linear combination does not bring a wider shape variety than the affine combination, it generates the same class of shapes but it just introduces an additional uniform scaling.

Another problem with a non-affine combination appears when interpolating surface attributes as for example color. Color is usually interpolated with the same weights as geometry [Par04] and it can happen that the resulting color jumps outside a color system. Then the color must be clamped to values within the color system. A similar problem appears for the normal vector computation, as long as we compute an affine combination of unit length normal vectors the result has still a unit length, but in the case of non-affine combination, the normal vector has to be renormalized.

5.5.2. Shape synthesis – linear combination of morphing vectors

We discussed the motivation for the concept of a zero element shape in the Section 5.3.2. Next, let us remind that the linear combination does not restrict the sum of weights, but it is convenient that each individual weight is in the interval <0; 1>, otherwise we access an extrapolation area of the classical morphing.

We will demonstrate the concept of a zero element shape on an example of four hand gestures which are depicted in Figure 5.5a). We can identify a zero element shape as an "all fingers straight" gesture; simple gestures are left, middle and right finger bent. Simple gestures are added to the zero element shape to obtain more complex gestures, e.g., two fingers bent. With respect to the zero element shape we generated three morphing vectors $\mathbf{v}_1 = \mathbf{M}_1 - \mathbf{M}_0$, $\mathbf{v}_2 = \mathbf{M}_2 - \mathbf{M}_0$, $\mathbf{v}_3 = \mathbf{M}_3 - \mathbf{M}_0$. It is clear that by adding the shape $\mathbf{M}_0$ and a vector $\mathbf{v}_i$ we obtain the shape $\mathbf{M}_i$. By adding all three vectors we obtain an "all fingers bent" gesture.

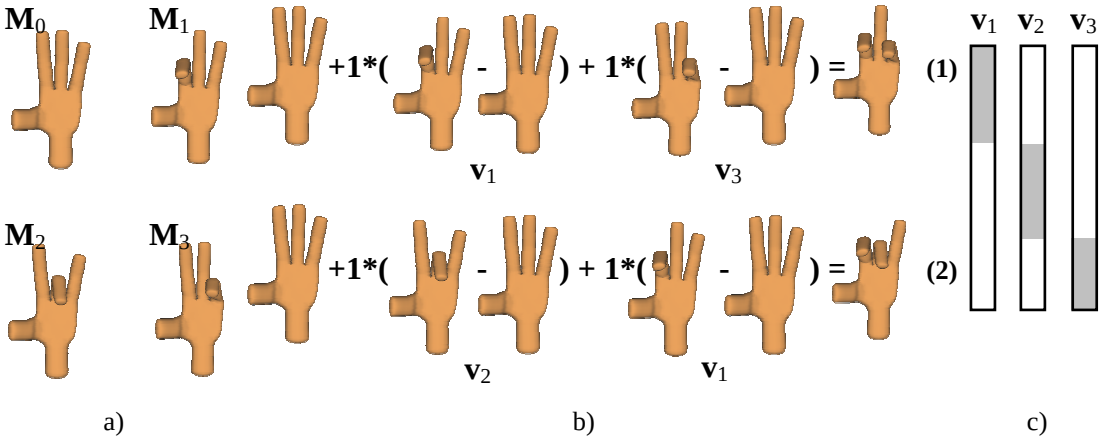


Figure 5.5: a) basis shapes, b) examples of a linear combination of basis shapes, c) a graphical representation of morphing vectors.

On the other hand, when combining shapes $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$ using a convex combination we would never achieve a gesture where two or three fingers are completely bent. Thus, the linear combination of morphing vectors allows us to generate a wider class of shapes than a convex combination. Examples of linear combinations of morphing vectors are shown in Figure 5.5b). The first symbolical expression combines shapes $\mathbf{M}_0$, $\mathbf{M}_2$ and $\mathbf{M}_3$, where the shapes $\mathbf{M}_2$ and $\mathbf{M}_3$ bend one

finger (left one and right one), the result is a shape with two fingers bent. Similarly, the second symbolical expression combines the shapes $\mathbf{M}_0$, $\mathbf{M}_1$ and $\mathbf{M}_2$, where $\mathbf{M}_1$ and $\mathbf{M}_2$ bend one finger (the middle one and the left one); the result is a shape with two fingers bent. Note that such shapes cannot be obtained using convex combination apparatus.

Note that the morphing vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ are really sparse because they contain just trajectory vectors to bend one finger. Also, in this example, we can work with shapes in the additive way, because each time we add some morphing vector to the neutral shape we add some specific gesture. By adding all three morphing vectors we obtain a complex "all fingers bent" gesture. Morphing vectors are depicted in Figure 5.5c), where the white regions represent zero components and the gray regions represent the non-zero component. It can be seen that the non-zero components are disjoint among morphing vectors which correspond to the fact that each morphing vector represents a movement of a local part of the mesh which is disjoint among $\mathbf{M}_1$, $\mathbf{M}_2$ and $\mathbf{M}_3$.

5.5.3. Shape analysis

Now let us have a reversed problem, given a set of shapes of interest, we want to analyze the set, to find the basis shapes and to try to express the elements of the set with respect to the basis shapes. Usually we want the dimension of the morphing space as low as possible so that the elements of the set can be expressed with respect to a relatively small number of basis shapes. The essential tool in the shape analysis is the orthogonal projection (Section 5.3.4) which can be used as an analogy of the inverse morphing function [Ale99].

We will demonstrate the orthogonal projection on a following example. We have 5 basis shapes $\mathbf{B}_i$, i=1,…, 5 which represent fish contours (Figure 5.6). Let us denote $\mathbf{L}$ a space of shapes generated by basis shapes $\mathbf{B}_1$, $\mathbf{B}_2$, …, $\mathbf{B}_5$. We computed 5 different shapes $\mathbf{M}_i$, i = 1, …, 5 as a convex combination of basis shapes (Figure 5.6b). Each shape is represented by 5 weights which express a contribution of the basis shapes. As a subspace $\mathbf{L}_0$ of $\mathbf{L}$ we choose 1 dimensional space of shapes spanned by the shapes $\mathbf{B_1}$ and $\mathbf{B_5}$. Using the algorithm described in Section 5.3.4 we computed projections of shapes from Figure 5.6a) to the subspace $\mathbf{L}_0$. The projected shapes are shown in Figure 5.6c). The key benefit is that each shape in $\mathbf{L}_0$ is represented by one weight (in contrast to 5 weights needed to express a shape in $\mathbf{L}$) while the difference between the original shapes and their projections is small. The shapes $\mathbf{M}_1$, $\mathbf{M}_5$ are approximated exactly because they form a basis morphing vector of $\mathbf{L}_0$. The difference between the original shapes and their projections can be seen in Figure 5.6d), where the black shapes are original shapes $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$ (elements of $\mathbf{L}$) and the blue shapes $\mathbf{M}_2$', $\mathbf{M}_3$', $\mathbf{M}_4$' $\in \mathbf{L}_0$ are projections of $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4 \in \mathbf{L}$.

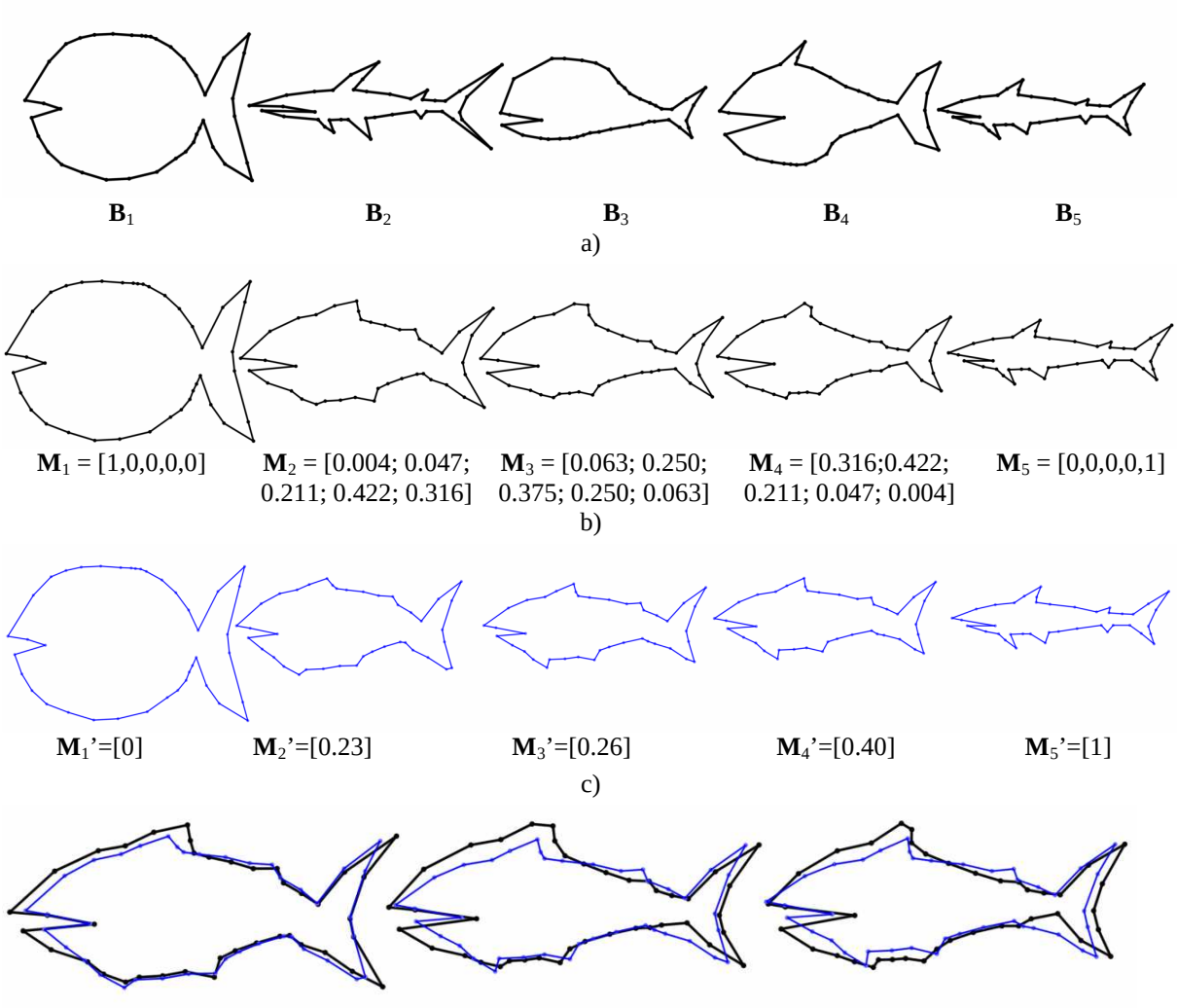


Figure 5.6: a) basis shapes, b) combination of basis shapes, c) orthogonal projections, d) comparison of original shapes (black) and their approximation (blue).

In the previous example we showed how the orthogonal projection can be used do reduce the dimension of the morphing space by choosing specific basis shapes and projecting elements of the original morphing to the lower dimensional morphing space. We also showed the difference between the shape in the original space and the shape in the space with lower dimension. The concept of the orthogonal projection can be also used for an analysis of unknown shapes, i.e., shapes where we have a geometrical description but we do not know the representation of the shape with respect to the basis shapes. For instance, let us have a shape processing system which has some built-in basis shapes. New shapes which enter the system are expressed in terms of built-in basis shapes by projecting a new shape to the space of shapes generated by the built-in basis shapes.

In the following example (Figure 5.7) we will demonstrate a computation of linearly independent set of basis shapes. Let us have a set of four basis shapes – $\mathbf{B}_0$, $\mathbf{B}_1$, $\mathbf{B}_2$ and $\mathbf{B}_3$. Intuitively, the shape $\mathbf{B}_3$ can be obtained by combining shapes $\mathbf{B}_1$ and $\mathbf{B}_2$ with $\mathbf{B}_0$ as a zero element shape. This can be confirmed by computing the Gram matrix (Section 5.3.4) of MVS spanned by $\mathbf{B}_0$, …, $\mathbf{B}_3$ which is singular. Alternatively, we can try to

project the shape $\mathbf{B}_3$ in a subspace spanned by the shapes $\mathbf{B}_0$, $\mathbf{B}_1$, $\mathbf{B}_2$. Note that it is possible because the shapes $\mathbf{B}_0$, $\mathbf{B}_1$, $\mathbf{B}_2$ are linearly independent. As a result, of the orthogonal projection, coefficients $\lambda_1 = 1.0$, $\lambda_2 = 1.0$ are obtained. Thus, the projected shape is

$$\mathbf{B}_3' = \mathbf{B}_0 + 1.(\mathbf{B}_1 - \mathbf{B}_0) + 1.(\mathbf{B}_2 - \mathbf{B}_0).$$

By computing the distance (Section 5.3.3) between the projected shape $\mathbf{B}_3'$ and the original shape $\mathbf{B}_3$ it can be seen that the shapes are the same, which means that the shape $\mathbf{B}_3$ can be expressed in terms of shapes $\mathbf{B}_0$, $\mathbf{B}_1$, $\mathbf{B}_2$. Hence, by removing the shape $\mathbf{B}_3$ from the set a linearly independent set is obtained. Note that, instead of removing $\mathbf{B}_3$, e.g., the shape $\mathbf{B}_1$ can be removed. Then, the shape $\mathbf{B}_1$ can be still obtained as:

$$\mathbf{B}_1 = \mathbf{B}_0 + (-1).(\mathbf{B}_2 - \mathbf{B}_0) + 1.(\mathbf{B}_3 - \mathbf{B}_0)$$

and the shapes $\mathbf{B}_0$, $\mathbf{B}_2$, $\mathbf{B}_3$ form a linearly independent set.

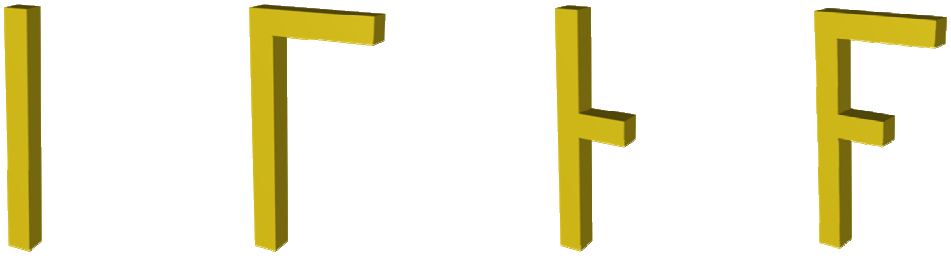


Figure 5.7: An example of linearly dependent shapes.

5.5.4. Exploration of space of shapes

In this example we used four hand gestures as basis shapes. We used the Bézier morphing to produce an animation of a waving hand. It can be seen that the basis shapes are too extreme for the waving hand animation, i.e., the fingers are bent too much. The Bézier morphing attenuates the influence of the intermediate shapes. We used a sequence of basis shapes $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_0$, i.e., the animation starts and ends in the shape $\mathbf{M}_0$. The basis shapes are depicted in Figure 5.8a) and some frames of the animation are depicted in Figure 5.8b).

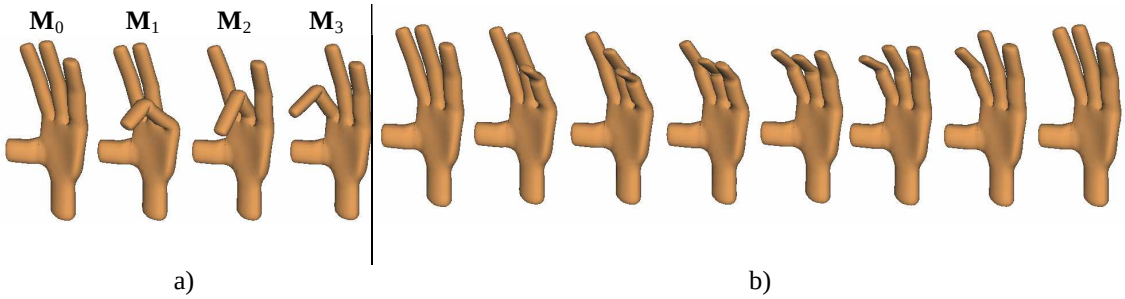


Figure 5.8: a) basis shapes, b) some frames of the resulting waving hand animation.

Note that this animation can be produced by any tool which supports a linear combination of basis shapes (e.g., 3ds max). However, using the linear combination, the user must control all four weights to precisely model the final shape. On the other hand,

using the Bézier morphing the user manipulates just the curve parameter, while the weights are generated automatically by Bernstein polynomials. Clearly, controlling the animation by direct manipulation of the weights is more general, but the Bézier morphing is easier to control.

5.6. Summary and possible extensions

In this chapter we described a generalization of morphing, called multimorphing, which extends the idea of classical morphing between two shapes to morphing between multiple shapes. The classical morphing generates shapes between the initial and the final shape and the morphing transition is controlled by one transition parameter (in animation understood as a time). By gradually changing the transition parameter a 1-dimensional space of shapes is generated. In the multimorphing setting it is not straightforward how to systematically generate shapes; therefore we proposed an approach where we handle the morphing space as an analogy of an affine space and a vector space.

In the examples we showed how to generate new shapes by affine and linear combinations. We can also use our apparatus to analyze some existing set of shapes – we showed how to find basis shapes and how to express elements of the set with respect to the basis shapes. A general technique of orthogonal projection is used to compute a representation of shapes with respect to the basis shapes. Using the orthogonal projection we can project shapes from generally higher dimensional space to a lower dimensional space, thereby reduce the data needed to express the shape. By introducing the norm on the space of shapes we also know the error of the projection. We also discussed some user interaction aspects of generation of new shapes (i.e., barycentric coordinates and curves in the morphing space), which could help when implementing an editor for shape generation.

We considered shapes represented as triangular meshes and polygons; nonetheless, the method works just with vertex positions, so it can be used in any representation where the shape is induced by the points (e.g., point based representation). Additionally, the idea of Bézier morphing using de Casteljau algorithm is very general, so we think it could be used for morphing of any representation as long as a morphing between two objects is defined.

By introducing an analogy of the morphing space and a vector space and by introducing a norm on the morphing space we open a big area for a future research, because we can now generalize concepts well known from the Euclidean space as, e.g., the area. Another interesting field is the analysis of sets of shapes, where we used a concept of orthogonal projection. In this chapter we showed how to choose linearly independent basis shapes to represent some set of shapes, in the future research, it would be interesting to compute basis shapes not from the set of interest but to compute some artificial basis shapes so that the number of basis shapes is as low as possible and all shapes of the set can be represented with respect to the artificial basis. This would probably lead to some optimization problem, e.g., least squares fitting.

We also think that our approach can be generalized for other representations as well. For example, raster images consist of pixels. In a similar manner we could define affine combination of raster images, a vector representation describing a difference between zero element image and some basis image, inner product on a space of images, etc.

6. Normal computation for deformable meshes

In this chapter we will focus on one of the later stages of the morphing process – normal computation. In the case of triangular meshes, the normal vectors are essential for rendering. At this stage, we consider that some morphing technique computed two isomorphic meshes which will be interpolated linearly. Together with the shape interpolation, the normal vectors must be interpolated as well to be able to render the intermediate shapes. We have already described one solution in Section 4.3. In this chapter we will describe a drawback of this method and we will suggest some alternative solutions. The contribution of this chapter does not apply only for mesh morphing – generally it can be used to compute normal vectors of any deformable meshes, e.g., garment simulation, collision detection of deformable meshes, etc.

In the further text we will briefly review a related work (Section 6.1). Then we will introduce a t-variant cross product which is a basis of a new method for the computation of normal vectors for deformable isomorphic meshes (Section 6.2). In the case of the triangular meshes, we distinguish between face normals (normals of triangles) and vertex normals. It will be described in the sections 6.3 and 6.4.

This research was done in cooperation with Anders Hast, Univesity of Gävle, Sweden, who sketched many interesting ideas. Some of them were further investigated and they also appear in this chapter.

6.1. Related work

A normal vector to a surface is a vector perpendicular to it. In the differential geometry the normal vector of a smooth surface can be computed by taking cross-product of partial derivatives. However, the triangular mesh is a piecewise linear approximation of a real surface, thus the normal is usually computed only for the vertices of the mesh and for the rest of the surface the normals are interpolated. Basically, the normal vector determines an orientation of the surface; therefore it is extensively used in shading, collision detection or mesh editing.

6.1.1. Vertex normal computation

Vertex normals can be computed from an arbitrary triangle mesh in many different ways. One of the most often used approaches is to compute the vertex normal as a

weighted average of normals of faces which are incident to the vertex. An overview and a comparison of different weighting schemes is given in [Jin05]. The basic relation is:

$$\mathbf{n}_V = \sum_{i=1}^{n} w_i \mathbf{n}_i \,, \tag{6.1}$$

where $\mathbf{n}_V$ is the vertex normal of the vertex $\mathbf{V}$, $\mathbf{n}_i$ is a normal of a face adjacent to the vertex $\mathbf{V}$ and w_i is a weight of the face normal $\mathbf{n}_i$. The simplest form of the weighting scheme is to set all weights w_i to 1.0, then each face normal contributes equally to the resulting vertex normal. Other schemes consider the fact that faces adjacent to the processed vertex have different size and therefore they contribute differently to the resulting vertex normal. Some weighting schemes are summarized in Table 6.1.

Weighting scheme	**Description**
equal weighting	all faces contribute equally to the resulting face normal, regardless the area or the angle
angle weighting	considers the angle under which the face is incident to the vertex, i.e., faces with larger angles contribute more to the resulting vertex normal
area weighting	considers areas of faces incident to the vertex, i.e., larger faces contribute more to the resulting vertex normal
inverse area weighting	considers areas of faces incident to the vertex, smaller faces contribute more to the resulting vertex normal

Table 6.1: Some weighting schemes for the vertex normal computation.

The use of a specific weighting scheme depends on the application domain. Max [Max99] tested the aforementioned weighting schemes on random analytical cubic surfaces. On his testing data the inverse area weighting scheme generated the most accurate vertex normals. Jin et al. [Jin05] tested the weighting schemes on parameterized surfaces and marching tetrahedral-tessellated surfaces. On these testing data the area weighting scheme generated the most accurate results. The area weighting scheme was also suggested by Lengyel [Len04] as a method giving more appealing[19] vertex normals for some models. The vertex normal computed using the area weighting scheme is computed as follows:

$$\mathbf{n}_v = \sum_{i=1}^{n} \mathbf{n}_i \left\|\mathbf{e}_i\right\| \left\|\mathbf{e}_{i+1}\right\| \sin\theta_i = \sum_{i=1}^{n} \mathbf{n}_i \left\|\mathbf{e}_i \times \mathbf{e}_{i+1}\right\| = \sum_{i=1}^{n} \mathbf{e}_i \times \mathbf{e}_{i+1} \,, \tag{6.2}$$

where $\mathbf{n}_i$ is the normal of the face $\mathbf{f}_i$ formed by the vertex $\mathbf{V}$ and edges $\mathbf{e}_i$, $\mathbf{e}_{i+1}$, i = 1, …, n, where n is the number of faces adjacent to the vertex $\mathbf{V}$. Note, that faces $\mathbf{f}_i$ may or may not form a closed triangle fan. The main advantage of the area weighting scheme is that individual face normals which appear in the sum in Eq. 6.2 do not need to be normalized. Normalization is computationally expensive unless hardware is used, not the least depending on the square root involved.

[19] The vertex normals were used for smooth shading.

6.1.2. Deformable meshes

The area of computer animation could be divided into a rigid-body motion and a soft-body motion. In the rigid-body motion the relative position of each two vertices stays fixed during the transformation and the object transforms as one entity [Kar04]. Examples of the rigid-body motion are a rotation or a translation.

The rigid-body transformation can be expressed by a transformation matrix **A**, then the normal field of the transformed object under a rigid-body motion is transformed by the inverse transpose of the Jacobian matrix **J** of the transformation matrix **A**, i.e., $(\mathbf{J}^{-1})^T$ [Gom99]. Moreover, if the transformation is linear, the normal field is transformed by the matrix **A** directly, because in the case of a linear transformation **A** it holds that $(\mathbf{J}^{-1})^T=\mathbf{J}=\mathbf{A}$.

In the soft-body motion there are no restrictions on change of a relative position of two vertices; each vertex can travel along its trajectory independently on other vertices. Hence, no global transformation can be applied on the normal field, as in the rigid-body motion case. In this chapter we will deal with soft-body motion, however, our techniques are general enough so that they can be used for linear transformations as well.

A lot of methods for computing vertex normals for static meshes exist. In the case of deformable meshes, there are basically two approaches how to compute normal vectors – deform the mesh and *recompute* the normal vectors using some standard approach or *interpolate* the normal vectors during the mesh deformation. The recomputation approach takes usually more time but it computes exact normals (with respect to the current mesh shape). The interpolation approaches are usually faster, but they may be inaccurate because the interpolation need not reflect the true mesh shape.

Besides the recomputation and the interpolation approach, Alexa et al. [Ale00a] suggest to compute normals only for the first frame of the animation and leave them unchanged for the remaining frames. This may work in the cases when a mesh does not deform very dramatically and when only several in-between frames are needed, however it is not very useful for morphing because morphing might involve a dramatic shape transformation and a lot of in-between frames.

6.2. t-variant cross product

In this section we will describe an essential tool for the computation of a normal vector of a moving plane.

Let us recall that the cross product is an operator which takes two non-parallel vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and computes a vector **n** which is perpendicular to both $\mathbf{v}_1$ and $\mathbf{v}_2$. In the computer graphics, it is often used to compute a normal of a triangle. We generalized the cross product for a computation of a normal of a deforming triangle. The triangle can arbitrarily deform as long as the vertices of the triangle travel along straight lines with a constant velocity (so-called *linear motion*). The situation is depicted in Figure 6.1.

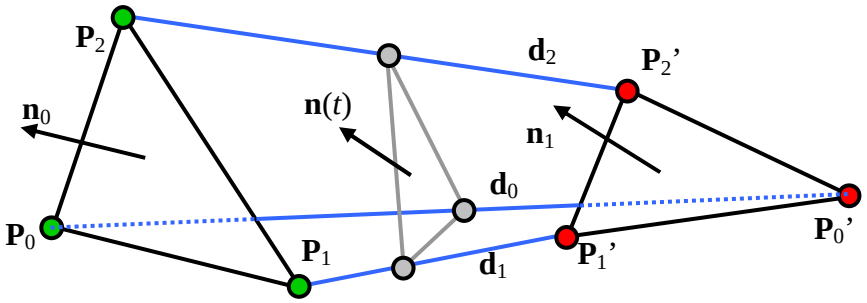


Figure 6.1: A linear motion of a triangle from an initial position (vertices $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$) to a final position (vertices $\mathbf{P}_0$', $\mathbf{P}_1$', $\mathbf{P}_2$'). Vertices travel along linear trajectories $\mathbf{P}_0(t)$, $\mathbf{P}_1(t)$ and $\mathbf{P}_2(t)$, $\mathbf{n}_0$ is the normal of the triangle in the initial position, $\mathbf{n}_1$ is the normal of the triangle in the final position and $\mathbf{n}(t)$ is the t-variant face normal depending on vertex trajectories $\mathbf{P}_0(t)$, $\mathbf{P}_1(t)$ and $\mathbf{P}_2(t)$.

If the vertices move from the source position $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ to the target position $\mathbf{P}_0$', $\mathbf{P}_1$', $\mathbf{P}_2$' along a straight line, the trajectory can be described as $\mathbf{P}_0(t)=\mathbf{P}_0+t(\mathbf{P}_0'-\mathbf{P}_0)$ (and analogously $\mathbf{P}_1(t)$, $\mathbf{P}_2(t)$). The normal of the triangle in the source position can be computed as $\mathbf{n}_0=(\mathbf{P}_1-\mathbf{P}_0)\times(\mathbf{P}_2-\mathbf{P}_0)$. If we want a relation for the t-variant face normal, we have to substitute the vertex trajectories $\mathbf{P}_0(t)$, $\mathbf{P}_1(t)$, $\mathbf{P}_2(t)$ into the cross product, i.e.:

$$\mathbf{n}(t) = (\mathbf{P}_1(t) - \mathbf{P}_0(t)) \text{ x } (\mathbf{P}_2(t) - \mathbf{P}_0(t)) = [n_x(t), n_y(t), n_z(t)], \text{ where}$$

$$\begin{aligned} n_x(t) &= [P_{1y}(t)-P_{0y}(t)].[P_{2z}(t)-P_{0z}(t)]-[P_{2y}(t)-P_{0y}(t)].[P_{1z}(t)-P_{0z}(t)] \\ n_y(t) &= -[P_{1x}(t)-P_{0x}(t)].[P_{2z}(t)-P_{0z}(t)]+[P_{2x}(t)-P_{0x}(t)].[P_{1z}(t)-P_{0z}(t)] \\ n_z(t) &= [P_{1x}(t)-P_{0x}(t)].[P_{2y}(t)-P_{0y}(t)]-[P_{2x}(t)-P_{0x}(t)].[P_{1y}(t)-P_{0y}(t)]. \end{aligned} \quad (6.3)$$

Each component $n_x(t)$, $n_y(t)$, $n_z(t)$ of the normal vector $\mathbf{n}(t)$ is a degree two polynomial[20]. After expanding Eq. 6.3 and organizing the terms by the power of t, Eq. 6.4 is obtained:

$$\begin{aligned} \mathbf{n}(t) = {} & (\mathbf{P}_1\times\mathbf{P}_2 - \mathbf{P}_1\times\mathbf{P}_0 - \mathbf{P}_0\times\mathbf{P}_2) + \\ & t(\mathbf{d}_1\times\mathbf{P}_2 + \mathbf{P}_1\times\mathbf{d}_2 - (\mathbf{d}_1\times\mathbf{P}_0 + \mathbf{P}_1\times\mathbf{d}_0) - (\mathbf{d}_0\times\mathbf{P}_2 + \mathbf{P}_0\times\mathbf{d}_2)) + \\ & t^2(\mathbf{d}_1\times\mathbf{d}_2 - \mathbf{d}_1\times\mathbf{d}_0 - \mathbf{d}_0\times\mathbf{d}_2), \end{aligned} \quad (6.4)$$

where $\mathbf{d}_i = \mathbf{P}_i'-\mathbf{P}_i$, i = 1, 2, 3 is the vector of the trajectory. Eq. 6.4 represents the *t-variant cross product* (TCP). Note, that if two trajectory vectors $\mathbf{d}_i$, $\mathbf{d}_j$ are the same, i.e., $\mathbf{d}_i=\mathbf{d}_j$, $i\neq j$, i, j=1, 2, 3, then TCP is linear, i.e., the quadratic term disappears. If all trajectory vectors are the same, i.e., when the plane just translates, then TCP is constant. It is clear that if two points of the face do not move then TCP is linear, if all three points do not move then TCP is constant.

This fact can be used for an analysis of the motion and for simplification of TCP. We can compare[21] the trajectory vectors and use the appropriate form of TCP. For instance, if two trajectory vectors are almost the same, the coefficient of the quadratic term would yield a very small absolute value, which may cause numerical problems when evaluating the polynomial. However, we can explicitly unify similar trajectories, which turns TCP to a linear polynomial. By unifying the trajectories we know exactly what

[20] The trajectory is expressed by a degree one polynomial; in the cross product, two degree one polynomials are multiplied, therefore the resulting polynomial will have degree two.

[21] Since we compare vectors, we have to compare a direction (i.e., an angle between the vectors) and a length.

kind of error we introduce, i.e., the angle difference or the trajectory length difference. On the other hand, similar trajectories lead to small absolute values of the polynomial coefficients. If the coefficient value is very small, we could disregard it, but we would have to set some artificial limit value, so that the coefficient values smaller than the limit value are disregarded. Additionally, it is not completely clear how big error we introduced by disregarding some term. Also, very small values of the coefficients may cause numerical problems when evaluating the polynomial[22].

Note that the coefficients are computed using a vector addition and the cross product, thus it can be precomputed very efficiently on hardware with a vector operation support.

After computing the coefficients of the degree two polynomials of the t-variant cross product the intermediate normals can be evaluated by evaluating the degree two polynomials. Generally, a degree two polynomial $at^2 + bt + c$ can be evaluated by 3 multiplications and 3 additions. It can be further optimized by using the Horner scheme. It transforms the quadratic polynomial to the form $t(at + b) + c$. To evaluate this form, 2 multiplications and 2 additions are required. In the case of t-variant cross product we have to evaluate three degree two polynomials (one for each component of the normal vector), thus we need 6 multiplications and 6 additions. Horner scheme is the fastest method for evaluating a polynomial at a single point. However, an animation usually requires evaluating a polynomial at several evenly spaced values. In this case, an incremental scheme based on the forward differencing can be used. For degree two polynomials, the forward differencing method requires 2 additions per one evaluation (plus some setup needed to initialize the incremental scheme). Thus, the t-variant cross product can be evaluated by 6 additions using the forward differencing method.

Sometimes, the normal vectors are required to be unit length. In this case the intermediate normal vectors obtained by evaluating the t-variant cross product must be normalized. On the other hand the length of the normal vector obtained by evaluating the t-variant cross product is proportional to the area of the triangle formed by the moving vertices $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$ (Figure 6.1), i.e., the normal vector is *implicitly* weighted by an area.

6.3. Face normal computation

In this section we will deal with face normal vector computation of deforming triangular meshes. Face normals are essential vectors for vertex normal computation using some weighting schemes (see Section 6.1.1); they can be also used for the shading.

As stated before, there are basically two approaches for the normal vector computation – a recomputation approach and an interpolation approach. In the case of the recomputation approach a mesh is deformed and the face normals are recomputed from the scratch. In the case of the interpolation approach we set several key-frames for which exact face normals are computed (i.e., interpolation constrains) and the intermediate (i.e., between the key-frames) face normals are interpolated using some interpolation technique. In the further text we will propose several approaches how to compute face normals.

[22] Such problems are related to the floating point numbers representation and they usually appear when doing numerical operations between a very big and a very small number.

6.3.1. t-variant cross product

The t-variant cross product can be directly used for face normal computation. The advantage of this approach is that the coefficients of the degree two polynomials can be precomputed during a preprocessing stage. During the mesh deformation we just evaluate 3 degree two polynomials for each face. Of course, the precomputed coefficients must be stored somehow. The space cost is 9 floating point values (i.e., 3 quadratic expressions, one for each component of the normal vector) for each triangle.

The recomputation approach consists of a computation of two linearly independent vectors $\mathbf{v}_1$, $\mathbf{v}_2$ and a cross product computation between $\mathbf{v}_1$, $\mathbf{v}_2$. The vectors $\mathbf{v}_1$, $\mathbf{v}_2$ are usually computed by taking the vertices of the faces, i.e., $\mathbf{v}_1 = \mathbf{P}_1 - \mathbf{P}_0$, $\mathbf{v}_2 = \mathbf{P}_2 - \mathbf{P}_0$. It requires 6 additions. Furthermore, the cross product requires 6 multiplications and 3 additions. Thus, the recomputation approach requires 9 additions and 6 multiplications per face. In the case of triangular meshes, an edge is shared by two triangles (except the boundary edges); therefore, the computation of the vectors $\mathbf{v}_1$, $\mathbf{v}_2$ can be used for the neighboring triangles as well. Using the Horner scheme, the face normal computation based on the t-variant cross product requires 6 multiplications and 6 additions per face which is less than the number of arithmetical operations required by the recomputation approach.

The length of the normal is proportional to the area of the face $\mathbf{P}_0$, $\mathbf{P}_1$, $\mathbf{P}_2$. If normalization is required, the normalized t-variant cross product is defined as follows:

$$\hat{\mathbf{n}}(t) = \frac{(\mathbf{P}_2(t) - \mathbf{P}_1(t)) \times (\mathbf{P}_2(t) - \mathbf{P}_0(t))}{\left|(\mathbf{P}_2(t) - \mathbf{P}_1(t)) \times (\mathbf{P}_2(t) - \mathbf{P}_0(t))\right|} \tag{6.5}$$

The resulting relation is clearly more complicated that the t-variant cross product because it requires computation of the square root.

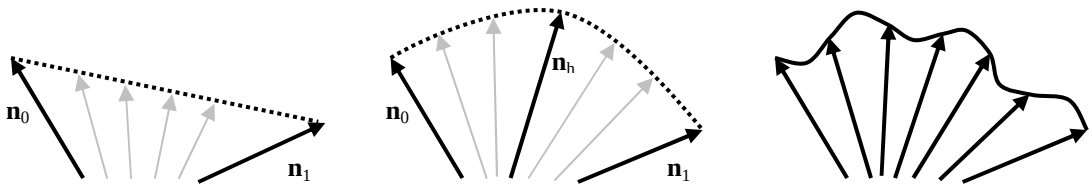
6.3.2. Lagrange interpolation

Linear interpolation (degree one Lagrange interpolation) of normal vectors is described by the following expression:

$$\mathbf{n}(t) = \mathbf{n}_0 + t(\mathbf{n}_1 - \mathbf{n}_0), \tag{6.6}$$

where $\mathbf{n}_0$ is an initial normal and $\mathbf{n}_1$ is a final normal. This approach is used, e.g., for spatial normal interpolation in Phong shading. In fact, the Lagrange interpolation interpolates the vector as if it was a point. It is fast but not sufficient because the intermediate normals are far from being perpendicular to the triangle; furthermore the intermediate normals are not of unit length (Figure 6.2a).

Higher degree Lagrange interpolation fits better the true normal behavior but there is always a tradeoff between a better fit and an oscillation due to the higher degree of an interpolation function. Also, unit length is not preserved. Figure 6.2b) shows degree two Lagrange interpolation which needs an additional intermediate normal $\mathbf{n}_h$ to fit a quadratic interpolation curve. Figure 6.2c) shows a higher degree Lagrange interpolation where a number of intermediate normals is required to precompute the interpolation curve, it can be seen that due to the high degree of the interpolation polynomial the normal is oscillating.



a) b) c)

Figure 6.2: a) a linear interpolation, b) a quadratic interpolation, c) a higher degree interpolation.

6.3.3. Vector SLERP

SLERP (Spherical Linear intERPolation) is a technique for interpolation of vectors, which maintains unit length. It is defined as:

$$\mathrm{SLERP}(\mathbf{n}_0, \mathbf{n}_1, t) = \frac{\sin((1-t)f)\mathbf{n}_0 + \sin(tf)\mathbf{n}_1}{\sin f}, \qquad (6.7)$$

where $\mathbf{n_0}$, $\mathbf{n_1}$ is the initial and target normal, respectively, f is the angle between $\mathbf{n}_0$, $\mathbf{n}_1$. This approach preserves unit length normals but again the direction of the normal is far from being perpendicular to the intermediate triangle.

6.3.4. Spherical de Casteljau

Slightly better idea based on [Ježek, F., personal communication, 2005] is to compute several intermediate normals exactly (as in the higher degree Lagrange interpolation) and interpolate these vectors on the surface of the unit sphere. In this case, a generalized de Casteljau algorithm for spherical interpolation can be used. The de Casteljau algorithm is well known for a fast generation of Bézier curves. It is basically a recursive subdivision of the control polygon which converges to the Bézier curve. The generalization of de Casteljau algorithm for fast vector interpolation means to replace line segments with the shortest great circle arcs, i.e., the line segment subdivision step is replaced by SLERP of consecutive intermediate normals. It is demonstrated in Figure 6.3a), there is an initial normal $\mathbf{n}_{0,0}$, a final normal $\mathbf{n}_{0,3}$ and two intermediate normals $\mathbf{n}_{0,1}$ and $\mathbf{n}_{0,2}$ (e.g., in the time t=0.33 and t=0.66). To compute the normal $\mathbf{n}_f$ at the time t the normals $\mathbf{n}_{1,0}$, $\mathbf{n}_{1,1}$, $\mathbf{n}_{1,2}$ are computed by applying SLERP on pairs of successive normals, i.e., $\mathbf{n}_{1,0}$=SLERP($\mathbf{n}_{0,0}$, $\mathbf{n}_{0,1}$, t), $\mathbf{n}_{1,1}$=SLERP($\mathbf{n}_{0,1}$, $\mathbf{n}_{0,2}$, t), $\mathbf{n}_{1,2}$=SLERP($\mathbf{n}_{0,2}$, $\mathbf{n}_{0,3}$, t). This process is repeated until one single normal $\mathbf{n}_{3,1}$ is obtained.

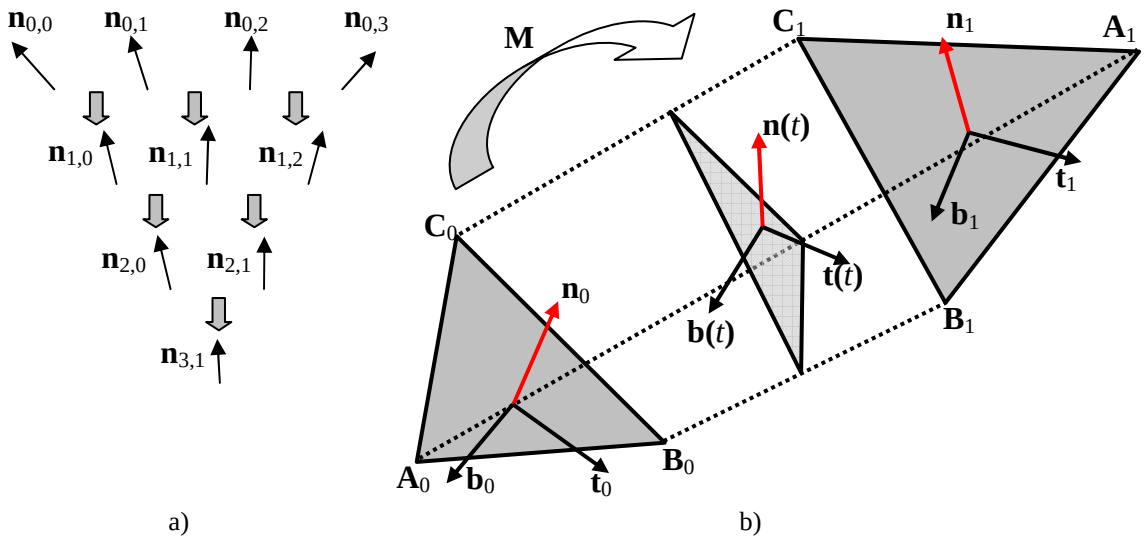


Figure 6.3: a) a demonstration of de Casteljau algorithm for vectors, the gray thick arrow represents application of SLERP on two successive normals, b) quaternion interpolation of face normal, transformation **M** transforms triangle $\mathbf{A}_0$, $\mathbf{B}_0$, $\mathbf{C}_0$ to the triangle $\mathbf{A}_1$, $\mathbf{B}_1$, $\mathbf{C}_1$ and $\mathbf{n}_0$ to $\mathbf{n}_1$. Frame $\mathbf{F}_1=(\mathbf{t}_1, \mathbf{b}_1, \mathbf{n}_1)$ is computed by transforming $\mathbf{F}_0=(\mathbf{t}_0, \mathbf{b}_0, \mathbf{n}_0)$ by the transformation **M**.

6.3.5. Quaternion SLERP

In this section we will describe a new idea which uses quaternions to interpolate face normals. A brief introduction to quaternions is in Appendix B. First, let us recall two important identities which we will use in the following description:

(i) Rotation in 3d around an axis **a** by an angle φ is represented by a 3x3 rotation matrix **R** or by a quaternion **q** [Sho85, Ebe04]. Both representations are equivalent.

(ii) Matrix **R** of rotation transformation is orthogonal and its columns (and rows) are of unit length, thus the matrix of rotation forms an orthonormal *frame*. By the term frame we understand here an orthogonal basis of $\mathbf{E}^3$. The orthonormal frame is formed by three unit length vectors.

The central idea is to set an orthogonal frame $\mathbf{F}_0 = (\mathbf{n}_0, \mathbf{t}_0, \mathbf{b}_0)$ for the initial face, set an orthogonal frame $\mathbf{F}_1 = (\mathbf{n}_1, \mathbf{t}_1, \mathbf{b}_1)$ for the final face and interpolate between $\mathbf{F}_0$ and $\mathbf{F}_1$ using QSLERP. To set a frame $\mathbf{F} = (\mathbf{n}, \mathbf{t}, \mathbf{b})$ means to associate one vector of the frame with the normal **n** of the face, to choose the tangent vector **t** which lies in the plane of the face (e.g., an edge of the triangle) and compute the binormal vector **b** by taking the cross product of **n** and **t**, i.e., $\mathbf{b} = \mathbf{n} \times \mathbf{t}$. By organizing the column vectors **t**, **b**, **n** into a 3x3 matrix $\mathbf{R} = [\ \mathbf{t} \mid \mathbf{b} \mid \mathbf{n}\ ]$ a rotation matrix **R** is obtained (ii). The matrix **R** can be converted into a quaternion representation **q** [Ebe04]. By converting the frames $\mathbf{F}_0$, $\mathbf{F}_1$ into the quaternion representation, quaternions $\mathbf{q}_0$, $\mathbf{q}_1$ are obtained. They represent the initial and the final orientation of the face. To obtain intermediate normals $\mathbf{n}(t)$, quaternions are interpolated using QSLERP. The interpolated quaternion $\mathbf{q}(t)$ can be converted back to the orthogonal matrix $\mathbf{R}(t)$ and the normal is extracted from the last column of $\mathbf{R}(t)$. It is demonstrated in Figure 6.3b) where the triangle $\mathbf{A}_0$, $\mathbf{B}_0$, $\mathbf{C}_0$ with the frame $\mathbf{F}_0$ is transformed into the triangle $\mathbf{A}_1$, $\mathbf{B}_1$, $\mathbf{C}_1$ with the frame $\mathbf{F}_1$. The intermediate quaternion $\mathbf{q}(t)$ is converted to the frame $(\mathbf{t}(t), \mathbf{b}(t), \mathbf{n}(t))$ and the intermediate normal $\mathbf{n}(t)$ is extracted.

The question is how to set frames $\mathbf{F}_0$ and $\mathbf{F}_1$. One possibility is to use the following method. First we compute a transformation matrix **M** which transforms the vertices of the initial triangle into the vertices of the final triangle; moreover, we want the transformation **M** to transform also the initial normal to the final normal. All conditions can be expressed by a matrix equation, i.e.:

$$\mathbf{M}.[\mathbf{A}_0 \mid \mathbf{B}_0 \mid \mathbf{C}_0 \mid \mathbf{n}_0] = [\mathbf{A}_1 \mid \mathbf{B}_1 \mid \mathbf{C}_1 \mid \mathbf{n}_1], \tag{6.8}$$

where $\mathbf{A}_0$, $\mathbf{B}_0$, $\mathbf{C}_0$, $\mathbf{n}_0$ can be written as column vectors, e.g., $\mathbf{A}_0 = [a_{0x}, a_{0y}, a_{0z}, 1.0]^T$. The matrix **M** can be computed as:

$$\mathbf{M} = [\mathbf{A}_1 \mid \mathbf{B}_1 \mid \mathbf{C}_1 \mid \mathbf{n}_1].[\mathbf{A}_0 \mid \mathrm{B}_0 \mid \mathbf{C}_0 \mid \mathbf{n}_0]^{-1}. \tag{6.9}$$

Then, we set an arbitrary frame $\mathbf{F}_0$ for the initial face. The vectors $\mathbf{b}_1$ and $\mathbf{t}_1$ of the frame $\mathbf{F}_1$ are computed as follows:

$$\begin{aligned} \mathbf{b}_1 &= \mathbf{M}\,.\mathbf{b}_0 \\ \mathbf{t}_1 &= \mathbf{n}_1 \times \mathbf{b}_1 \end{aligned} \tag{6.10}$$

Since the initial frame is chosen arbitrarily, the problem is that there is an infinite number of the initial and the final frame configurations. The choice of the initial and the final frame influences the quality of the interpolation. Therefore, the best configuration of frames (from the interpolation quality point of view) is needed. Unfortunately, we did not succeed in deriving the rule for the best frame configuration. Clearly, the best configuration of frames can be approximately computed by brute force testing of random configurations, however, it is very time consuming. Therefore, an algebraic solution of this problem belongs to our future work.

6.3.6. Comparisons and discussion

We compared approaches described in the sections 6.3.1 – 6.3.5 from two points of view. We observe an error of the interpolation and the time consumption. The error of the interpolation is measured as follows. The time interval (usually <0, 1>) is sampled and in each sample an angle between an exact normal (computed by the cross product) and an interpolated normal (computed by some method described above) is computed. The error of the interpolation scheme is expressed as a sum of angles:

$$e = \sum_{i=0}^{n-1} |a_i|, \tag{6.11}$$

where n is the number of samples, a_i is the angle between the exact normal and the interpolated normal in the sample i.

We tested different interpolation approaches on four morphing animations, where a mesh composed of triangles deforms from one shape into the other shape so that the trajectories of individual vertices are linear. Numbers in Table 6.2 represent an error of the interpolation for the whole mesh. It is computed as a sum of interpolation errors e_j of individual triangles, i.e., $E = \Sigma e_j$, j=0, ..., m-1, where m is the number of triangles of the mesh and e_j is the interpolation error of the j-th triangle (Eq. 6.11). The numbers in Table 6.2 must be always viewed with respect to the number of triangles and the

number of samples. Rather than the absolute values it is more important to compare ratios between different methods, e.g., it can be seen that the quadratic interpolation is almost 4-times better than a simple linear interpolation. The second, third and fourth row show results of the normal interpolation described in the Section 6.3.2. The "t-variant cross product" row shows results of the approach from the Section 6.3.1. The row "Vector SLERP" shows results of method described in Section 6.3.3. Results of vector interpolation using generalized de Casteljau algorithm (Section 6.3.4) are shown in the "Spherical de Casteljau" row. The last row shows the quality of normal interpolation using quaternions (Section 6.3.5).

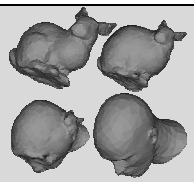
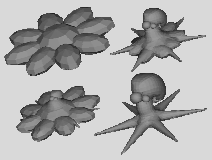
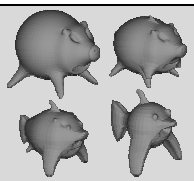
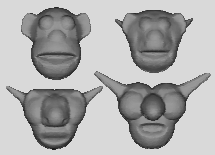
Normal interpolation approach	 **# faces: 15300**	 **# faces: 44700**	 **# faces: 35180**	 **# faces: 21040**
Linear	18916	55756	44078	20061
Quadratic	4820	15792	9681	5425
Cubic	1906	6444	3640	1572
t-variant cross product	0	0	0	0
Vector SLERP	19427	57999	44972	20861
Spherical de Casteljau	7039	21772	16068	7647
Quaternion SLERP	17766	53492	42831	19202

Table 6.2: Comparison of error of different normal interpolation approaches. The images in the table heading show the example mesh deformation. The numbers of faces of the example deformations are indicated below the images.

From Table 6.2 it is clear that the best approach from the interpolation quality point of view is the t-variant cross product approach which produces exact normals, i.e., it is not really an interpolation approach. Other proposed approaches (except the quaternion approach) handle the normal vectors as usual vectors, i.e., it is not respected that normals vector are perpendicular to some surface and in fact, these approaches can be used for an interpolation of any vectors. The worst but one results are obtained by the linear interpolation, better results can be achieved by the quadratic or the cubic interpolation (up to 90% improvement). The worst results were achieved by SLERP of normal vectors. The quality of interpolation of the spherical de Casteljau approach depends on how many intermediate normals we use. In this case we used initial, final normal and two additional intermediate normals. Quaternion SLERP approach has slightly better results (on average about 5%) than simple linear interpolation.

Next we will compare various approaches from the time consumption point of view. We will not present an exact timing since it is dependent on how various elementary operations (SLERP, cross-product, polynomial evaluation, etc.) are implemented. Some of them can be implemented in hardware so that their execution can be very fast. We will express the time consumption in terms of elementary operations, so that the reader must decide which approach is the most suitable according to the actual application platform.

Elementary operations used in Table 6.3 are polynomial evaluation, SLERP and quaternion to matrix conversion. Polynomial evaluation is used in the Lagrange interpolation approach and in the t-variant cross product approach. Polynomials can be evaluated by Horner scheme which saves some multiplications in comparison with usual evaluation of polynomial in monomial form (see Section 6.2). SLERP is used in the Vector SLERP approach and in the de Casteljau approach. SLERP requires evaluation trigonometric functions which are computationally expensive, but it can be speeded up by an incremental approach described by Barrera et al. [Bar04]. Quaternion to matrix conversion is used in the Quaternion SLERP approach. Note that in our case we need to extract only one column from the matrix, i.e., the normal.

It is also important to decide whether we need a "random access" to the deformation or just a "sequential access". If we consider that the mesh deformation is parametrized by the time, the random access means that we can jump from one time instant to another without any limitation. The sequential access means that the mesh deformation is evaluated is evenly spaced time instants in an increasing or a decreasing order. In this case, incremental methods (using temporal coherence) for computing SLERP [Bar04] or polynomial evaluation [Has03] can be used.

Method	Normalization	Computation
Linear interpolation	Yes	3 linear interpolations (Eq. 6.6)
Quadratic, cubic interpolation	Yes	3 evaluation of degree two (quadratic interpolation) or degree three (cubic interpolation) polynomials
t-variant cross product	yes (Eq. 6.4) no (Eq. 6.5)	3 evaluation of degree two polynomials (Eq. 6.4) 3 evaluation of rational polynomial (Eq. 6.5)
Vector SLERP	No	1 SLERP (Eq. 6.7)
Spherical de Casteljau	No	n(n-1)/2 SLERPs, where n is the number in precomputed normals
Quaternion SLERP	No	1 SLERP, quaternion to matrix conversion

Table 6.3: Comparison of various normal interpolation approaches from time consumption point of view in terms of elementary operations.

6.4. Vertex normal computation

The goal of this section is to show a new vertex normal computation approach based on the t-variant cross product. The basic idea is that we use a weighting scheme approach for the vertex normal computation so that the individual face normals are computed using the t-variant cross product. We consider the area weighting scheme since the normal vectors computed by the t-variant cross product are implicitly weighted by the area. First we will show the computation for the general case, later we will show a simplification for the case when a vertex is surrounded by a fan of triangles.

6.4.1. A general case

First let us formalize the input setting. A number of n triangles share the same vertex[23] denoted $\mathbf{P}_0$. We want to compute the vertex normal at the vertex $\mathbf{P}_0$. Every vertex $\mathbf{P}_k$,

[23] Note that the triangles need not to form a closed triangle fan.

k=0, …, n+1, where n is the number of faces adjacent to the vertex $\mathbf{P}_k$, has a linear trajectory:

$$\mathbf{P}_k(t) = \mathbf{P}_k + t\mathbf{d}_k \,, \tag{6.12}$$

where

$$\mathbf{d}_k = \mathbf{P'}_k - \mathbf{P}_k \,, \tag{6.13}$$

where $\mathbf{P'}_k$ is the final position of $\mathbf{P}_k$ and $\mathbf{d}_k$ is a trajectory vector. Figure 6.4 shows how a mesh is deformed and which variables are involved in the process.

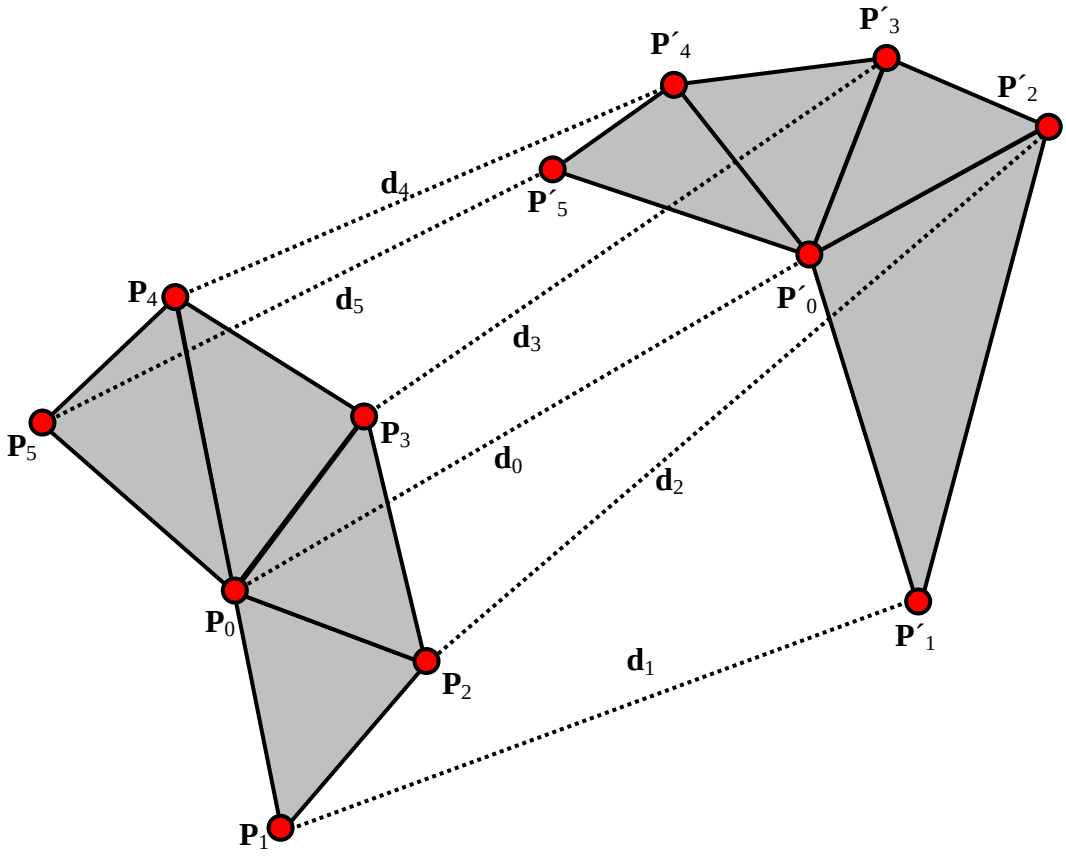


Figure 6.4: A part of a triangle mesh under linear deformation. Each vertex travels from its initial position $\mathbf{P}_k$ towards its final position $\mathbf{P'}_k$ along a linear trajectory.

Every edge $\mathbf{e}_j$, j=1, …, n+1 can be described as a vector

$$\mathbf{e}_j(t) = \mathbf{P}_j(t) - \mathbf{P}_0(t) \,. \tag{6.14}$$

And the face normal $\mathbf{n}_i$, i=1, …, n becomes

$$\mathbf{n}_i(t) = \mathbf{e}_i(t) \times \mathbf{e}_{i+1}(t) \,. \tag{6.15}$$

This implies that the edges must have a sorted order in each fan for computing the face normal. By substituting Eq. 6.14 into Eq. 6.15 we obtain:

$$\mathbf{n}_i(t) = \mathbf{P}_i(t) \times \mathbf{P}_{i+1}(t) - \mathbf{P}_i(t) \times \mathbf{P}_0(t) - \mathbf{P}_0(t) \times \mathbf{P}_{i+1}(t) \tag{6.16}$$

Expanding each cross product from Eq. 6.16 gives:

$$\begin{aligned}
\mathbf{P}_i(t) \times \mathbf{P}_{i+1}(t) &= \mathbf{P}_i \times \mathbf{P}_{i+1} + t(\mathbf{d}_i \times \mathbf{P}_{i+1} + \mathbf{P}_i \times \mathbf{d}_{i+1}) + t^2(\mathbf{d}_i \times \mathbf{d}_{i+1}) \\
\mathbf{P}_i(t) \times \mathbf{P}_0(t) &= \mathbf{P}_i \times \mathbf{P}_0 + t(\mathbf{d}_i \times \mathbf{P}_0 + \mathbf{P}_i \times \mathbf{d}_0) + t^2(\mathbf{d}_i \times \mathbf{d}_0) \\
\mathbf{P}_0(t) \times \mathbf{P}_{i+1}(t) &= \mathbf{P}_0 \times \mathbf{P}_{i+1} + t(\mathbf{d}_0 \times \mathbf{P}_{i+1} + \mathbf{P}_0 \times \mathbf{d}_{i+1}) + t^2(\mathbf{d}_0 \times \mathbf{d}_{i+1})
\end{aligned} \tag{6.17}$$

Now, the vertex normal can be computed as:

$$\mathbf{n}(t) = \sum_{i=1}^{n} \mathbf{n}_i(t) \tag{6.18}$$

We can now put Eq. 6.17 into Eq. 6.18 and get:

$$\mathbf{n}(t) = \sum_{i=1}^{n} \begin{pmatrix} \mathbf{P}_i \times \mathbf{P}_{i+1} - \mathbf{P}_i \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_{i+1} \\ + t(\mathbf{d}_i \times \mathbf{P}_{i+1} + \mathbf{P}_i \times \mathbf{d}_{i+1} - (\mathbf{d}_i \times \mathbf{P}_0 + \mathbf{P}_i \times \mathbf{d}_0) - (\mathbf{d}_0 \times \mathbf{P}_{i+1} + \mathbf{P}_0 \times \mathbf{d}_{i+1})) \\ + t^2(\mathbf{d}_i \times \mathbf{d}_{i+1} - \mathbf{d}_i \times \mathbf{d}_0 - \mathbf{d}_0 \times \mathbf{d}_{i+1}) \end{pmatrix} \tag{6.19}$$

Eq. 6.19 is general enough so it can be used for computing a single face normal as well. In this case we just set n=1, i.e., the sum disappears and Eq. 6.19 turns to the standard t-variant cross product.

The constant and quadratic terms in Eq. 6.19 behave in quite a similar way. We shall see that they can be simplified. The sum of constant terms is

$$\begin{aligned}
&\mathbf{P}_1 \times \mathbf{P}_2 - \mathbf{P}_1 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_2 \\
&+ \mathbf{P}_2 \times \mathbf{P}_3 - \mathbf{P}_2 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_3 \\
&+ \mathbf{P}_3 \times \mathbf{P}_4 - \mathbf{P}_3 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_4 \\
&+ \cdots \\
&+ \mathbf{P}_n \times \mathbf{P}_{n+1} - \mathbf{P}_n \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_{n+1}
\end{aligned} \tag{6.20}$$

Examining this sum further reveals that the second term in the second row cancels out the last term on the first row. This behavior is the same in each row. Hence we can rewrite the sum in Eq. 6.19 as:

$$\sum_{i=0}^{n+1} \mathbf{P}_i \times \mathbf{P}_{(i+1) \bmod (n+2)} \tag{6.21}$$

Thus the quadratic term becomes:

$$\sum_{i=0}^{n+1} \mathbf{d}_i \times \mathbf{d}_{(i+1) \bmod (n+2)} \tag{6.22}$$

The linear term is:

$$\begin{aligned}
&\mathbf{d}_1 \times \mathbf{P}_2 + \mathbf{P}_1 \times \mathbf{d}_2 - (\mathbf{d}_1 \times \mathbf{P}_0 + \mathbf{P}_1 \times \mathbf{d}_0) - (\mathbf{d}_0 \times \mathbf{P}_2 + \mathbf{P}_0 \times \mathbf{d}_2) \\
&+ \mathbf{d}_2 \times \mathbf{P}_3 + \mathbf{P}_2 \times \mathbf{d}_3 - (\mathbf{d}_2 \times \mathbf{P}_0 + \mathbf{P}_2 \times \mathbf{d}_0) - (\mathbf{d}_0 \times \mathbf{P}_3 + \mathbf{P}_0 \times \mathbf{d}_3) \\
&+ \mathbf{d}_3 \times \mathbf{P}_4 + \mathbf{P}_3 \times \mathbf{d}_4 - (\mathbf{d}_3 \times \mathbf{P}_0 + \mathbf{P}_3 \times \mathbf{d}_0) - (\mathbf{d}_0 \times \mathbf{P}_4 + \mathbf{P}_0 \times \mathbf{d}_4) \\
&+ \cdots \\
&+ \mathbf{d}_n \times \mathbf{P}_{n+1} + \mathbf{P}_n \times \mathbf{d}_{n+1} - (\mathbf{d}_n \times \mathbf{P}_0 + \mathbf{P}_n \times \mathbf{d}_0) - (\mathbf{d}_0 \times \mathbf{P}_{n+1} + \mathbf{P}_0 \times \mathbf{d}_{n+1})
\end{aligned} \tag{6.23}$$

Once again the second term in the second row cancels out the last term in the first row. This behavior is repeated for each row.

$$\begin{aligned}
&\mathbf{d}_1 \times \mathbf{P}_2 + \mathbf{P}_1 \times \mathbf{d}_2 - (\mathbf{d}_1 \times \mathbf{P}_0 + \mathbf{P}_1 \times \mathbf{d}_0) \\
&+ \mathbf{d}_2 \times \mathbf{P}_3 + \mathbf{P}_2 \times \mathbf{d}_3 \\
&+ \mathbf{d}_3 \times \mathbf{P}_4 + \mathbf{P}_3 \times \mathbf{d}_4 \\
&+ \cdots \\
&+ \mathbf{d}_n \times \mathbf{P}_{n+1} + \mathbf{P}_n \times \mathbf{d}_{n+1} - (\mathbf{d}_0 \times \mathbf{P}_{n+1} + \mathbf{P}_0 \times \mathbf{d}_{n+1})
\end{aligned} \tag{6.24}$$

Rearranging the terms gives

$$\begin{aligned}
&\mathbf{d}_0 \times (\mathbf{P}_1 - \mathbf{P}_{n+1}) + \mathbf{d}_1 \times (\mathbf{P}_2 - \mathbf{P}_0) \\
&+ \mathbf{d}_2 \times (\mathbf{P}_3 - \mathbf{P}_1) \quad + \mathbf{d}_3 \times (\mathbf{P}_4 - \mathbf{P}_2) \\
&\cdots \\
&+ \mathbf{d}_{n+1} \times (\mathbf{P}_0 - \mathbf{P}_n)
\end{aligned} \tag{6.25}$$

This can be written as:

$$\sum_{i=0}^{n+1} \mathbf{d}_i \times (\mathbf{P}_{(i+1) \bmod (n+2)} - \mathbf{P}_{(i-1) \bmod (n+2)}) \tag{6.26}$$

The normal can finally be written as:

$$\mathbf{n}(t) = \sum_{i=0}^{n+1} \mathbf{P}_i \times \mathbf{P}_{(i+1) \bmod (n+2)} + t \sum_{i=0}^{n+1} \mathbf{d}_i \times (\mathbf{P}_{(i+1) \bmod (n+2)} - \mathbf{P}_{(i-1) \bmod (n+2)}) + t^2 \sum_{i=0}^{n+1} \mathbf{d}_i \times \mathbf{d}_{(i+1) \bmod (n+2)} \tag{6.27}$$

It should be noted that these sums can be pre-computed once before the actual mesh deformation, more specifically, each sum represent a coefficient (i.e., absolute, linear and quadratic) of a degree-two polynomial. Eq. 6.27 is in vector form, which means that for each coordinate component (x, y and z) we have a separate degree-two polynomial. Then, as the mesh deforms, only the degree-two polynomials for each of x, y and z coordinate components are evaluated to obtain the vertex normal in the particular time instant. It must be said that the resulting vertex normal is not of unit length; therefore it must be additionally normalized if necessary (e.g., when it is computed for shading purposes).

6.4.2. Simplification of the Circular Case

The presented formulas can be used for any number of adjacent triangles. This implies that one or several adjacent faces can be omitted from the computation. This can be useful if there is a sharp edge in the fan of faces. However, if all faces are included in the computation, then the last edge is the same as the first edge, i.e. the faces form a closed fan. The computation can then be reduced. The constant term is

$$\begin{aligned} &\mathbf{P}_1 \times \mathbf{P}_2 - \mathbf{P}_1 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_2 \\ &+ \mathbf{P}_2 \times \mathbf{P}_3 - \mathbf{P}_2 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_3 \\ &+ \mathbf{P}_3 \times \mathbf{P}_4 - \mathbf{P}_3 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_4 \\ &+ \mathbf{P}_4 \times \mathbf{P}_5 - \mathbf{P}_4 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_5 \\ &+ \cdots \\ &+ \mathbf{P}_{n-1} \times \mathbf{P}_n - \mathbf{P}_{n-1} \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_n \\ &+ \mathbf{P}_n \times \mathbf{P}_1 - \mathbf{P}_n \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_1 \end{aligned} \tag{6.28}$$

Once again there are terms that cancel out terms in other rows. After simplification we have:

$$\begin{aligned} &\mathbf{P}_1 \times \mathbf{P}_2 + \mathbf{P}_2 \times \mathbf{P}_3 \\ &+ \mathbf{P}_3 \times \mathbf{P}_4 + \mathbf{P}_4 \times \mathbf{P}_5 \\ &+ \cdots \\ &+ \mathbf{P}_{n-1} \times \mathbf{P}_n + \mathbf{P}_n \times \mathbf{P}_1 \end{aligned} \tag{6.29}$$

This can be expressed as a sum:

$$\sum_{i=1}^{n} \mathbf{P}_i \times \mathbf{P}_k, k = \begin{cases} 1, i = n \\ i+1, i \neq n \end{cases} \tag{6.30}$$

In a similar way we can compute a sum for the quadratic term.

The linear term for the simplified case can be derived from Eq. 6.24, where $\mathbf{P}_{n+1}=\mathbf{P}_1$ and $\mathbf{d}_{n+1}=\mathbf{d}_1$ (because of the circular nature), which gives:

$$\begin{aligned}&\mathbf{d}_1\times(\mathbf{P}_2-\mathbf{P}_n)+\mathbf{d}_2\times(\mathbf{P}_3-\mathbf{P}_1)\\&\mathbf{d}_3\times(\mathbf{P}_4-\mathbf{P}_2)+\mathbf{d}_4\times(\mathbf{P}_5-\mathbf{P}_3)\\&\dots\\&\mathbf{d}_n\times(\mathbf{P}_1-\mathbf{P}_{n-1})\end{aligned} \tag{6.31}$$

This can be expressed as a sum:

$$\sum_{i=1}^{n}\mathbf{d}_i\times(\mathbf{P}_k-\mathbf{P}_l),\ k=\begin{cases}1, i=n\\ i+1, i\neq n\end{cases}\quad l=\begin{cases}n, i=1\\ i-1, i\neq 1\end{cases} \tag{6.32}$$

6.4.3. Examples

In this section we will show that the usual linear interpolation of the normal vectors is not correct. Then we will show that our normal computation scheme (t-variant vertex normals) is better than the linear interpolation and it has similar results to the recomputation approach. We will also show that our normal computation scheme is faster than the recomputation approach.

In the first example we will show that the linear interpolation of normal vectors is not good for some animations. Note that in the morphing animations the source and the target mesh could be highly dissimilar and the transformation of individual faces could be quite dramatic. If the shape transformation is dramatic we need a lot of in-between frames to represent the shape transformation, so it is not possible to "fake" the normal field by some approximation or even by leaving normals unchanged during the transformation as suggested in [Ale00a]. Figure 6.5a) shows a deforming mesh sequence with linearly interpolated normals. Figure 6.5b) and c) shows three frames of this sequence plus a detail of the problematic region (the region where the fish's fin disappears in the octopus body). Figure 6.5b) shows the result of the linear interpolation of normals. In Figure 6.5b) right, vertex normals are depicted. It can be seen that linearly interpolated vertex normals do not reflect the true shape of the mesh (i.e., they are not perpendicular to the mesh) and therefore they lead to an incorrect shading. Figure 6.5c) shows the normals computed using recomputation approach[24]. On the detail view (Figure 6.5c) right) it can be seen that these normals reflect the true shape of the mesh and therefore the shading is correct.

[24] We used the area weighting scheme.

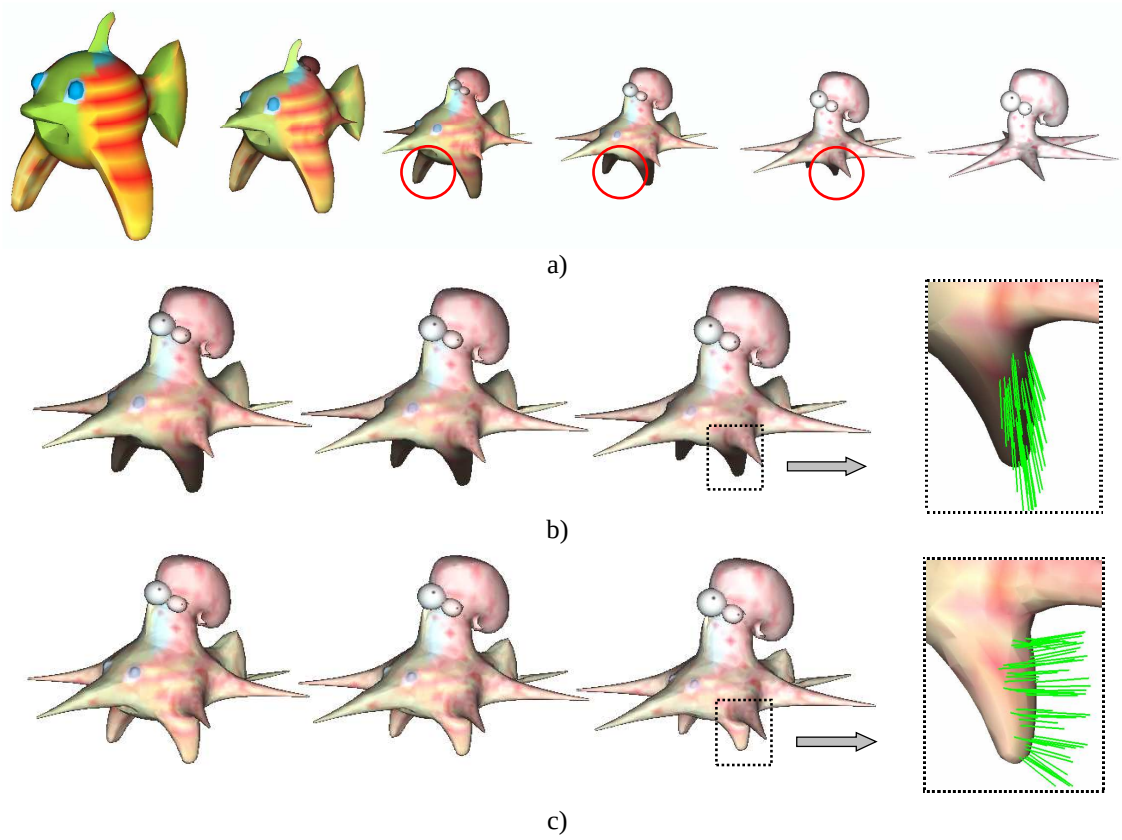


Figure 6.5: a) Deforming mesh from the shape of fish to the shape of octopus (circles mark the incorrectly shaded region). b) Three frames of the shape transformation with linearly interpolated normals plus detail of the problematic region with depicted vertex normals. c) Three frames of the shape transformation with recomputed normals plus detail of the same region as in b).

In the next example we will show that our normal computation scheme is better than the linear interpolation and that it has the same results as the recomputation approach. Figure 6.6a) is an example of linear normal interpolation. The problems can be seen in the third image, where the arising antennas are badly shaded. The detail of this area is depicted in Figure 6.6b), where the top row shows the linear normal interpolation, the middle row shows the result of our t-variant vertex normal approach and the bottom row shows deforming mesh rendered with the recomputed normals. It can be seen that the mesh rendered with t-variant vertex normals is almost the same as the mesh rendered with recomputed normals. In the last column (t=0.40) of Figure 6.6b there are also the vertex normals depicted. It can be seen that linearly interpolated normals do not reflect the mesh shape and our time vertex normals as well as recomputed normals do reflect the mesh shape and thus result in better shading.

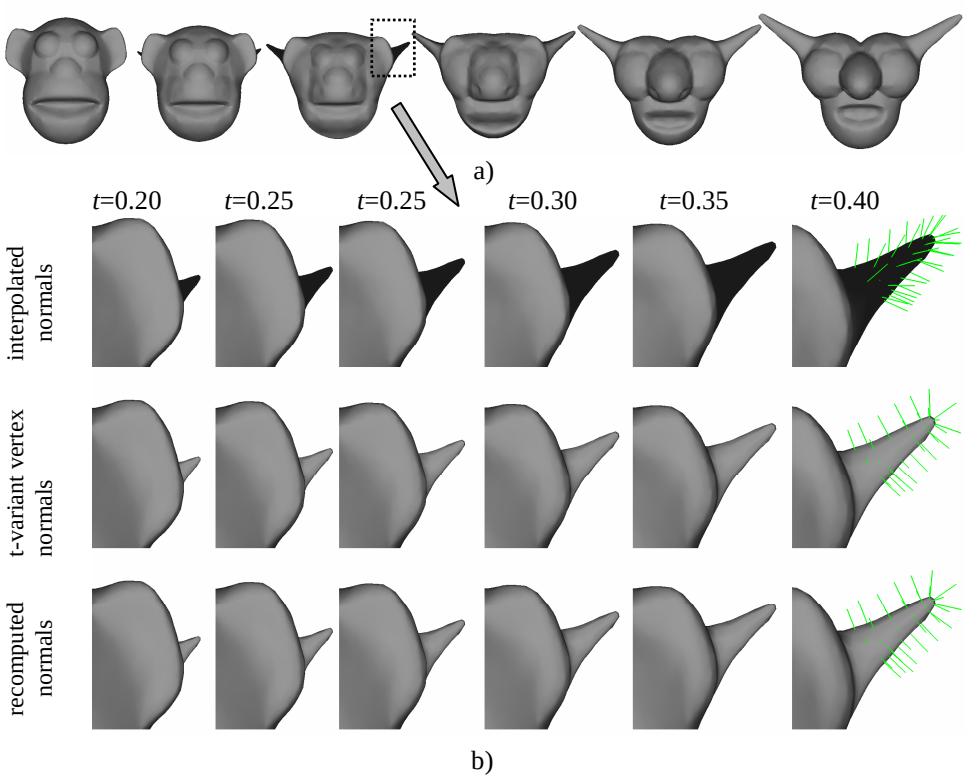


Figure 6.6: a) An example of the morphing transition between two faces using linear normal interpolation. b) The detail of the problematic region – linear normal interpolation (top row sequence), our t-variant vertex normal approach (middle row), recomputed normals (bottom row). Last column (t=0.40) shows also vertex normals in the problematic region.

If we compare the preprocessing time required for usual linear normal interpolation and the proposed approach, then we find that for linear normal interpolation we must first compute interpolation constraints, i.e., all vertex normals for the starting mesh and all the vertex normals for the final mesh. It means that we have to run the normal computation twice and normalize the initial and the final normal. The proposed approach requires that the coefficients for the degree-two polynomials are precomputed, i.e. to compute 3 sums (Eq. 6.27). It should be noted that both approaches require renormalization of the interpolated normals.

To evaluate the t-variant vertex normals we have to store the coefficients of the polynomials computed in the preprocessing stage (Eq. 6.27). 9 floating point numbers per normal must be stored. Usual linear interpolation requires storing 6 floating point numbers, i.e. interpolation constraints. The recomputation approach requires a data structure which contains the incident faces to a particular vertex which is usually represented by a list of integer indices to faces, i.e., the number of indices required is at least $\sum d_i$, i=1, ..., n, where n is the number of vertices and d_i is the degree of the i-th vertex. Additionally, the closed form functional description of the normal behavior allows us to analyze normal behavior by means of function analysis, e.g., to approximate degree-two polynomial by degree-one polynomial and reduce the amount of data which is stored in the preprocessing stage. For example, the degree-two polynomial can be approximated by the first two terms of Taylor expansion, which yields a degree-one polynomial. Using Taylor expansion we also know the exact error caused by the approximation.

The recomputation approach is dependent on the actual connectivity of the mesh, i.e., the time needed for computation of one vertex normal depends on how many face normals contribute. It requires a fast data structure which contains, for each vertex, a set of incident faces. Of course the number of faces incident to a particular vertex is different for different vertices. During evaluation of Eq. 6.1 the data structure must be traversed to obtain the incident faces. Our t-variant approach requires traversing the data structure containing incident faces just in the preprocessing stage (computation of polynomial coefficients). During the vertex normal evaluation a constant time is required to evaluate three degree-two polynomials. Thus the proposed approach is not dependent on the actual mesh connectivity and underlying data structures.

We measured computational time of our t-variant vertex normal (TVVT) computation and the recomputation approach (RA). We used four animations with different number of vertices. For each animation we generated 200, 500 and 1000 in-between frames. We measured only the pure vertex normal computation, i.e., times needed for a data structure update and the rendering were excluded from the time measurement. For the measurement we used P IV, 3 GHz, 1 GB RAM running on Windows 2003 Server. The measurement is summarized in Table 6.4. First three rows contain times for 200, 500, 1000 in-between frames, "per frame" row shows times for vertex normals evaluation per one frame of the animation and the "per vertex" row shows times needed to evaluate one vertex normal. It is clear that the computational time increases linearly with the number of vertices and in-between frames. It can be seen that average time needed to compute one vertex normal is approximately the same for both approaches. Our technique is approximately four times faster than the usual recomputation approach which was verified also on different animations and different hardware configurations.

	9902 vertices, 19800 faces		**39802 vertices, 79600 faces**		**89702 vertices, 179400 faces**		**159602 vertices, 319200 faces**	
	RA	**TVVN**	**RA**	**TVVN**	**RA**	**TVVN**	**RA**	**TVVN**
200 fr. [ms]	2653.00	668.80	10481.00	2678.00	23806.20	6031.20	45997.00	10581.20
500 fr. [ms]	6628.40	1662.20	25815.60	6609.60	59593.60	14868.60	116472.20	26684.40
1000 fr. [ms]	13400.20	3334.20	58484.40	15366.00	119186.80	29706.20	229999.80	52881.00
per frame [ms]	13.31	3.33	54.17	13.99	119.14	29.87	230.98	53.05
per vertex [µs]	1.34	0.34	1.36	0.35	1.33	0.33	1.45	0.33
speed-up		**3.99**		**3.88**		**3.99**		**4.35**

Table 6.4: Time comparison of the t-variant vertex normal approach (TVVN) and the recomputation approach (RA).

6.5. Quaternion correction

In this section we will propose a new vector interpolation scheme which can be used for a normal vectors interpolation of meshes originating from the topology merging process (Section 4). For better understanding we will first describe the basic idea for an

interpolation of scalar quantities, and then we will generalize it for vector quantities. Then, two applications of the proposed approach will be showed.

6.5.1. Basic idea

Let us have two pairs of scalar values – an initial pair ($v_{\mathbf{e}}^0, v_{\mathrm{f}}^0$) and a final pair ($v_{\mathbf{e}}^1, v_{\mathrm{f}}^1$). The pair contains a correct value $v_{\mathbf{e}}^0$ and a "fake" value v_{f}^0. The idea is that the correct value is computed by some elaborated method which has exact results but it is, for instance, computationally expensive. On the other hand, the fake value is computed by some simple method which does not produce exact results but it is, for instance, very fast. The goal is to use the simple method (which produces the fake values) to compute the intermediate values between $v_{\mathbf{e}}^0$ and $v_{\mathbf{e}}^1$ and correct these values towards the correct values. First, we compute the initial error value e^0 between the initial fake value and the initial correct value, i.e., $e^0 = v_{\mathbf{e}}^0 - v_{\mathrm{f}}^0$, similarly the final error value e^1 is computed as $e^1 = v_{\mathbf{e}}^1 - v_{\mathrm{f}}^1$. The core idea is to compute the intermediate value using the simple method and compensate the error by adding a linearly interpolated error value, i.e.:

$$\mathrm{v_c}(t) = \mathrm{f}(t) + (e^0 + t(e^1 - e^0)),\ t \in \langle 0; 1 \rangle, \tag{6.33}$$

where f(t) is the simple method for computing the fake values, e^0 the initial error value, e^1 is the final error value and $\mathrm{v_c}(t)$ is the corrected value. Note that f(0) = v_{f}^0 and f(1) = v_{f}^1. Also note that the corrected values $\mathrm{v_c}(0) = v_{\mathbf{e}}^0$ and v(1) = $v_{\mathbf{e}}^1$. Of course, the intermediate values are not exact, but in a relatively simple way, the values around t=0 and t=1 are close to the correct values. Simplicity of this approach together with a small computation overhead is a motivation for generalization of this method for the interpolation of vector quantities.

The generalization of the proposed approach for the vector quantities is as follows. The initial error is expressed as a correction quaternion $\mathbf{q}^0 = (\mathbf{a}^0, \alpha^0)$, where α^0 is computed as the angle between the initial correct vector and the initial fake vector, $\mathbf{a}^0$ is computed as $\mathbf{a}^0 = \mathbf{v}_{\mathrm{f}}^0 \times \mathbf{v}_{\mathrm{e}}^0$. Note that $\mathbf{a}^0$ is essentially an axis of rotation when we want to rotate the vector $\mathbf{v}_{\mathrm{f}}^0$ so that it coincides with $\mathbf{v}_{\mathrm{e}}^0$, α^0 is then an angle of rotation (Figure 6.7 left). Similarly, the final error is expressed as a correction quaternion $\mathbf{q}^1 = (\mathbf{a}^1, \alpha^1)$, where α^1 is the angle between the final correct vector and the final fake vector, $\mathbf{a}^1$ is computed as $\mathbf{a}^1 = \mathbf{v}_{\mathrm{f}}^1 \times \mathbf{v}_{\mathrm{e}}^1$ (Figure 6.7 right). The intermediate correction quaternion $\mathbf{q}(t)$ is computed using the QSLERP (Eq. B.4). The correction quaternion $\mathbf{q}(t)$ is used to correct the fake vector values computed by some simple method by rotating the fake values using the correction quaternion (Eq. B.3), i.e.:

$$\mathbf{v}_{\mathrm{c}}(t) = \mathbf{q}(t) \cdot \mathrm{f}(t) \cdot \mathbf{q}^*(t), \tag{6.34}$$

where f(t) represents some simple method for vector computation which produces the fake vectors, $\mathbf{q}(t)$ is the correction quaternion, $\mathbf{q}^*(t)$ is the conjugate correction quaternion and $\mathbf{v}_{\mathrm{c}}(t)$ is the vector corrected using the quaternion correction approach.

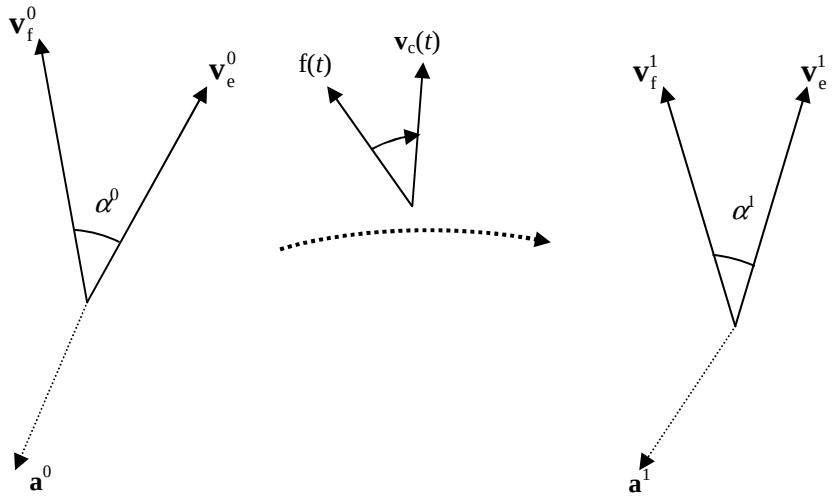


Figure 6.7: A demonstration of the quaternion correction idea.

6.5.2. Applications

In the first application we will use the quaternion correction to interpolate normal vectors of deforming meshes. The deforming meshes are constructed using the topology merging technique. The problem of the topology merging technique is that it produces a mesh with an irregular connectivity and unequally sized triangles. If we use such connectivity to compute vertex normals using a weighting scheme and if we use such normals for smooth shading, we obtain disturbing shading artifacts. Fortunately, using the method described in Section 4.3 we can compute vertex normals for the source shape and the target shape which produce plausible shading without any shading artifacts. It is demonstrated in Figure 6.8, Figure 6.8a) and b) shows the input meshes of the topology merging technique; Figure 6.8c) shows the supermesh in the shape of the flower, Figure 6.8d) shows the connectivity of the supermesh. Figure 6.8e) left shows the mesh rendered with vertex normals computed by a weighting scheme while Figure 6.8e) right shows the mesh rendered with vertex normals computed using the face mapping approach (Section 4.3), Figure 6.8e) center shows the detailed views.

The problem is that the face mapping approach computes normals only for the supermesh in the shape of the source mesh and the target mesh. The intermediate normals must be somehow interpolated. As shown before (Section 6.4), the linear normal interpolation is not completely correct for some kind of motions, because it does not respect the true mesh shape. On the other hand, the recomputation of the vertex normals using the weighting scheme produces bad normals because of bad connectivity. Recall that the result of the normal vectors recomputation is equivalent to t-variant vertex normal computation described in Section 6.4. Therefore, we suggest using the quaternion correction to diminish the problems caused by the bad connectivity.

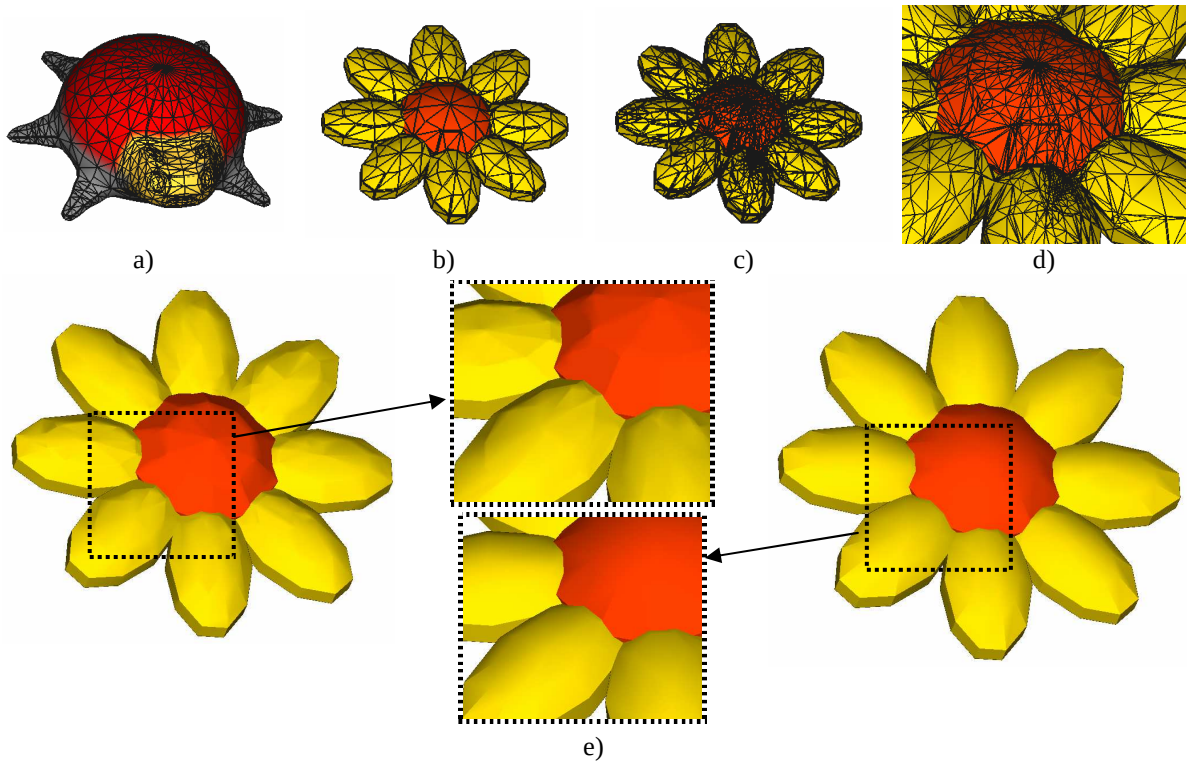


Figure 6.8: a), b) – the input meshes of the topology merging technique, c) the supermesh in the shape of the target mesh, d) a detail view of the connectivity after the topology merging process, e) a topology merging mesh with normals computed using a weighting scheme (left) and the face mapping approach (right).

The quaternion correction approach is applied as follows. The correct normals are computed using the face mapping approach. The fake normals are computed using the t-variant vertex normals. Clearly, the t-variant vertex normals can be computed during the whole deformation, but the face mapping approach can be used only for the initial and the final mesh. Thus, the t-variant vertex normal computation acts as a simple method and the results are corrected using the correct normals computed by the face mapping approach.

The results are shown in Figure 6.9. Figure 6.9a) shows the mesh rendered with linearly interpolated vertex normals. The artifacts caused by the linear interpolation are visible in the second and the third frame of the animation. Figure 6.9b) shows the mesh rendered with t-variant vertex normals, the shading artifacts caused by bad connectivity are visible especially in the first frame of the animation. Figure 6.9c) shows the mesh rendered with t-variant vertex normals corrected using the quaternion correction, it can be seen that the first frame does not have the shading artifacts caused by the bad connectivity and the intermediate frames do not have shading artifacts caused by the linear normal interpolation.

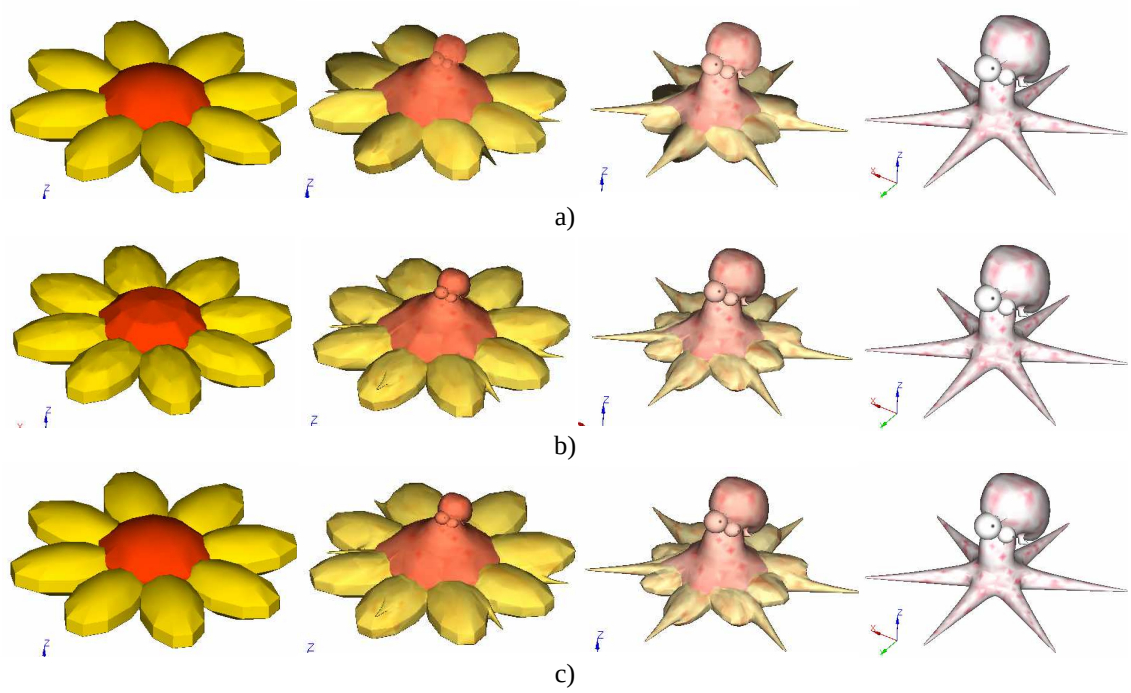


Figure 6.9: a) linear interpolation, b) t-variant vertex normals, c) t-variant vertex normals corrected using the quaternion correction.

In the second application we use the quaternion correction to mimic different weighting schemes in the vertex normal computation. The setting is as follows, we can compute vertex normals of deforming meshes using the t-variant vertex normal approach. The t-variant vertex normals use the area weighting scheme. However, sometimes we might want to use another weighting scheme. The problem is that we do not have a simple expression (as Eq. 6.27) for the other weighting schemes. Therefore, we propose to compute vertex normals using the t-variant vertex normals approach. These normals act as fake normals. Then, using the desired weighting scheme, we compute the "more accurate" vertex normals for the mesh in the initial shape and for the mesh in the final shape. Using the difference between the "fake" and "more accurate" normals we can construct the correction quaternions as in the previous example. The intermediate normals computed by the t-variant vertex normals approach are corrected towards the desired weighting scheme by an interpolated correction quaternion.

To compare two methods for vertex normal computation we use two measures: Accumulated Angle Error (AAE) and Angular Discrepancy Histogram (ADH) [Jin05]. Two normal vectors $\mathbf{n}_c$, $\mathbf{n}_f$ are compared by computing the angle between them. If we have two normal fields, AAE is computed as sum of absolute values of angles between corresponding normals[25]. If AAE is zero than the two normal fields are identical. ADH is computed by evaluating angles between pairs of corresponding normals and organizing the angle values into a histogram. From the histogram we can read a percentage of normal vectors which have the discrepancy in a certain interval (given by the width of the histogram bin).

To test the quaternion correction, we generated 20 frames of a morphing animation. In each animation frame we computed the vertex normals using the quaternion correction

[25] Thus, the unit of AAE is the radian (if an angle is computed in the circular measure) or the degree (if an angle is computed in degrees).

approach and an “exact” normals using some elaborated weighting scheme. In each frame we computed AAE and ADH to compare the normals computed by the quaternion correction and by the direct evaluation of the weighting scheme. AAE for 3 different weighting schemes (angle weighting scheme, inverse area weighting scheme and equal weighting) can be seen in Figure 6.10. It can be seen that for the initial and the final frame the AAE is zero since the normal vectors computed by the t-variant vertex normal are corrected towards the desired weighting scheme. It can be also seen that the angle weighting scheme was mimicked better using the quaternion correction that the inverse area weighting scheme or equal weighting.

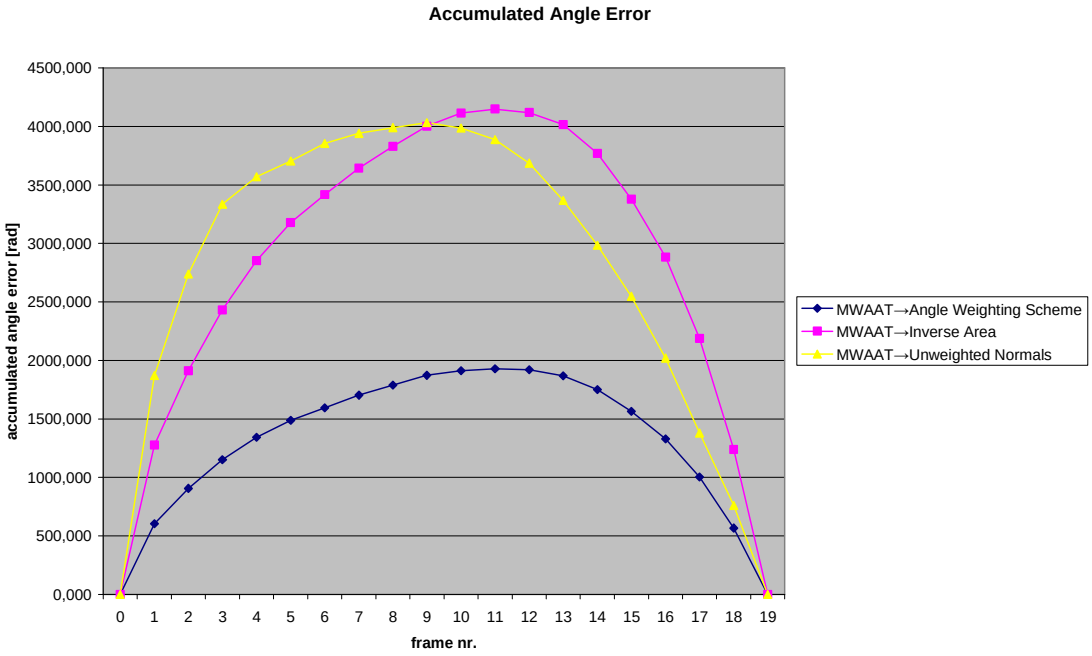


Figure 6.10: Accumulated angle error (AAE) of an animation consisting of 20 frames. The blue curve shows AAE when mimicking angle weighting scheme using area weighting scheme and QC, the yellow curve shows AAE when mimicking equally weighted normals and the magenta curve shows AAE when mimicking inverse area weighting scheme.

The biggest accumulated error occurs in the frames 12, 13, 14. It is given by the nature of the correction approach which corrects the values at the beginning and at the end of the interval the most while in the middle in the interval the correction influence is the smallest. To examine the distribution of error in the most problematic frames we will show ADH (Figure 6.11) when trying to mimic the angle weighting scheme using the quaternion correction. It can be seen that approximately 60% to 70% of vertex normals falls into the first histogram bin which corresponds to the angular discrepancy 0°-3°.

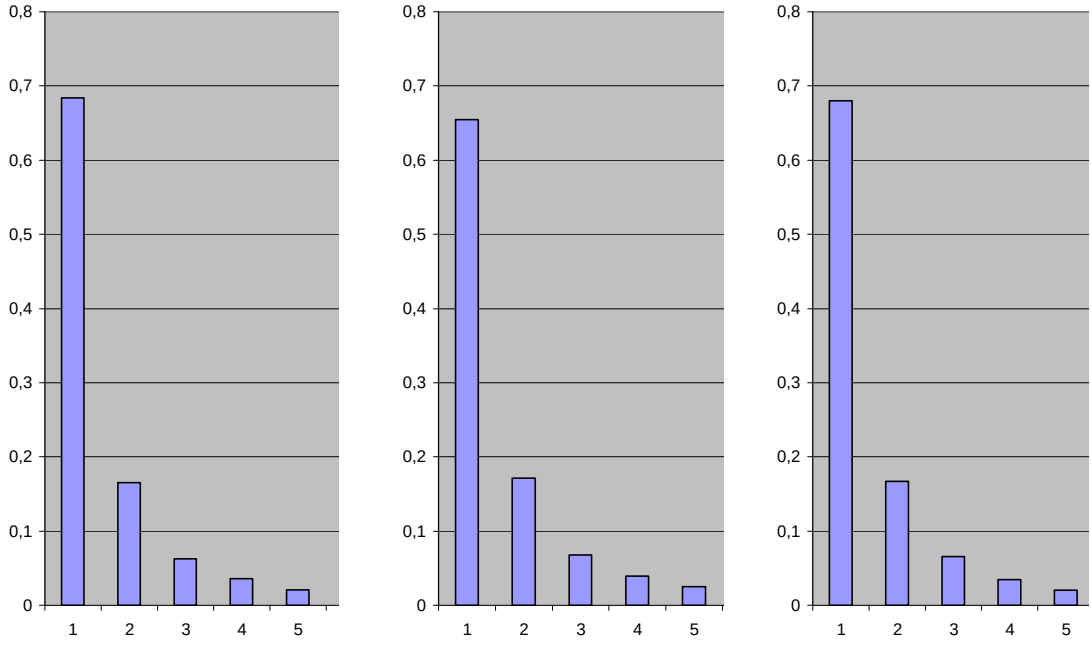


Figure 6.11: ADH of vertex normals when trying to mimic the angle weighting scheme using the quaternion correction. The horizontal axes represent histogram bins; the vertical axes represent a fraction of normal vectors in a histogram bin.

6.6. Summary

In this chapter we described some methods for computation of normal vectors of deforming meshes. We described a mathematical apparatus called t-variant cross product which can be used for fast computation of normal vectors of planes under linear motion. Subsequently, we used the t-variant cross product for fast vertex normal computation.

We also outlined two ideas which need some further research – quaternion SLERP and quaternion correction. Quaternion SLERP requires determining a best pair of frames configuration. In the case of the quaternion correction we showed some preliminary results, however, as a potentially general technique for error compensation, it requires some more validation.

7. Continuous collision detection

Collision detection is an important part of every simulation system ranging from computer games to surgical simulators. In the simulation, rigid or deformable objects can be involved. In this chapter we will show elementary predicates for the collision detection of deforming meshes which are produced by morphing. We focused on continuous collision detection because it considers the deformation as a continuous process in contrast to standard approaches which consider the deformation as a discrete sequence of shapes.

The elementary predicates include point/plane intersection test and line/line intersection test. These tests are essential when computing intersections of more complicated entities. In the area of continuous collision detection, many elaborated techniques based on space subdivision and hierarchies of bounding volumes exist. Even though these techniques significantly improve a performance of a collision detection system, still a big amount of time is spent by computing intersections between individual primitives. Therefore an analysis and robust implementation of the elementary predicates is important. The elementary predicates have been already described by some authors [Pro97, Hut07], but we present here a full derivation which allows us to point out some special cases which may cause numerical problems when not handled properly.

7.1. Introduction

The goal of the collision detection system is to discover collisions or interpenetrations of objects involved in a simulation and generate an appropriate response.

In collision detection we are usually interested in finding the time of the collision, the contact point and the contact normal [Ber03]. The contact point is a place where two objects first touch, the contact normal is the normal of the plane which contains the contact point and which separates colliding objects near the contact point (in some ε-neighborhood). The contact normal is important in the context of collision response.

Collision detection approaches can be divided into two categories – discrete and continuous. In discrete methods the time is considered to be discrete in contrast to the real world where the time is assumed to be continuous, thus collision or interpenetrations are investigated in discrete time samples only, which may result in missing or late detection of the collision (especially if the objects move rapidly or if they are very small). It can be solved by increasing the sampling rate of the time but it

brings higher computational demands. Another possibility is to use a backtracking method. Here, if a penetration occurs, the time interval is recursively subdivided in a binary search way until the first time of collision (FTC) is found. The computational cost of backtracking can be very high for complex objects and it can fail if objects are not connected [Red05]. On the other hand, continuous methods consider not only individual time samples but also an in-between motion, i.e., movement of objects between the individual time samples.

Another possible division of the collision detection area is whether it deals with rigid bodies or with soft-bodies (deformable objects). Rigid bodies do not deform, formally, the relative position of each two points of the body stays fixed during the motion and the body transforms as one entity [Kar04]. Hence, it is just a translation and/or a rotation type of motion. On the other hand, there are no restrictions on the change of the relative position of two points in the case of soft-body motion; each point can travel along its trajectory independently of the other points. Soft-bodies appear in computer graphics, e.g., in surgery simulation, morphing, cloth simulation, etc. It is clear that handling collision detection for soft-bodies is something more difficult than for the rigid-body case.

Collision detection algorithms can also be distinguished according to the representation of the objects involved in the simulation system. It can either be a point based representation, a volume representation or a boundary representation. Each representation requires different data structures and different algorithms. To speed up the computation of collision detection, two paradigms are usually used – hierarchy of bounding volumes (e.g., AABBs, OOBBs, k-DOPs, bounding spheres, convex hulls, etc.) and spatial subdivision (3d grid, octree, BSP, range trees, etc). Bounding volumes approximate the original objects by some simple volume with which it is easier to compute intersections. If there is no intersection, it is not necessary to further investigate intersections of the possibly more complex object. Bounding volumes are usually organized into hierarchies. Spatial subdivision can quickly answer which two objects may intersect. Usually, a combination of both techniques is used. No matter which bounding volume and spatial subdivision is used, sometimes (e.g., when bounding volumes intersect) it is necessary to investigate the exact intersection. Then it is important how the 3d object is represented. It may be a point based representation, volume representation or boundary representation. The exact intersection is computed by enumerating all elements involved in the collision (points, voxels or polygons depending on the actual representation) and tests them for the mutual intersection.

An elementary collision detection test between triangular meshes is a computation of collision between two triangles. Mathematically, the intersection computation between triangles $\mathbf{T}_0$ and $\mathbf{T}_1$ involves computation of intersection line $\mathbf{l}$ of two planes ρ_0 (given by $\mathbf{T}_0$) and ρ_1 (given by $\mathbf{T}_1$) and further test whether $\mathbf{l}$ is common to $\mathbf{T}_0$ and $\mathbf{T}_1$. From an implementation point of view, a so called “early reject” test is important [Mol97]. The early reject test can decide very quickly whether two triangles can intersect without the need for a whole triangle/triangle intersection test. For example, if all points of $\mathbf{T}_0$ lie in the same half-space given by ρ_1, the triangles $\mathbf{T}_0$, $\mathbf{T}_1$ cannot intersect.

This chapter deals with continuous collision between a moving point and a moving plane (Figure 7.1a) and between two moving lines (Figure 7.1b). These elementary tests are used for continuous collision between two moving triangles. We assume that the

behavior of objects between samples is linear (i.e., the linear motion, Section 6.2). The linear motion is simple enough so that the continuous collision detection can be computed by solving a cubic equation (so called *collision equation*) which can be solved algebraically.

a) b)

Figure 7.1: a) a collision between a triangle under linear motion and a point moving along linear trajectory, b) a collision between two edges, where each vertex of the edge moves independently along a linear trajectory.

7.2. Related work

A survey of discrete and continuous collision detection algorithms for deformable objects is given in [Tes04].

Redon et al. [Red00] approximated the in-between motion by so-called "arbitrary in-between motion". The basic idea is as follows. Given an initial and a final position and orientation of an object in time samples t_n and t_{n+1}. The in-between motion is given by screw motion [Kim03] – a combination of translation and rotation which transforms an object from the initial position at t_n to the final position at t_{n+1}. If the trajectory is known then it is possible to compute algebraically the collision time by solving a polynomial equation. This approach was used for rigid-bodies and articulated structures [Red04].

Cameron [Cam90] proposed to compute continuous collision detection by computing 4d swept volumes of moving objects. It turns the dynamic 3d problem into a static 4d problem. This approach requires the computation of 4d intersections. The main problem is that 4d swept volumes may be very complicated. Therefore, this approach is suitable only for a certain class of objects and motions.

Larsson and Akeine-Möller [Lar03] proposed a discrete collision detection algorithm for meshes deformed by morphing, where the vertex positions are convex combinations of some reference models. The morphing model used for vertices is used for bounding volumes hierarchy (k-DOPs and spheres) as well. The hierarchy of bounding volumes is used to avoid expensive low level primitive vs. primitive tests.

In [Her90, Sny93] a continuous collision detection approach for time-dependent parametric and implicit surfaces is presented.

Provot [Pro97] showed similar relations to ours in the context of collision and self-collision in cloth model. Other similar relations to ours were showed in [Hut07]. They showed that continuous collision detection between moving points, planes and lines can be generally formulated as a test for coplanarity of four points. Both [Pro97] and [Hut07] do not describe special cases which may in many cases simplify the solution of collision equation and which may cause numerical problems.

7.3. Moving point/plane test

Let us recall the t-variant cross product which describes a behavior of a face normal under a linear motion (see also Section 6.2):

$$\begin{aligned}\mathbf{n}(t) = {} & (\mathbf{P}_1 \times \mathbf{P}_2 - \mathbf{P}_1 \times \mathbf{P}_0 - \mathbf{P}_0 \times \mathbf{P}_2) + \\ & t(\mathbf{d}_1 \times \mathbf{P}_2 + \mathbf{P}_1 \times \mathbf{d}_2 - (\mathbf{d}_1 \times \mathbf{P}_0 + \mathbf{P}_1 \times \mathbf{d}_0) - (\mathbf{d}_0 \times \mathbf{P}_2 + \mathbf{P}_0 \times \mathbf{d}_2)) + \\ & t^2(\mathbf{d}_1 \times \mathbf{d}_2 - \mathbf{d}_1 \times \mathbf{d}_0 - \mathbf{d}_0 \times \mathbf{d}_2)\,.\end{aligned} \tag{7.1}$$

Recall that the degree of the polynomial in Eq. 7.1 depends on the type of motion. If two trajectory vectors are the same, then the polynomial in Eq. 7.1 is linear. If all trajectory vectors are the same, i.e., when the plane just translates, then the polynomial in Eq. 7.1 is constant. It is clear that if two points of the face do not move, then the polynomial in Eq. 7.1 is linear, if all three points do not move, then the polynomial in Eq. 7.1 is constant. Also note that the coefficients of Eq. 7.1 can be precomputed in a preprocessing stage and stored for each investigated time interval or they can be computed on the fly using GPU as it needs only a number of vector operations.

Next, let us recall a general equation of the plane, which is:

$$\mathbf{n}\mathbf{x} + d = 0, \tag{7.2}$$

where **n** is a normal vector of the plane. If the normal has a unit length then d represents a distance of the plane from the origin. A plane is defined by three points. If the plane moves in time then it can be described by the movement of three points. Let us remind that each of the three points could travel independently along a linear trajectory. Then, the t-variant general equation of the plane is:

$$\mathbf{n}(t)\mathbf{x} + \mathrm{d}(t) = 0\,. \tag{7.3}$$

From Eq. 7.1 we have $\mathbf{n}(t)$. It remains to compute $\mathrm{d}(t)$. For a static plane it is computed by substituting an arbitrary point $\mathbf{v}=(v_x, v_y, v_z)$ of the plane into Eq. 7.2, i.e.:

$$d = -av_x - bv_y - cv_z\,. \tag{7.4}$$

Analogously, to compute $\mathrm{d}(t)$ we can substitute a moving point $\mathbf{v}(t)=v_0+t(v_1 - v_0)$ into Eq. 7.3. Note that the point $\mathbf{v}(t)$ must be a point of the moving plane. Now, Eq. 7.3 can be rewritten as:

$$\mathbf{n}(t)\mathbf{x} - \mathbf{n}(t)\mathbf{v}(t) = 0\,, \tag{7.5}$$

where $\mathbf{x} = (x, y, z)$ is a point on the plane. If we want to test whether a moving point $\mathbf{q}(t) = \mathbf{q}_0 + t(\mathbf{q}_1 - \mathbf{q}_0)$ hits the moving plane, then we substitute the query point trajectory $\mathbf{q}(t)$ into (7.5 and solve the equation, i.e.,

$$\mathbf{n}(t)\mathbf{q}(t) - \mathbf{n}(t)\mathbf{v}(t) = 0\,. \tag{7.6}$$

By solving Eq. 7.6 the time t_{col} of the collision between a moving plane and a moving point is computed. Of course, the contact normal is given by evaluating $\mathbf{n}(t_{\mathrm{col}})$, where t_{col} is the collision time.

Eq. 7.6 is generally cubic, since the normal behavior is quadratic and it is further multiplied by a linear term (query point trajectory $\mathbf{q}(t)$ and an arbitrary point of the plane trajectory). A cubic equation has three roots, so the transforming plane can be intersected up to three times by a point moving along a linear trajectory. A hybrid approach for finding roots of cubic equations is described in Appendix C.

Next let us discuss some special cases which may in some situations lead to a lower degree of Eq. 7.6. We already discussed the degree of $\mathbf{n}(t)$ which depends on the trajectories of the points of the moving plane. If $\mathbf{n}(t)$ is linear then Eq. 7.6 is quadratic, if $\mathbf{n}(t)$ is constant then Eq. 7.6 is linear. Note that the case when $\mathbf{n}(t)$ is constant represents a translating plane, which is often the case when objects just translate and do not deform, then Eq. 7.6 becomes:

$$\mathbf{nq}(t) - \mathbf{nv}(t) = 0, \tag{7.7}$$

which is a simple linear equation. Thus, it can be seen that Eq. 7.6 does not hold only for deforming planes, but it can be used more generally, too.

If the point or plane is static, the degree of Eq. 7.6 may be also reduced. It is described in Table 7.1.

Point	**Plane**	**Equation**
moving	moving	$\mathbf{n}(t)\mathbf{q}(t)$-$\mathbf{n}(t)\mathbf{v}(t) = 0$ (Eq. 7.6)
moving	static	$\mathbf{nq}(t)$-$\mathbf{nv} = 0$
static	moving	$\mathbf{n}(t)\mathbf{q}$-$\mathbf{n}(t)\mathbf{v}(t) = 0$
static	static	$\mathbf{nq} - \mathbf{nv} = 0$

Table 7.1: Forms of Eq. 7.6 for special kinds of linear motion.

Note that the case of a moving point and a static plane (line 2 in Table 7.1) leads to the usual ray-triangle intersection. The case when both point and plane do not move (line 4 in Table 7.1) leads to the usual point-in-plane test.

Another special case which reduces the degree of the polynomial by one is the case when the trajectory vector of the query point is the same as a trajectory vector of some point of the plane.

Eq. 7.6 holds for a point $\mathbf{q}(t)$ lying in the plane. If the point $\mathbf{q}(t)$ does not lie in the plane then Eq. 7.6 can be used for computation of a t-variant signed distance between the moving point and the moving plane, i.e.,

$$\mathrm{dist}(t) = \mathbf{n}(t)\mathbf{q}(t) - \mathbf{n}(t)\mathbf{v}(t). \tag{7.8}$$

Note that $\mathrm{dist}(t)$ does not represent the Euclidean distance, since $\mathbf{n}(t)$ does not have a unit length, but in collision detection we are interested mainly in cases when $\mathrm{dist}(t) = 0$, then it is not necessary to have $\mathbf{n}(t)$ normalized.

Let us also remind that Eq. 7.8 is just a degree three polynomial. Real roots of Eq. 7.6 represent collision times, extremes of Eq. 7.8 represent extreme distances of a point $\mathbf{q}(t)$ to the plane. Analysis of extremes of Eq. 7.8 can be used as an *early reject test* which

can quickly eliminate cases, when there is no collision at all, from further computation. Details of the early reject test are described in Appendix C.

Next we will describe continuous collision test between two moving lines which can be further extended to test between two moving edges.

7.4. Moving lines test

The key idea for the edge/edge collision detection algorithm is that we can check if vertex $\mathbf{b}(t)$ lies in a plane spanned by two vectors $\mathbf{v}_1(t)$ and $\mathbf{v}_2(t)$ as shown in Figure 7.2. In other words, lines $\mathbf{P}_0(t)$ $\mathbf{P}_1(t)$ and $\mathbf{P}_2(t)$ $\mathbf{P}_3(t)$ intersect iff $\mathbf{P}_1(t)$ lies in the plane formed by $\mathbf{P}_0(t)$, $\mathbf{P}_2(t)$ and $\mathbf{P}_3(t)$.

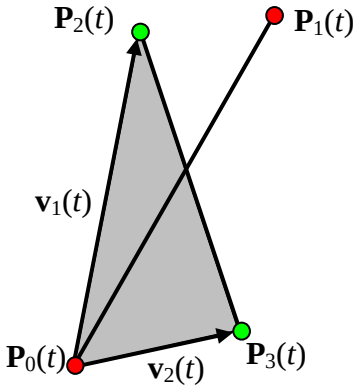


Figure 7.2: A collision between two moving edges $\mathbf{P}_0(t)$ $\mathbf{P}_1(t)$ and $\mathbf{P}_2(t)$ $\mathbf{P}_3(t)$. Edges collide if $\mathbf{P}_1(t)$ lies in the plane spanned by vectors $\mathbf{v}_1(t)$ and $\mathbf{v}_2(t)$.

As in the previous section we can construct the general t-variant equation of the plane by taking $\mathbf{n}(t) = \mathbf{v}_1(t) \times \mathbf{v}_2(t)$ and $d(t) = -\mathbf{n}(t).\mathbf{v}(t)$, where $\mathbf{v}(t)$ is again one vertex of the moving plane, e.g., $\mathbf{v}(t) = \mathbf{a}(t)$. So, we have to solve:

$$\mathbf{n}(t)\mathbf{P}_1(t) - \mathbf{n}(t)\mathbf{P}_0(t) = 0\,. \qquad (7.9)$$

Eq. 7.9 is again a cubic equation. By solving it we compute the collision time between two moving lines. The contact normal is given by evaluation $\mathbf{n}(t_{col})$, where t_{col} is the collision time.
We can expect similar special cases as in the previous point/plane test. First, if one of the lines translates, then the degree of Eq. 7.9 is reduced by one. If both lines translate, then the degree of Eq. 7.9 is reduced by two. In some cases one of the lines (or both of them) may be static, then Eq. 7.9 will be as follows:

Line 1	Line 2	Equation
moving	moving	$\mathbf{n}(t)\mathbf{P}_1(t)$-$\mathbf{n}(t)\mathbf{P}_0(t) = 0$ (Eq. 7.9)
moving	static	$\mathbf{n}(t)\mathbf{P}_1$-$\mathbf{n}(t)\mathbf{P}_0 = 0$
static	static	$\mathbf{nP}_1 - \mathbf{nP}_0 = 0$

Table 7.2: Forms of Eq. 7.9 for special kinds of linear motion.

7.5. Moving triangle/triangle test

Two triangles **A** and **B** may collide in two different ways – a vertex of **A** with the triangle **B** and vice-versa (Figure 7.3a) and edge of **A** and edge of **B** (Figure 7.3b).

a) b)

Figure 7.3: Two possible types of triangle vs. triangle collision – a) vertex of one triangle collides with the other triangle, b) two triangles collide by edges.

Thus, the low level tests are: point/triangle and an edge/edge. The test described in Section 7.3 can be used for continuous collision detection between a moving triangle $\mathbf{T}(t)$ and a moving point $\mathbf{p}(t)$. The triangle $\mathbf{T}(t)$ defines a plane $\rho(t)$. By solving Eq. 7.6 we compute a collision time t_{col} between the plane $\rho(t)$ and the point $\mathbf{p}(t)$. Then, it is necessary to check whether $\mathbf{p}(t_{col})$ lies inside the triangle $\mathbf{T}(t_{col})$.

Analogously, the test described in Section 7.4 can be used for continuous collision detection between two moving line segments $\mathbf{l}_A(t)$ and $\mathbf{l}_B(t)$. By solving Eq. 7.9 we compute a collision time t_{col} between infinite lines $\mathbf{l}_A(t)$ and $\mathbf{l}_B(t)$. Then we have to check whether lines segments $\mathbf{l}_A(t_{col})$ and $\mathbf{l}_B(t_{col})$ really intersect because the line collision does not imply the line segment collision.

The continuous triangle vs. triangle intersection test can be composed of six "point vs. triangle tests" (each vertex of one triangle against the other triangle and vice versa) and nine "edge vs. edge tests" (each edge of one triangle against the edges of the other triangle and vice versa). Note that we have to perform all tests, since we need the earliest collision time.

As in the static triangle intersections tests [Mol97] we can use the early reject test. The distance function dist(t) (Eq. 7.8) can be analyzed and if it monotonous and does not have a root on an investigated interval then the query vertex cannot collide with the plane (Appendix C). If all three vertices of one triangle cannot collide, it is not necessary to perform any other tests since the triangles cannot collide at all.

The continuous collision detection between the triangle $\mathbf{T}_0(t)$ (formed by vertices $\mathbf{v}_0(t)$, $\mathbf{v}_1(t)$, $\mathbf{v}_2(t)$ and edges $\mathbf{e}_{0,0}(t)$, $\mathbf{e}_{0,1}(t)$, $\mathbf{e}_{0,2}(t)$) and $\mathbf{T}_1(t)$ (formed by vertices $\mathbf{u}_0(t)$, $\mathbf{u}_1(t)$, $\mathbf{u}_2(t)$ and edges $\mathbf{e}_{1,0}(t)$, $\mathbf{e}_{1,1}(t)$, $\mathbf{e}_{1,2}(t)$) is done in the following steps:

1. Perform early reject test for vertices $\mathbf{v}_0(t)$, $\mathbf{v}_1(t)$, $\mathbf{v}_2(t)$ with respect to the plane $\boldsymbol{\rho}_0(t)$ (given by $\mathbf{T}_0(t)$). Exit computation if early reject test rejects all vertices.
2. Repeat step 1 with vertices $\mathbf{u}_0(t)$, $\mathbf{u}_1(t)$, $\mathbf{u}_2(t)$ and the plane $\boldsymbol{\rho}_1(t)$.
3. Compute the collision between moving points $\mathbf{v}_0(t)$, $\mathbf{v}_1(t)$, $\mathbf{v}_2(t)$ and moving plane $\boldsymbol{\rho}_0(t)$ using the test described in Section 7.3. If there is a collision between a vertex $\mathbf{v}(t)$ and a plane $\boldsymbol{\rho}(t)$ in time t_{col}, check if the $\mathbf{v}(t_{col})$ lies inside the triangle $\mathbf{T}(t_{col})$. If $\mathbf{v}(t_{col})$ lies inside $\mathbf{T}(t_{col})$ add t_{col} in a collision list $\mathbf{L}$.
4. Repeat the step 3 with vertices $\mathbf{u}_0(t)$, $\mathbf{u}_1(t)$, $\mathbf{u}_2(t)$ and the plane $\boldsymbol{\rho}_1(t)$.
5. Compute the collision between each pair of moving lines ($\mathbf{e}_{0,i}(t)$, $\mathbf{e}_{1,j}(t)$), i=0, 1, 2; j=0, 1, 2 using the test described in Section 7.4. If there is a collision between a line $\mathbf{e}_{0,i}(t)$ and $\mathbf{e}_{1,j}(t)$ in the time t_{col}, check if the edge $\mathbf{e}_{0,i}(t_{col})$ intersects $\mathbf{e}_{1,j}(t_{col})$. If so, add t_{col} in $\mathbf{L}$.
6. Sort $\mathbf{L}$ and return the earliest collision time t_{first}.

7.6. Experiments

We tested our equations on various configurations of moving elements (i.e., points, lines and planes). First, we computed a collision time t_{col} by solving a collision equation. Using the collision time we computed position of moving elements at the time t_{col} and we checked if they really intersect using a static intersection test. The static intersection test was in our case a point-in-plane test and a line/line intersection test. By this test we verified that t_{col} really is the time of collision.

By solving the collision equation we may end up with three collisions, however, we are usually interested in the first time of collision (FTC), i.e., the time when two elements first touch. To verify that we really found the FTC by solving the collision equation we used a sampling approach. This approach samples the movement of elements and in each sample the static intersection test is computed. Of course the sampling must be dense enough to avoid missing or late detection of the collision.

In fact the sampling approach is used in the classical (i.e., non-continuous) collision detection framework. The density of sampling is usually a system variable which influences accuracy and also time consumption. If the sampling density is high, the collision is detected exactly but it takes more time. If the sampling is lower, the computation is faster but elements may interpenetrate or the collision may be missed. One advantage of the Continuous Collision Detection (CCD) approach is that the computation (i.e., a solution of the collision equation) takes almost constant time whereas the time consumption of the sampling approach depends on the density of the sampling. It is clear that if the collision between two elements occurs at the beginning of the time interval (early collision) then fewer samples are processed, on the other hand, if the collision occurs at the end of the time interval (late collision) then almost all samples of the time interval must be processed. The extreme case is when two elements do not collide at all, then the sampling approach has to sample whole time interval. This case is usually avoided by spatial subdivision or hierarchy of bounding volumes, but still in some cases it may appear.

To show the difference between the CCD approach and the sampling approach and to further verify our equations we computed collisions between many configurations of two moving triangles. The CCD test for moving triangles involves both a point/plane and a line/line test. The static intersection test for the sampling approach involves a triangle/triangle intersection test. For the static triangle/triangle intersection test we used

a reference implementation of an algorithm by Akeinne-Moller [Mol97] available at [URL1]. The time measurement was made on an AMD Athlon XP-M, 2800+, 512 MB RAM; we also performed tests on different configurations to verify that the general trend is always the same.

In the following comparison we focused on four general cases which may appear when computing the collision between two moving triangles – early collision (triangle collide in the beginning of the time interval), late collision (triangles collide in the end of the time interval), fast movement (one triangle moves very fast) and a case when two triangles do not collide at all (no collision). To show the behavior we picked for each case one particular instance and we measured the time complexity and accuracy. In Table 7.3 there are results computed by the CCD approach, it contains the first time of collision (FTC column) and the *t* time needed to compute FTC (i.e., the time needed to solve the collision equation). It can be seen that the time needed to compute FTC is approximately always the same. Let us suppose that FTC values computed by CCD approach are exact, i.e., exact within the representation of real numbers in case of an algebraical solution or within the accuracy of a numerical solution of the collision equation. For solution of the collision equation we used a hybrid approach described in Appendix C.

Type of collision	**FTC [s]**	***t* [ms]**
Early collision	0,135952	0,032250
Late collision	0,816131	0,035200
Fast movement	0,474108	0,036650
No collision	-	0,034550

Table 7.3: Results of CCD approach.

In Table 7.4 we show an accuracy and a time consumption of the sampling approach. The accuracy Δt is computed as $\Delta t = |FTC_{CCD} - FTC_S|$, where FTC_{CCD} is the exact FTC computed by CCD approach and FTC_S is FTC computed by the sampling approach. It is clear that both accuracy and time consumption depends on the density of the sampling which is also shown in the "samples" column. The density of the sampling is application dependent and it is usually some system variable of the collision detection framework. It can be seen how Δt fall with the growing number of samples, but of course the time consumption is also higher. Note that in the case of early collision only 13% of samples were processed, in the case of late collision 81% of samples were processed and in the case of no collision 100% of samples were processed. For example if we want the FTC with $\Delta t < 0.0001$ then the early collision case required to process approximately 5000 samples which took 0,260 ms while the late collision case required approximately 5000 samples too but it took 1,232 ms. In the case of fast movement it can be seen that 10 samples were not enough to capture the collision. In the extreme case of no collision it can be seen that the sampling 10000 which were in previous cases sufficient to achieve $\Delta t < 0.0001$ took 3.605 ms which is approximately 100 times more than using CCD approach.

samples	early collision		late collision		fast movement		no collision	
	Δt	t [ms]	Δt	t [ms]	Δt	t [ms]	Δt	t [ms]
10	0,086270	0,010	-	0,010	-	0,010	-	0,010
100	0,005462	0,010	0,002050	0,020	0,000639	0,010	-	0,110
500	0,000321	0,030	0,001504	0,131	0,000842	0,071	-	0,260
1000	0,000184	0,051	0,000685	0,300	0,000366	0,130	-	0,371
2000	0,000116	0,110	0,000277	0,501	0,000129	0,240	-	0,721
5000	7,54E-05	0,260	3,19E-05	1,232	0,000187	0,611	-	1,803
10000	6,18E-05	0,521	5,03E-05	2,503	3,92E-05	1,222	-	3,605
20000	5,03E-06	1,041	9,46E-06	4,997	1,55E-05	2,443	-	7,220
100000	9,59E-06	5,158	6,81E-06	25,126	6,56E-06	12,268	-	35,882

Table 7.4: Time comparison of the continuous collision detection and the resampling approach.

7.7. Summary

For collision detection of rigid objects auxiliary data structures to speed up the collision detection computation are usually built. A generalization for deformable objects is not straightforward since the data structures must be updated more frequently and usually complicated algorithms are required. Also, during the object deformation, very complex shapes may arise, e.g., non-convex or even disconnected, which may be a problem for some collision detection algorithms.

Therefore we described elementary tests which can be used for continuous collision detection between linearly deforming meshes. Our tests are not specialized only for linear deformation. It can be used for continuous collision detection between rigid objects as well; then the degree on the collision equation is lower. Our point/plane test can be also used for continuous collision detection of point based representation.

Note that in collision detection we are usually interested in FTC which mathematically involves a computation of an intersection between generally nonparallel and nonintersecting lines (edge/edge collision) or point-in-triangle test (vertex/triangle collision). These computations require comparison of two floating point numbers. Because of a limited precision of the floating points numbers represented in the computer, the comparison is usually implemented using some ε-neighborhood, where ε must be estimated empirically. Using our CCD approach, we compute the collision by solving a cubic collision equation. The collision equation can be solved algebraically. The algebraic solution requires comparison of two real numbers too (e.g., discriminant computation). Therefore, we suggest to analyze the input data (i.e., the trajectory vectors) of the collision equation according to the detailed study of the special cases presented in this chapter. By analyzing the trajectory vectors we can immediately expect a lower degree of the collision equation. Also, we can make "almost the same" trajectory vectors "exactly the same" which avoid very small values of coefficient of the collision equation. Of course, we might discover that some trajectory vectors were "almost the same" by analyzing the coefficients of the collision equation. For example, if the cubic coefficient is very small (e.g., 1.10^{-8}) we may guess that some trajectory vectors were almost the same. However, the question is, should we use an apparatus for solving the cubic equation or should we disregard the value of the cubic coefficient and solve the collision equation as a quadratic equation? What error did we cause by

disregarding a very small cubic coefficient? If we correct the trajectories **before** the computation of the collision equation, we know exactly how much the trajectories differ and how big error do we introduce by making them explicitly the same.

Let us recall that the core of the collision equation is the t-variant cross product (Eq. 7.1)[26]. Therefore, the coefficients of the t-variant cross product can be used when rendering a mesh as well as when evaluating the collision detection.

[26] Properties and computational aspects of the t-variant cross product are described in Section 6.2 in the context of normal computation of deformable meshes.

8. Core-increment morphing

In this chapter we will describe a novel morphing technique for 2d polygons; it is based on the decomposition paradigm[27] and it is motivated by the process of growing in nature. It is suitable for situations when some parts are expected to grow while some parts are expected to disappear during the shape transformation process. Our solution does not require correspondence computation and/or user interaction; however, different effects can be achieved by a change of the mutual position of the objects. Our method works with 2d polygons; however the description of the method can be viewed also as an analysis of the general technique – i.e., a growing motivated shape transformation. During the method development we kept in mind a possible generalization in 3d or perhaps a use of a different shape representation. However, the generalizations belong to the future work.

This research was done in cooperation with Ing. Martina Málková who helped with an experimental verification of the proposed ideas. She now carries out an independent research which deals with 3d version of the core-increment morphing.

8.1. Introduction

The goal of morphing is to compute a shape transformation (represented by an animation) between two shapes. There are usually many different ways how to deform one shape to another. The goal is to choose such a transformation which is visually plausible. The visual plausibility is strongly application dependent. Sometimes we might want to explode the initial shape into particles and form the final shape from the particles; sometimes we might want a continuous transformation during which the volume is preserved, etc. Current algorithms are mainly designed to morph between similar shapes where some common features can be identified. However, in some cases it is not possible to find common features (e.g., a seed to palm tree morphing). Therefore, in this chapter we will describe an approach for morphing which has a "growing-like" nature, i.e., we focus on a shape transformation which mimics growing (or disappearing) of some parts, which is quite a common process in nature.

Let us have two input shapes – a source and a target shape – which partially overlap. An intersection of the input shapes is computed. The intersection is called a core and it is a fixed part which will not change during a shape transformation. When transforming between the source shape and the target shape, parts of the source shape will disappear

[27] See Section 3.2.

in the core (i.e., they die away) whereas parts of the target shape will grow out of the core. The process of disappearing can be viewed as a reversed process of growing, thus algorithmically we need to solve just one process.

This way, we decoupled a complicated task of morphing of two polygons into several less complicated tasks of morphing of polygon chains. In contrast to typical morphing approaches, our method does not require a computation of a correspondence and/or some user interaction. On the other hand, a user can adjust a mutual orientation of the input shapes or an algorithm for growing and disappearing effects can be explicitly specified for each part.

8.2. The proposed solution

8.2.1. General idea

A polygon **P** is an ordered set of vertices $\mathbf{V}_i$, i = 1,..., n. An edge $\mathbf{e}_i$ of polygon is a line segment with endpoints $\mathbf{V}_i$, $\mathbf{V}_{i+1}$. A simple polygon is a polygon whose consecutive edges $\mathbf{e}_i$, $\mathbf{e}_{i+1}$ intersect only in the endpoint $\mathbf{V}_{i+1}$. An unclosed sequence of edges is called a *polygon chain*.

Now let us describe the idea of the core increment approach for simple polygons. The core **C** is obtained by computing an intersection of input polygons **A** and **B**, i.e., $\mathbf{C} = \mathbf{A} \cap \mathbf{B}$. For simplicity, let us suppose that the core consists of one part. If the intersection of **A** and **B** contains multiple parts we choose one representative part as a core which will act as a fixed part which does not change during the shape transformation. Our algorithm is not designed for the case when the intersection of **A** and **B** does not exist, or it consists of one of the polygons **A** or **B** (one polygon is inside the other one), where some other algorithms should be used instead.

The parts $\mathbf{P} = \cup\mathbf{P}_i$ which will disappear in the core are computed as $\mathbf{P} = \mathbf{A}\text{-}\mathbf{B}$ and the parts $\mathbf{Q} = \cup\mathbf{Q}_j$ which will grow out of the core are computed as $\mathbf{Q} = \mathbf{B}\text{-}\mathbf{A}$. Also suppose that $\mathbf{P} \neq 0$ or $\mathbf{Q} \neq 0$, i.e., one polygon is not entirely inside the other polygon. Algorithmically, the process of disappearing is just reversed process of growing so for now we will concentrate only on the disappearing of a part $\mathbf{P}_i$ in the core **C**.

A part $\mathbf{P}_i$ consists of two polygon chains $\mathbf{C}_0$, $\mathbf{C}_1$, where $\mathbf{C}_0$ is a common polygon chain of the core **C** and the part $\mathbf{P}_i$, $\mathbf{C}_1$ is the other part of the polygon $\mathbf{P}_i$ which remains after removing $\mathbf{C}_0$ from $\mathbf{P}_i$. The polygon chains $\mathbf{C}_0$, $\mathbf{C}_1$ are separated by intersection vertices $\mathbf{v}_{I0}$, $\mathbf{v}_{I1}$ (Figure 8.1). An intersection vertex lies in the intersection of the input polygons **A** and **B**. If $\mathbf{P}_i \neq \mathbf{A}$ and $\mathbf{P}_i \neq \mathbf{B}$ then $\mathbf{P}_i$ has two intersection vertices. By morphing the polygon chain $\mathbf{C}_1$ to the polygon chain $\mathbf{C}_0$ we achieve the effect of disappearing of the part $\mathbf{P}_i$ in the core **C**. Hereby, we decompose the polygon morphing problem into several polygon chain morphing problems.

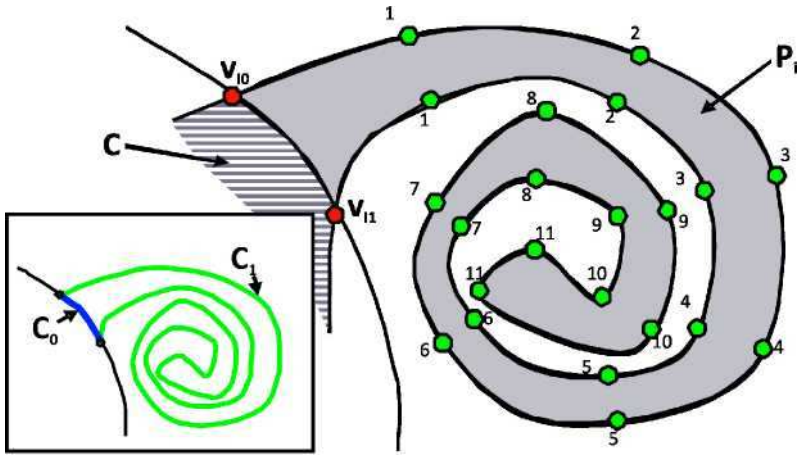


Figure 8.1: A highly nonconvex part $\mathbf{P}_i$ emerges from the core **C** (the hatched part). The vertices are labeled according to the topological distance from the intersection vertices $\mathbf{V}_{I0}$, $\mathbf{V}_{I1}$. The part $\mathbf{P}_i$ will grow out of the core **C** by morphing the polygon chain $\mathbf{C}_0$ to $\mathbf{C}_1$.

The morphing between a polygon chain $\mathbf{C}_0$ and $\mathbf{C}_1$ will be described using a vertex path for each vertex of the polygon chain $\mathbf{C}_1$ excluding the intersection vertices. The vertex path of a vertex $\mathbf{V}_i$ is a list of tuples $(t_j, \mathbf{P}_j)$, where t_j is a time and $\mathbf{P}_j$ is a position of the vertex $\mathbf{V}_i$ at the time t_j. A vertex path is usually defined on a canonical time interval <0; 1>. The vertex path has at least two elements, i.e., the initial position of the vertex at the time t=0.0 and a final position of the vertex at the time t=1.0. A movement of a vertex is obtained by computing intermediate positions of a vertex. The intermediate positions are computed by interpolating the position values $\mathbf{P}_k$ within the vertex path. We can use an arbitrary interpolation technique, e.g., a piecewise linear interpolation or cubic spline interpolation. Once a vertex path representing the growing process is computed, it can be interpreted in a reversed order to represent the disappearing process.

A vertex path is computed using a concept of a topological distance. The topological distance $d(\mathbf{V}_i,\mathbf{V}_j)$ between vertices $\mathbf{V}_i$ and $\mathbf{V}_j$ is the minimal number of edges on the polygon between $\mathbf{V}_i$ and $\mathbf{V}_j$. We compute the topological distance with respect to the intersection vertices. Because there are always two intersection vertices, we use the minimal topological distance $d_{min}(\mathbf{V}_i) = \min(d(\mathbf{V}_i, \mathbf{V}_{I0}), d(\mathbf{V}_i, \mathbf{V}_{I1}))$. Intuitively, the topological distance establishes an order in which the vertices will grow in order to form the whole part. The vertices with a smaller topological distance will finish earlier than vertices with larger topological distance. This avoids self-intersections during the growing process. To distinguish between topological distances of vertices lying on the polygon chains $\mathbf{C}_0$ and $\mathbf{C}_1$, we add the negative sign to the vertices lying on the polygon chain $\mathbf{C}_0$. Then d_{min}, d_{max} are minimal and maximal topological distances of the part $\mathbf{P}_i$.

In the following text we will describe three methods how to compute the vertex paths to be able to morph between polygon chains $\mathbf{C}_0$, $\mathbf{C}_1$ and simulate a process of disappearing of part $\mathbf{P}_i$ in this way. By simultaneous growing and disappearing of all parts we will achieve the effect of morphing between two simple polygons.

8.2.2. Perimeter growing

The first method to be described is called Perimeter growing, because all the vertices $\mathbf{V}_i$ lying on $\mathbf{C}_1$ travel along the perimeter of the part $\mathbf{P}_i$, i.e., their vertex path

contains only vertices of $\mathbf{P}_i$. The problem is to determine at which vertex $\mathbf{V}_j$ with the topological distance d_j is the specific vertex path supposed to end. Vertex path of a vertex $\mathbf{V}_i$ with the topological distance d_i contains vertices with topological distances (d_{i-1}, d_{i-2}, ..., d_0, d_{-1},..., d_j) (Figure 8.2). There are two rules concerning the last vertex of the vertex path, vertex $\mathbf{V_j}$. First, it must belong to the polygon chain $\mathbf{C}_0$. Second, we need at least one vertex path to end at each vertex that belongs to the polygon chain $\mathbf{C}_0$ (to form the shape of the other polygon). Therefore we use the following approach: the vertex path of the vertex with d_{max} always ends at the vertex with d_{min}. The vertex path of the vertex with d_{max-1} should end at the vertex with d_{min+1}. Generally, a vertex path of the vertex with d_{max-i} should end at the vertex with d_{min+i} (Figure 8.2).

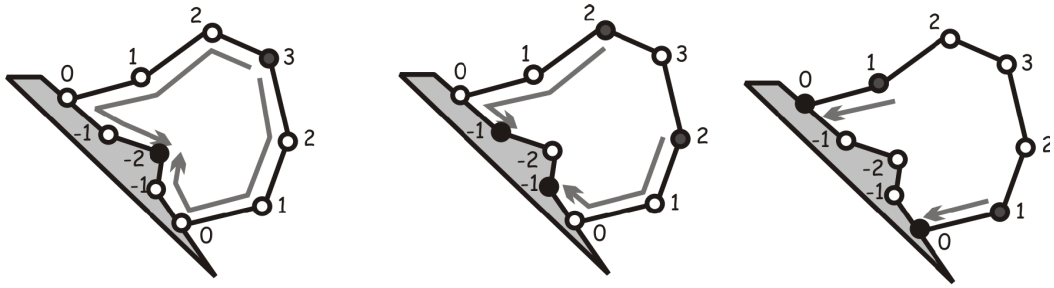


Figure 8.2: Vertex paths (d_{max} = 3, d_{min} = -2): a) vertex path for the vertex with d=d_{max} ends at the vertex with d = d_{min}, b) for the vertex with d = d_{max-1} it ends at the vertex with d = d_{min+1}, c) and so on.

However, such a vertex does not always lie on $\mathbf{C}_0$. If we denote n_0, n_1 the number of vertices of $\mathbf{C}_0$, $\mathbf{C}_1$ respectively, we can distinguish the following three cases:

- $n_0 = n_1$ (Figure 8.3a), each vertex of $\mathbf{C}_1$ ends its path at one vertex of $\mathbf{C}_0$,
- $n_0 > n_1$ (Figure 8.3b), there are some vertices of $\mathbf{C}_0$ that do not belong to any vertex path. It means that some of the vertices of $\mathbf{C}_1$ need to be duplicated, in such a case, we use such vertices of $\mathbf{C}_1$ that have the topological distance equal to one and duplicate them as many times as is necessary to cover all the vertices of $\mathbf{C}_0$ that are left,
- $n_0 < n_1$ (Figure 8.3c), some vertices of $\mathbf{C}_1$ cannot end their paths at the supposed vertex, in such a case their vertex paths end at the intersection vertices.

Because the vertex path computed by this method follows the perimeter of the part, the results always seem as if something was really growing from the core. Two things are ruining the nice effect. The former thing is the top of the growing part, which is always a straight line connecting the vertices with the same topological distance. This causes that the method is not suitable for parts, where some vertices with the same topological distance are wide apart (Figure 8.4a). On the other hand, it has really good results for the parts that are narrow and/or highly non-convex, like parts of a spiral or curly type or long and straight parts (Figure 8.4c,d). The latter thing is that following the shape of the part is not always what we wanted - for example if we have a part as in Figure 8.4b), the part will first grow from the core and then come back a little, and until that it will continue growing. But that coming back is something we do not expect.

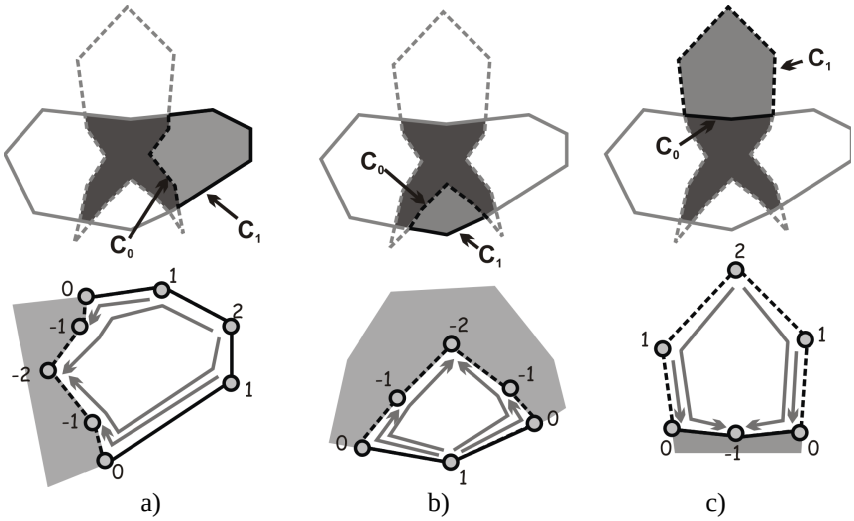


Figure 8.3: Three possible inputs for computing vertex paths: a) $n_0 = n_1$, b) $n_0 > n_1$, c) $n_0 < n_1$ (dark grey: core, grey: selected part, a full line: the first polygon, a dashed line: the second polygon, light grey: a part of the core, grey arrows: vertex paths).

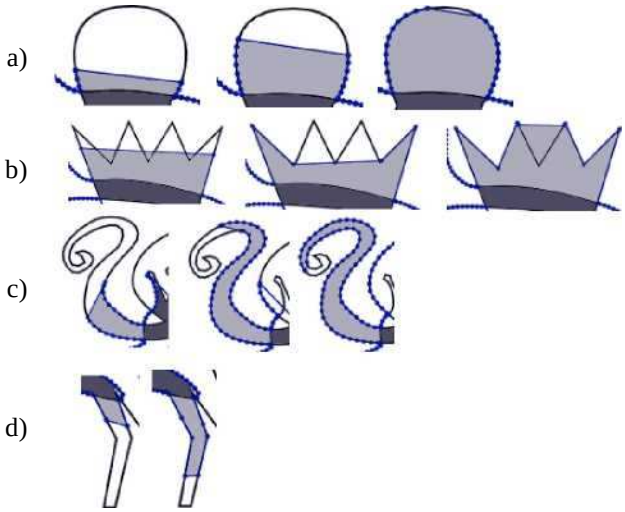
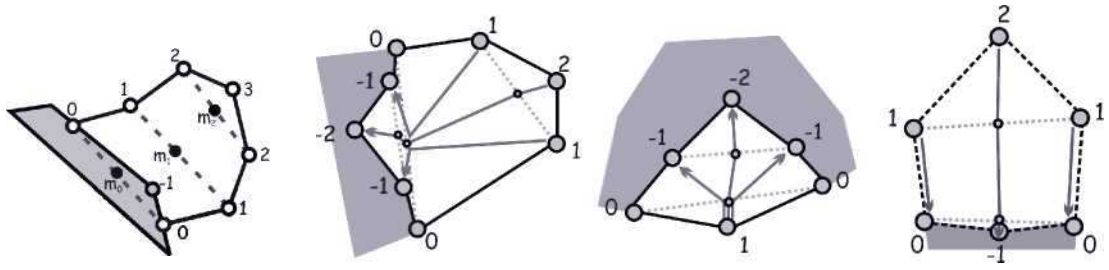


Figure 8.4: a), b) examples when not to use the Perimeter growing, c), d) examples when the Perimeter growing is suitable (dark gray: part of the core, light gray: part growing out, black: input polygons).

8.2.3. Half-line growing

The method called Half-line growing is similar to the Perimeter growing with a slight variance. Instead of using the vertices of $\mathbf{C}_1$ as the elements of the vertex paths, we use the midpoints of line segments defined by the vertices with the same topological distance (Figure 8.5a). Let us denote $\mathbf{M}_i$ a midpoint of the line segment $\mathbf{V}_i\mathbf{V}_j$ where $d_i = d_j$. Then, the vertex path of a vertex $\mathbf{V}_i$ with the topological distance d_i contains vertices ($\mathbf{M}_{i-1}$, $\mathbf{M}_{i-2}$, ...,$\mathbf{M}_0$, ...,$\mathbf{M}_{j-1}$, $\mathbf{V}_j$). The vertex $\mathbf{V}_j$ is computed according to the rules described in Section 8.2.2. The results for all three cases with a different relation between n_0, n_1 are shown in Figure 8.5b), c), d). When $n_0 = n_1$ (Figure 8.5b), each vertex of $\mathbf{C}_1$ ends its path at one vertex of $\mathbf{C}_0$. If $n_0 > n_1$ (Figure 8.5c), some of the vertices of $\mathbf{C}_1$ need to be duplicated (those with the topological distance equal to one). If $n_0 < n_1$ (Figure 8.5d), some vertices of $\mathbf{C}_1$ cannot end their paths at the supposed vertex (so they end at the intersection vertices).



a) b) c) d)

Figure 8.5: Midpoints (black): points in the middle of the line segment (dashed) connecting the vertices with the same topological distance.

The Half-line growing method is suitable for similar cases as was the Perimeter growing, with the slight difference that the top line of the growing part is not straight, but it is in the shape of a spire. This can result in better outputs for growing prickles or anything that is sharp, because the spire is there from the beginning, showing the future shape of the part (Figure 8.6).

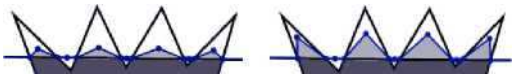


Figure 8.6: When to use the Half-line growing (dark gray: part of the core, light gray: part growing out, black: input polygons).

8.2.4. Projection growing

The last method to be described is the Projection growing. Its first difference from the previous methods is that all vertices of $\mathbf{C}_1$ have the same number of elements in their vertex paths. We call "one step projection" such a method, where there are only two elements in each vertex path of a vertex – the vertex itself and its destination. The "two step projection" represents a method, where the vertex path has also one element in the middle. Both of them work in the same way. First, the vertices of $\mathbf{C}_1$ are mapped (projected) onto the line segment defined by the intersection vertices (vertices with zero topological distance). An equidistant mapping is used – the line segment is divided into n+1 parts, where n is the number of the vertices of $\mathbf{C}_1$. We assign the vertices of $\mathbf{C}_1$ chronologically to the new vertices on the line segment (Figure 8.7a). The next step is to map the vertices of $\mathbf{C}_0$ in the same way (Figure 8.7b). Then we sort the projected vertices of $\mathbf{C}_0$ and $\mathbf{C}_1$ into one sorted list in the order in which they appear on the line segment defined by the intersection vertices. The last step is to go through this sorted list as follows (Figure 8.7c):

1. Go through the list until a vertex of $\mathbf{C}_0$ is reached. All the vertices of $\mathbf{C}_1$ that are before this vertex will have it in their vertex path.
2. Until the next vertex of $\mathbf{C}_0$ is reached, all the vertices of $\mathbf{C}_1$ will have the recent vertex of $\mathbf{C}_0$ in their vertex path.
3. Repeat step 2 until the end of the list is reached.

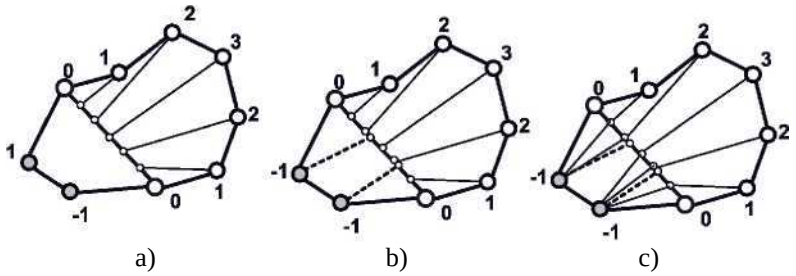


Figure 8.7: Computing vertex paths in the Projection growing: a) mapping the vertices of $\mathbf{C}_1$ onto the line segment between the intersection vertices, b) the same mapping of the vertices of $\mathbf{C}_0$, c) choosing the vertices of $\mathbf{C}_0$ for the vertex paths of the vertices of $\mathbf{C}_1$ (dashed: mapped vertices, thin lines: vertex paths).

For the Two step projection method, the vertex into which the vertex $\mathbf{v}_i$ was mapped also belongs to the vertex path of $\mathbf{v}_i$. For the One step projection method, only the vertex $\mathbf{v}_i$ itself and the assigned vertex of $\mathbf{C}_0$ are in the vertex path of $\mathbf{v}_i$ (Figure 8.8).

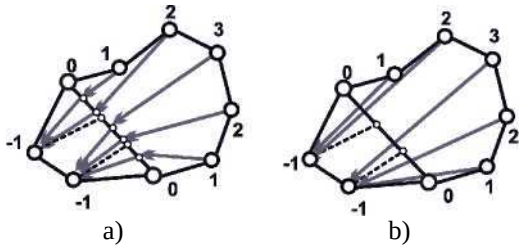


Figure 8.8: Vertex paths (grey) for a) Two step projection b) One step projection.

The projection growing is a method that provides its outputs somewhere between the Perimeter (or Half-line) growing and typical algorithms based on the correspondence. There is still dependence of the outputs on the core, however, the parts do not follow any shapes of the original polygons, they grow directly from the core. That results in its usability when the shape of the part is convex or when it is non-convex, but not curled or spiral (Figure 8.9).

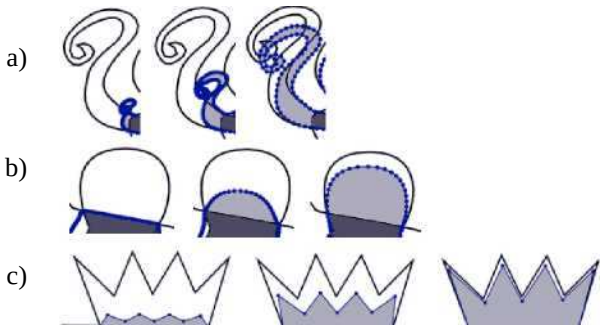


Figure 8.9: Where not to use (a) and where to use (b,c) the Projection growing (dark gray: part of the core, light gray: the part growing out, black: input polygons).

8.2.5. Merging

Until now each part of the polygon was handled separately. Now we have to merge all parts so that the result is one polygon with a vertex path for each vertex. Also remember that vertex paths of the growing parts must be reverted because we considered only the disappearing. The merging process is motivated by Weiler-Atherton algorithm for polygons intersection [Wei77]. It processes part by part by copying vertices with positive topological distance to the new list of vertices. It skips between the adjacent parts at the intersection vertices. The new list of vertices forms

the new polygon. The complete core increment morphing algorithm can be seen in Figure 8.10 and the details of the merging process are in Figure 8.11.

Input: Two polygons **A**, **B** (both represented by a list of vertices) which partially overlap.

Output: Polygon **C** (represented by a list of vertices), where each vertex contains a vertex path determining its behavior during the time.

The algorithm:

1. Compute the core $\mathbf{C} = \mathbf{A} \cap \mathbf{B}$. Compute the parts $\mathbf{P} = \mathbf{A}\text{-}\mathbf{B}$, $\mathbf{Q} = \mathbf{B}\text{-}\mathbf{A}$.
2. Compute the topological distance d_i of each vertex $\mathbf{V}_i$ of each part in **P**, **Q**. First, compute d_{i0}, d_{i1} as the minimal number of edges on the part between the vertex $\mathbf{V}_i$ and the two intersection vertices. Then $d_i = \min(d_{i0}, d_{i1})$.
3. Use a specific method (the Perimeter, Half-line or Projection growing) to compute vertex path of each vertex $\mathbf{v}_i$ where $d_i > 0$.
4. Merge the parts **P**, **Q** (see in detail in Figure 8.11).

Figure 8.10: The whole algorithm.

Input: List of parts $\mathbf{R} = \cup\mathbf{R}_i$, lists of vertices of each part $\mathbf{l}_i = (\mathbf{V}_1, \ldots, \mathbf{V}_n)$. The lists $\mathbf{l}_i$ are circular (so the next vertex to $\mathbf{V}_n$ is $\mathbf{V}_0$ and the previous vertex to $\mathbf{V}_0$ is $\mathbf{V}_n$). Each part has a different number of vertices in its list, but each part shares exactly two vertices with two other parts (the intersection vertices).

Output: One list of vertices containing such vertices $\mathbf{V}_j$ from the lists $\mathbf{l}_i$ that have $d_j > 0$.

The merging algorithm:

1. Choose an arbitrary part $\mathbf{R}_i$ from the list of input parts (for example the first one). Start from the first vertex in $\mathbf{R}_i$. Go through $\mathbf{l}_i$ until the first intersection vertex $\mathbf{V}_j$ is found. Add $\mathbf{V}_j$ to the resulting list (which now contains only $\mathbf{V}_j$).
2. Check the vertex $\mathbf{V}_{j+1}$ if its topological distance is positive. If so, continue forward, otherwise backward in $\mathbf{l}_i$. Add each visited vertex to the resulting list until the next intersection vertex $\mathbf{V}_k$ is added. Delete the part $\mathbf{R}_i$ from the list of parts.
3. Because $\mathbf{V}_k$ was the intersection vertex, either one of the parts in the list contains it (in such a case use this part and continue by 2), or the list of parts is empty ($\mathbf{V}_k$ is the intersection vertex from step 1). In such a case, the algorithm is finished.

Figure 8.11: The merging algorithm.

8.2.6. Improvements

Both Perimeter and Half-line algorithm do not take into account lengths of edges of the polyline they morph, they compute only with the topological distance of the points. That results in a different behavior for the same-shaped polygons with a different number of vertices. A solution is to include a preprocessing part to this algorithm, when the polygons are "resampled", so that all their edges are of the same length. If we include

such resampling into our algorithm, the result is even better. Not only it results in a smoother and controlled movement of each part, but also the segments that are about the same length have approximately the same number of vertices.

Another improvement comes from the fact that each part $\mathbf{P}_i$ is computed separately. It means that the parts do not have to be computed by the same method, the methods can be arbitrarily combined. As each method is suitable for a different kind of shapes (as is discussed in Section 8.3), combining the methods can result in a more interesting output.

8.3. Experiments

The results of our algorithm were compared with the Sedeberg's and Greenwood's[28] algorithm [Sed93a] and with the Carmel's and Cohen-Or's[29] algorithm [Car97], well-known correspondence-based algorithms.

In the following examples we will demonstrate a behavior of different methods of vertex path computation of our algorithm for different shapes. Although it is possible to morph each part of a shape independently, in our examples we will morph all parts using only one method, so that we can clearly demonstrate its suitability for the given shape and the given type of effect.

As already mentioned, our algorithm is completely suitable for the cases when one would expect some parts of the polygon **B** to grow out of the polygon **A** (or some parts of **A** disappearing in **B**). Those parts can be horns, prickles, fingers or tails, usually in situations when the user wants them to appear (grow out), or disappear in something. However, our algorithm is not suitable for similar polygons which are only transformed (e.g., moved, rotated, scaled), where the user expects the polygon only to move according to the transformation, not to change its shape.

Examples in sections 8.3.1 and 8.3.2 show the case where the user expects some parts of the polygon to grow or to disappear, the example in Section 8.3.3 shows morphing of completely different polygons.

8.3.1. Parts of a spiral type

For the shape of a spiral type, the Perimeter (Figure 8.12) or the Half-line growing are better then the Projection (Figure 8.13), where the result contains many self-intersections.

[28] An implementation by P. Celba, Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic, http://iason.zcu.cz/~kolinger/AVG/Metamorfoza_Celba.zip.
[29] An implementation obtained from http://w3.impa.br/~morph/software/softw-2d-morphing.html.

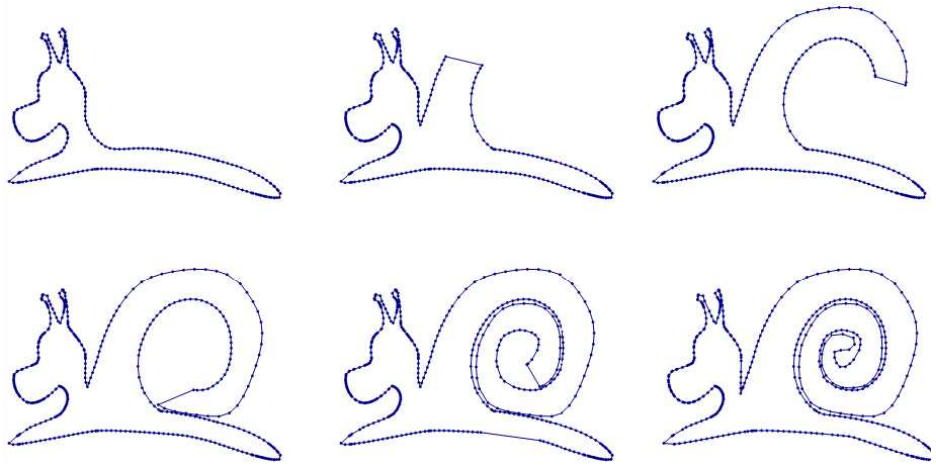


Figure 8.12: The perimeter growing.

Although the Projection growing is not producing results we would expect when we want something of a spiral type to grow out of the core, it can sometimes give interesting esthetical results when we fill the polygons by some color (because the overlapping parts have the color of the background).

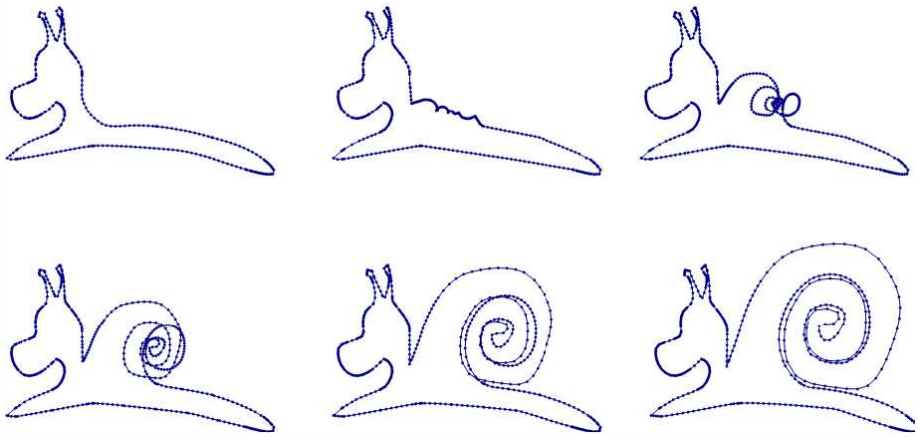
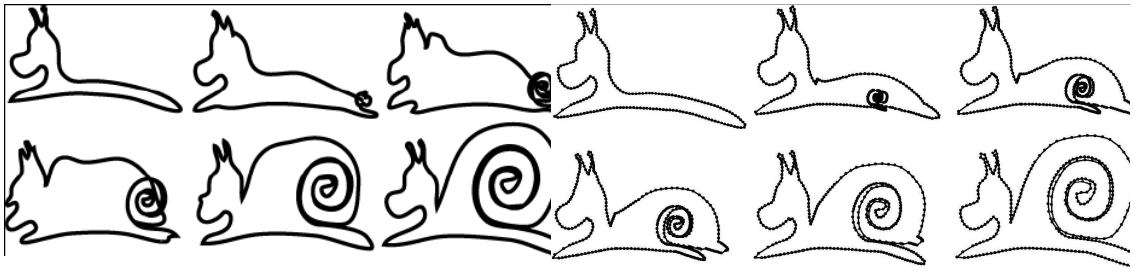


Figure 8.13: The projection growing.

Carmel's and Cohen-Or's algorithm (Figure 8.14a) and Sedeberg's and Greenwood's algorithm (Figure 8.14b) give results containing many self-intersections here.

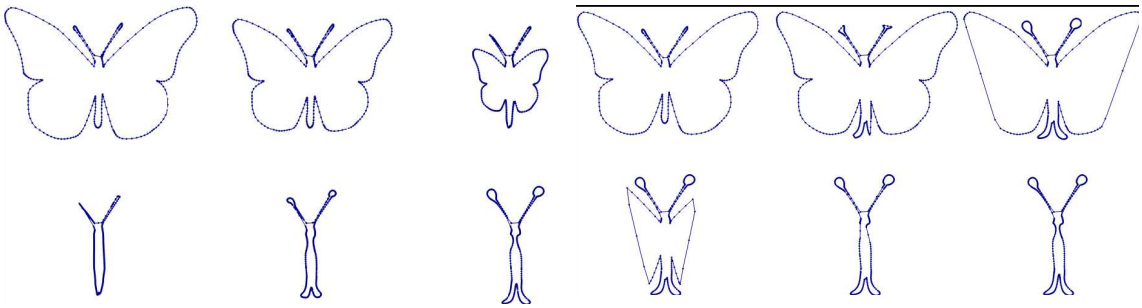


a) b)

Figure 8.14: a) the Carmel's and Cohen-Or's algorithm, b) the Sedeberg's and Greenwood's algorithm.

8.3.2. Convex Parts

For parts which are convex or nearly convex the Projection growing (Figure 8.15a) is more suitable then the Perimeter or Half-line growing (Figure 8.15b).

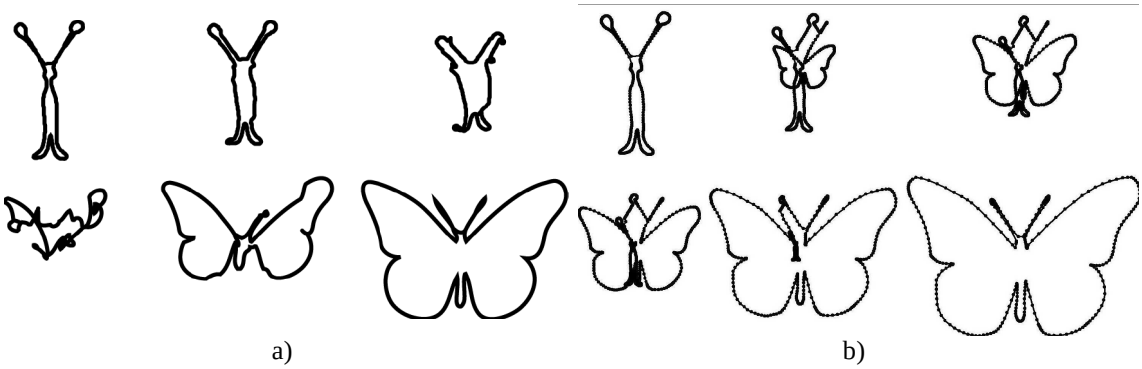


a) b)

Figure 8.15: a) the Projection growing, b) the Half-line growing.

In the example of a butterfly and an alien, the body of the butterfly is similar to the alien, so one would probably expect the wings of the butterfly to disappear in the body of the alien, and the eyes (at the end of the antenna) to grow out from the antenna of the butterfly.

Both Carmel's and Cohen-Or's algorithm (Figure 8.16a) and Sedeberg's and Greenwood's algorithm (Figure 8.16b) result in many self-intersections in this case.



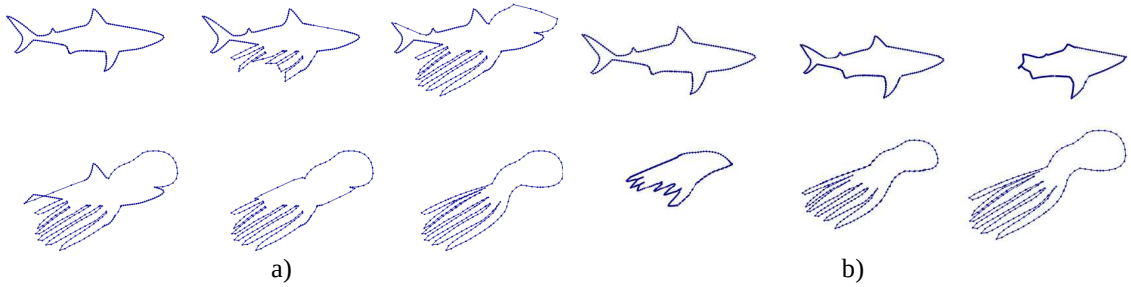
a) b)

Figure 8.16: a) The Carmel's and Cohen-Or's algorithm, b) the Sedeberg's and Greenwood's algorithm.

8.3.3. Long and more or less straight parts

When the shape of the parts is long and more or less straight, the Projection (Figure 8.17a), Perimeter and Half-line (Figure 8.17b) growing have results of a similar quality. In this example, such a case appears at octopus' fingers. For the other parts, the Projection growing appears to be more suitable.

The case shown in this particular example is not the case, where one would naturally expect the growing behavior, because the octopus and the shark do not have any similar part. However, growing of the octopus' fingers is probably the only way how to morph from the shark's stomach into them without an intersection.



a) b)

Figure 8.17: a) the Half-line growing, b) the Projection growing.

The result Carmel's and Cohen-Or's algorithm (Figure 8.18a) is quite similar as our growing algorithm for the case of octopus' fingers. The Sedeberg's and Greenwood's algorithm (Figure 8.18b) experiences a few intersections here.

a) b)

Figure 8.18: a) the Carmel's and Cohen-Or's algorithm, b) the Sedeberg's and Greenwood's algorithm.

In the previous three examples we showed that our algorithm produces expected results for the case when some parts of one polygon are supposed to grow out of the other polygon or disappear in it. However, as we can see in the last example, the algorithm can be also used in some cases where growing is not expected and still produce acceptable results. On the other hand, it is not very suitable for the polygons that are similar or nearly similar with dissimilarities of non-growing type (like faces with different expressions, bent and straight finger etc.).

8.4. Conclusion and future work

In this chapter we presented an algorithm for computing morphing between two simple polygons that partially overlap. Our algorithm does not compute the correspondence between the two polygons, which results in a completely different behavior than the correspondence-based methods have. It also does not require user interaction; however, the user can adjust the mutual position of the polygons or specify a different growing method for each part.

We compared the results of our algorithm with two other correspondence-based methods. The experiments confirmed that our algorithm is completely suitable for the cases, where user expects some parts of the polygon to grow out from the intersection or disappear in it. Our algorithm can be suitable also for some other than grow-like cases, but it is not useful for the polygons that are of the same shape and they are only transformed (rotated, translated etc.).

There is already an ongoing research carried out by Ing. Martina Málková who deals with a generalization of the core-increment technique in 3d dimensions. She used the 2d version of the core-increment technique as an analysis of the problem. Challenging problems of the 3d version are: generalization of the topological distance and a generalization of the growing technique (i.e., perimeter growing, half-line growing or projection growing).

9. Conclusion

In this thesis we dealt with morphing of geometrical objects in boundary representation. It is a very wide topic and there are many ways how to do research in this area. Our research can be roughly divided into two groups – extensions of existing solutions and new techniques. The extensions improve some well established techniques to operate faster, to produce better results, etc. The new techniques are derived from some existing approaches as well, but the novelty is considerably bigger than in the case of extensions. From this perspective, the extensions are:

- improvements of the topology merging technique – we extended the topology merging technique so that it takes into account surface attributes and we also described some ways how to improve a quality of meshes,
- computation of normal vectors – since the computation of normal vectors is an every day task a fast computation is needed, therefore we showed some methods how to compute it for triangular meshes under a linear motion,
- elementary predicates for continuous collision detection – we described a derivation of a collision equation based on t-variant cross product and we identified singular cases which lead to a simplification of the collision equation.

As new techniques, the following contribution can be considered:

- multimorphing – a space of shapes is considered as an affine space where we defined an inner product and an orthogonal projection; we also showed some new ways how to produce new shapes and animations,
- core-increment morphing – a new technique which is motivated by the process of growing.

Another possible contribution of this thesis is a “refactoring” of existing approaches (Chapter 3), i.e., we identified some recurring “design patterns” which are common in various approaches. It is useful for study and understanding of existing approaches. For instance, in the Chapter 3 we showed that the topology merging technique is closely related with parametrization. However, the first paper [Ken92] where the topology merging appears, does not mention this relation. Instead it develops an ad hoc approach which leads to a limitation of the technique so that it worked with star-shaped meshes only. Therefore, an ability to identify the design patterns in an existing approach is

important for assessing the approach, for instance, when considering it for a practical use.

Another possible division of the contributions presented in this thesis is how far the given contribution can be used in practice. Clearly, some ideas are more practically oriented, while some ideas belong more to a fundamental research, i.e., they show some potential but a practical use is still an open question. The practically oriented contributions are tuned for a specific application (i.e., triangular meshes, linear motion, etc.) while the fundamental research related contributions have possibly a wider application domain (not only in the field of the computer graphics). From this perspective, we consider the following contribution to be oriented more practically:

- extension of the topology merging technique, especially the handling of attributes,
- vertex normal computation based on the t-variant cross product,
- elementary predicates for continuous collision detection.

On the other hand, the following contributions belong more to the area of the fundamental research and we believe that they can be used as a theoretical basis or an inspiration for a future research:

- multimorphing – some fundamental analogies were outlined, the apparatus was tested on artificial examples, however, a use in real world applications must be considered,
- face normal computation of deforming meshes – some ways how to interpolate the normal vectors were suggested, however a use a specific technique must be considered with respect to the application domain and available resources,
- core-increment technique – since our main interest was the 3d version of the algorithm, we used the 2d version as an analysis of the problem so that we were able to identify the main difficulties.

Let us recall that the shape interpolation is not well defined, there are many different ways how to interpolate between two shapes. Clearly, there are some constrains which eliminate some morphs, but still a huge number of possibilities remains. In this case, it is important that the user of the morphing has a possibility to control the shape transformation. Ideally, the user should be allowed to change the morphing, however the technique should work without the user input as well. We try to pursue this goal in the Bézier morphing and in the core increment morphing. The Bézier morphing works without the user input, it generates an animation which approximates the control shapes in the same way as the Bézier curve approximates the control vertices. However, the Bézier animation can be adjusted by increasing multiplicity of some shape. It causes that the animation approximates closer the multiplied shape. Additionally the Bézier morphing can be extended to Rational Bézier morphing where the user has yet more control over the resulting animation because it is possible to adjust weights of the individual control shapes. The advantage of the core-increment morphing is that it is able to morph complicated shapes without user interaction. But still, it is possible to adjust the animation by changing a mutual position of the input shapes.

9.1. Summarization of the future work

Although directions for a further research were given in each chapter containing our contribution, let us briefly summarize it in the following list:

- validation of improvements of quality of meshes originating from the topology merging process (Section 4.4),
- sketch of an algorithm which generalizes the topology merging for multiple meshes (Section 4.5),
- practical use of the apparatus related to the multimorphing (Chapter 5),
- quaternion SLERP method (Section 6.3.5) – a derivation of rules for the best frame configuration,
- verification and validation of quaternion correction (Section 6.5),
- extension of the core-increment technique into 3d (Section 8.4).

9.2. Ongoing work

The core-increment approach was researched in cooperation with Ing. Martina Málková. Even though the 2d version of the core-increment technique was researched mainly to analyze the problem and to identify the pitfalls, the implementation showed some interesting results. Therefore it was presented as a standalone technique in [Mal07]. Moreover, Ing. Martina Málková defended a master degree thesis [Mal08a] which deals with an extension of 2d core-increment morphing technique into 3d. Results of the thesis together with some extensions were also published in [Mal08b]. Briefly, conclusions drawn by this work are following:

- it is possible to generalize core-increment technique into 3d,
- the technique is suitable for 3d shape transformation where it is expected that some parts will grow or disappear in a common core (e.g., tentacles, leafs, etc.),
- the main obstacle is that for the interpolation, the parts which grow or disappear must be represented as isomorphic meshes.

An example of 3d core-increment morphing is shown in Figure 9.1. Even though the research of core-increment technique showed some interesting results, it seems that the boundary representation limits the potential of the technique. While it was relatively easy to solve it in 2d, it appeared to be more problematic in 3d. Therefore, in the future research it would be good to abstract away from the underlying data representation and formulate the core-increment morphing in the terms of differential geometry.

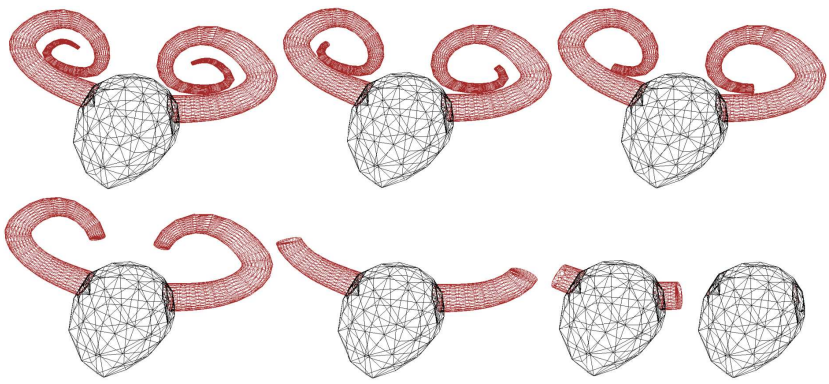


Figure 9.1: An example of 3d core-increment morphing technique (taken from [Mal08a]).

Appendix A Spherical geometry

A.1. Point in Spherical Triangle

To test if a point lies in a spherical triangle we adapted a standard point-in-triangle test. An edge of a spherical triangle is replaced by a plane formed by two edge endpoints and the center of the sphere. The point-in-triangle test then turns to checking whether the query point **Q** is oriented in the same way with respect to all three planes formed by triangle edges and the center of the sphere, i.e.:

$$((\mathbf{V}_0 \times \mathbf{V}_1).\mathbf{Q} \geq 0) \wedge ((\mathbf{V}_0 \times \mathbf{V}_1).\mathbf{Q} \geq 0) \wedge ((\mathbf{V}_0 \times \mathbf{V}_1).\mathbf{Q} \geq 0), \tag{A.1}$$

where $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$ are the vertices of the triangle and **Q** is the query point. The term $(\mathbf{V}_0 \times \mathbf{V}_1).\mathbf{Q}$ is the orientation test of the query point **Q** and the plane which is given by vertices $\mathbf{V}_0$, $\mathbf{V}_1$ and the center of the sphere. In fact it is a dot product between the normal of the plane and the vector formed by the query point **Q** and the center of the sphere.

A.2. Spherical Barycentric Coordinates

To compute barycentric coordinates of a point **Q** (which lies on a unit sphere) with respect to a spherical triangle $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$ we first compute barycentric coordinates of **Q** with respect to the planar triangle $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$. It results in values *t, u, w*. Since the point **Q** does not lie in the planar triangle $\mathbf{V}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$, we normalize the barycentric coordinates as follows:

$$t' = t / (t + u + v),\ u' = u / (t + u + v),\ \mathrm{w}' = w / (t + u + v) \tag{A.2}$$

A.3. Arc-Arc intersection

In this section we will describe a computation of arc-arc intersection where the arcs are parts of great circles of a sphere with a center **C**. Denote $\mathbf{P}_1$, $\mathbf{P}_2$ the endpoints of the first arc and $\mathbf{Q}_1$, $\mathbf{Q}_2$ the endpoints of the other arc. As the arcs are parts of great circles, we first investigate an intersection of great circles and then check whether the intersection is a common point of both arcs. The endpoints $\mathbf{P}_1$, $\mathbf{P}_2$ together with a central point **C** form a plane. So the intersection of two great circles lies on the intersections of planes formed by $\mathbf{P}_1$, $\mathbf{P}_2$, **C** and $\mathbf{Q}_1$, $\mathbf{Q}_2$, **C**. The intersection of planes is a line with a vector **r**:

$$\mathbf{r} = \pm(\mathbf{P}_1 \times \mathbf{P}_2) \times (\mathbf{Q}_1 \times \mathbf{Q}_2)\ , \tag{A.3}$$

where $(\mathbf{P}_1 \times \mathbf{P}_2)$, $(\mathbf{Q}_1 \times \mathbf{Q}_2)$, respectively, gives the perpendicular vectors to the plane $\mathbf{P}_1$, $\mathbf{P}_2$, **C** and $\mathbf{Q}_1$, $\mathbf{Q}_2$, **C**, respectively. By normalizing the vector **r** we have two intersections of great circles and one of them could be the desired intersection of arcs. To check whether the intersection lies on both arcs, we solve the system:

$$\begin{aligned} t_\mathrm{p}.\mathbf{r} &= \mathbf{P}_1 + s_\mathrm{p}(\mathbf{P}_2 - \mathbf{P}_1) \\ t_\mathrm{q}.\mathbf{r} &= \mathbf{Q}_1 + s_\mathrm{q}(\mathbf{Q}_2 - \mathbf{Q}_1) \end{aligned}, \tag{A.4}$$

where t_p, t_q and s_p, s_q are unknowns. The intersection is a common point of two arcs if $s_\mathrm{p}, s_\mathrm{q} \in \langle 0; 1 \rangle$ and $t_\mathrm{p}, t_\mathrm{q} > 0$.

Appendix B Quaternions

In this appendix we will briefly review quaternions and their use for representation of rotations. Quaternions were introduced to computer graphics community by Shoemake [Sho85] but they date back to 1843 when they were first described by Hamilton as an extension of complex numbers. A comprehensive description of quaternions can be found in [Ebe04].

A quaternion is a tuple (**a**, *s*), where **a** is a vector and *s* is a scalar. Quaternions are used in computer graphics to represent 3d rotations. An arbitrary rotation can be described by a quaternion **q** as:

$$\mathbf{q} = \cos(s/2) + \mathbf{a}\sin(s/2)\,, \tag{B.1}$$

where **a** is the axis of the rotation, *s* is the angle of rotation around the axis **a**. In the same way as rotation matrices are compounded using the matrix multiplication, the rotations represented by quaternions can be compounded by quaternion multiplication. The multiplication of quaternions $\mathbf{q}_0 = (\mathbf{v}_0, s_0)$, $\mathbf{q}_1 = (\mathbf{v}_1, s_1)$ is described as:

$$\mathbf{q}_0\mathbf{q}_1 = (s_0 s_1 - \mathbf{v}_0\mathbf{v}_1,\ s_0\mathbf{v}_1 + s_1\mathbf{v}_0 + \mathbf{v}_0 \times \mathbf{v}_1). \tag{B.2}$$

A rotation of a vector **v** using a quaternion **q** is expressed as:

$$\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^*\,, \tag{B.3}$$

where $\mathbf{q}^*$ is the conjugate quaternion defined as $\mathbf{q}^* = (-\mathbf{a}, s)$ and **v** is considered as a quaternion with a zero real part, i.e., $\mathbf{v} = (v_x, v_y, v_z, 0)$.

Note that the rotation represented by quaternions requires 4 values whereas the rotation represented by a rotation matrix requires 9 values. Therefore, the quaternion representation is more economical. Also, it is less vulnerable to the error accumulation when many rotations are compound.

Quaternions are also useful for an interpolation of orientation. A quaternion spherical linear interpolation (QSLERP) is used to smoothly interpolate between two quaternions. QSLERP between quaternions $\mathbf{q}_0=(\mathbf{v}_0, s_0)$, $\mathbf{q}_1=(\mathbf{v}_1, s_1)$ is defined as:

$$\mathbf{q}(t) = \frac{\sin((1-t)\alpha)\mathbf{q}_0 + \sin(t\alpha)\mathbf{q}_1}{\sin\alpha}\,, \tag{B.4}$$

where $\alpha = \mathbf{v}_0\mathbf{v}_1 + s_0 s_1$ (i.e., the dot product of quaternions $\mathbf{q}_0$, $\mathbf{q}_1$) and *t* is the interpolation parameter.

Appendix C Hybrid approach for cubic equation solution

In this appendix we will describe a hybrid approach for finding real roots of a cubic polynomial $P(t)$:

$$P(t) = at^3 + bt^2 + ct + d. \tag{C.1}$$

It can be solved algebraically by Cardano's formulas but it involves computations with complex numbers. In the solution of the collision equation (Section 7.3) we are interested in real roots only since they represent the time of the collision between a point and a plane. Furthermore we are interested only in real roots within a certain interval. Here, without loss of generality, let us consider a canonical interval <0; 1>.

Local extremes of $P(t)$ are computed as:

$$e_{1,2} = \frac{-b \pm \sqrt{b^2 - 3ac}}{3a}, \tag{C.2}$$

where e_1 is a local maximum and e_2 is a local minimum. Denote $D = b^2 - 3ac$. There are three possible cases:

- $D < 0$, $P(t)$ is monotonous and it has one real root and two complex roots.
- $D = 0$, $P(t)$ has one real triple root.
- $D > 0$, $P(t)$ has at least one real root.

Since the cubic polynomial with real coefficients has at least one real root, we suggest to find one real root r_0 numerically by the Newton method. Then, the original cubic polynomial $P(t)$ is divided by the term $(t-r_0)$. The division can be done by the Horner scheme. It results in a quadratic polynomial. Roots of the quadratic polynomial can be computed algebraically.

The case when $D < 0$ is used as an early reject test. If $D < 0$ there is only one real root of $P(t)$. Since we are interested only in real roots within interval <0; 1> we can check if $\text{sgn}(P(0)) \neq \text{sgn}(P(1))$, which means that there is a real root of $P(t)$ on the interval <0; 1>, otherwise there is no real root on the interval <0; 1> and thus there can be no collision. Hence, if $D < 0$ and $\text{sgn}(d) = \text{sgn}(a+b+c+d)$ then we can skip the computation of real roots on the interval <0; 1>.

References

[Ale99] Alexa, M., Müller W.: The Morphing Space, Proceedings of WSCG'99, Pilsen, Czech Republic, pp. 329-336, 1999.

[Ale00a] Alexa, M., Behr, J., Müller, W.: The Morph Node, VRML '00: Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML), Monterey, California, United States, pp. 29-34, 2000.

[Ale00b] Alexa, M.: Merging Polyhedral Shapes with Scattered Features, The Visual Computer, Vol. 16, No. 1, pp. 26-37, 2000.

[Ale00c] Alexa, M., Cohen-Or, D., Levin, D.: As-Rigid-As-Possible Shape Interpolation, Proceedings of SIGGRAPH 2000, pp. 157-164, 2000.

[Ale00d] Alexa, M., Müller, W.: Representing Animations by Principal Components, Computer Graphics Forum, Vol. 19, No. 3, pp. 411-418, 2000.

[Ale01a] Alexa, M.: Recent Advances in Mesh Morphing, Computer Graphics Forum, Vol. 21, No. 2, pp. 173-197, 2002.

[Ale01b] Alexa, M.: Local Control for Mesh Morphing, Proceedings of Shape Modeling International 2001, pp. 209-215, 2001.

[Ale03] Alexa, M.: Differential Coordinates for Local Mesh Morphing and Deformation, The Visual Computer, Vol. 19, No. 2-3, 2003, pp. 105-114.

[Bar04] Barrera, T., Hast, A., Bengtsson, E.: Incremental Spherical Linear Interpolation, SIGRAD 2004, pp. 7-10, 2004.

[Bar84] Barr, A., H.: Global and Local Deformations of Solid Primitives, SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques, pp. 21-30, 1984.

[Bei92] Beier, T., Neely, S.: Feature-Based Image Metamorphosis, SIGGRAPH '92: Proceedings of the 19th annual conference on computer graphics and interactive techniques, pp. 35-42, 1992.

[Ber03] G. Bergen: Collision Detection in Interactive 3D Environments, Morgan Kaufman, 2003.

[Cam90] Cameron, S.: Collision Detection by Four-dimensional intersection testing, IEEE Transactions on Robotics and Automation, Vol. 6, No. 3, pp. 291-302, 1990.

[Car97] Carmel, E., Cohen-Or, D.: Warp-guided object space morphing, The Visual Computer, Vol. 13, No. 9-10, pp. 465-478, 1997.

[Din05] Dinh, H., Q., Yetti, A., Turk, G.: Texture transfer during shape transformation, ACM Trans. Graph., Vol. 24, No. 2, pp. 289-310, 2005.

[Ebe04] Eberly, D., H.: Game Physics, Morgan Kaufman, 2004.

[Flo05] Floater, S., M., Horman, K.: Surface Parametrization: a Tutorial and Survey, Advances in Multiresolution for Geometric Modelling, pp. 157-186, 2005.

[Gom99] Gomes, J., Darsa, L., Costa, B., Velho, L.: Morphing and Warping of Graphical Objects, Morgan Kaufmann, 1999

[Gre99] Gregory, A., State, A., Lin, M., C., Manocha, D., Livingston, M., A.: Interactive Surface Decomposition for Polyhedral Morphing, The Visual Computer, Vol. 15, pp. 453-470, 1999.

[Has03] Hast, A., Barrera, T., Bengtsson, E.: Improved Shading Performance by Avoiding Vector Normalization, Proceedings of WSCG'01, pp. 1-8, 2001.

[He94] He, T., Wang, S., Kaufamn, A.: Wavelet-Based Volume Morphing, VIS '94: Proceedings of the conference on Visualization '94, IEEE Computer Society Press, pp. 85-92, 1994.

[Her90] Herzen, B., Barr, A., H., Zatz, H. R.: Geometric Collision for time-dependent parametric surfaces, ACM Computer Graphics, Vol. 24, No. 4, pp. 39-48, 1990.

[Hop96] Hoppe, H.: Progressive Meshes, Computer Graphics, Vol. 30, pp. 99-108, 1996.

[Hug92] Hughes, J., F.: Scheduled Fourier Volume Morphing, SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques, pp. 43-46, 1992.

[Hut01] Hutton, T., J., Buxton, B., F., Hammond, P.: Dense Surface Point Distribution Models of the Human Face, IEEE Workshop on Mathematical Methods in Biomedical Image Analysis, pp. 153-160, 2001.

[Hut07] Hutter, M., Fuhrman, A.: Optimized Continuous Collision Detection for Deformable Triangle Meshes, Proceedings of WSCG, 2007.

[Jin05] Jin, S., Lewis, R., R., West, D.: A Comparison of Algorithms for Vertex Normal Computation, The Visual Computer, Springer-Verlag GmbH, Vol. 21, No 1-2, pp. 71 – 82, 2005.

[Jir04] Jirka, T., Skala, V.: Isosurface vertex normal computation, Zeszyty Naukowe Politechniki Slaskiej; no. 1615, Geometria i grafika inżynierska, No. 6, pp. 27-32, 2004.

[Joh02] Johnstone, J., K., Wu, X.: Morphing two polygons into one, 40th Annual Southeast ACM Conference, 2002.

[Kan97] Kanai, T., Suzuki, H., Kimura, F.: 3D Geometric Metamorphosis Based on Harmonic Map, Proceedings of the 5th Pacific Conference on Computer Graphics and Applications, pp. 97-104, 1997.

[Kar04] Karni, Z., Gotsman, C.: Compression of soft-body animation sequences, Computers and Graphics, Vol. 28, pp. 25-34. 2004.

[Ken92] Kent, J., R., Carlson, W., E., Parent, R., E.: Shape Transformation for Polyhedral Objects, Computer Graphics, Vol. 26, Issue 2, pp. 47-54, 1992.

[Kim03] Kim, B., Rossignac, J.: Collision Prediction for Polyhedra under Screw Motions, SM'03: Proceedings of the eights ACM symposium on Solid modeling and applications, pp. 4-10, 2003.

[Kor98] Korfiatis, I., Paker, Y.: The Three-dimensional object metamorphosis through energy minimization, Computers & Graphics, Vol. 22, No. 2-3, pp. 195-202, 1998.

[Kra04] Kraevoy, V., Sheffer, A.: Cross-parameterization and compatible remeshing of 3D models, ACM Trans. Graph., ACM Press, Vol. 23, No. 3, pp. 861-869, 2004.

[Lar03] Larsson, T., Akeine-Moller, T.: Efficient collision detection for models deformed by morphing, The Visual Computer, Vol. 19, No. 2-3, pp. 164-174, 2003.

[Laz98] Lazarus, F., Verroust, A.: Three-dimensional metamorphosis: a survey, The Visual Computer, Vol. 14, Issue 8 - 9, pp. 373 – 389, 1998.

[Lee99] Lee, A., W., F., Dobkin, D., Sweldens, W., Schröder, P.: Multiresolution Mesh Morphing, SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pp. 343-350, 1999.

[Len04] Lengyel, E.: Mathematics for 3D Game Programming & Computer Graphics, 2nd Edition, Charles River Media, pp. 179, 180, 2004.

[Ler95] Lerios, A., Garfinkle, C., D., Levoy, M.: Feature-Based Volume Metamorphosis, SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, pp. 449-456, 1995.

[Lip04] Lipman, Y., Sorkine, O., Cohen-Or, D., Levin, D., Rössl, C., Seidel, H.: Differential Coordinates for Interactive Mesh Editing, Proceedings of Shape Modeling International, pp. 181-190, 2004.

[Mal07] Málková, M.: A new core-based morphing algorithm for polygons, CESCG 2007, pp. 39-46, 2007.

[Mal08a] Málková, M.: Morphing of geometrical objects in boundary representation, Master degree thesis, University of West Bohemia, 2008.

[Mal08b] Málková, M., Kolingerová, I., Parus, J.: Core-based morphing algorithm for triangle meshes, SIGRAD 2008, Linköping University Electronic Press, Linköping, Sweden, pp. 39-46, 2008.

[Max99] Max, N.: Weights for computing vertex normals from facet normals. Journal of Graphics Tools, 4(2), pp.1-6, 1999.

[Mey02] Meyer, M., Lee, H., Barr, H., Desbrun, M.: Generalized Barycentric Coordinates on Irregular Polygons, Journal of Graphics Tools, Vol. 7, No. 1, pp. 13-22, 2002.

[Mic01] Michikawa, T., Kanai, T., Fujita, M., Chiyokura, H.: Multiresolution Interpolation Meshes, Proceedings of the 9th Pacific Conference on Computer Graphics and Applications, pp. 60-69, 2001.

[Mol97] Akeine-Moller, T: A Fast Triangle-Triangle Intersection Test, Journal of Graphics Tools, Vol. 2, No. 2, 25-30, 1997.

[Par05] Parus, J.: Morphing of Meshes, Technical Report DCSE/TR-2005-02, University of West Bohemia, 2005.

[Pas04] Pasko, G., Nieda, T., Pasko, A., Kunii, T., L.: Space-time modeling and analysis, SCCG '04: Proceedings of the 20th spring conference on Computer graphics, pp. 13-20, 2004.

[Pro97] Provot, X.: Collision and self-collision handling in cloth model dedicated to design garments, Proceedings of Graphics Interface'97, pp. 177-189, 1997.

[Red00] Redon, S., Kheddar, A., Coquillart, S:. An Algebraic Solution to the Problem of Collision Detection for Rigid Polyhedral Objects, Proceedings of IEEE International Conference on Robotics and Automation, 2000.

[Red04] Redon, S:. Continuous Collision Detection for Rigid and Articulated Bodies, ACM SIGGRAPH Course Notes, 2004.

[Red05] Redon, S:. Fast Continuous Collision Detection and Handling for Desktop Virtual Prototyping, Virtual Reality, 8, 1, 63-70, 2005.

[Sed93a] Sedeberg, W., T., Greenwood, E.: A Physically Based Approach to 2-D Shape Blending, SIGGRAPH '92: Proceedings of the 19th annual conference on computer graphics and interactive techniques, pp. 25-34, 1992.

[Sed93b] Sedeberg, W., T., Gao, P., Wang, G., Mu, H.: 2-D Shape Blending: An Intrinsic Solution to the Vertex Path Problem, Computer Graphics, Vol. 27, pp. 15-18, 1993.

[Sha95] Shapira, M., Rappoport, A.: Shape Blending Using the Star-Skeleton Representation, IEEE Computer Graphics and Applications, Vol. 15, No. 2, pp. 44-50, 1995.

[Sho85] Shoemake, K.: Animating rotation with quaternion curves, SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp. 245-254, 1985.

[Sny93] Snyder, J., M.,Woodbury, A. R, Fleischer, K., Currin, B., Barr, A., H.: Interval methods for multi-point collisions between time-dependent curved surfaces, SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, pp. 321–334, 1993.

[Sun97] Sun, Y., M., Wang, W., Chin, F., Y, L.: Interpolation Polyhedral Models Using Intrinsic Shape Parameters, Journal of Visualization and Computer Animation, Vol. 8, 81-96, 1997.

[Sur01] Surazhsky, V., Gotsman, C.: Controllable morphing of compatible planar triangulations, ACM Transactions on Graphics, Vol. 20, No. 4, pp. 203-231, 2001.

[Sur04] Surazhsky, V., Gotsman, C.: Intrinsic Morphing of Compatible Triangulations, International Journal of Shape Modeling, Vol. 9, No. 2, pp. 191-201, 2003.

[Tes04] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Rahupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser and P. Volino. Collision Detection for Deformable Objects, STAR EG 04, 2004.

[Tur99] Turk, G., O'Brien, J., F.: Shape transformation using variational implicit functions, SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, pp. 335-342, 1999.

[URL1] A Fast Triangle-Triangle Intersection Test, http://jgt.akpeters.com/papers/Moller97

[Vla01] Vlachos, A., Peters, J., Boyd, C., Mitchell, J., L.: Curved PN Triangles, I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics, pp. 159-166, 2001.

[Wei77] Weiler, K., Atherton, P.: Hidden surface removal using polygon area sorting, SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques, pp. 214-222, 1977.

[Zoc00] Zockler, M., Stalling, D., Hege, H.: Fast and Intuitive Generation of Geometric Shape Transitions, The Visual Computer, Vol. 16, No. 5, pp. 241-235, 2000.

Activities

Publications related to the dissertation:

- Málková, M., Kolingerová, I., Parus, J.: Core-based morphing algorithm for triangle meshes, SIGRAD 2008, Linköping University Electronic Press, Linköping, Sweden, pp. 39-46, 2008.
- Hast A., Parus J., Kolingerová, I.: Improved Normal Computation by Quaternion Correction, 7th International Conference APLIMAT 2008, Bratislava, Slovakia, pp. 829-837, 2008.
- Parus, J., Hast, A., Kolingerová, I.: Elementary Predicates for Continuous Collision Detection, 13th International Conference on Geometry and Graphics (ICGG2008), Dresden, Germany, 2008.
- Parus, J., Hast, A., Kolingerová, I.: Fast Computation of Vertex Normals for Linearly Deforming Meshes, Journal of Graphics Tools, Vol. 12, No. 4, A. K. Peters, Ltd., pp. 47-58, 2007.
- Hast, A., Parus J., Kolingerová, I.: Smooth Shading for Soft-Body Animation of Meshes Obtained by Topology Merging, Using Quaternion Correction (poster), Symposium on Computer Animation 2006, Vienna, Austria, 2006.
- Parus, J., Hast A., Kolingerová, I.: Temporal Face Normal Interpolation, SIGRAD 2006, Skövde, Sweden, pp. 12-16, 2006.
- Parus, J., Kolingerová, I.: Normal Evaluation for Soft-body Deforming Meshes, Simulation and Visualization 2006 (SimVis2006), Magdeburg, Germany, pp. 157-168, 2006.
- Parus, J., Kolingerová, I.: Morphing of Color Surfaces, Electronics Computers and Informatics (ECI 2004), Herlany, Slovakia, pp. 415-420, 2004.
- Parus, J., Kolingerová, I.: Morphing of Meshes with Attributes, Spring Conference on Computer Graphics 2004 (SCCG2004), Budměrice, Slovakia, pp. 69-78, 2004.

  Cited by:

  - Sigal, I., Hardisty, M., Whyne, C.: Mesh-morphing algorithms for specimen-specific finite element modeling, Journal of Biomechanics – 2008, Vol. 41, Issue 7, pp. 1381 – 1389, 2008.

Non-reviewed:

- Parus, J.: Morphing of Meshes, Technical Report DCSE/TR-2005-02, University of West Bohemia, 2005.

  Cited by:

  - Chen, D., Liu, J.: A Survey of Mobile 3D Animation, COMPUTER KNOWLEDGE AND TECHNOLOGY (Academic Exchange), No. 4, pp. 125-128, 2006.
- Parus, J.: Morphing 3D geometrických objektů, MSc Thesis, University of West Bohemia, 2003.

- Parus, J., Kolingerová, I.: Mesh Morphing, International Conference and Competition Student EEICT2003, Brno, Czech Republic, pp. 298-302, 2003 (best paper award).

Accepted for publication:

- Parus, J., Kolingerová, I., Málková, M.: Multimorphing: a tool for shape synthesis and analysis, Advances in Engineering Software (IF 2007: 0.529), Elsevier (doi:10.1016/j.advengsoft.2008.06.001)

Under review:

- Málková, M., Parus, J., Kolingerová, I., Beneš, B.: An intuitive polygon morphing (submitted to The Visual Computer)

Other publications:

- Ramos, F., Chover, M., Parus, J., Kolingerová, I.: Level-of-Detail Triangle Strips for Deforming Meshes, International Conference on Computer Science (ICCS 2008), Krakow, Poland, pp. 86-95, 2008.
- Varnuška, M., Parus, J., Kolingerová, I.: Simple Holes Triangulation in Surface Reconstruction, Algoritmy 2005, Podbanska, Slovakia, pp. 280-289, 2005.

  Cited by:

  - Tseng, J.: Surface Reconstruction and Simplification Based on Shape Geometric Properties, PhD Thesis, Department of Electronic Engineering, Chung Yuan Christina University, Taiwan, 2005.
  - Li, P.: Object Simplification on Static and Dynamic Models, PhD Thesis, Department of Electronic Engineering, Chung Yuan Christian University, Taiwan, 2006.
- Parus, J., Vaněček, P., Kolingerová, I.: Stripification of Meshes With Attributes, 1st Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2005), Znojmo, Czech Repulic, pp. 23-30, 2005.

Related talks:

- Parus, J.: Morphing trojrozměrných objektů, VŠB Ostrava, Czech Republic, November 2007.
- Parus, J.: Morphing povrchové reprezentace, VŠB Ostrava, Czech Republic, November 2006.
- Parus, J.: Morfování trojrozměrných těles, Faculty of Antropology, Charles University, Prague, Czech Republic, February 2006.
- Parus, J.: Deforming meshes, Centre of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, May 2006.

- Parus, J.: Isomorphic Meshes, Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, June 2006.
- Parus, J.: Deforming meshes, Comenius University in Bratislava, Faculty of Mathematics, Physics and Informatics, Slovakia, November 2005.
- Parus, J.: Alternativní reprezentace 3D objektů, Centre of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, April, 2005.
- Parus, J.: Morphing povrchových modelů, Centre of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, March, 2005.
- Parus, J.: Morphing povrchových modelů, VŠB Ostrava, Czech Republic, December 2004.
- Parus, J.: Morphing of Meshes, Technical University of Graz, Austria, September 2004.
- Parus, J.: Morphing trojrozměrných objektů, Centre of Computer Graphics and Data Visualization, University of West Bohemia, Pilsen, Czech Republic, May 2004.

Grants participation:

- The Use of Strip Representation for Morphing (FRVŠ 1509/2005/G1), project leader,
- Surface Reconstruction from Point Clouds (FRVŠ 1349/2004/G1),
- Center of Computer Graphics - National Network of Fundamental Research Centers (LC06008), Ministry of Education, Youth and Sports, Czech Republic,
- Computer Graphics and Visualization with C# (2003-187), Microsoft Research Ltd., UK,
- Information technologies - sub-project: Computer Graphics and Data Visualization (MSMT 235200005), Ministry of Education, Youth and Sports, Czech Republic.

Stays abroad:

- 06/2006 – Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia,
- 11/2005 – Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia,
- 12/2005 – Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava, Bratislava, Slovakia,
- 9/2004 – Technical University of Graz, Graz, Austria,
- 02/2002 – 06/2002 – E.T.S.I Informática, University of Granada, Spain.

Teaching activities:

- Fundamentals of Computer Graphics, Department of Computer Science and Engineering, University of West Bohemia, seminars,
- Fundamentals of Information Technologies, Department of Computer Science and Engineering, University of West Bohemia, seminars,
- Computer Aided Creativity, Department of Computer Science and Engineering, University of West Bohemia, seminars.