Library Declaration Form

University of Otago Library

Author's full name and year of birth: Alexis Angelidis,
(for cataloguing purposes) 11 October 1978

Title of thesis: Shape Modeling by Swept Space Deformation

Degree: Doctor of Philosophy

Department: Department of Computer Science

Permanent Address:

I agree that this thesis may be consulted for research and study purposes and that reasonable quotation may be made from it, provided that proper acknowledgement of its use is made.

I consent to this thesis being copied in part or in whole for

i) a library

ii) an individual

at the discretion of the University of Otago.

Signature:

Date:

Shape Modeling by Swept Space Deformation

Alexis Angelidis

a thesis submitted for the degree of

Doctor of Philosophy

at the University of Otago, Dunedin,

New Zealand.

Abstract

In Computer Graphics, in the context of shape modeling on a computer, a common characteristic of popular techniques is the possibility for the artist to operate on a shape by modifying directly the shape's mathematical description. But with the constant increase of computing power, it has become increasingly realistic and effective to insert interfaces between the artist and the mathematics describing the shape. While in the future, shape descriptions are likely to be replaced with new ones, this should not affect the development of new and existing shape interfaces. Space deformation is a family of techniques that permits describing an interface independently from the description. Our thesis is that while space deformation techniques are used for solving a wide range of problems in Computer Graphics, they are missing a framework for the specific task of interactive shape modeling. We propose such a framework called sweepers, together with a set of related techniques for shape modeling. In sweepers, we define simultaneous-tools deformation, volume-preserving deformation, topology-changing deformation and animated deformation. Our swept-fluid technique introduces the idea that a deformation can be described as a fluid. In fact, the sweepers framework is not restrained to shape modeling and is also used to define a new fluid animation technique. Since the motion of a fluid can be considered locally as rigid, we define a formalism for handling conveniently rigid transformations. To display shapes, we propose a mesh update algorithm, a point-based shape description and a discrete implicit surface, and we have performed preliminary tests with inverse-raytracing. Finally, our technique called spherical-springs can be used to attach a texture to our shapes.

Contents

Notation **3**

1 Introduction **7**
- 1.1 Context . . . 7
- 1.2 Motivation . . . 8
- 1.3 Limit of scope . . . 9
- 1.4 Contributions . . . 9
- 1.5 Thesis overview . . . 10

2 State of the Art **11**
- 2.1 Axial space deformations . . . 12
- 2.2 Lattice-based space deformations . . . 19
- 2.3 Tool-based space deformations . . . 25
- 2.4 Conclusion . . . 32

3 Sweepers and Related Techniques **35**
- 3.1 Definitions . . . 36
- 3.2 Solution to foldover . . . 43
- 3.3 Fast expressions for simple sweepers . . . 45
- 3.4 Tool influence . . . 47
- 3.5 Changing topology . . . 51
- 3.6 Animation . . . 58
- 3.7 Conclusion . . . 65

4 Volume Preserving Modeling **67**
- 4.1 Swirling-sweepers . . . 68
- 4.2 Swept-fluid . . . 76
- 4.3 Conclusion . . . 82

5 Rendering **85**
- 5.1 Updated mesh . . . 85
- 5.2 Point-based representation . . . 89
- 5.3 Discrete implicit . . . 93
- 5.4 Ray-tracing . . . 98
- 5.5 Texturing with spherical springs . . . 102

6 Simulation of Smoke based on Vortex Filament Primitives 111
6.1 The physics of vorticity and filaments 113
6.2 Our choice of representation and solver 115
6.3 Practical approximations and extensions 118
6.4 Our primitives of vorticity 120
6.5 Smoke particles 124
6.6 Interactive design of flows 126
6.7 Results 127
6.8 Conclusion 130

7 Hexanions 133
7.1 Motivation 133
7.2 Overview 134
7.3 Algebraic properties 134
7.4 Hexanion subgroups 136
7.5 Closed-form exponential 138
7.6 Hexanion product 139
7.7 Properties 140
7.8 Matrix to hexanion conversion: 140
7.9 Conclusion 142

Conclusion 143

A Affine Transformations 145
A.1 Translation 145
A.2 Scale 146
A.3 Non-uniform scale 146
A.4 Rotation 147

B Constant-Volume Swirl 149
B.1 Constant volume basic swirl 149
B.2 Swirl angle 151

C Monotonic Mostly-C^1 Interpolation 153
C.1 Monotonic Mostly-C^1 Interpolation 153
C.2 Discretization 154

References 155

Acknowledgment

Special thanks to Sui-Ling Ming-Wong for being such a fantastic adviser and listener, she made everything possible. Thanks to my co-adviser Brendan McCane for his cheerful advice and curiosity of mathematics. Thanks to my adviser Geoff Wyvill for his general culture of computer graphics, and for letting me lots of freedom. Thanks to my external adviser Marie-Paule Cani for her always immediate feedbacks. Thanks to Fabrice Neyret for sharing his knowledge and intuition of fluid dynamics. Thanks to the examiners, especially Gerrard Liddell for his insightful comments. Thanks to the System Administrators of the University of Otago for being responsive, including Barry, Cathy and Dave. Apologies to my family and friends for being too focused on my work. Thanks to little Alex for training regularly my reflexes at chess. Thanks to Sylvain Lefèbvre, a very talented programmer with whom I did my first project in computer graphics. Thanks to Jérome Darbon for showing up in New Zealand, I hope we'll have the opportunity to do some project some day. Thanks to everyone else who hanged around the Graphics Lab during those days: Geoff & Yerin, Phil, Natalie, Annabel, Matthew, Billy, Andreas, Mike, Mads, Jayson & Emily, Scott, Tami & Graham, Andrew, Chris, Nathan, Kevin, Dave & Irene and everyone else. Thanks to the DEA IVR 2001 friends and classmates, with whom studying graphics in Grenoble was a great experience: Sylvain, Thomas, the other Sylvain, Philippe, Stéphane, Yohan, the other Stéphane, Fani, Thomas…

Notation

We begin by describing the notation we use. We advise the reader to skip this section, and refer to it while reading the manuscript in cases where an equation is unclear. The purpose of the notation adopted is to deal exclusively with problems in three-dimensional spaces, in a coordinate system described by an origin and three clockwise-oriented unit orthogonal vectors $(\mathrm{o}, \vec{\mathrm{e}}_x, \vec{\mathrm{e}}_y, \vec{\mathrm{e}}_z)$.

Scalars are denoted in italic characters by:

$$s \in \mathbb{R} \tag{1}$$

Spatial components along $\vec{\mathrm{e}}_x$, $\vec{\mathrm{e}}_y$, $\vec{\mathrm{e}}_z$ are denoted with x y and z subscripts. A *point* is a position in space, with components represented in a column array:

$$\mathrm{p} = \mathrm{o} + p_x\vec{\mathrm{e}}_x + p_y\vec{\mathrm{e}}_y + p_z\vec{\mathrm{e}}_z = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = (p_x, p_y, p_z)^\top \in \mathbb{R}^3 \tag{2}$$

We may refer to the three coordinates of point p with a dummy subscript, i.e. p_i where $i \in \{x, y, z\}$. A *vector* is a direction in space with a magnitude. A vector is denoted with an arrow, with components represented in a column array:

$$\vec{\mathrm{v}} = v_x\vec{\mathrm{e}}_x + v_y\vec{\mathrm{e}}_y + v_z\vec{\mathrm{e}}_z = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = (v_x, v_y, v_z)^\top \in \vec{\mathbb{R}}^3 \tag{3}$$

We may refer to the three coordinates of vector $\vec{\mathrm{v}}$ with a dummy subscript, i.e. v_i where $i \in \{x, y, z\}$. A vector may be obtained by subtracting two points. We denote the dot product between two vectors $\vec{\mathrm{a}}$ and $\vec{\mathrm{b}}$ by:

$$\vec{\mathrm{a}} \cdot \vec{\mathrm{b}} = a_x b_x + a_y b_y + a_z b_z \in \mathbb{R} \tag{4}$$

We denote with superscript 2 the dot product of a vector with itself:

$$\vec{\mathrm{v}}^2 = \vec{\mathrm{v}} \cdot \vec{\mathrm{v}} \in \mathbb{R}^+ \tag{5}$$

We denote with double pipes $\|.\|$ the magnitude of a vector, i.e. its Euclidean length:

$$\|\vec{\mathrm{v}}\| = \sqrt{\vec{\mathrm{v}}^2} \in \mathbb{R}^+ \tag{6}$$

We denote the cross product between two vectors by:

$$\vec{\mathrm{a}} \times \vec{\mathrm{b}} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix} \in \vec{\mathbb{R}}^3 \tag{7}$$

Although dot and cross products involving points do not define genuine dot and cross products, they may appear in our equations, for the sake of algebraic simplicity. Note that in those cases, the operation still corresponds to a real dot or cross product once factored. Thus for example, we tolerate the cross products on the right hand side of the following expression (if $\mathrm{q} = \mathrm{o}$, the last term would vanish):

$$\vec{\mathrm{v}} \times (\mathrm{p} - \mathrm{q}) = \vec{\mathrm{v}} \times \mathrm{p} - \vec{\mathrm{v}} \times \mathrm{q} \tag{8}$$

Thus by convention, the cross product of two points is a vector.
We denote ∇ the derivative vector with respect to spatial coordinates along $\vec{\mathrm{e}}_x$, $\vec{\mathrm{e}}_y$ and $\vec{\mathrm{e}}_z$:

$$\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} = \left(\tfrac{\partial}{\partial x}, \tfrac{\partial}{\partial y}, \tfrac{\partial}{\partial z} \right)^\top \tag{9}$$

We handle ∇ like a non-commutative vector: its dot product with a vector field $\nabla \cdot \vec{\mathrm{v}}$ is called the divergence, its cross product with a vector field $\nabla \times \vec{\mathrm{v}}$ is called the curl, its product with a scalar field $\nabla \cdot \phi$ is called the gradient and the divergence of the gradient $\nabla^2 \phi$ is called the Laplacian. We denote *matrices* with capital letters, and matrix components with zero based indices. The components of a 3×3 matrix are denoted:

$$M = \begin{pmatrix} m_{xx} & m_{xy} & m_{xz} \\ m_{yx} & m_{yy} & m_{yz} \\ m_{zx} & m_{zy} & m_{zz} \end{pmatrix} \in \mathbb{R}^{3 \times 3} \tag{10}$$

We may refer to the component of a matrix M with dummy subscripts, i.e. m_{ij} where $i, j \in \{x, y, z\}$. We denote with superscript $\top$ the *transpose* of a matrix, which swaps element m_{ij} and m_{ji}:

$$\text{if } L = M^\top \text{ then } l_{ji} = m_{ij} \tag{11}$$

We use $\cdot$ to denote the matrix product $M_0 \cdot M_1$ as well as the matrix-vector product $M \cdot \vec{\mathrm{v}}$ or matrix-point product $M \cdot \mathrm{p}$. The product is defined by componentwise multiplication of the rows of the left operand with the columns of the right operand. We consider vectors as 3×1 matrices, thus the dot product can also be written:

$$\vec{\mathrm{a}} \cdot \vec{\mathrm{b}} = \vec{\mathrm{a}}^\top \cdot \vec{\mathrm{b}} \in \mathbb{R} \tag{12}$$

Although the left hand side of the following looks similar to the above, it is a matrix:

$$\vec{\mathrm{a}} \cdot \vec{\mathrm{b}}^\top = \begin{pmatrix} a_x b_x & a_x b_y & a_x b_z \\ a_y b_x & a_y b_y & a_y b_z \\ a_z b_x & a_z b_y & a_z b_z \end{pmatrix} \in \mathbb{R}^{3 \times 3} \tag{13}$$

We require 4×4 matrices as well in order to represent the translational part of transformations. The components of a 4×4 matrix are denoted:

$$N = \begin{pmatrix} n_{xx} & n_{xy} & n_{xz} & n_{xh} \\ n_{yx} & n_{yy} & n_{yz} & n_{yh} \\ n_{zx} & n_{zy} & n_{zz} & n_{zh} \\ n_{hx} & n_{hy} & n_{hz} & n_{hh} \end{pmatrix} \in \mathbb{R}^{4\times 4} \tag{14}$$

Since we deal exclusively with three-dimensional spaces, we make assumptions to simplify notation involving 3D coordinates and the 4th homogeneous coordinate. We will only consider two kinds of 4×4 matrices: $\{n_{hx} = n_{hy} = n_{hz} = n_{hh} = 0\}$ or $\{n_{hx} = n_{hy} = n_{hz} = 0, n_{hh} = 1\}$. We may refer to the component of a matrix N with dummy subscripts, i.e. n_{ij} where $i, j \in \{x, y, z, h\}$. To define operations between 3×3 and 4×4 matrices, we assume that 3×3 matrices have a fourth homogeneous row and column filled with 0; thus for instance for matrix M given above we assume $m_{ih} = m_{hj} = 0$. This makes non-ambiguous the writing of the matrix sum $M + N$ or matrix product $N \cdot M$.

To define the product of a 4×4 matrix with a point, we assume that points have a fourth homogeneous dummy coordinate always equal to 1:

$$N \cdot \mathrm{p} = \begin{pmatrix} n_{xx}p_x + n_{xy}p_y + n_{xz}p_z + n_{xh} \\ n_{yx}p_x + n_{yy}p_y + n_{yz}p_z + n_{yh} \\ n_{zx}p_x + n_{zy}p_y + n_{zz}p_z + n_{zh} \end{pmatrix} \in \mathbb{R}^3 \cup \vec{\mathbb{R}}^3 \tag{15}$$

If all four components of the fourth row of N are all equal to 0 then the result of the product $N \cdot \mathrm{p}$ is a vector $N \cdot \mathrm{p} = \vec{\mathrm{v}}$, otherwise it is a point $N \cdot \mathrm{p} = \mathrm{q}$. To define the product of a matrix with a vector, we assume that vectors have a fourth homogeneous dummy coordinate always equal to 0 symbolized by the arrow:

$$N \cdot \vec{\mathrm{v}} = \begin{pmatrix} n_{xx}v_x + n_{xy}v_y + n_{xz}v_z \\ n_{yx}v_x + n_{yy}v_y + n_{yz}v_z \\ n_{zx}v_x + n_{zy}v_y + n_{zz}v_z \end{pmatrix} \in \vec{\mathbb{R}}^3 \tag{16}$$

The exponential of a matrix is defined as follows:

$$\exp M = \mathrm{I} + M + \frac{1}{2}M^2 + \frac{1}{6}M^3 \cdots = \sum_{k=0}^{\infty} \frac{M^k}{k!} \tag{17}$$

The logarithm is defined as an inverse of the exponential, as follows:

$$\log(\mathrm{I} - M) = -M - \frac{1}{2}M^2 - \frac{1}{3}M^3 \cdots = -\sum_{k=1}^{\infty} \frac{M^k}{k} \tag{18}$$

A matrix is raised to the power of a real number exponent as follows:

$$M^s = \exp(s \log M) \tag{19}$$

A *quaternion* is a 4-dimensional element of quaternion space H, denoted in bold font and whose components are represented in a column array:

$$\mathbf{q} = \begin{pmatrix} q_s \\ q_x \\ q_y \\ q_z \end{pmatrix} \in \mathrm{H} \tag{20}$$

For convenience, the components of a quaternion are denoted with a scalar $s \in \mathbb{R}$ and a vector $\vec{\mathrm{v}} \in \vec{\mathbb{R}}^3$ as follows: $\mathbf{q} = (s, \vec{\mathrm{v}})^\top$. Let us define two quaternions $\mathbf{q} = (s, \vec{\mathrm{v}})^\top$ and $\mathbf{q'} = (s', \vec{\mathrm{v}}')^\top$. The *addition* of two quaternions is obtained by adding the corresponding components. We denote by $*$ the quaternion *product*, defined as follows:

$$\mathbf{q} * \mathbf{q}' = (ss' - \vec{\mathrm{v}} \cdot \vec{\mathrm{v}}', s\vec{\mathrm{v}}' + s'\vec{\mathrm{v}} + \vec{\mathrm{v}} \times \vec{\mathrm{v}}')^\top \tag{21}$$

The most remarkable elements are the unit quaternions H_1, which can be written $\mathbf{q} = (\cos(\theta), \sin(\theta)\vec{\mathrm{n}})^\top$, and whose norm $s^2 + \vec{\mathrm{v}}^2$ is equal to 1. The set of unit quaternions constitute a unit sphere in a four-dimensional space. They provide a convenient way to handle pure rotations. The *logarithm* of quaternion $\mathbf{q}$ is a 3D element $\mathrm{p} = \theta\vec{\mathrm{n}}$, and the *exponential* of p is $\mathbf{q}$. From the above definitions arise the *exponentiation* of $\mathbf{q} \in H_1$:

$$\mathbf{q}^t = \exp^{t \log \mathbf{q}} \tag{22}$$

A 3D point $\mathrm{p} = (p_x, p_y, p_z)^\top$ can be represented with a *point-quaternion*: $\mathbf{p} = (0, \mathrm{p})^\top$. With points, unit quaternion $\mathbf{q}$ does not only represent an element on the unit 4D sphere, but also a rotation of angle $\theta/2$ around axis $\vec{\mathrm{v}}$. The rotation of $\mathbf{p}$ with $\mathbf{q}$ is:

$$\mathbf{q} * \mathbf{p} * \bar{\mathbf{q}} \tag{23}$$

where $\bar{\mathbf{q}} = (s, -\vec{\mathrm{v}})^\top$ is the *conjugate* of $\mathbf{q}$.
We denote by $\circ$ the composition of functions. Let us consider for instance functions $f_i : \mathbb{R}^3 \mapsto \mathbb{R}^3$:

$$f_j(f_i(\mathrm{p})) = (f_j \circ f_i)(\mathrm{p}) \tag{24}$$

The operator Ω expresses the finite repeated composition of functions:

$$\underset{i=1}{\overset{n}{\Omega}} f_i(\mathrm{p}) = (f_n \circ \cdots \circ f_1 \circ f_0)(\mathrm{p}) \tag{25}$$

CHAPTER 1

Introduction

Shape modeling in Computer Graphics is the use of computers to assist an artist to create a numerical shape. Shape modeling may find applications in many disciplines, such as medicine, art, cinema or games. In the design process, the artist brings a shape through stages by means of a series of inputs, until the targeted shape is obtained.

Unfortunately the process of shape modeling is ill-defined: an initial shape can take an infinite number of paths to produce exactly the same target shape. In practice, shape modeling software provides the artist with a set of shape operations, and leaves the choice of the path to the artist. It goes without saying that the behavior of the shape under the proposed operations determines the efficiency of the modeling software.

Although existing techniques provide the artists with a rich set of shape behaviors, this set is far from complete. The paths the artist can choose are limited. This thesis attempts to define new behaviors, unified in a single framework called *sweepers*. We are tempted to say that sweepers are intuitive, easy and natural to use for modeling shapes, although this is very subjective. Rather we claim that sweepers allow the artist to take new *useful* and *predictable* paths toward the target shape. The behaviors we propose can prevent foldovers, preserve volume or change the shape's topology.

Sweepers is a framework that belongs to a family of techniques called space deformation. This family of techniques has the great advantage of separating the shape's behavior from the shape's mathematical representation.

1.1 Context

On a computing device, an artist models a shape by means of a series of inputs. A shape modeling technique can be characterized by the *shape's behavior* in response to these inputs. The shape's behavior must be at least predictable and interactive.

On the other hand, the shape's *representation* is a piece of mathematics that describes which points of $\mathbb{R}^3$ belong to the shape's surface and/or volume. In most popular techniques, the shape's behavior and the shape's representations are symbiotic. Some of these popular techniques are for example meshes, subdivision surfaces, NURBS, and to some extent implicit surfaces. It can be observed for instance that for a subdivision surface, a control point serves both as a tool for deforming the shape and a part of the mathematics defining the surface.

This symbiosis between a shape's behavior and its representation is an advantage when it comes to time efficiency. However some operations would be very tedious to perform by the artist if he/she were restricted to use the built-in behavior. With the increasing performance of computing devices, it is realistic to handle higher level operations with a computer. The necessity of a higher level of interaction has long been recognized. In 1977, R. Parent defined an operation for meshes called a *warp*, that acts on a set of vertices [Par77]. Later, A. Barr discovered that by deforming $\mathbb{R}^3$, a shape embedded in that space would be subject to that deformation [Bar84]. His operations are the first published of a very large family of techniques, referred to in the literature as *space deformation*[1]. The advantage of a space deformation technique is that it entirely describes a shape's behavior, independently of the shape's representation. This makes space deformation a versatile way of specifying new shape behaviors for modeling. We present an overview of existing space deformation techniques in Chapter 2.

1.2 Motivation

The literature on space deformation techniques is vast. The majority of existing techniques are specialized in achieving particular behaviors. In our work, instead of proposing a list of specialized or unrelated space deformation techniques, we define a unifying framework called *sweepers*, in which we propose several techniques. If complexity and simplicity were measurable quantities, we believe that the ratio between the complexity of behavior and the simplicity of input is an important factor. In any non-virtual handicraft, gesture is the basis of creation. By analogy with this, we focus our work on defining deformations that take as direct input one or more gestures from the artist. A gesture may be defined for instance using a mouse, or by means of any other user interface. We use these gestures to achieve space deformations that satisfy rules automatically, for example preventing surface foldovers or preserving volume.

For software robustness reasons or for compatibility with post-processing operations on the shape such as texturing, rendering or animation, it is desirable for a shape to exhibit some coherence. Objectively, coherency is described by a set of criteria that a shape guarantees to satisfy. The set of acceptable criteria that describe coherency is not strict, and depends on one's standard. In the context of shape representation, we believe that the following is a reasonable criterion:

A shape defines points inside and outside, separated by a boundary, without ambiguity.

Although such coherency need only to be satisfied by the end-product of the modeling process, enforcing it during the modeling process gives to the artist some expectations about his/hers shape's geometry, at any time. We also believe that a technique capable

---

[1] The term Free-form has also been used, but we will not use it since it also includes operations of another kind.

of producing incoherent in-between shapes would probably not be specific enough as a dedicated tool for shape modeling. Thus we believe that the following is a reasonable criterion in the context of an interactive modeling technique:

A point inside a shape remains inside that shape (and similarly for a point outside)[2].

Although space deformation techniques perform versatile operations independently from the shape's representation, it is easily possible to maintain this coherency, with the sole condition that the deformation be reversible, i.e. foldover-free. The foldover-free property is also necessary for an inverse deformation to exist, which is useful for defining an undo operation, or rendering the deformed shape with inverse-raytracing (Chapter 5).

Finally, in a virtual modeling context, there is no material: no wax, clay, wood or marble. A challenge for computer graphics is to provide a virtual tool that convinces the artist that there is material. To complete the illusion, a shape must behave in accordance with a suitable modeling metaphor. Volume is one of the most important factors influencing the manner in which an artist models with real materials. We introduce techniques that preserve volume, and help the artist believe he is interacting with virtual material. Also, modeling while preserving the available amount of material produces shapes with a style that other virtual modeling methods can only achieve with more effort.

1.3 Limit of scope

A modeling technique is intuitive when it behaves in an expected way. Depending on individual experience with modeling, artists will have different expectations. It is difficult to claim objectively that one method is more intuitive than another without a thorough psychological study. Proving the intuitiveness of sweepers is beyond the scope of this thesis, and our judgment is based only on our experience.

1.4 Contributions

We organize our contributions in nine points. The first five are techniques strongly related to our framework called sweepers. We have also developed techniques that can be applied in conjunction with more general space deformation techniques.

- We have developed sweepers, a framework for modeling by deformation. A sweeper is a geometric tool together with a motion path; its effect is to move space underneath the tool along the path, smoothly. We describe a normalized blending formula, allowing us to use multiple sweepers simultaneously, and without artifacts. We provide efficient formulations for a single tool following a simple path. We propose a solution to prevent the surface from self-intersecting, which is part of the definition of sweepers.

---

[2] This means that a point is tagged explicitly or implicitly with the information of being inside or outside the shape.

- We have developed a technique called animated-sweepers, useful for animating the modeling of a shape. It allows an artist to edit the keyframes of the deformation, cameras and lighting until he/she is satisfied with the result.
- We have developed sweepers that change the topology of space, and allow controlled changes of topology of the shape.
- We propose techniques for describing shapes dedicated to modeling with sweepers. We have developed a mesh refinement and decimation algorithm that takes advantage of our swept deformations. We also have applied sweepers to a point-based shape description and to a discrete implicit surface, and we have performed preliminary tests with inverse-raytracing.
- We have developed a technique called swirling-sweepers, for modeling while preserving the shape's volume. We have discovered that in conjunction with other space deformation [Dec96], a rich set of tools can be used to define volume-preserving swept deformations.
- We have developed a technique called swept-fluid, which stands out of the sweeper framework. This technique uses the Navier-Stokes equations to define a deformation, and also preserves volume.
- We have developed a technique called spherical-springs, useful for spreading evenly the texture coordinates of a shape homeomorphic to a sphere.
- We have derived from our shape modeling framework a technique for animating fluids.
- We have defined a formalism for handling rigid transformation.

1.5 Thesis overview

The first chapter reviews the background of space deformation techniques. The fundamental principles of sweepers are then described in Chapter 3, together with efficiency improvements, techniques for changing a shape's topology and techniques for animation. Volume preserving matters are then described in Chapter 4. We have discovered a case of volume preserving sweepers which does not require us to compute the volume of the shape: *swirling-sweepers*. Preserving the amount of material has led us to develop a physically-based approach that achieves comparable results. The repeated application of space deformations produces distortions of the shape's surface. This causes a few complications for displaying the shape, and we have developed several solutions: a mesh refinement and decimation algorithm, a point-based representation, an inverse-raytracing technique, a discrete implicit surface technique, and a relaxation technique for spreading more evenly the texture coordinates of shapes homeomorphic to a sphere. These rendering-related topics are described in Chapter 5. We have applied our formalism to the more general topic of fluid dynamics, which we present in Chapter 6. Our last chapter presents a formalism for handling rigid transformations, which we foresee as an important part of the theory for developing future space deformation techniques.

Chapter 2

State of the Art

Space deformation provides a formalism to specify any modeling operation by successively deforming the space in which an initial shape $S(k_0)$ is embedded. A deformed shape is given by the *modeling equation*, where k_i parameterizes the evolution of the shape:

$$S(k_n) = \left\{ \underset{i=0}{\overset{n-1}{\Omega}} f_{k_i \mapsto k_{i+1}}(\mathrm{p}) \mid \mathrm{p} \in S(k_0) \right\} \tag{2.1}$$

$$\text{where } \underset{i=0}{\overset{n-1}{\Omega}} f_{k_i \mapsto k_{i+1}}(\mathrm{p}) = f_{k_{n-1} \mapsto k_n} \circ \cdots \circ f_{k_0 \mapsto k_1}(\mathrm{p}) \tag{2.2}$$

The operator Ω expresses the finite repeated composition of functions[1]. The series of functions $f_{k_i \mapsto k_{i+1}} : \mathbb{R}^3 \to \mathbb{R}^3$ constitute the shape's history of deformation. Each function deforms a point p of the shape $S(k_i)$ into a point of the shape $S(k_{i+1})$. The common feature of space deformation techniques is that they apply to the space in which the model is embedded and therefore can specify shape behavior independently of the shape. This means that they define a deformation of the portion of space where there is no surface, if required. Note that the operations do not commute under function composition, $\circ$. Note also that the definition of an individual operation $f_{k_i \mapsto k_{i+1}}$ is independent of the shape's history.

This chapter reviews existing space deformation techniques, organized in three groups: axial deformations, lattice-based deformations and tool-based deformations. For the sake of clarity, we present existing space deformations aligned with the axes $\vec{\mathrm{e}}_x$, $\vec{\mathrm{e}}_y$ and $\vec{\mathrm{e}}_z$ and within the unit cube $[0, 1]^3$, whenever possible. But a mere change of coordinates enables the artist to place the deformation anywhere in space. Note that affine transformations are the simplest case of space deformations. They are described in Appendix A.

---

[1] $(f \circ g)(\mathrm{p}) = f(g(\mathrm{p}))$

2.1 Axial space deformations

Axial space deformations are a subset of space deformations whose control-points are geometrically connected along a curve. The curve may be initially straight or bent. To compare existing deformation techniques from the same point of view, we use $\vec{e}_z$ as the common axis of deformation, which leads to slight reformulation in a few cases.

2.1.1 Global and local deformations of solid primitives

A. Barr defines space tapering, twisting and bending via matrices whose components are functions of one space coordinate [Bar84]. We denote $(x, y, z)^\top$ the coordinates of a point. We show in Figures 2.1, 2.2, and 2.3 the effects of these operations, and we give their formula in the form of 4 × 4 homogeneous matrices to be applied to the coordinates of every point in space to be deformed.

Tapering operation: The function r is monotonic in an interval, and is constant outside that interval.

$$\begin{pmatrix} r(z) & 0 & 0 & 0 \\ 0 & r(z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

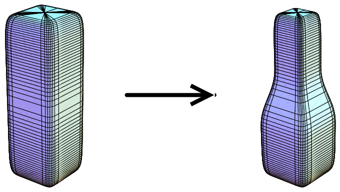


Figure 2.1: *Taper deformation of a super-ellipsoid shape. A description of the shape can be found in [Gla89].*

Twisting operation: The function θ is monotonic in an interval, and is constant outside that interval.

$$\begin{pmatrix} \cos(\theta(z)) & -\sin(\theta(z)) & 0 & 0 \\ \sin(\theta(z)) & \cos(\theta(z)) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

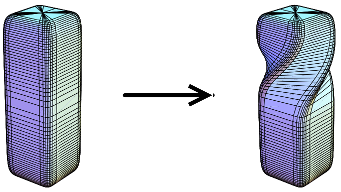


Figure 2.2: *Twist deformation of a super-ellipsoid.*

Bending operation: This operation bends space along the axis y, in the $0 < z$ half-space. The desired radius of curvature is specified with ρ. The angle corresponding to ρ is $\theta = \hat{z}/\rho$. The value of $\hat{z}$ is the value of z, clamped in the interval $[0, z_{\max}]$.

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & \rho - \rho\cos\theta - \hat{z}\sin\theta \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & \rho\sin\theta - \hat{z}\cos\theta \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

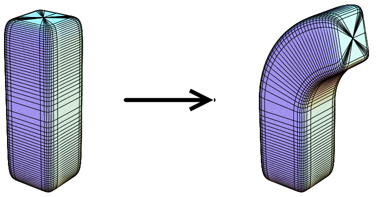


Figure 2.3: *Bend deformation of a super-ellipsoid.*

A. Barr observes that rendering the deformed shape with rays of light is equivalent to rendering the undeformed shape with curves of light. The curves of light are obtained by applying the inverse of the deformation to the rays. Because the deformation he proposes are not local, the portions of the rays to deform can be quite large.

2.1.2 A generic implementation of axial procedural deformation techniques

C. Blanc extends A. Barr's work to mold, shear and pinch deformations [Bla94]. Her transformations use a function of one or two components. She calls this function the *shape function*. Examples are shown in Figures 2.4, 2.5, and 2.6.

$$\begin{pmatrix} r(z) & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

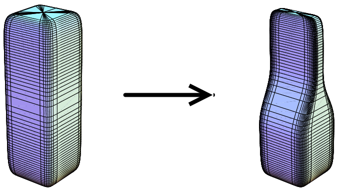


Figure 2.4: *Pinch deformation of a super-ellipsoid.*

$$\begin{pmatrix} r(\tan^{-1}(x,y)) & 0 & 0 & 0 \\ 0 & r(\tan^{-1}(x,y)) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

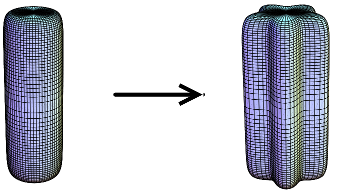


Figure 2.5: *Mold deformation of a super-ellipsoid.*

$$\begin{pmatrix} 1 & 0 & 0 & s(z) \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

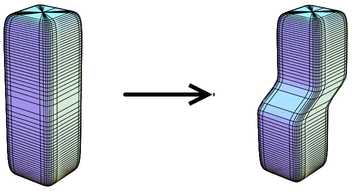
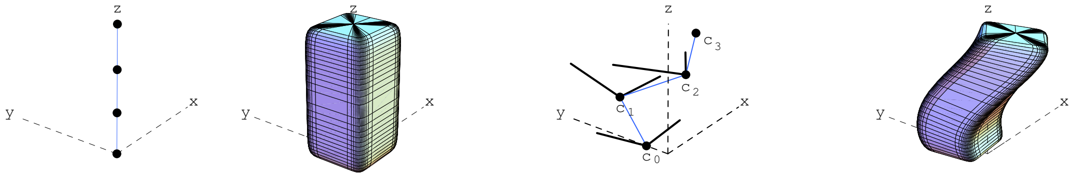


Figure 2.6: *Shear deformation of a super-ellipsoid.*

2.1.3 A generalized de Casteljau approach to 3D free-form deformation

Y.K. Chang and A.P. Rockwood propose a polynomial deformation that efficiently achieves "Barr"-like deformations and more [CR94], using a Bézier curve with coordinate sets defined along $\vec{e}_z$ at the curve's control knots $(z_0, z_1 \dots, z_n) \in [0,1]^{n+1}$. A reference straight segment, $z \in [0,1]$, is deformed by specification of coordinate sets $(\mathrm{c}_i, \vec{\mathrm{u}}_i, \vec{\mathrm{v}}_i, \vec{\mathrm{w}}_i)$ along that segment. The shape follows the deformation of the segment, as shown in Figure 2.7.



straight axis initial shape control points and handles deformed shape

Figure 2.7: *Example of the deformation of Y.K. Chang and A.P.Rockwood applied to a super-ellipsoid. There is no need to define a pair of handles for the end control point.*

To compute the image q of a point p of the original shape, the matrix transforming a point to a local coordinate set is needed:

$$M_i = \begin{pmatrix} u_{i,x} & v_{i,x} & w_{i,x} & c_{i,x} \\ u_{i,y} & v_{i,y} & w_{i,y} & c_{i,y} \\ u_{i,z} & v_{i,z} & w_{i,z} & c_{i,z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.3}$$

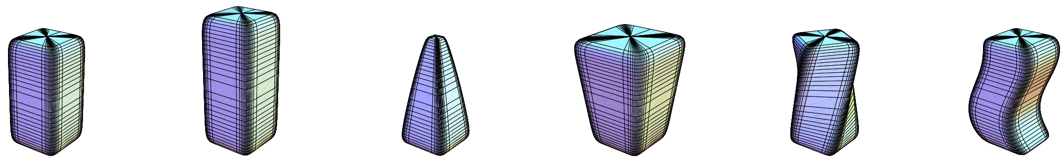
where $\vec{\mathrm{w}}_i = \mathrm{c}_{i+1} - \mathrm{c}_i$, and $\vec{\mathrm{u}}_i$, $\vec{\mathrm{v}}_i$ are the handles.

Using this matrix, the deformation of a point is obtained recursively with the de Casteljau algorithm for evaluating a Bézier curve:

$$f_i^j(\mathrm{p}) = (1 - \mathrm{p}_z) f_i^{j-1}(\mathrm{p}) + \mathrm{p}_z f_{i+1}^{j-1}(\mathrm{p}) \tag{2.4}$$

$$\text{where } f_i^0(\mathrm{p}) = M_i \cdot \mathrm{p}$$

The original generalized de Casteljau algorithm presented by Y.K. Chang and A.P. Rockwood is a recursion on affine transformations rather than on points. As we show in Figure 2.8, this method is capable of performing "Barr"-like deformations and more. Note that away from the control axis, the deformation may not be bijective. The same remark can be made about Sections 2.1.1 and 2.1.2.



initial stretch taper swell twist bend

Figure 2.8: *Deformation of a super-ellipsoid.*

2.1.4 Axial deformation

The limitation of the methods presented so far is the initial rectilinear axis. If the shape is initially excessively bent, the manipulation of an initially straight control axis will not induce a predictable behavior of the shape. F. Lazarus et al. develop an extension of axial-based deformations using an initially curved axis [LCJ94]. Let us define a parametric curve $\mathrm{C}(u)$. A point p in space is attached to local coordinates along the curve. The origin of this local coordinate system is the closest point to p on the curve, and the axes are those of an extended Frenet frame that discards vanishing points [Blo90]. To find the closest point to p on curves, B. Crespin proposes an efficient algorithm based on subdivision [Cre99]. The axes are computed by propagating along the curve a frame defined at one extremity of the curve. The axes consist of three vectors: a tangent $\vec{\tau}(u)$, a normal $\vec{\mathrm{n}}(u)$ and a binormal $\vec{\mathrm{b}}(u)$. The propagated frame is computed as follows:

- the unit tangent at the origin is given by the equation of the curve: $\vec{\tau}(0) = \frac{\mathrm{dC}(0)}{\mathrm{d}u} / \| \frac{\mathrm{dC}(0)}{\mathrm{d}u} \|$.
- the normal and binormal are given by the Frenet frame, or can be any pair of unit vectors such that the initial frame is orthonormal.

To compute the next frame, a rotation matrix is needed. The purpose of this matrix is to minimize torsion along the curve. Although we provide a formula for a rotation matrix in Appendix A, numerous constructions of the rotation matrix justify a less expensive formulation:

$$R = \begin{pmatrix} a_{xx}+c & a_{xy}+b_z & a_{zx}-b_y \\ a_{xy}-b_z & a_{yy}+c & a_{yz}+b_x \\ a_{zx}+b_y & a_{yz}-b_x & a_{zz}+c \end{pmatrix} \tag{2.5}$$

$$\text{where} \quad \begin{array}{ll} (a_x, a_y, a_z)^\top = \frac{\vec{\tau}(u_i) \times \vec{\tau}(u_{i+1})}{\|\vec{\tau}(u_i) \times \vec{\tau}(u_{i+1})\|} & \alpha = 1 - c \\ c = \vec{\tau}(u_i) \cdot \vec{\tau}(u_{i+1}) & \beta = \sqrt{1 - c^2} \end{array} \tag{2.6}$$

$$\begin{array}{lll} a_{xx} = \alpha a_x^2 & a_{xy} = \alpha a_x a_y & b_x = \beta a_x \\ a_{yy} = \alpha a_y^2 & a_{yz} = \alpha a_y a_z & b_y = \beta a_y \\ a_{zz} = \alpha a_z^2 & a_{zx} = \alpha a_z a_x & b_z = \beta a_z \end{array} \tag{2.7}$$

Given a frame at parameter u_i, the next axes of a frame at u_{i+1} are computed as follows:

- the tangent is defined by the equation of the curve: $\vec{\tau}(u_{i+1}) = \frac{\mathrm{dC}(u_{i+1})}{\mathrm{d}u} / \| \frac{\mathrm{dC}(u_{i+1})}{\mathrm{d}u} \|$.
- the normal is given by the rotation of the previous normal: $\vec{\mathrm{n}}(u_{i+1}) = R \cdot \vec{\mathrm{n}}(u_i)$.
- the binormal is given by a cross product: $\vec{\mathrm{b}}(u_i) = \vec{\tau}(u_i) \times \vec{\mathrm{n}}(u_i)$.

The choice of the size of the step, $u_{i+1} - u_i$, depends on the trade-off between accuracy and speed. B. Crespin extends the axial deformation to a surface deformation [Cre99].

2.1.5 Wires: a geometric deformation technique

K. Singh and E. Fiume introduce *wires*, a technique which can easily achieve a very rich set of deformations with curves as control features [SF98]. Their technique is inspired by the armatures used by sculptors.

A wire is defined by a quadruple $(\mathrm{R}, \mathrm{W}, s, r)$: the reference curve R, the wire curve W, a scaling factor s that controls bulging around the curve, and a radius of influence r. The set of reference curves describes the armature embedded in the initial shape, while the set of wire curves defines the new pose of the armature.

On a curve C, let p_C denote the parameter value for which $\mathrm{C}(\mathrm{p}_C)$ is the closest point to p. Let us also denote $\mathrm{C}'(\mathrm{p}_C)$ the tangent vector at that parameter value.

The reference curve, R, generates a scalar field $\mathrm{F} : \mathbb{R}^3 \mapsto [0, 1]$. The function F which decreases with the distance to R, is equal to 1 along the curve and equals 0 outside a neighborhood of radius r. The algorithm to compute the image $\mathrm{p}_{\mathrm{def}}$ of a point p influenced by a single deformation consists of three steps, illustrated in Figure 2.9:

- Scaling step. The scaling factor is modulated by F. The image of a point p after scaling is: $\mathrm{p}_s = \mathrm{R}(\mathrm{p}_R) + (1 + s\mathrm{F}(\mathrm{p}))(\mathrm{p} - \mathrm{R}(\mathrm{p}_R))$, where p_R denotes the parameter value for which $\mathrm{R}(\mathrm{p}_R)$ is the closest to p.
- Rotation step. Let θ be the angle between the tangents $\mathrm{R}'(\mathrm{p}_R)$ and $\mathrm{W}'(\mathrm{p}_R)$. The point p_s is rotated around axis $\mathrm{R}'(\mathrm{p}_R) \times \mathrm{W}'(\mathrm{p}_R)$ about center $\mathrm{R}(\mathrm{p}_R)$ by the modulated angle θ F(p). This results in point p_r
- Translation step. Finally, a translation is modulated to produce the image $\mathrm{p}_{\mathrm{def}} = \mathrm{p}_r + (\mathrm{W}(\mathrm{p}_R) - \mathrm{R}(\mathrm{p}_R))$.

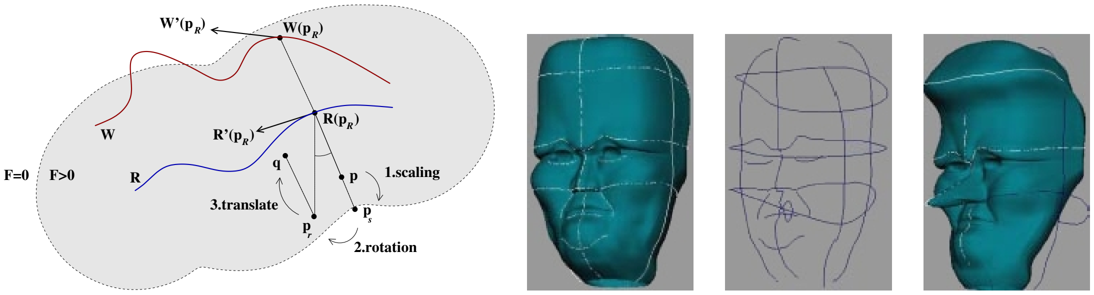


Figure 2.9: *Left: deformation of a point by a single wire: the reference curve is in blue and the wire curve is in red. Right: deformation of a shape with multiple wires (the three images on the right are from [SF98]). The first image shows the initial shape, the second shows the reference curves and the third shows the wire curves and the deformed shape.*

They propose different blending methods in the case when a point is subject to multiple wires. These methods work by taking weighted combinations of the individually deformed point. Let us denote p_i the deformation of p by wire i. Let $\Delta\mathrm{p}_i = \mathrm{p}_i - \mathrm{p}$. The simplest deformation is:

$$\mathrm{p}_{\mathrm{def}} = \mathrm{p} + \frac{\sum_{i=1}^{n} \Delta\mathrm{p}_i \|\Delta\mathrm{p}_i\|^m}{\sum_{i=1}^{n} \|\Delta\mathrm{p}_i\|^m}$$

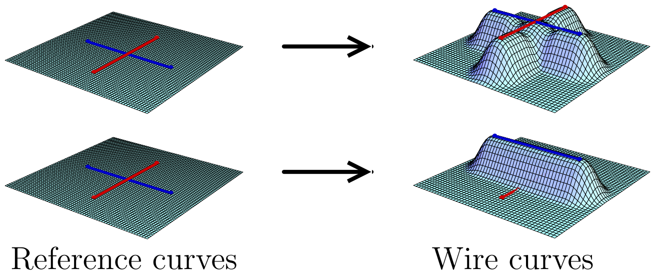


Figure 2.10: *Blending weights based on summed displacement magnitudes. This blending is not free from artifacts: notice the creases around the intersection in the upper-right figure.*

The scalar m is defined by the artist. This expression is not defined when m is negative and $\|\Delta\mathrm{p}_i\|$ is zero. To fix this, they suggest to omit the wires for which this is the case. Their second solution is to use another blending defined for both positive and negative values of m:

$$\mathrm{p}_{\mathrm{def}} = \mathrm{p} + \frac{\sum_{i=1}^{n} \Delta\mathrm{p}_i \prod_{j \neq i} \|\Delta\mathrm{p}_j\|^{|m|}}{\sum_{i=1}^{n} \prod_{j \neq i} \|\Delta\mathrm{p}_j\|^{|m|}}$$

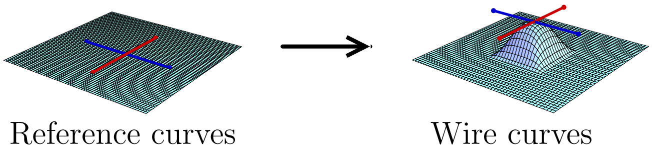


Figure 2.11: *Blending weights based on multiplied displacement magnitudes. The deformation is defined at the intersection of the reference curves.*

In order to use unmoved wires as anchors that hold the surface, they use $\mathrm{F}_i(\mathrm{p})$ instead of $\Delta\mathrm{p}_i$ as a measure of proximity:

$$\mathrm{p}_{\mathrm{def}} = \mathrm{p} + \frac{\sum_{i=1}^{n} \Delta\mathrm{p}_i \mathrm{F}_i(\mathrm{p})^m}{\sum_{i=1}^{n} \mathrm{F}_i(\mathrm{p})^m}$$

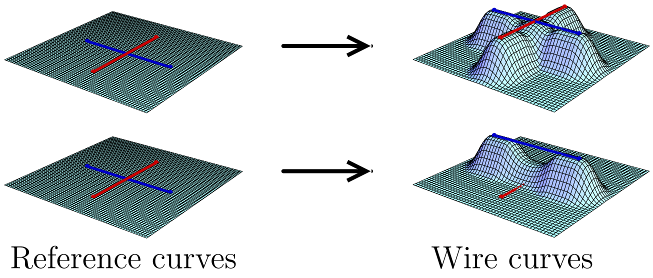


Figure 2.12: *Blending weights based on influence function. The unmoved wire holds space still. This blending is not free from artifacts: notice the creases around the intersection in the upper-right figure.*

Other capabilities of wires can be found in the original paper [SF98]. Note that the expensive part of the algorithm is computing the distance from each curve to each deformed surface point.

2.1.6 Blendeforming: ray traceable localized foldover-free space deformation

As explained in the introduction, there are practical reasons for which a space deformation should be foldover-free. D. Mason and G. Wyvill introduce blendeforming [MW01]. A deformation is specified by moving a point or the control points of a curve along a constrained direction. Space follows the deformation of these control features in a predictable manner.

They define the blendeforming deformation as a bundle of non-intersecting streamlines. The streamlines are parallel, and described by a pair of functions: $b_{\mathrm{x,y}} : \mathbb{R}^2 \rightarrow [-d_{\max}, d_{\max}]$ and $b_{\mathrm{z}} : [0,1] \rightarrow [0,1]$. Function $b_{\mathrm{x,y}}$ controls the amount of deformation for each individual z-streamlines, and the choice of function b_{z} affects the maximum compression of space along the streamlines. The deformation of point $\mathrm{p} = (x, y, z)^\top$ is

$$\begin{aligned} \mathrm{p}_{def} &= (x, y, z_{def})^\top \\ \text{where } z_{def} &= z + b_{\mathrm{x,y}}(x,y)\ b_{\mathrm{z}}(z) \end{aligned} \tag{2.8}$$

It is the definition of b_{z} together with a corresponding threshold $d_{\max}$ that prevents foldovers, as shown in Figure 2.13. The following function is a possible choice for $b_{\mathrm{z}}(z)$, used in the example:

$$b_{\mathrm{z}}(z) = \begin{cases} 16z^2(1-z)^2 & \text{if } z \in [0,1] \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$
$$\text{with } \ d_{\max} = \frac{3\sqrt{3}}{16} \simeq 0.324$$

Functions permitting larger values for $d_{\max}$ can be found in the original paper. Since $b_{\mathrm{x,y}}$ is independent of z, any function with values in $[-d_{\max}, d_{\max}]$ can be used for it, regardless of the slope. Because the amplitude of the effect of a blendeforming function is bounded by the $d_{\max}$ threshold, it is obvious that modeling an entire shape uniquely with blendeforming functions can be rather tedious. In the original paper, the authors also propose bending blendeforming functions, defined in cylindrical coordinates.

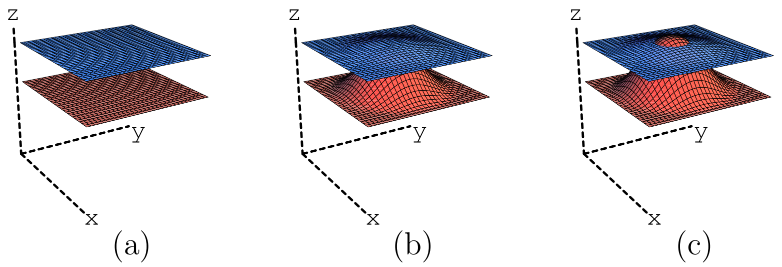


Figure 2.13: *(a) Initial scene: two parallel planes. (b) Blendeforming, with* $b_{\mathrm{x,y}}(x,y) = (x^2 - x + y^2 - y - 1/2)^2$*. The value of* $d_{\max}$ *guarantees that the two planes will never intersect. (c) With* $d_{\max} < d$*, foldover occurs: the lower plane intersects the higher plane.*

2.2 Lattice-based space deformations

The limitation of axial-based or surface-based space deformation comes from the arrangement of the controls along a curve or on a surface. Note that this statement is untrue only for wires, which permits the blending of the controls [SF98]. Lattice-based space deformations are techniques that allow control points to be connected along the three dimensions of space. There are two ways of understanding lattice-based deformation, related to the manner in which the artist expresses the deformation. Let us denote the space deformation function by f.

In the first interpretation of lattice-based deformations, the artist provides pairs of points: a source point and a destination point, $(\mathrm{p}_i, \mathrm{q}_i)$. The deformation f will interpolate or approximate the pairs in this way $f(\mathrm{p}_i) = f_{\mathrm{p}}(\mathrm{p}_i) \approx \mathrm{q}_i$. The function f_{p} is a position field. A position field does not have any physical equivalent to which the artist or scientist can relate, and requires a certain amount of imagination to be visualized.

In the second interpretation of lattice-based deformations, the artist provides a source point and a displacement of that point, $(\mathrm{p}_i, \vec{\mathrm{v}}_i)$. The deformation f will interpolate or approximate the pairs in this way $f(\mathrm{p}_i) = \mathrm{p}_i + f_{\vec{\mathrm{v}}}(\mathrm{p}_i) \approx \mathrm{p}_i + \vec{\mathrm{v}}_i$. The function $f_{\vec{\mathrm{v}}}$ is a vector field. There is a convenient physical analogy to a vector field. Vector fields are used in fluid mechanics to describe the motion of fluids or to describe fields in electromagnetics [Rut90, Gri99]. This analogy is of great help for explaining and creating new space deformations.

While the effect of using either a position field or a vector field is equivalent, the vector field also gives more insight in the process of deforming space: in lattice-based space deformations, the path that brings the source point onto the desired target point is a straight translation using a vector. In this section on lattice-based space deformation, we will therefore consider the construction of a vector field rather than a position field whenever possible.

2.2.1 Free-form deformation of solid geometric models

The effect of Free-Form Deformation (FFD) on a shape is to embed this shape in a piece of flexible plastic. The shape deforms along with the plastic that surrounds it [SP86].

The idea behind FFD is to interpolate or approximate vectors defined in a 3d regular lattice. The vectors are then used to translate space. In their original paper, T. Sederberg and S. Parry propose to use the trivariate Bernstein polynomial as a smoothing filter. Let us denote by $\vec{\mathrm{v}}_{ijk}$ the $(l+1) \times (m+1) \times (n+1)$ *control vectors* defined by the artist. The smoothed vector field is a mapping $\mathrm{p} \in [0,1]^3 \mapsto \mathbb{R}^3$.

$$\begin{aligned}&\vec{\mathrm{v}}(\mathrm{p}) = \\ &\textstyle\sum_{i=0}^{l} \binom{i}{l} (1-x)^{l-i} x^i \left(\sum_{j=0}^{m} \binom{j}{m} (1-y)^{m-j} y^j \left(\sum_{k=0}^{n} \binom{k}{n} (1-z)^{n-k} z^k \right) \right) \vec{\mathrm{v}}_{ijk}\end{aligned} \tag{2.10}$$

Then the deformation of a point is a translation of that point

$$\mathrm{p}_{def} = \mathrm{p} + \vec{\mathrm{v}}(\mathrm{p}) \tag{2.11}$$

In order for the deformation to be continuous across the faces of the FFD cube, the boundary vectors should be set to zero. A drawback of using the Bernstein polynomial is that a control vector $\vec{\mathrm{v}}_{ijk}$ has a non-local effect on the deformation. Hence updating the modification of a control vector requires updating the entire portions of shape within the lattice. For this reason, J. Griessmair and W. Purgathofer propose to use B-Splines [GP89].

In commercial software, the popular way to let the artist specify the control vectors is to let him move the control points of the lattice, as shown in Figure 2.14(c). A drawback often cited about this interface is the visual self occlusion of the control points. This problem increases with the increase in resolution of the lattice. Another drawback is the manipulation of control points, which requires high skills in spacial apprehension from the artist. Clearly, practical FFD manipulation through control-points can only be done with reasonably small lattices.

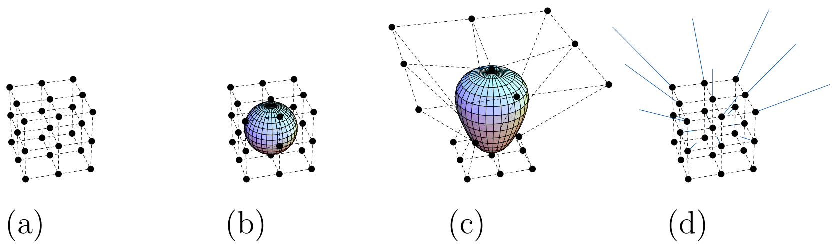

(a) (b) (c) (d)

Figure 2.14: *FFD deformation. (a) Lattice of size* 3^3*. (b) Initial shape. (c) The popular interaction with an FFD lattice consists of displacing the control points. (d) The discrete vectors.*

2.2.2 Extended free-form deformation (EFFD)

Due to the practical limit of the size of the FFD-lattice, the major restriction of an FFD is strongly related to the arrangement of control-points in parallelepipeds. The parallelepipeds are also called *cells*. To provide the artist with more control, S. Coquillart proposes a technique with non-parallelepipedic and arbitrarily connected *cells*. The technique is called Extended Free-Form Deformation (EFFD) [Coq90].

To model with EFFD, the artist first builds a lattice by placing the extended cells anywhere in space, and then manipulates the cells to deform the shape. An extended cell is a small FFD of size 4^4. The transformation from the cell's *local coordinates* $\mathrm{s} = (u, v, w)^\top$ to world coordinates is:

$$\mathrm{p}(\mathrm{s}) = \sum_{i=0}^{3}\binom{i}{3}(1-u)^{3-i}u^i \left(\sum_{j=0}^{3}\binom{j}{3}(1-v)^{3-j}v^j \left(\sum_{k=0}^{3}\binom{k}{3}(1-w)^{3-k}w^k\right)\right) \mathrm{p}_{ijk} \quad (2.12)$$

The eight corners $\mathrm{p}_{ijk\in\{0,3\}^3}$ of a cell are freely defined by the artist. The position of the remaining $4^4 - 8$ are constrained by the connection between cells, so that continuity is maintained across boundaries. This is done when the artist connects the cells. Because the lattice is initially deformed, finding a point's coordinates s in a cell is not straightforward. The local coordinates of a point p in a cell are found by solving

Equation (2.12) in s using a numerical iteration. This can be unstable in some cases, although the authors report they did not encounter such cases in practice. Once s is found, the translation to apply to p is found by substituting in Equation (2.12) the control points p_{ijk} with the control vectors $\vec{\mathrm{v}}_{ijk}$. Note that specifying the control points, the cells and the control vectors is rather tedious, and results shown in the paper consist essentially of imprints. An example is shown in Figure 2.15.

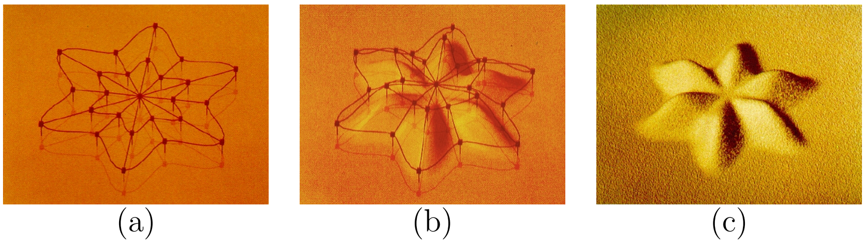

(a) (b) (c)

Figure 2.15: *EFFD deformation, images from [Coq90]. (a)Control lattice. (b)Deformed lattice. (c)Result: a sand-pie.*

2.2.3 Free-form deformations with lattices of arbitrary topology (SFFD)

R.A. MacCracken and K.I. Joy have established a method that allows the user to define lattices of arbitrary shape and topology [MJ96]. The method is more stable than EFFD since it does not rely on a numerical iteration technique.

Their method is based on subdivision lattices. We will refer to it as SFFD, for subdivision FFD. The user defines a control lattice, L: a set of vertices, edges, faces and cells. A set of refinement rules are repeatedly applied to L, creating a sequence of increasingly finer lattices $\{L_1, L_2, \dots L_l\}$. The union of the cells define the deformable space. After the first subdivision, all cells can be classified into cells of different type: type-n cells, $n \geq 3$. See [MJ96] for the rules.

Although there is no available trivariate parameterization of the subdivision lattice, the correspondence between world coordinates and lattice coordinates is possible thanks to the subdivision procedure. The location of a vertex embedded in the deformable space is found by identifying the cell that contains it. Then, for a type-3 cell, trilinear parameterization is used. For a type-n cell, the cell is partitioned in $4n$ tetrahedra, in which the vertex takes a trilinear parameterization. Each point is tagged with its position in its cell.

Once a point's location is found in the lattice, finding the point's new location is straightforward. When the artist displaces the control points, the point's new coordinates are traced through the subdivision of the deformed lattice.

2.2.4 Direct manipulation of free-form deformations (DMFFD)

The manipulation of individual control points makes FFD and EFFD tedious methods to use. Two groups of researchers, P. Borrel and D. Bechmann, and W.M. Hsu et al. propose a similar way of doing direct manipulation of FFD control points (DMFFD) [BB91, HHK92]. The artist specifies translations $\vec{\mathrm{v}}_i$ at points p_i in the form $(\mathrm{p}_i, \vec{\mathrm{v}}_i)$.

The DMFFD algorithm finds control vectors that satisfy, if possible, the artist's desire. Let us define a single input vector $\vec{\mathrm{v}}$ at point p. The FFD Equation (2.10) must satisfy

$$\vec{\mathrm{v}} = \mathrm{B}(\mathrm{p})(\vec{\mathrm{v}}_{ijk}) \tag{2.13}$$

Let $\nu = (3(l+1)(m+1)(n+1))$. The matrix B is the $3 \times \nu$ matrix of the Bernstein coefficients, which are functions of point p. Note that their method is independent of the chosen filter: instead of the Bernstein polynomials, W.M. Hsu et al. use B-Splines and remark that Bernstein polynomials can be used. P. Borrel and D. Bechmann on the other hand found that using simple polynomials works just as well as B-Splines. The size of the vector of control vectors $(\vec{\mathrm{v}}_{ijk})$ is $3(l+1)(m+1)(n+1)$. When the artist specifies μ pairs $(\mathrm{p}_i, \vec{\mathrm{v}}_i)$, the FFD Equation (2.10) must satisfy a larger set of equations:

$$\begin{pmatrix} \vec{\mathrm{v}}_1 \\ \vdots \\ \vec{\mathrm{v}}_\mu \end{pmatrix} = \mathbf{B} \cdot \begin{pmatrix} \vec{\mathrm{v}}_{ijk} \\ \vdots \\ \vec{\mathrm{v}}_{ijk} \end{pmatrix} \quad \text{where } \mathbf{B} = \begin{pmatrix} \mathrm{B}(\mathrm{p}_1) \\ \vdots \\ \mathrm{B}(\mathrm{p}_\mu) \end{pmatrix} \tag{2.14}$$

This set of equations can either be overdetermined or under determined. In either case, the matrix $\mathbf{B}$ cannot be inverted in order to find the $\vec{\mathrm{v}}_{ijk}$. The authors use the Moore-Penrose pseudo-inverse, $\mathbf{B}^+$. If the inverse of $\mathbf{B}^\top \cdot \mathbf{B}$ exists, then

$$\mathbf{B}^+ = (\mathbf{B}^\top \cdot \mathbf{B})^{-1} \cdot \mathbf{B}^\top \tag{2.15}$$

It is however preferable to compute the Moore-Penrose pseudo-inverse using singular value decomposition (SVD). The $\mu \times \nu$ matrix $\mathbf{B}$ can be written

$$\mathbf{B} = U \cdot D \cdot V^\top \tag{2.16}$$

where U is an $\mu \times \mu$ orthogonal matrix, V is an $\nu \times \nu$ orthogonal matrix and D is an $\mu \times \nu$ diagonal matrix with real, non-negative elements in descending order.

$$\mathbf{B}^+ = V \cdot D^{-1} \cdot U^\top \tag{2.17}$$

Here, the diagonal terms of D^{-1} are simply the inverse of the diagonal terms of D.

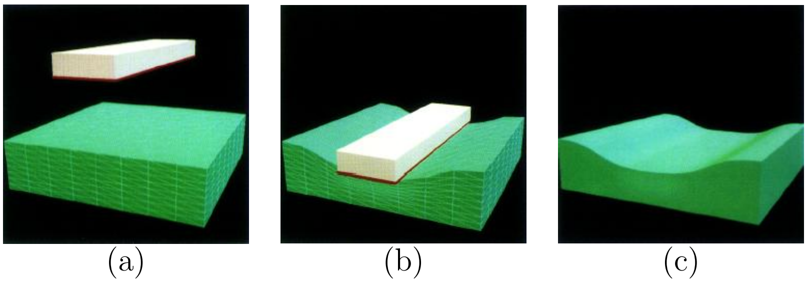

(a) (b) (c)

Figure 2.16: *DMFFD deformation, images from [HHK92]. (a) Initial scene. (b) The deformation is created according to the displacement of several vertices of the green object. (c) Result. The authors do not describe how the vertices on the green object are selected.*

The size of the basis, or, equivalently the number of control points, has a direct effect on the locality of the deformation around the selected point. In their approach, P. Borrel and D. Bechmann pursue the reasoning even further, and define a technique suitable for n-dimensional objects [BB91]. In the context of shape animation, i.e. in $\mathbb{R}^4$ with time as the fourth dimension, the Bernstein, B-Splines or simple polynomials are inappropriate. They propose to use a basis that does not change the initial time, t_0, and final time, t_f, of an object:

$$\mathrm{B}_t(\mathrm{p}, t) = \big((t - t_0)(t - t_f)\ ,\ (t - t_0)(t - t_f)t\ ,\ (t - t_0)(t - t_f)t^2\ ,\ \dots\big)^\top \tag{2.18}$$

2.2.5 Simple constrained deformations for geometric modeling and interactive design (scodef)

In simple constrained deformations (scodef), P. Borrel and A. Rappoport propose to use DMFFD with radial basis functions (RBF) [BR94]. The artist defines constraint triplets $(\mathrm{p}_i, \vec{\mathrm{v}}_i, r_i)$: a point, a vector that defines the desired image of the point, and a radius of influence. Let $\phi_i(\mathrm{p})$ denote the scalar function $\phi(\frac{\|\mathrm{p}-\mathrm{p}_i\|}{r_i})$ for short. The motivation of using RBF is to keep the deformation local, in the union of spheres of radius r_i around the points p_i. A naive vector field would be:

$$\vec{\mathrm{v}}(\mathrm{p}) = \sum_{i=1}^{n} \phi_i(\mathrm{p})\vec{\mathrm{v}}_i \tag{2.19}$$

Unless the points p_i are far apart enough, Equation (2.19) will not necessarily satisfy the artist's input $\vec{\mathrm{v}}(\mathrm{p}_i) = \vec{\mathrm{v}}_i$ if the functions ϕ_i overlap. However, this can be made possible by substituting the vectors $\vec{\mathrm{v}}_i$ with suitable vectors $\vec{\mathrm{w}}_i$.

$$\vec{\mathrm{v}}(\mathrm{p}) = \sum_{i=1}^{n} \phi_i(p)\vec{\mathrm{w}}_i \tag{2.20}$$

These vectors $\vec{\mathrm{w}}_i$ can be found by solving a set of $3n$ equations:

$$\vec{\mathrm{v}}_i^\top = (\phi_1(\mathrm{p}_i) \dots \phi_n(\mathrm{p}_i)) \cdot \begin{pmatrix} \vec{\mathrm{w}}_1^\top \\ \vdots \\ \vec{\mathrm{w}}_n^\top \end{pmatrix} \text{ where } i \in [1, n] \tag{2.21}$$

Let us take the transpose, and arrange the n equations in rows. The following equation is the equivalent of Equation (2.14), but with radial basis functions:

$$\begin{pmatrix} \vec{\mathrm{v}}_1^\top \\ \vdots \\ \vec{\mathrm{v}}_n^\top \end{pmatrix} = \begin{pmatrix} \phi_1(\mathrm{p}_1) & \dots & \phi_n(\mathrm{p}_1) \\ \vdots & & \\ \phi_1(\mathrm{p}_n) & \dots & \phi_n(\mathrm{p}_n) \end{pmatrix} \cdot \begin{pmatrix} \vec{\mathrm{w}}_1^\top \\ \vdots \\ \vec{\mathrm{w}}_n^\top \end{pmatrix} \text{ where } i \in [1, n] \tag{2.22}$$

Let $\mathbf{\Phi}$ be the $n \times n$ square matrix of Equation (2.22). This matrix takes the role of $\mathbf{B}$ in Equation (2.14). Since $\mathbf{\Phi}$ can be singular, the authors also use its pseudo-inverse $\mathbf{\Phi}^+$ to find the vectors $\vec{\mathrm{w}}_i$.

2.2.6 Dirichlet free-form deformation (DFFD)

With DFFD, L. Moccozet and N. Magnenat-Thalmann propose a technique that builds the cells of a lattice automatically [MMT97], relieving the artist from a tedious task. The lattice cells are the cells of a Voronoï diagram of the control points, shown in Figure 2.17. The location of a point within a cell is neatly captured by the Sibson coordinates. The naive deformation of a point p is given by interpolating vectors defined at the control points with the Sibson coordinate.

$$\mathrm{p} \mathrel{+}= \sum_{i=1}^{n} \frac{a_i}{a} \vec{\mathrm{v}}_i \tag{2.23}$$

Where a_i is the volume of cell i stolen by p, and a is the volume of the cell of p. This interpolation is only C^0. They use a method developed by G. Farin [Far90] to define a continuous parameterization on top of the Sibson coordinates. The interpolation is made of four steps:

- build the local control net
- build *Bézier abscissa*
- define *Bézier ordinates* such that the interpolant is C^1
- evaluate the multivariate Bernstein polynomial using Sibson coordinates.

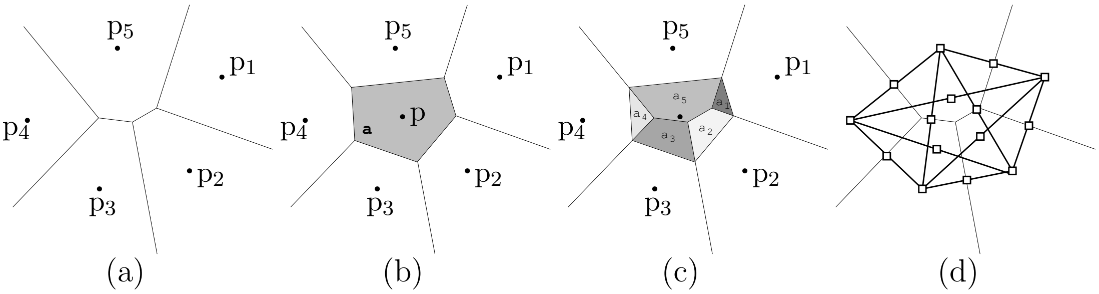


Figure 2.17: *2D illustration of the Sibson coordinates (a) Voronoï cells of the control points. (b) Voronoï diagram is updated after the insertion of point* p*. (c) The areas stolen by the point* p *from its natural neighbors give the Sibson coordinates* a_i/a*. (d) Local control net, with Bézier abscissa.*

2.2.7 Preventing self-intersection under free-form deformation

In FFD, EFFD and DMFFD, if the magnitude of a control-vector is too high, the deformation may produce a self-intersection of the shape's surface (see a self-intersection in Figure 2.13). Once the shape's surface self-intersects, there is no space deformation that can remove the self-intersection. The appearance of this surface incoherency is the result of a space foldover: the deformation function is a surjection of $\mathbb{R}^3$ onto $\mathbb{R}^3$, not a bijection. J. Gain and N. Dodgson present foldover detection tests for DMFFD deformations that are based on uniform B-Splines [GD01]. They argue that a necessary and sufficient test is too time consuming, and present an alternative sufficient test. Let us define q_{ijk}, the deformed control points of the lattice. If the determinants of all the following 3×3 matrices are all positive, there is no foldover.

$$\begin{array}{l} \phi_{ijk} = s \ \det \left(\ \mathrm{q}_{i\pm1jk} - \mathrm{q}_{ijk} \quad , \quad \mathrm{q}_{ij\pm1k} - \mathrm{q}_{ijk} \quad , \quad \mathrm{q}_{ijk\pm1} - \mathrm{q}_{ijk} \ \right) \\ \text{where the sign } s \text{ is 1 if } (i \pm 1, j \pm 1, k \pm 1) \text{ are clockwise, else } -1. \end{array} \tag{2.24}$$

The idea underlying the test is that the determinant of three column vectors is the volume of the parallelepiped defined by these vectors. A negative volume detects a possible singularity in the deformation. A technique for efficiently testing several determinants at once can be found in the original paper.

This test can then be used to repair the DMFFD. Let us define $(\mathrm{p}_i, \vec{\mathrm{v}}_i)$, the pairs of points and vectors defining the DMFFD. If a foldover is detected, the DMFFD is recursively split into two parts: $(\mathrm{p}_i, \vec{\mathrm{v}}_i/2)$ and $(\mathrm{p}_i + \vec{\mathrm{v}}_i/2, \vec{\mathrm{v}}_i/2)$. The procedure eventually converges, and the series of DMFFDs obtained are foldover-free and can be applied safely to the shape.

2.3 Tool-based space deformations

Lattice-based techniques are capable of building a wide range of vector fields. But when dealing with a problem in animation, modeling or visualization, a technique tailored for that specific problem will be more suitable. This section is about techniques that focus on a particular unresolved problem of space deformation, and solve it in an original way.

2.3.1 Interactive space deformation with hardware assisted rendering

Y. Kurzion and R. Yagel present *ray deflectors* [KY97]. The authors are interested in rendering the shape by deforming the rays, as opposed to directly deforming the shape. To deform the rays, one needs the inverse of the deformation that the artist intends to apply to the shape. Rather than defining a deformation and then trying to find its inverse, the authors directly define deformations by their inverse. Their tool can translate, rotate and scale space contained in a sphere, locally and smoothly. Moreover they define a discontinuous deformation that allows the artist to cut space, and change a shape's topology. A tool is defined within a ball of radius r around a center c. Let ρ be the distance from the center of the deflector c to a point p.

$$\rho = \|\mathrm{p} - \mathrm{c}\| \tag{2.25}$$

Translate deflector: To define a translate deflector, the artist has to provide a translation vector, $\vec{\mathrm{t}}$. The effect of the translate deflector will be to transform the center point, c, into $\mathrm{c} + \vec{\mathrm{t}}$.

$$f_{\mathrm{T}}(\mathrm{p}) = \left\{ \begin{array}{ll} \mathrm{p} - \vec{\mathrm{t}}(1 - \frac{\rho^2}{r^2})^2 & \text{if } \rho < r \\ \mathrm{p} & \text{otherwise} \end{array} \right. \tag{2.26}$$
$$\text{where } \theta \in \mathbb{R}$$

Rotate deflector: To define a rotate deflector, the artist has to provide an angle of rotation, θ, and a vector, $\vec{\mathrm{n}}$, about which the rotation will be done. The reader can find the expression of a rotation matrix, $R_{\theta',\vec{\mathrm{n}},\mathrm{c}}$, in Appendix A. Let us call θ' an angle of rotation that varies in space:

$$\theta' = -\theta(1 - \frac{\rho^2}{r^2})^4$$
$$f_{\mathrm{R}}(\mathrm{p}) = \begin{cases} R_{\theta',\vec{\mathrm{n}},\mathrm{c}} \cdot \mathrm{p} & \text{if } \rho < r \\ \mathrm{p} & \text{otherwise} \end{cases} \tag{2.27}$$
$$\text{where } \|\vec{\mathrm{t}}\| \in [0, \frac{3\sqrt{3}r}{8}]$$

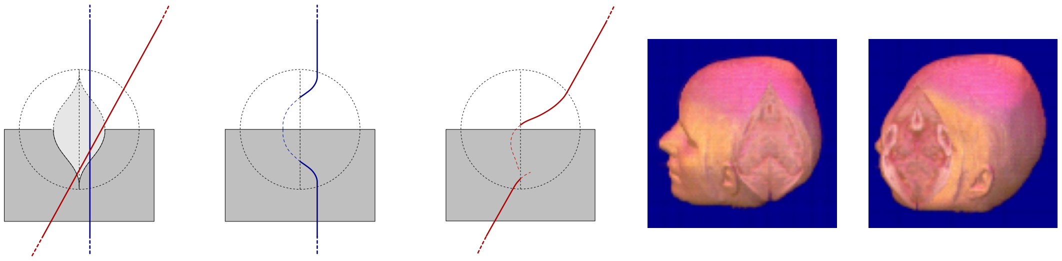
Scale deflector: To define a scale deflector, the artist has to provide a scale factor s. The scale deflector acts like a magnifying glass.

$$f_{\mathrm{S}}(\mathrm{p}) = \begin{cases} \mathrm{p} - (\mathrm{p} - \mathrm{c})(1 - \frac{\rho^2}{r^2})^4 s & \text{if } \rho < r \\ \mathrm{p} & \text{otherwise} \end{cases} \tag{2.28}$$
$$\text{where } s \in [-1, 1]$$

Discontinuous deflector: To define a discontinuous deflector, the artist has to provide a translation vector, $\vec{\mathrm{t}}$. The deflector is split into two halves, on each side of a plane going through c and perpendicular to $\vec{\mathrm{t}}$. In the half pointed at by $\vec{\mathrm{t}}$, the discontinuous deflector will transform c, into $\mathrm{c} + \vec{\mathrm{t}}$, while in the other half, the discontinuous deflector will transform c, into $\mathrm{c} - \vec{\mathrm{t}}$. The effect will be to cut space. The deformation applied to the rays is:

$$f_{\mathrm{D}}(\mathrm{p}) = \begin{cases} \mathrm{p} - \vec{\mathrm{t}}(1 - \frac{\rho^2}{r^2})^2 & \text{if } \rho < r \text{ and } 0 < (\mathrm{p} - \mathrm{c}) \cdot \vec{\mathrm{t}} \\ \mathrm{p} + \vec{\mathrm{t}}(1 - \frac{\rho^2}{r^2})^2 & \text{if } \rho < r \text{ and } (\mathrm{p} - \mathrm{c}) \cdot \vec{\mathrm{t}} < 0 \\ \mathrm{p} & \text{otherwise} \end{cases} \tag{2.29}$$
$$\text{where } \theta \in \mathbb{R}$$

Since this deformation is discontinuous on the disk separating the two halves of the deformation, a ray crossing that disk will be cut in two, as we show in Figure 2.18(c). Thus a shape intersection algorithm will have to march along the ray from the two sides of the ray, until each curve crosses the separating disk. This deformation assumes that the shape's representation has an inside and outside test. Note that other authors have extended FFD for dealing with discontinuities [SE04].



(a) (b) (c) (d) (e)

Figure 2.18: *(a) Discontinuous deflector as observed by the artist. Two arbitrary rays are shown. (b) Simple case, where the ray of light crosses only one hemisphere. (c) When the ray of light changes hemisphere, the curve of light is subject to a discontinuity. (d, e) Example of application, images from [KY97].*

2.3.2 Geometric deformation by merging a 3D object with a simple shape

P. Decaudin proposes a technique that allows the artist to model a shape by iteratively adding the volume of simple 3D shapes [Dec96]. His method is a metaphor of clay sculpture by addition of lumps of definite size and shape. His deformation function is a closed-form, as opposed to a numerical method that would explicitly control the volume [HML99].

Loosely speaking, this technique inflates space by blowing up a tool in space through a hole. This will compress space around the point in a way that preserves the outside volume. Hence if the tool is inserted inside the shape, the tool's volume will be added to the shape's volume. On the other hand, if the tool is inserted outside the shape, the shape will be deformed but its volume will remain constant. This is illustrated for the 2D case in Figure 2.21. A restriction on the tool is to be star-convex with respect to its center c. The deformation function is[2] (see Figure 2.20):

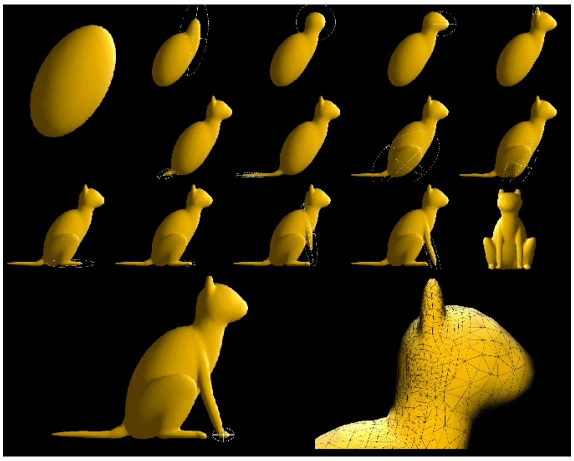


Figure 2.19: *Steps of the modeling of a cat, image from [Dec96].*

$$f_{3D}(\mathrm{p}) = \mathrm{c} + \sqrt[3]{\rho(\mathrm{p})^3 + r(\mathrm{p})^3}\ \vec{\mathrm{n}} \quad (2.30)$$

- $\rho(\mathrm{p})$ is the magnitude of the vector $\vec{\mathrm{u}} = \mathrm{p} - \mathrm{c}$.
- $r(\mathrm{p})$ is the distance between c and the intersection of the tool with the half-line $(\mathrm{c}, \vec{\mathrm{u}})$.
- $\vec{\mathrm{n}} = \vec{\mathrm{u}}/\|\vec{\mathrm{u}}\|$ is the unit vector pointing from c to p.

---

[2] The 2D case is obtained by replacing 3 with 2.

If the tool was not star-convex in c, then $r(\mathrm{p})$ would be ambiguous. The deformation is foldover-free. It is continuous everywhere except at the center c. The effect of the deformation converges quickly to identity with the increasing distance from c. The deformation can be considered local, and is smooth everywhere except at c. An example in $3D$ is shown in Figure 2.19. A feature of this space deformation which is rare, is that it has an exact yet simple inverse in the space outside the tool:

$$f_{3D}^{-1}(\mathrm{p}) = \mathrm{c} + \sqrt[3]{\rho(\mathrm{p})^3 - r(\mathrm{p})^3}\ \vec{\mathrm{n}} \tag{2.31}$$

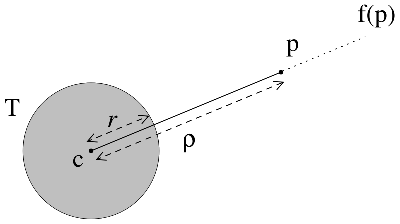


Figure 2.20: *The insertion of a tool at center* c *affects the position of point* p*. See the deformation in Equations (2.30).*

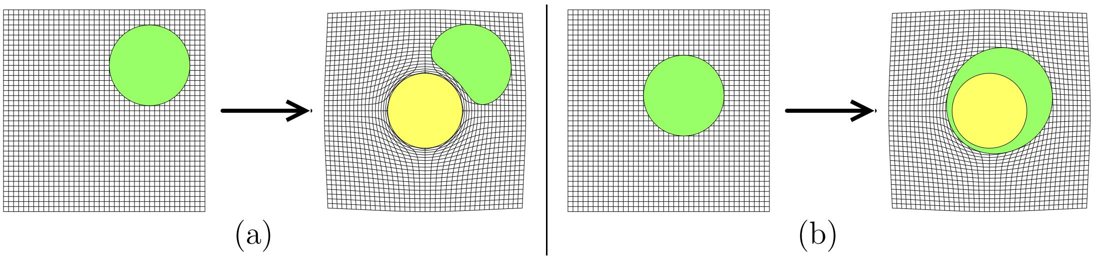


Figure 2.21: *(a) Deformation of a shape (green) by blowing up a tool (yellow) outside the shape. The shape's area is preserved. (b) Deformation of a shape by blowing up a tool inside the shape. The shape's area is increased by that of the tool.*

2.3.3 Implicit free-form deformations (IFFD)

B. Crespin introduces Implicit Free-Form Deformations (IFFD) [Cre99]. Note that though it is called implicit, the deformation is explicit. IFFD is rather a technique inspired by implicit surfaces, a vast branch of computer graphics whose presentation is beyond the scope of this manuscript [BBB$^+$97]. The field values $\phi \in [0,1]$ generated by a skeleton modulates a transformation, M, of points. The deformation of point p by a single function is:

$$f(\mathrm{p}) = \mathrm{p} + \phi(\mathrm{p})(M \cdot \mathrm{p} - \mathrm{p}) \tag{2.32}$$

He proposes two ways to combine many deformations simultaneously. Let us denote by p_i the transformation of p by deformation f_i. The first blending is shown in Figure 2.22. For M, we have used a translation matrix.

$$\mathrm{p}_{def} = \mathrm{p} + \frac{\sum_{i=1}^{n}(\mathrm{p}_i - \mathrm{p})\phi_i(\mathrm{p})}{\sum_{i=1}^{n}\phi_i(\mathrm{p})}$$

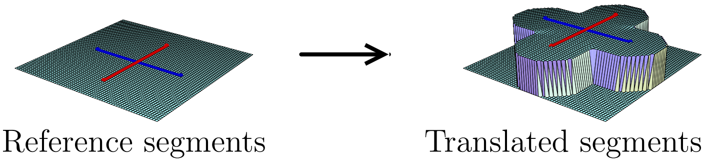


Figure 2.22: *Blending weights based on summed displacement magnitudes. The deformation is only defined where the amounts ϕ are not zero, and is discontinuous at the interface $\sum_i \phi_i = 0$. This blending is useful when the deformed shape is entirely contained within the field.*

The second blending attempts to solve the continuity issue, but requires the definition of supplementary profile functions, γ_i. The purpose of the index i is to assign individual profiles to skeletons.

$$\mathrm{p}_{def} = \mathrm{p} + \frac{\sum_{i=1}^{n}(\mathrm{p}_i - \mathrm{p})\phi_i(\mathrm{p})\gamma_i(\mathrm{p})}{\sum_{i=1}^{n}\phi_i(\mathrm{p})}$$

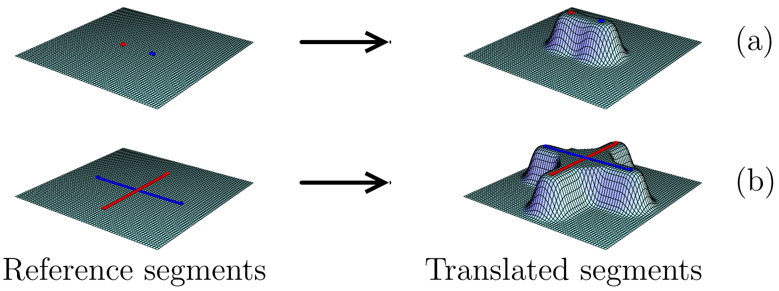


Figure 2.23: *Blending weights based on displacement magnitudes and profile functions. For control points, the technique works well. For segments, there is a discontinuity near their intersection.*

In order to produce Figure 2.23, the following γ_i function was used:

$$\gamma_i(\mathrm{p}) = \begin{cases} 1-(1-\sigma^2)^2 & \text{if } \sigma \in [0,1], \text{ where } \sigma = \sum_{i=1}^{n}\phi_i(\mathrm{p}) \\ 1 & \text{otherwise} \end{cases} \tag{2.33}$$

2.3.4 Twister

I. Llamas et al propose a method called twister in which a twist transformation of points is weighted with a scalar function [LKG+03], in a similar way to IFFD but with a transformation restrained to a twist. With this restriction, they propose to weight single twists along the trajectory of the transformation rather than weighting the displacement. They define a twist by transforming an orthonormal coordinate system $(\mathrm{o}, \vec{\mathrm{u}}, \vec{\mathrm{v}}, \vec{\mathrm{w}})$ into $(\mathrm{o}', \vec{\mathrm{u}}', \vec{\mathrm{v}}', \vec{\mathrm{w}}')$. The axis of the twist is defined by a direction $\vec{\mathrm{d}}$ and point a on the axis, while the angle of rotation around the axis is α and the amount of translation along the axis is d:

$$\begin{array}{rcl} \vec{\mathrm{d}} & = & \frac{\vec{\mathrm{g}}}{\|\vec{\mathrm{g}}\|} \quad \text{where} \quad \vec{\mathrm{g}} = (\vec{\mathrm{u}}'-\vec{\mathrm{u}})\times(\vec{\mathrm{v}}'-\vec{\mathrm{v}}) + (\vec{\mathrm{v}}'-\vec{\mathrm{v}})\times(\vec{\mathrm{w}}'-\vec{\mathrm{w}}) + (\vec{\mathrm{w}}'-\vec{\mathrm{w}})\times(\vec{\mathrm{u}}'-\vec{\mathrm{u}}) \\ \alpha & = & 2\arcsin\left(\frac{\|\vec{\mathrm{u}}'-\vec{\mathrm{u}}\|}{2\|\vec{\mathrm{d}}\times\vec{\mathrm{u}}\|}\right) \\ d & = & \vec{\mathrm{d}}\cdot(\mathrm{o}'-\mathrm{o}) \\ \mathrm{a} & = & \frac{\mathrm{o}+\mathrm{o}'-d\vec{\mathrm{d}}}{2} + \frac{\vec{\mathrm{d}}\times(\mathrm{o}-\mathrm{o}')}{2\tan(\alpha/2)} \end{array} \tag{2.34}$$

Their procedure for deforming a point p with a twist parameterized by t is:

1. Bring p into local coordinates: translate by $-\vec{a}$ and then rotate by a rotation that maps $\vec{d}$ onto $\vec{z}$.

2. Apply the twist in local coordinates: translate by $t\ d$ along $\vec{z}$ and rotate by $t\ \alpha$ around $\vec{z}$

3. Finally bring p back into world coordinates: rotate by a rotation that maps $\vec{z}$ onto $\vec{d}$ and translate by $\vec{a}$

To weight the twist, they propose to use a piecewise scalar function:

$$t(\mathrm{p}) = \begin{cases} \cos^2(\pi\|\mathrm{p}-\mathrm{o}\|/2r) & \text{if } \|\mathrm{p}-\mathrm{o}\| < r \\ 0 & \text{otherwise} \end{cases} \tag{2.35}$$

For operations that require simultaneous twists, they propose simply to add the displacement of the weighted twist. Details for defining a two-point constraint can be found in the paper.

2.3.5 Scalar-field guided adaptive shape deformation and animation (SFD)

J. Hua and H. Qin create a technique called SFD [HQ04]. They define a deformation by attaching space to the level-sets of an animated scalar field. The artist is offered three different techniques for animating a scalar field. Since there are many ways of attaching a point to a level-set of a scalar field, the authors choose the way that keeps the shape as rigid as possible.

They define $\phi(t, \mathrm{p}(t))$, the scalar field which is animated in time, t. Since a moving point, $\mathrm{p}(t)$, is attached to a level-set of the scalar field, the value of ϕ at p is constant in time:

$$\frac{\mathrm{d}\phi}{\mathrm{d}t} = 0 \tag{2.36}$$

The square of Equation (2.36) gives a constraint:

$$(\frac{\mathrm{d}\phi}{\mathrm{d}t})^2 = 0 \tag{2.37}$$

There are several ways of attaching a point to a level set while the scalar field is moving. The simplest way would be to make a point follow the shortest path, found when the magnitude of the point's speed, $\|\vec{\mathrm{v}}(t)\|$, is minimized. Another possibility, chosen by the authors, is to minimize the variation of velocity, so that the deformation is as rigid as possible. Instead of using the divergence of the speed to measure rigidity, they use an estimate by averaging the variation of speed between that point's speed, $\vec{\mathrm{v}}$, and its neighbors' speed, $\vec{\mathrm{v}}_k$:

$$(\nabla \cdot \vec{\mathrm{v}})^2 \approx \frac{1}{k}\sum_k (\vec{\mathrm{v}} - \vec{\mathrm{v}}_k)^2 \tag{2.38}$$

Since this is a constrained optimization problem [Weia], there exists a Lagrange multiplier λ such that:

$$\frac{\mathrm{d}}{\mathrm{d}\vec{\mathrm{v}}}(\frac{\mathrm{d}}{\mathrm{d}t}\phi(t,\mathrm{p}(t)))^2 + \lambda\frac{\mathrm{d}}{\mathrm{d}\vec{\mathrm{v}}}(\nabla\cdot\vec{\mathrm{v}})^2 = \vec{0} \tag{2.39}$$

According to the authors, λ is an experimental constant, used to balance the flow constraint and speed variation constraint. Its value ranges between 0.05 and 0.25. We rearrange this equation and expand the derivative of ϕ with the chain rule:

$$\frac{\mathrm{d}}{\mathrm{d}\vec{\mathrm{v}}}\left((\nabla\phi\cdot\vec{\mathrm{v}} + \frac{\partial\phi}{\partial t})^2 + \lambda(\nabla\cdot\vec{\mathrm{v}})^2\right) = \vec{0} \tag{2.40}$$

Let us define $\hat{\vec{\mathrm{v}}}$, the average of the velocity of all the adjacent neighbors connected with edges to point p. If we substitute $(\nabla\cdot\vec{\mathrm{v}})^2$ for its approximate given by Equation (2.38), and then apply the derivative with respect to $\vec{\mathrm{v}}$, we obtain:

$$(\nabla\phi\cdot\vec{\mathrm{v}} + \frac{\partial\phi}{\partial t})\nabla\phi + \lambda(\vec{\mathrm{v}} - \hat{\vec{\mathrm{v}}}) = \vec{0} \tag{2.41}$$

By solving Equation (2.41), the updated position is:

$$\vec{\mathrm{v}} = \hat{\vec{\mathrm{v}}} - \frac{\hat{\vec{\mathrm{v}}}\cdot\nabla\phi + \frac{\partial\phi}{\partial t}}{\lambda + (\nabla\phi)^2}\nabla\phi \tag{2.42}$$

The algorithm deforms a set of vertices in n sub-steps. If n is set to one, the deformation takes one step. In the first step, since all the speeds are zero, we suggest that they could be initialized with:

$$\vec{\mathrm{v}} = -\frac{\frac{\partial\phi}{\partial t}}{\lambda + (\nabla\phi)^2}\nabla\phi \tag{2.43}$$

The algorithm is:

```
for i = 1 to n do
    for all $\mathrm{p}_k$ in the list of vertices to update do
        Update the scalar field $\phi(t + \Delta t, \mathrm{p}_k)$.
        Deduce $\frac{\partial\phi}{\partial t} = \frac{\phi(t+\Delta t,\mathrm{p}_k)-\phi(t,\mathrm{p}_k)}{\Delta t}$
        Calculate $\nabla\phi$, possibly with finite differences.
        Compute $\hat{\vec{\mathrm{v}}}$ according to neighbors' updated velocities $\vec{\mathrm{v}}_k$.
        Deduce $\vec{\mathrm{v}}$ according to Equation (2.42).
        Update vertex positions with $\mathrm{p}_k(t + \Delta t) = \mathrm{p}_k(t) + \vec{\mathrm{v}}\frac{\Delta t}{n}$
        Improve surface representation using a mesh refinement and simplification strategy.
        if $\phi(t + \Delta t, \mathrm{p}_k(t + \Delta t)) \approx \phi(t, \mathrm{p}_k(t))$ then
            remove $\mathrm{p}_k$ from the list of vertices to update.
        end if
    end for
end for
```

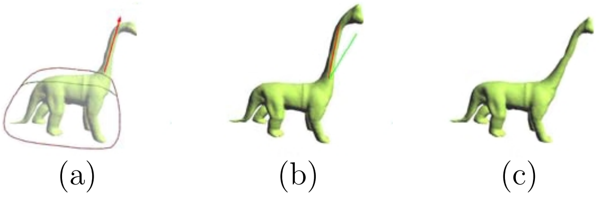

(a) (b) (c)

Figure 2.24: *SFD applied to a digitized model of a dinosaur, images from [HQ04].*

Firstly, this technique is not a very versatile space deformation technique since it requires an explicit surface in order to compute the divergence of the speed. Secondly, the advantage of a large set of possible SFD shape operations (as large as the set of possible animated scalar fields) is at the cost of making the artist's task rather tedious: specifying the animated field does not permit quick and repeated operations on the shape. Note that the authors do not mention how the surface refinement interacts with the update list. Also, results show the editing of imported shapes rather than shapes entirely modeled from scratch.

2.4 Conclusion

The large number of space deformation techniques can lead quickly to the naive conclusion that in any shape modeling by deformation scenario, the limitation of a technique may be simply circumvented by using another technique. This reasoning has several flaws. Firstly, from the point of view of a programmer, the amount of effort required to implement a space deformation Swiss-army knife for shape modeling would be considerable. Secondly, from the point of view of an artist, choosing quickly the most appropriate space deformation would require a vast amount of knowledge of the underlying mathematics of many techniques, which is a skill that should not be required. Thirdly, from a researcher's point of view, all space deformation techniques are not necessarily designed for the specific purpose of shape modeling, and there are surely efficient ways of dealing with specific problems. We will discuss this last point in the remainder of this section, i.e. we will overview the suitability of individual space deformation techniques for the purpose of interactive shape modeling.

Firstly, the subset of space deformations, whose effect on a shape is not local, makes these techniques unsuitable for the task of modeling shapes, since an artist's operation on a visible portion of the shape will have effects on portions that are further away [Bar84, Bla94, CR94, LCJ94].

Secondly, a large number of space deformation techniques requires the artist to specify a rather large number of control parameters [SP86, Coq90, MJ96, MMT97, HML99, HQ04]. We believe that increasing the number of parameters does not increase the amount of control by an artist, but rather it makes the task longer and more tedious. Many techniques illustrate their capabilities on imported models, that were either digitized or pre-modeled with conventional modeling techniques with a few exceptions [Dec96, HHK92, LKG$^+$03]. We believe that the absence of a model entirely developed in one piece with a single technique is some evidence that the technique is tedious to use for the dedicated purpose of modeling shapes.

Finally, many space deformation techniques do not prevent a surface from self-intersecting after deformation, aside from a couple of exceptions [MW01, GD01]. A self-intersecting surface is a rather annoying situation in modeling with deformation, since it is impossible for a space deformation to remove a previously introduced self-intersection. Thus we believe that space deformation operations for shape modeling should satisfy all the following criteria:

- Its effective span should be controllable.
- Its input parameters should be reduced to their strict minimum: a gesture.
- It should be predictable, in accordance with a metaphor.
- It should be foldover-free, and even revertible.
- It should be sufficiently fast for existing computing devices.

To our knowledge, the literature does not contain techniques satisfying all the above criteria. Rather than defining a set of unrelated techniques, we will specify a framework in which we will define deformation operations that satisfy the above. We will illustrate the modeling capabilities of our framework with techniques and examples.

Chapter 3

Sweepers and Related Techniques

s

The work presented in Sections 3.2, 3.3 and 3.4 extends the work published in SMI 2004 [AWC04] which was acknowledged with a best paper award. The work on animation in Section 3.6 has been published in CGI 2004 [AW04].

Sweepers is a class of space deformations suitable for interactive virtual sculpture. The artist describes a basic deformation as a path through which a tool is moved. Our tools are simply shapes, subsets of 3D space. So we can use shapes already created as customized tools to make more complex shapes or to simplify the modeling process.

When a tool is moved it causes a deformation of the working shape along the path of the tool. This is in accordance with a clay modeling metaphor, and is easy to understand and predict. More complicated deformations are achieved by using several tools simultaneously in the same region, using a *blending formula*. The deformation produced by blended tools is smooth and its behavior is predictable.

It is desirable that deformations for modeling are 'foldover-free', that is parts of deformed space cannot overlap so that the deformations are reversible. Applying a non foldover-free deformation to a shape may produce an incoherent shape. We have a proof that our deformations converge to foldover-free deformations and we propose a practical bound for interactive applications.

We have efficient formulations for a single tool following simple paths (translation, scaling or rotation specified for instance with a mouse) and a pair of tools following a path reflected about a mirroring plane. We can demonstrate the effects of multiple tools used simultaneously.

Sweepers can also be keyframed for describing animations of deformable objects. In the end of this chapter, we reformulate sweepers to take into account a varying time parameter, and present three techniques for animation.

3.1 Definitions

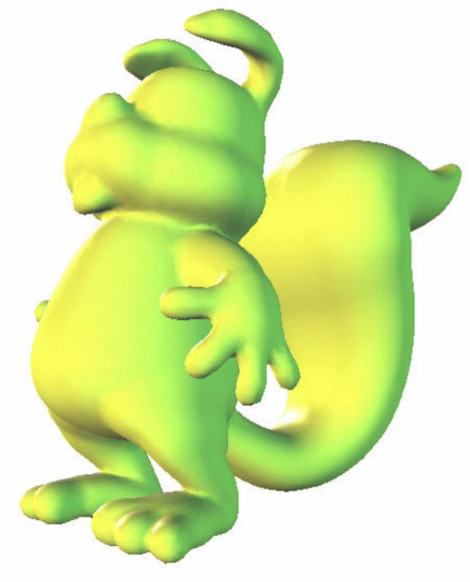


Figure 3.1: *Squirrel character modeled out of an initial ball. The artist modeled only one side, while the other is automatically made at the same time thanks to the simultaneous tool. There are no discontinuities caused by the symmetry.*

The process a sculptor uses to create a shape can be regarded as a definition of the shape. From this point of view, a representation such as a NURBS or implicit surface is merely an intermediate device between the acts of modeling and rendering. Foley and Van Dam remark, "The user interfaces of successful systems are largely independent of the internal representation chosen" [FvDF$^+$94]. This, surely, is evidence that the representations are inherently unsuitable modeling interfaces.

Our thesis is that the primary representation of a model must allow straightforward and predictable editing by an artist. By predictable, we mean that the editing operations must work in accordance with a consistent metaphor that is clear to the artist.

Existing mathematical representations are not directly suitable for editing operations, while most existing editing operations are not predictable according to a suitable metaphor. For most virtual modeling tools, this observation results from the fact that the mathematical representation is strongly linked to the editing operations; for example editing the control points of a NURBS patch manually. Space deformations stand apart from this, and can be used with any mathematical model, including implicit surfaces when the deformation is reversible. However, space deformation has had more success adjusting existing models than with creating entirely new ones, mainly because the deformation operations have not been developed to create a rich set of features. With the exception of [Dec96], [MW01] and [GD01], deformation operations do not prevent surfaces from self-intersecting. This is crucial, since space deformation cannot remove a self-intersection in a surface.

We see all the above as obstacles to the creativity of artists. This chapter introduces a framework for defining *operations for sculpture* independent of the shape's underlying mathematical model. These operations can be applied in principle to any standard model. All the examples in this chapter are deformations of an initially neutral shape, a sphere[1]. These deformation operations are specified intuitively as transformations of tools where a tool is any shape. They are continuous (at least C^0 and in most cases C^2). They are local operations, within some user-defined distance of the tools and, most importantly, they are foldover-free, preserving the shape's coherency.

3.1.1 Terminology and notation

As mentioned in the introduction, space deformation provides a formalism to specify any modeling operation by successively deforming the space in which an initial shape, $S(t_0)$, is embedded. For the moment, t may be interpreted as time. A deformed shape is given by the *modeling equation*:

$$S(t_n) = \left\{ \mathop{\Omega}\limits_{i=0}^{n-1} f_{t_i \mapsto t_{i+1}}(\mathrm{p}) \mid \mathrm{p} \in S(t_0) \right\} \tag{3.1}$$

where $f_{t_i \mapsto t_{i+1}} : \mathbb{R}^3 \to \mathbb{R}^3$ is a space deformation that deforms a point p of space at time t_i into a point of space at time t_{i+1}. The operator Ω expresses the finite repeated composition of functions $f_{t_{n-1} \mapsto t_n} \circ \cdots \circ f_{t_0 \mapsto t_1}(\mathrm{p})$. With sweepers, the artist describes the functions $f_{t_i \mapsto t_{i+1}}$ simply by moving tools. Before introducing sweepers in Section 3.1.2, we present our terminology in the following section.

The motion of tools

We call *influence* an animated scalar field $\phi_t : \mathbb{R}^3 \to [0, 1]$. To let the user specify influences conveniently, we compose the following C^2 piecewise polynomial function, $\mu : \mathbb{R} \to [0, 1]$, with the *distance to the shape of a tool*, $d_t : \mathbb{R}^3 \to \mathbb{R}$:

$$\mu_\lambda(d) = \begin{cases} 0 & \text{if } \lambda \leq d \\ 1 + (\frac{d}{\lambda})^3(\frac{d}{\lambda}(15 - 6\frac{d}{\lambda}) - 10) & \text{if } d < \lambda \end{cases} \tag{3.2}$$

We chose this function also because it is symmetric: $\mu_\lambda(d) = 1 - \mu_\lambda(\lambda - d)$. Other functions may be used. We will use the term *tool* to mean the shape of the tool. We define the *influence* of a tool as follows:

$$\phi_t(\mathrm{p}) = \mu_\lambda \circ d_t(\mathrm{p}) \tag{3.3}$$

The above composition is illustrated in Figure 3.2. For the moment, the reader may envision the tool as a ball. Tools are the topic of Section 3.4. To animate a scalar

---

[1] The use of a sphere makes clear that all the features on our shapes were genuinely created with our technique.

field ϕ_t, the artist is only required to animate the position, orientation and size of the tool. The tool's motion determines a deformation. In the following reasoning, in order to use the tool's motion we will have to refer to the minimum of the derivative of μ, which has the following value for our choice of μ_λ:

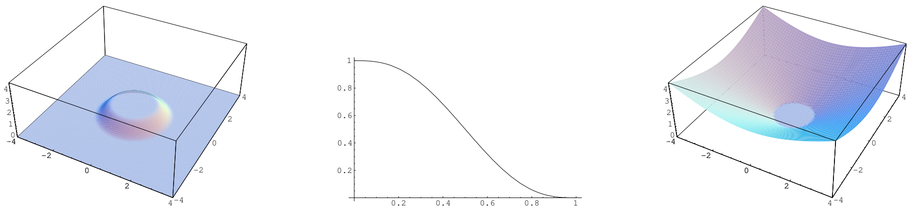
$$\min\left(\frac{\mathrm{d}\mu_\lambda}{\mathrm{d}d}\right) = -\frac{15}{8\lambda} \tag{3.4}$$

There is a wide range of choice for the function μ. If the reader wants to choose another one, we advocate the use of a function whose derivative has a small minimum, justified in Section 3.1.2. The scalar field ϕ_t has local support, and is C^2 if the distance function is smooth within a λ-neighborhood of the tool. We distinguish three zones defined by ϕ_t:

- the *inside* T_t, where $\phi_t(\mathrm{p}) = 1$
- the *coating* K_t, where $\phi_t(\mathrm{p}) \in (0, 1)$
- the *outside* O_t, where $\phi_t(\mathrm{p}) = 0$

We represent the path swept by a tool with discrete keyframes $\{t_0, \ldots t_n\}$, which correspond to positions of the tool. We encode positions along the path and transformations between the positions with 4×4 homogeneous matrices, which may be obtained by multiplying translations, scaling and rotations. We distinguish two kinds of matrices:

- *absolute* transformations M_{t_i}, that encode the *position* of the tool at time t_i. This is a transformation from local tool coordinates to world coordinates.
- *relative* transformations $M_{t_i \mapsto t_{i+1}} = M_{t_{i+1}} \cdot M_{t_i}^{-1}$, that encode the *displacement* of the tool from time t_i to time t_{i+1}. This is a transformation from world coordinates to world coordinates.



$$\phi_t(\mathrm{p}) \quad = \quad \mu_\lambda \quad \circ \quad d_t(\mathrm{p})$$

Figure 3.2: *2D scalar field for a disk of radius* 1, *with* $\lambda_j = 1$. *Right: the distance field. Middle: the function* μ_λ, *applied to the distance. Left: the resulting influence of a tool.*

Combination of transformations

Loosely speaking, the scalar field $\phi_t(\mathrm{p})$ encodes the *amount of deformation* induced by the tool at time t at p. To blend or to compute fractions of deformations, we were inspired by the formalism of M. Alexa [Ale02a], i.e. the multiplication operator $\odot$ and addition operator $\oplus$, which behave essentially like $\cdot$ and $+$ for scalars. Although we use Alexa's operators, we do not necessarily evaluate them numerically as proposed in his paper, since we are interested in combining particular kinds of matrices: translations, rotations, twists and uniform or non-uniform scales.

The operator $\odot$ is defined as $\alpha \odot M = \exp(\alpha \log M)$ and the operator $\oplus$ is defined as $M \oplus N = \exp(\log M + \log N)$. Because the matrices M_i we combine are particular, they can be defined by their logarithm $\log M_i$ in a unique manner, provided in Chapter 7 and Appendix A. Thus we combine our simple matrices M_i weighted by $w_i \in \mathbb{R}$ as follows:

$$\exp(\sum_{i=1}^{n} w_i \log M_i) \tag{3.5}$$

where the exponential of a matrix is defined as follows:

$$\exp M = \mathrm{I} + M + \tfrac{1}{2}M^2 + \tfrac{1}{6}M^3 \cdots = \textstyle\sum_{k=0}^{\infty} \frac{M^k}{k!} \tag{3.6}$$

Although the above is a possible way to evaluate exp, note that simple matrices such as rotation, translation, scale and twist have closed-forms for their exponential. The following shorthand is equivalent to Equation 3.5:

$$w_1 \odot M_1 \oplus w_2 \odot M_2 \oplus \ldots w_n \odot M_n = \bigoplus_{i=1}^{n} w_i \odot M_i \tag{3.7}$$

Note that if M is a translation, then $\alpha \odot M$ is a translation as well. This is also true for scales, rotations and twists. For more general matrices, exp *may be* computed numerically using a series expansions [Ale02a]. Note that When $\|\mathrm{I} - M\| < 1$, the logarithm series converges, and defines an inverse of the exponential. Since we define matrices by their logarithm, the following expression is never computed:

$$\log(\mathrm{I} - M) = -M - \frac{1}{2}M^2 - \frac{1}{3}M^3 \cdots = -\sum_{k=1}^{\infty} \frac{M^k}{k} \tag{3.8}$$

In the remainder of the thesis, when log is used, it will be understood to be restricted to the unit ball about the identity. When compositions of transformations are used, it is assumed that they satisfy this constraint. See Chapter 7 and Appendix A for further discussion.

3.1.2 Single sweepers and solution to foldovers

In order for the deformation induced by the input gestures on a tool to be predictable, we propose that the deformation should satisfy the following:

- the deformation should move every point of space that intersects the tool according to the tool's transformation $M_{t_i \mapsto t_{i+1}}$.
- the deformation should not affect points far away from the tool, i.e. to such points should be applied the identity I.

Naive equation with foldover

The influence of the tool ϕ_t is a scalar function that describes the amount of transformation induced by the tool. ϕ_t can be used conveniently to interpolate in $\mathbb{R}^3$ the matrices $M_{t_i \mapsto t_{i+1}}$ and I using the $\odot$ operator as follows:

$$f_{t_i \mapsto t_{i+1}}(\mathrm{p}) = (\phi_{t_i}(\mathrm{p}) \odot M_{t_i \mapsto t_{i+1}}) \cdot \mathrm{p} \tag{3.9}$$

The above conveniently satisfies $f_{t_i \mapsto t_{i+1}}(\mathrm{p} \in T_t) = M_{t_i \mapsto t_{i+1}} \cdot \mathrm{p}$ and $f_{t_i \mapsto t_{i+1}}(\mathrm{p} \in O_t) = \mathrm{I} \cdot \mathrm{p}$, and interpolates the transformations in-between.

The use of the deformation $f_{t_i \mapsto t_{i+1}}$ is however naive, since it may create a foldover. Foldovers may produce an incoherent shape, whose surface self-intersects. Self-intersections of the shape's surface are undesirable, because they make the location of the inside and outside of the shape ambiguous. Formally, a foldover occurs if:

$$\exists \mathrm{p}, \mathrm{q} \in \mathbb{R}^3 \ , \ \mathrm{p} \neq \mathrm{q} \mid f_{t_i \mapsto t_{i+1}}(\mathrm{p}) = f_{t_i \mapsto t_{i+1}}(\mathrm{q}) \tag{3.10}$$

If the transformation $f_{t_i \mapsto t_{i+1}}$ was a bijection, then it would not create foldovers.

Solution to foldovers

To illustrate a case where a foldover occurs, let us consider a simple scenario: if $M_{t_i \mapsto t_{i+1}}$ is a translation of magnitude larger than the coating thickness λ; it would map points from the inside T_{t_i}, onto points of the outside $O_{t_{i+1}}$, thus folding space onto itself as shown in Figure 3.3(left). However, if we decompose the transformation into a series of s small enough transformations, and apply each of them to the result of the previous one, foldovers can be avoided as shown in Figure 3.3(right). The decomposition in s steps for a general transformation can be expressed as follows:

$$\left\{ \begin{array}{rcl} f_{t_i \mapsto t_{i+1}}(\mathrm{p}) & = & f_{\tau_{s-1} \mapsto \tau_s} \circ \cdots \circ f_{\tau_0 \mapsto \tau_s}(\mathrm{p}) \\ \text{where } f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) & = & (\phi_{\tau_k}(\mathrm{p}) \odot M_{\tau_k \mapsto \tau_{k+1}}) \cdot \mathrm{p} \end{array} \right. \tag{3.11}$$

There exist a finite number of steps s such that the above is foldover-free. Had we found a closed-form expression for the deformation when $s \to +\infty$, we would not need to bother with finding a threshold to s. For local and C^2 influences, computing this closed-form is very tedious if not impossible, thus we propose a lower bound to s, and create equally spaced sub-keyframes $\{\tau_0, \dots \tau_s\}$, such that $\tau_0 = t_i$ and $\tau_s = t_{i+1}$. Note that in Equation (3.11), it could be tempting to replace the operator $\circ$ with a matrix product, but this would not work, since the influence function ϕ_t is a function of space and time.

Algorithm

In the following reasoning, we will focus on a single interval $[t_i, t_{i+1}]$, so for the sake of simplicity let us denote by M the relative transformation $M_{t_i \mapsto t_{i+1}}$. The following matrices are the s intermediate positions of the tool:

$$\left(\frac{k}{s} \odot M \right) \cdot M_{t_i}, \ k \in [0, s-1] \tag{3.12}$$

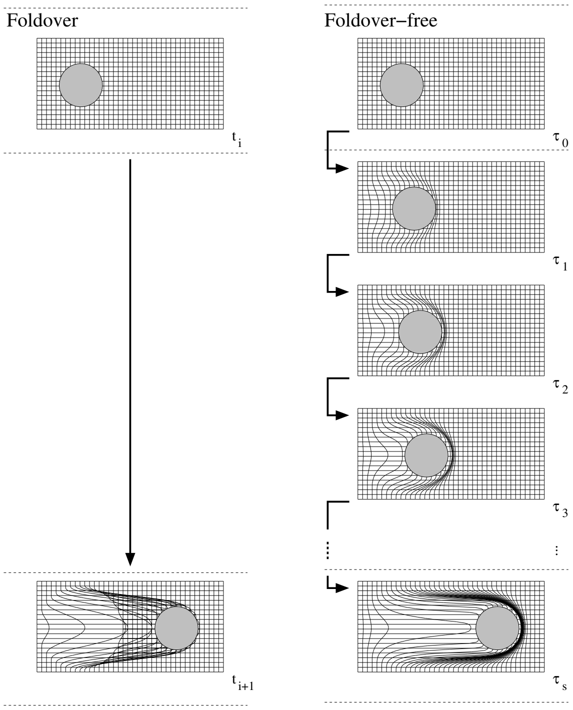


Figure 3.3: *2D illustration of our solution to foldovers. Left: the deformation maps space onto itself. Right: the deformation is decomposed into small foldover-free steps.*

The s in-between relative transformation matrices are all identical:

$$\frac{1}{s} \odot M \tag{3.13}$$

We propose the following as a lower bound to the required number of steps:

$$-\min\left(\frac{\mathrm{d}\mu}{\mathrm{d}d}\right)\max_{l\in[1,8]} ||\log M \cdot \mathrm{p}_l|| < s \tag{3.14}$$

where $\mathrm{p}_{l\in[1,8]}$ are the corners of a bounding box around K_{t_i}, as shown in Figure 3.4. It is apparent in Equation (3.14) that a good choice for the function μ is a function whose first derivative has a small minimum. The following is an algorithm to deform an array of points and normals:

```
Input: positions $M_{t_i}$ and displacement $M$.
Compute the number of required steps, $s$
Find the points contained in the deformation's bounding box, $B$
for each step $k$ from 0 to $s-1$ do
    for each point p in the sub-deformation's bounding box, $B_k$ do
        $\mathrm{p} = (\frac{\phi_{\tau_k}(\mathrm{p})}{s} \odot M) \cdot \mathrm{p}$
        Deform the normal of the point if there is one.
    end for
    Place the tool at the next position $\left(\frac{k+1}{s} \odot M\right) \cdot M_{t_i}$
end for
```

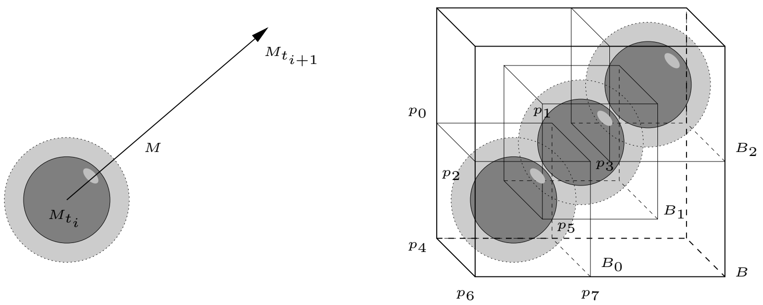


Figure 3.4: *Left: tool and its transformation. Right: the tool's motion is decomposed into $s = 3$ steps, with a bounding box for each step.*

Normal deformation: In order to deform the normals, the co-matrix[2] of the Jacobian is needed [Bar84]. We provide closed-form solutions in Section 3.3. Let us recall that the following expression is a convenient way to compute the co-matrix of $J = (\vec{\mathrm{j}}_x, \vec{\mathrm{j}}_y, \vec{\mathrm{j}}_z)$, where the vectors $\vec{\mathrm{j}}_x$, $\vec{\mathrm{j}}_y$ and $\vec{\mathrm{j}}_z$ are column vectors:

$$\mathrm{com}J = \left(\vec{\mathrm{j}}_y \times \vec{\mathrm{j}}_z, \vec{\mathrm{j}}_z \times \vec{\mathrm{j}}_x, \vec{\mathrm{j}}_x \times \vec{\mathrm{j}}_y\right) \tag{3.15}$$

3.1.3 Simultaneous sweepers

Applying more than one tool at the same time and the same place has applications in modeling, as shown in Figure 3.1, where we modeled a symmetric object by applying the same tool symmetrically about a plane. Manipulating tools simultaneously is also useful for defining a deformable tool, which is composed of several rigid parts; for instance a hand. The simultaneous manipulation of tools also allows the artist to pinch a shape. Pinching will be useful when we extend our method to incorporate topology changes, in Section 3.5.

Naive blending equation

Let us define n moving tools T_j, whose positions are defined at keyframes t_i. To each tool T_j corresponds an influence ϕ_{j,t_i}. To each tool also corresponds a set of relative transformations, $M_{j,t_i \mapsto t_{i+1}}$, between keyframes t_i and t_{i+1}. We denote $\phi_j = \phi_{j,t_i}(\mathrm{p})$ and $M_j = M_{j,t_i \mapsto t_{i+1}}$ for the sake of simplicity. The following expression provides a piecewise smooth[3] combination of all the transformations at any point p in space[4]:

$$\begin{cases} \mathrm{I} & \text{if } \sum_{k=1}^{n} \phi_k = 0 \\ \bigoplus_{j=1}^{n} \left(\left(\frac{1-\prod_{k=1}^{n}(1-\phi_k)}{\sum_{k=1}^{n} \phi_k} \phi_j \right) \odot M_j \right) & \text{if } \sum_{k=1}^{n} \phi_k \neq 0 \end{cases} \tag{3.16}$$

For a practical implementation of the above function, we suggest the use of the following expression in which each $\log(M_j)$ needs to be computed only once for all the deformed

---

[2] Matrix of the co-factors

[3] as smooth as the ϕ_i.

[4] The operator $\bigoplus$ expresses a repetitive sum: $\bigoplus_{i=1}^{n} M_i = M_1 \oplus M_2 \oplus \cdots \oplus M_n$.

points:

$$f_{t_i \mapsto t_{i+1}}(\mathrm{p}) = \begin{cases} \mathrm{p} & \text{if} \sum_{k=1}^{n} \phi_k = 0 \\ \exp\left(\frac{1-\prod_{k=1}^{n}(1-\phi_k)}{\sum_{k=1}^{n}\phi_k} \sum_{j=1}^{n} \left(\phi_j \log(M_j)\right)\right) \cdot \mathrm{p} & \text{if } \sum_{k=1}^{n} \phi_k \neq 0 \end{cases} \tag{3.17}$$

where:

- $\frac{1}{\sum_{k=1}^{n}\phi_k(\mathrm{p})}$ is required to produce a **normalized** combination of the transformations. This prevents for instance two translations of vector $\vec{\mathrm{d}}$ producing a translation of vector $2\vec{\mathrm{d}}$, which would send a point far away from the tools. This unwanted behaviour was also identified by K. Singh and E. Fiume [SF98].
- $1 - \prod_{k=1}^{n}(1 - \phi_k(\mathrm{p}))$ **smooths** the deformation in the entire space, required in the boundary between $K_{t_i}^j$ and $O_{t_i}^j$. Indeed, smoothness would be lost if we only used the normalization above.

Figure 3.5 shows a comparison between a naive additive blending and the one just described. An interesting feature of Equation (3.17) is that there is no scalar field other than ϕ_j required to ensure locality and continuity in $\mathbb{R}^3$, as opposed to the solution proposed by B. Crespin [Cre99] presented in Section 2.3.3.

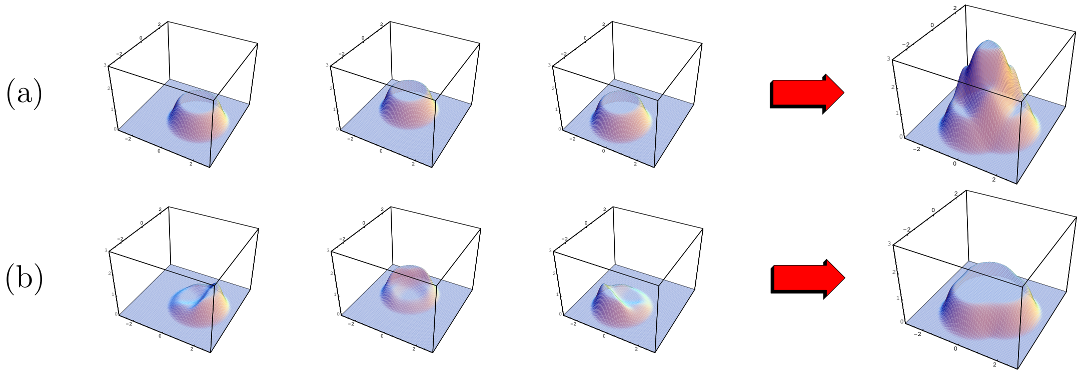


Figure 3.5: *Scalar fields of three 2d tools. To illustrate the behaviour of our blending, we combine the scalar fields instead of using them to modulate a transformation. (a) Adding the scalar fields. (b) By multiplying each field with* $(1 - \prod(1 - \phi_k))/\sum \phi_k$*, the sum of the fields is normalized.*

3.2 Solution to foldover

Equation (3.17) is naive for similar reasons to the ones discussed for a single tool in Section 3.1.2. If we decompose it in small steps, foldovers can be avoided:

$$\boxed{\begin{aligned} f_{t_i \mapsto t_{i+1}}(\mathrm{p}) &= \mathop{\Omega}_{k=0}^{s-1} f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) \\ \text{where } f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) &= \begin{cases} \mathrm{p} & \text{if } \sum_j \phi_{j,\tau_{k+1}} = 0 \\ \exp\left(\frac{1-\prod_j (1-\phi_{j,\tau_{k+1}})}{\sum_j \phi_{j,\tau_{k+1}}} \sum_j \left(\phi_{j,\tau_{k+1}} \log(M_j)\right)\right) \cdot \mathrm{p} & \text{otherwise} \end{cases} \end{aligned}} \tag{3.18}$$

The following expression is a lower bound to the required number of steps, generalizing the single tool condition:

$$\boxed{-\sum_j \min(\frac{\mathrm{d}\mu_j}{\mathrm{d}d}) \max_{l\in[1,8]} \left|\left|\log M_j \cdot \mathrm{p}_{l_j}\right|\right| < s} \tag{3.19}$$

where $\mathrm{p}_{l_j\in[1,8]}$ are the corners of the bounding box around $K^j_{t_i}$.

Algorithm

To speed up the computation, bounding boxes are used, as shown in Figure 3.6. In some special cases discussed below, this algorithm can be speeded. The following is an algorithm to deform an array of points and normals:

```
Input: tools' positions $M_{j,t_i}$ and transformations $M_j$.
Compute the number of steps, $s$, using Equation (3.19).
Find the points contained in the deformation's bounding box, $B$.
for each step $k$ from 0 to $s-1$ do
    Compute the union of all the tools' influence bounding boxes, $B_k$.
    for each point $\mathrm{p} \in B_k$ do
        $\sigma = 0$
        $\psi = 1$
        $L = 0$
        for each tool $T_j$ do
            if $\phi_j \neq 0$ then
                $\sigma = \sigma + \phi_j$
                $\psi = (1 - \phi_j)\ \psi$
                $L = L + \frac{\phi_j}{s} \log M_j$
            end if
        end for
        if $\sigma \neq 0$ then
            $\mathrm{p} = \exp(\frac{1-\psi}{\sigma} M) \cdot \mathrm{p}$
            Deform the normal of the point if there is one.
        end if
    end for
    Place all the tools at their next position, defined by $\left(\frac{k+1}{s} \odot M_j\right) \cdot M_{j,t_i}$
end for
```

Normal deformation: In order to deform the normals, we need to compute the co-matrix of the Jacobian [Bar84]. Even though a closed-form can be derived from the above transformation, its length makes it difficult to code and time consuming. In practice, computing the Jacobian with finite differences works well enough. Computing the deformation of the normal with multiple tools can be done efficiently by storing the influences ϕ_j in an array, and then considering only the tools whose influence is not zero.

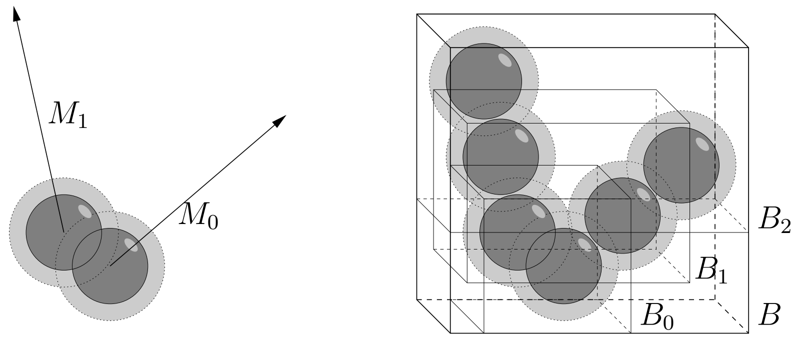


Figure 3.6: *Left: two tools and their transformation. Right: the tools' motions are decomposed into $s = 3$ steps, with a bounded box for each step.*

3.3 Fast expressions for simple sweepers

When using multiple tools simultaneously, the time of the scene is frozen in order to move each tool one at a time. This is not however the case when using a single tool.

3.3.1 Single tool

In a single tool scenario, the transformations that are convenient to input for the artist are pure translations, non-uniform and uniform scaling and rotations. With these simple transformations, the deformations of a point and its normal are much simpler to compute, as there is a closed-form to the logarithm of simple matrices. In this section, in addition to efficient expressions for computing the number of required steps, we provide fast deformation functions for a vertex and its normal. The formulas that we give have been derived from the expressions of log and exp given in Appendix A. For deforming the normal, computing the Jacobian's co-matrix is not always required: $(\mathrm{com}J^t) \cdot \vec{\mathrm{n}}$ leads to much simpler expressions. Note that the normal's deformations do not preserve the normal's length. It is therefore necessary to divide the normal by its magnitude. We denote $\vec{\gamma}_t = (\gamma_x^t, \gamma_y^t, \gamma_z^t)^\top$ the gradient of ϕ_t at p.

If M is a translation:
The use of $\odot$ can be simplified with translation vector $\vec{\mathrm{d}}$. The minimum number of steps is:

$$-\min(\frac{\mathrm{d}\mu}{\mathrm{d}d})\|\vec{\mathrm{d}}\| < s \tag{3.20}$$

The s vertex deformations are:

$$f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \mathrm{p} + \frac{\phi_{\tau_k}(\mathrm{p})}{s}\vec{\mathrm{d}} \tag{3.21}$$

The s normal deformations are:

$$g_{\tau_k \mapsto \tau_{k+1}}(\vec{\mathrm{n}}) = \vec{\mathrm{n}} + \frac{1}{s}(\vec{\gamma}_{\tau_k} \times \vec{\mathrm{n}}) \times \vec{\mathrm{d}} \tag{3.22}$$

If M is a uniform scaling operation:

Let us define the center of the scale c, and the scaling factor σ. The minimum number of steps is:

$$-\min(\frac{\mathrm{d}\mu}{\mathrm{d}d})\sigma\log(\sigma)d_{\max} < s \tag{3.23}$$

where $d_{\max}$ is the largest distance between a point in the deformed area and the center c, approximated using a bounding box. The s vertex deformations are:

$$f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \mathrm{p} + (\sigma^{\frac{\phi_{\tau_k}(\mathrm{p})}{s}} - 1)(\mathrm{p} - \mathrm{c}) \tag{3.24}$$

Let $\vec{\chi} = \frac{\log(\sigma)}{s}(\mathrm{p} - \mathrm{c})$. The s normal deformations are:

$$g_{\tau_k \mapsto \tau_{k+1}}(\vec{\mathrm{n}}) = \vec{\mathrm{n}} + (\vec{\gamma}_{\tau_k} \times \vec{\mathrm{n}}) \times \vec{\chi} \tag{3.25}$$

If M is a non-uniform scaling operation:

Let us define the center of the scale c, its direction of scale as the unit vector $\vec{\mathrm{n}}$, and its scaling factor σ. The minimum number of steps is:

$$-\min(\frac{\mathrm{d}\mu}{\mathrm{d}d})\sigma\log(\sigma)d_{\max} < s \tag{3.26}$$

where $d_{\max}$ is the largest distance between a point in the deformed area and the plane of normal $\vec{\mathrm{n}}$ passing through c. The s vertex deformations are:

$$f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \mathrm{p} + (\sigma^{\frac{\phi_{\tau_k}(\mathrm{p})}{s}} - 1)((\mathrm{p} - \mathrm{c}) \cdot \vec{\mathrm{n}})\vec{\mathrm{n}} \tag{3.27}$$

Let $\vec{\chi} = \frac{\log(\sigma)}{s}(\mathrm{p} - \mathrm{c})$. The s normal deformations are:

$$g_{\tau_k \mapsto \tau_{k+1}}(\vec{\mathrm{n}}) = \vec{\mathrm{n}} + \sigma^{\frac{\phi_{\tau_k}(\mathrm{p})}{s}}((\vec{\mathrm{v}} + (\vec{\mathrm{v}} \cdot \vec{\chi})\vec{\gamma}_{\tau_k}) \times \vec{\mathrm{n}}) \times \vec{\mathrm{v}} \tag{3.28}$$

It is appropriate to remark here that the tool is also subject to the scale, and that the influence function ϕ_t must be defined in an appropriate way, as described in Section 3.4.

If M is a rotation:

Let us define a rotation angle θ, center of rotation r and vector of rotation $\vec{\mathrm{v}} = (v_x, v_y, v_z)^\top$. The minimum number of steps is:

$$-\min(\frac{\mathrm{d}\mu}{\mathrm{d}d})\theta r_{\max} < s \tag{3.29}$$

where $r_{\max}$ is the distance between the axis of rotation and the farthest point from it, approximated using a bounding box. The s vertex deformations are:

$$f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \mathrm{p} + (\cos\frac{\phi_{\tau_k}\theta}{s} - 1)\vec{\xi} \times \vec{\mathrm{n}} + \sin\frac{\phi_{\tau_k}\theta}{s}\vec{\xi} \tag{3.30}$$
$$\text{where } \vec{\xi} = \vec{\mathrm{v}} \times (\mathrm{p} - \mathrm{r})$$

The s normal deformations are:

$$\begin{aligned} g_{\tau_k \mapsto \tau_{k+1}}(\vec{\mathrm{n}}) &= (\vec{\mathrm{n}} \cdot \vec{\mathrm{v}})\vec{\mathrm{v}} + \vec{\mathrm{v}} \times (\cos(h)\vec{\mathrm{n}} \times \vec{\mathrm{v}} - \sin(h)\vec{\mathrm{n}}) \\ &\quad + \theta\vec{\gamma} \times (\vec{\mathrm{n}} \times \vec{\xi} + ((\cos(h) - 1)(\vec{\mathrm{n}} \times \vec{\xi}) \cdot \vec{\mathrm{v}} + \sin(h)\vec{\mathrm{n}} \cdot \vec{\xi})\vec{\mathrm{v}}) \\ \text{where } h &= \tfrac{\phi_{\tau_k}\theta}{s} \end{aligned} \tag{3.31}$$

3.3.2 Symmetric tool

With a symmetric tool, the transformation matrices are of the same type, so blending them leads to simple expressions. Let us consider two tools T_0 and T_1, with influences ϕ_t^0 and ϕ_t^1. If the influence of both tools is zero at p, that is if $\phi_t^0(\mathrm{p}) = 0$ and $\phi_t^1(\mathrm{p}) = 0$, then the deformation is the identity. If the influence of one tool is zero at p, that is if $\phi_t^0(\mathrm{p}) = 0$ or $\phi_t^1(\mathrm{p}) = 0$, then the deformation equation is that of a single tool. When both influences are not zero at p, that is if $\phi_t^0(\mathrm{p}) \neq 0$ and $\phi_t^1(\mathrm{p}) \neq 0$, then the deformation induced at p by the tools' motion must be computed using Equation (3.17). In the rest of this section, we have simplified the blending equation for simple symmetric transformations of the same type.

Translation

The number of steps is:

$$-\min(\frac{\mathrm{d}\mu}{\mathrm{d}d})(\|\vec{\mathrm{d}}_0\| + \|\vec{\mathrm{d}}_1\|) < s \tag{3.32}$$

Where ϕ_0 and ϕ_1 are not trivial, the deformation of a point is:

$$f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \mathrm{p} + \frac{1}{s}(1 - \frac{\phi_0\phi_1}{\phi_0 + \phi_1})(\phi_0\vec{\mathrm{d}}_0 + \phi_1\vec{\mathrm{d}}_1) \tag{3.33}$$

Rotation, scale and non-uniform scale

Where ϕ_0 and ϕ_1 are not trivial, the deformation of a point is:

$$f_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \exp(\frac{1}{s}(1 - \frac{\phi_0\phi_1}{\phi_0 + \phi_1})(\phi_0 \log M_0 + \phi_1 \log M_1)) \tag{3.34}$$

3.4 Tool influence

An artist specifies a sweeper space deformation by moving tools. Each tool generates an influence, that encodes the tool's amount of transformation. The set of possible tools an artist can use determines the modeling possibilities. There are two possible ways of defining an influence. Either the influence is defined in a local coordinate system of the tool, or is defined directly in world coordinates.

Matrix transformation of a scalar field

The influence at a point is defined by applying a smoothing function to the distance from the point to the shape, as defined by Equation (3.3). There are two possible ways of computing the distance to a shape. These two equations are equivalent in the case where the eigenvalues of the matrix M_{t_i} are all equal, i.e. when M_{t_i} is isotropic. Let us denote S the shape of the tool. The distance can be evaluated in local coordinates:

$$d_{t_i}(\mathrm{p}) = \det(M_{t_i})^{\frac{1}{3}} \min_{\mathrm{q} \in S} \|M_{t_i}^{-1} \cdot \mathrm{p} - \mathrm{q}\| \tag{3.35}$$

The scalar $\det(M_{t_i})$ is the increase of volume since M_{t_i} is isotropic. Thus the factor $\det(M_{t_i})^{\frac{1}{3}}$ is required to rescale the distance along each axis. The distance can also be evaluated in world coordinates:

$$d_{t_i}(\mathrm{p}) = \min_{\mathrm{q} \in S} \|\mathrm{p} - M_{t_i} \cdot \mathrm{q}\| \tag{3.36}$$

Directly using the distance definition of Equation (3.35) is prohibitive, and an algorithmic trick has to be used: clustering the points of the shape, deriving a closed-form formula, or defining a numerical procedure. Using Equation (3.35) allows the programmer to precompute a data structure in local coordinates, and hence to speed up the evaluation of distance functions. There is a drawback however with using Equation (3.35) when M_{t_i} is a non-uniform scale: a distance field scaled non-uniformly does not satisfy the properties of a distance field any longer. If the smoothing function μ_λ is applied to the non-uniformly scaled distance field, the slope of the influence is also scaled non-uniformly, and the resulting slope is uncontrollable (it is a function of space). Since we do not have a convenient way to show to the artist the slope of influence in space, we strongly recommend using Equation (3.36) in the case of non-uniform scales. Figure 3.7 shows the difference between the scaled influence of a shape and the influence of a scaled shape; recall that the artist is only shown the ellipsoid.

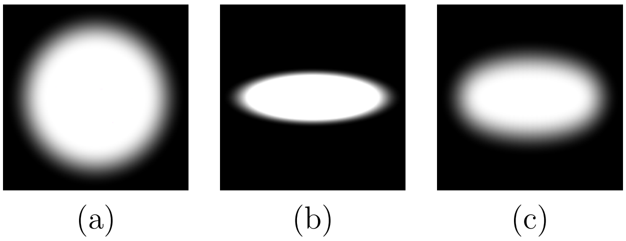


Figure 3.7: *(a) Influence of a shape, a disk. White corresponds to a value of* 1*, black corresponds to no influence at all. (b) Scaled influence of the shape. The slope of the gradient is dramatically affected by the scale. (c) Influence of the scaled shape, using the distance to a filled ellipse.*

3.4.1 Ball tool

The distance to a ball has a simple expression in local coordinates, in a way similar to that of Equation (3.35):

$$d_{sphere}(\mathrm{p}) = \begin{cases} 0 & \text{if } \|M_{t_i}^{-1} \cdot \mathrm{p}\|^2 \leq 1 \\ \det(M_{t_i}^{\frac{1}{3}})(\|M_{t_i}^{-1} \cdot \mathrm{p}\| - 1) & \text{otherwise} \end{cases} \tag{3.37}$$

If the artist wishes to apply a non-uniform scale to the sphere, it would turn into an ellipsoid, and Equation (3.37) would not be usable.

3.4.2 Filled ellipsoid tool

The ellipsoid is defined in local coordinates as a unit sphere, whose position in world coordinates is encoded in a possibly non-uniform matrix M_{t_i}. To compute the distance to a filled ellipsoid, we use the numerical method described in [Ebe01].

$$d_{\text{ellipsoid}}(\mathrm{p}) = \begin{cases} 0 & \text{if } \|M_{t_i}^{-1} \cdot \mathrm{p}\|^2 \leq 1 \\ \min\limits_{\mathrm{q} \in S} \|\mathrm{p} - M_{t_i} \cdot \mathrm{q}\| & \begin{array}{l}\text{otherwise,} \\ \text{where } S \text{ is the unit sphere at the origin}\end{array} \end{cases} \tag{3.38}$$

3.4.3 Mesh tool

The easiest tools that can be implemented are simple objects (e.g. point, ball, ellipsoid, cube) for which it is possible to derive closed-form expressions for calculating their distance to a point. It is however convenient for an artist to choose or manufacture his own tools, as every artist has an original way of modeling. For this purpose, we offer to the artist the possibility of *baking* pieces of clay in order to use them as tools (see Figure 3.8). By baking, we mean precomputing a data structure such that the distance field can be computed efficiently. Various algorithms exist, and presenting them is beyond the scope of this thesis. More information can found in [Gué01]. In our implementation, we precompute a BSP of the Voronoï diagram of the vertices, and compute the distance using the surrounding triangles. For the sake of completeness, the method implemented is described in this section.

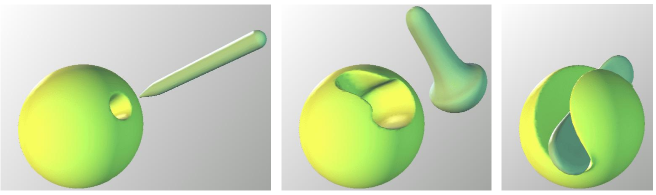


Figure 3.8: *Example of customized tools deforming a sphere.*

Precomputed structure

The distance to a polygonal surface is the shortest distance to each triangle of the surface. To avoid computing the distance to each triangle, some clustering of space needs to be done: to each triangle, T, corresponds a polygonal cell, Vor(T), such that if p $\in$ Vor(T), then T is the closest triangle to p, else there is another triangle closer to p. Ideally, we would like to compute the Vor(T) cells, which unfortunately happen to be very complex in shape. Our solution is approximate, but happens to be exact in most cases.

1. We precompute a BSP of the Voronoï diagram of sample vertices on the tool's shape. Thus an initial guess can be found in $log(N)$ (for N vertices). At each leaf, the index of the corresponding vertex is stored. Border cells of the BSP can be infinite, so we represent the vertices of the cells using projective geometry.

2. To compute the distance to the tool when modeling with it, the closest vertex is fetched, and the tested triangles are the ones connected to this vertex. To cope with the approximation of the surface with points, the query point can be jittered within a neighborhood $k_{\min}$, the minimum length of an edge. For each triangle,

we use the point to triangle algorithm of [Ebe01]. The distance to the tool is the smallest distance to each triangle.

This algorithm returns the distance to an *empty* tool, i.e. the shape is a membrane and the distance is zero only on the *surface* of the shape. If the user wants to grab the piece of clay with the tool, our distance algorithm must return the value 0 if the point is inside the tool. Thus, we add an inside test.

Distance to a volume algorithm: The inside test depends on the triangle features to which the query point is the closest (see Figure 3.9). The distance-to-triangle algorithm that we use gives the feature to which the closest point belongs [Ebe01].

- If the closest point is on the inside of the triangle, the point is inside the tool if it is behind the triangle.
- If the closest point is on an edge of the triangle, the point is inside the tool if either: (a) the edge is convex and the point is behind the two triangles sharing that edge, or (b) the edge is concave and the point is behind either one triangle or the other sharing the edge.
- If the closest point is a vertex, the point is inside the tool if it is in front of all the planes perpendicular to the concave edges around that vertex, as shows a simple case in Figure 3.9(c).

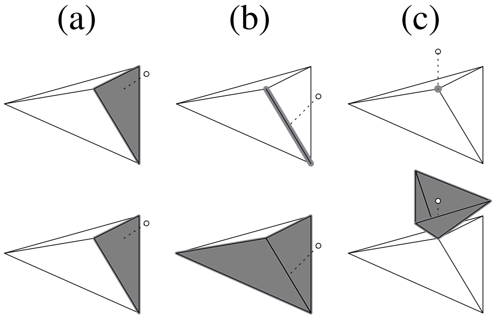
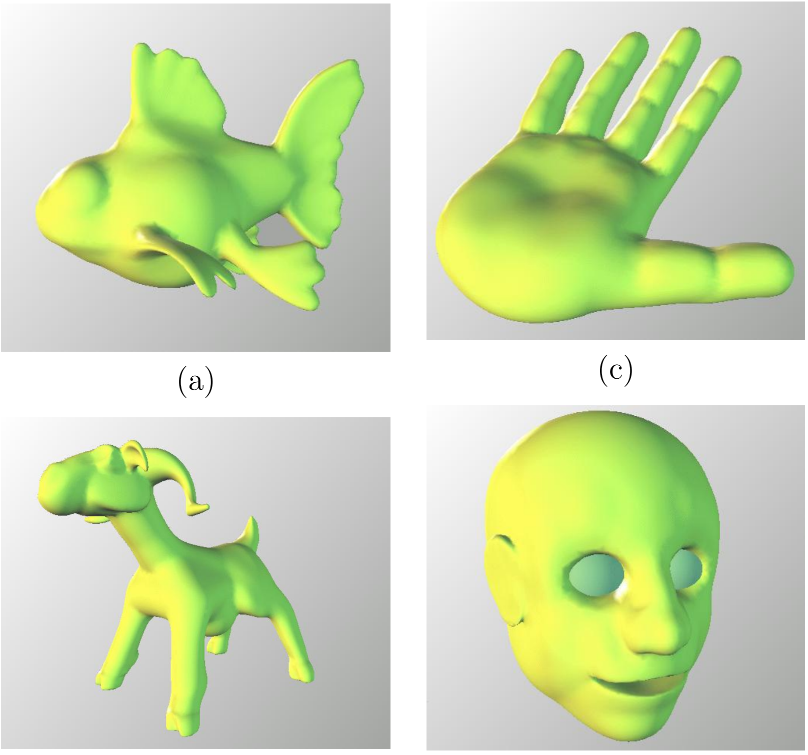


Figure 3.9: *Closest feature ((a) surface, (b) edge, (c) vertex), with corresponding planes used for the inside test. In the case where the feature is a vertex, planes are considered if edges are concave.*

Limitation of the algorithm: This distance field is inaccurate in the case of a very thin surface, i.e. when the closest point is found behind a triangle. It works fine in the case where the surface is finely sampled or not too twisted.

Results

The shapes in Figure 3.10 and Figure 3.11 were modeled with sweepers in at most one hour, starting with a sphere. In Figure 3.11(a), the first modeling step was to squash the sphere into a very thin disk. In Figure 3.10(d), eyeballs were added and the mouth was sculpted with a custom shape. The technique used for representing the surface of these shapes is presented in Chapter 5.


(a) (c)

(b) (d)

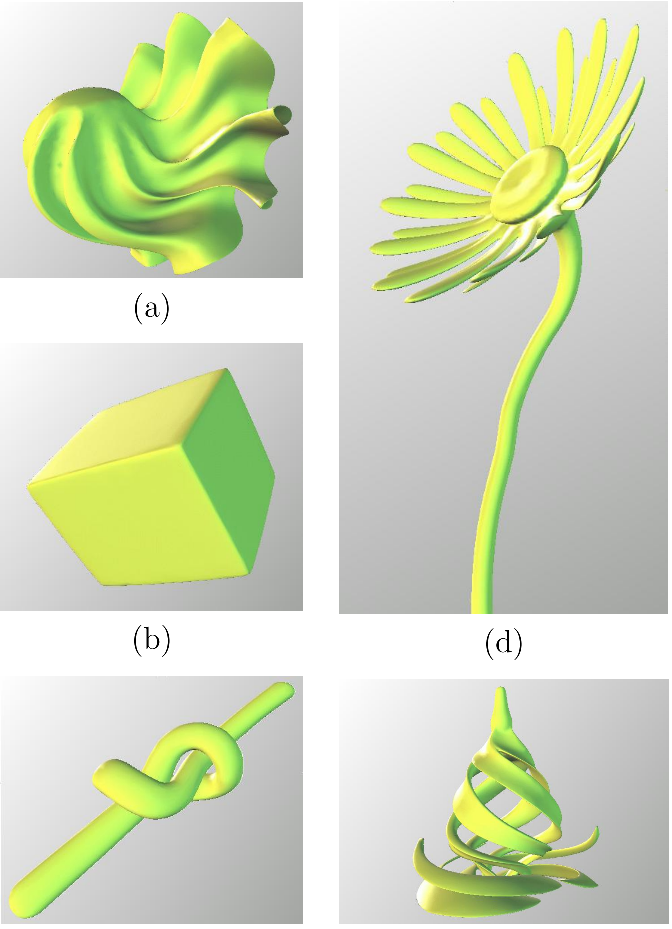
Figure 3.10: *All these shapes were modeled with sweepers in at most one hour, starting with a sphere. In (d), eyeballs were added.*

3.5 Changing topology

The purpose of this section is to show that a space deformation can both prevent foldovers and change topology: these are not contradictory properties. We present two topology changing deformation techniques: one adds a hole, the other deletes a hole. Describing the shape of a hole is beyond the scope of this section. Although shape description is the topic of Chapter 5, it is important to make clear that we do not have a satisfactory solution to the problem of finding a geometric representation of a shape whose topology can be changed by the deformation techniques described here. The examples shown here use an oversampled mesh with explicit topological change at a precise point.

Our philosophy of modeling by space deformation is that the entire surface of the final shape should be visible from the start on the initial shape: there is a bijection from the initial surface to the final surface shown in Figure 3.12(a). No "new" surface is revealed to the artist, as for instance when carving, shown in Figure 3.12(b). Considering this, changing the topology by deformation is more subtle than just "removing" a portion of the shape to make a hole, as with CSG operations (Constructive Solid Geometry, see for instance [Gla89]).

The topology of a manifold is described by its *genus*, loosely speaking the number of holes in the surface. In our view, a modeling system that allows holes to appear and disappear arbitrarily during the modeling of a shape provides low control over the



(a) (b) (c) (d) (e)

Figure 3.11: *All these shapes were modeled with sweepers in at most one hour, starting with a sphere. In (a), the first modeling step was to squash the sphere into a very thin disk.*

targeted result, as for instance with implicit surface modeling[5] [FCG02, AJC02].

The space deformations we have defined are diffeomorphisms of space, and as such they preserve the topology of space, hence they preserve the number of holes of any shape embedded in that space.

However, during the modeling process, the artist may want to increase or decrease the genus of the shape. In this section we define deformations that allow the artist to do that. Combined with the original sweepers, the genus of a shape is precisely controlled.

In his observations on morphogenesis [Koe90], J. Koenderink presents two deformation paths between a sphere and a torus: the "pinched sphere", shown in Figure 3.13(a), and the "strangled torus", shown in Figure 3.13(c). These two paths suggest a pair of tools which are able to increase and decrease the genus of a shape. Eventually, such tools have been created by A. Verroust and M. Finiasz, in the context of editing the vertices of a polyhedral mesh [VF02]. In their implementation, the smoothness of the

---

[5] With implicit surfaces, the effect of uncontrolled topology on a shape is called *unwanted blending.*

surface is not preserved.

Note that pinching is somehow the practical inverse deformation of strangling, as shown in the pairs of sequences (a, d) and (b, c) in Figure 3.13. An intrinsic property of this pair of tools is that the change of topology, insertion or deletion of a hole, is located at points on the surface. This raises the question: how can a point on a surface be located using space deformation, since space deformation is applied blindly to $\mathbb{R}^3$? The answer lies in the use of a space deformation that is discontinuous along a segment for pinching, and a space deformation that is discontinuous along a disk for strangling.

As remarked by A. Verroust and M. Finiasz, pinching and strangling produce at some stage of the operation a shape which is singular for two topologies. Thus a second transformation has to be applied to produce a non-singular shape. The latter doesn't have to be discontinuous, and just helps to pull the shape apart (see the transition from the third to the fourth columns of Figure 3.13).

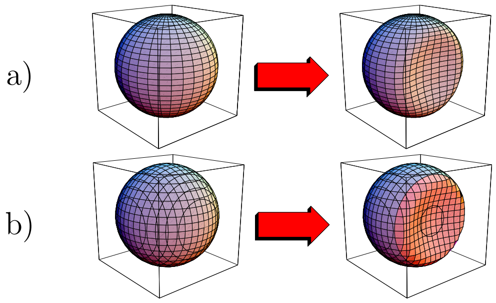


Figure 3.12: *Deforming a surface (a) and Carving a volume (b). In the former, the entire surface of the final shape is visible on the initial shape. In the latter, obtained using an implicit surface, the new surface appears behind the carved out portion (in red).*

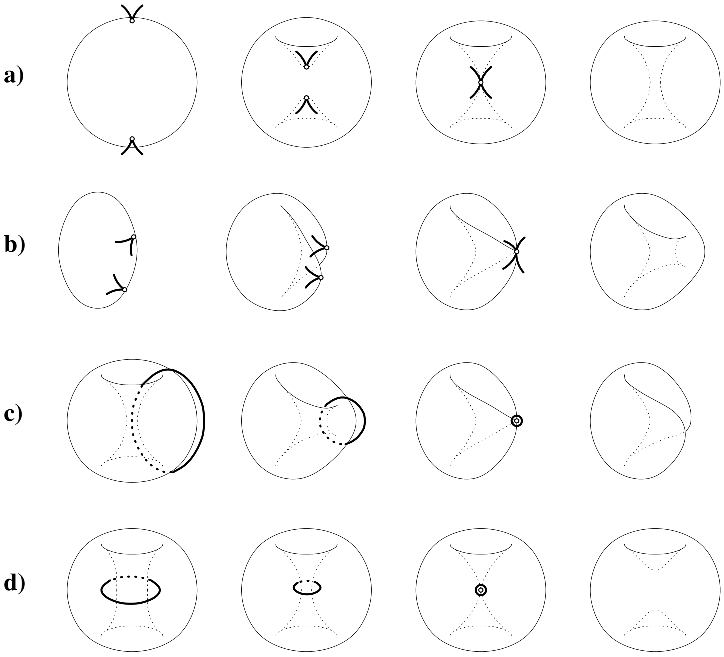


Figure 3.13: *Four deformation paths between a sphere and a torus. The pinched sphere (a) and the strangled torus (c). Similar deformations can be obtained by strangling (b) and pinching (d) the shape from the inside.*

3.5.1 Adding a hole

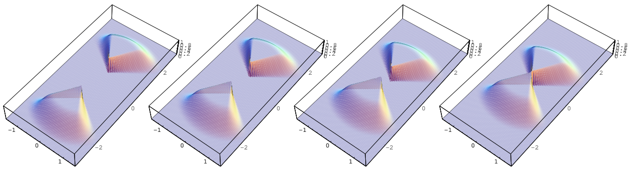


Figure 3.14: *2D plot of the needles' scalar fields. The deformation is discontinuous along the segment joining the tip of the needles.*

The pinch-tool deformation is made of two main steps: pinching and pulling. The pinching deformation is discontinuous only at two points which move toward each other: when they meet, the shape is singular for two topologies. Thus the pinch-tool deformation is discontinuous along a segment. Elsewhere, the deformation is smooth. Let us first focus on the pinching deformation. The pinching deformation itself is made of two simultaneous deformations, that we refer to as *needles*. Each needle is defined with a scalar field discontinuous only at one point, i.e. the tip of the needle. The two needles move toward each other, until the tips meet and poke an infinitely small hole in the surface of the shape. The moving scalar fields are shown in Figure 3.14.
In a local coordinate set, the discontinuous scalar field ϕ of a single needle whose tip is centered at the origin and whose body is aligned with the axis $\vec{\mathrm{e}}_z$ is the following:

$$\phi(\mathrm{p}) = \alpha(\mathrm{p})\rho(\mathrm{p}) \in \mathbb{R} \tag{3.39}$$

where α is a cone-like field discontinuous at $\o$, and ρ is a sphere-like field centered at $\o$ that bounds the effect of a needle, as shown in Figure 3.15 and given as follows:

$$\begin{array}{rcl} \alpha(\mathrm{p}) = \beta(\frac{a(\mathrm{p})-a_{\mathrm{out}}}{a_{\mathrm{in}}-a_{\mathrm{out}}}) & & \rho(\mathrm{p}) = \beta(\frac{r(\mathrm{p})-r_{\mathrm{out}}}{r_{\mathrm{in}}-r_{\mathrm{out}}}) \\ \text{where } a(\mathrm{p}) & = & \arccos(p_z/r(\mathrm{p})) \\ r(\mathrm{p}) & = & \|\mathrm{p}\| \\ \beta(v) & = & \begin{cases} 0 & \text{if } v \leq 0 \\ 1 & \text{if } 1 \leq v \\ v^3(10 - v(15 - 6v)) & \text{otherwise} \end{cases} \end{array} \tag{3.40}$$

Note that the role of β is only to smooth out the transition between the inside and the outside of the needle, and any other C^2 function could be considered. For poking the surface, we choose to follow the simplest path possible: in a local coordinate system, the needles face each other and each tip follows the point $(0, 0, 1-u)^\top$ and $(0, 0, -1+u)^\top$, where u is animated monotonically from 0 to 1, so two fields ϕ weight two opposed translations. Because of the discontinuous point, the number of steps required to animate the needles should be infinite; in practice however a large number is used. After applying the deformation, the local position of the shape between the two needles is known, and the topology can be changed at a single

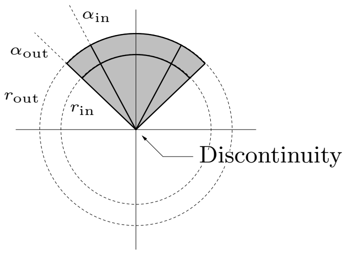


Figure 3.15: *Parameters that describe the size of the discontinuous scalar field of a single needle: angles a_{in}, a_{out} and radii r_{in}, r_{out}.*

point. Changing the topology at a point depends on the shape's description. In the case of a mesh, this has to be done explicitly, thus we proceed as follows (see Figure 3.16):

- find the intersections between the surface's triangles and the segment between the two needle tips.
- if the number of intersections is not 2, abort.
- insert two extremely small triangles on the intersected triangles and tag the 6 new vertices as discontinuous to prevent their edges from splitting or collapsing.
- apply the pinch deformation, splitting an edge only if none of its vertices are discontinuous. Now the two infinitely small triangles meet at a point.
- blow the infinitely small triangles into very small triangles in the plane perpendicular to the needles.
- remove the polygon and insert vertices to make hexagons. Find the vertex correspondence that minimizes the sum of squared distance (remember each hexagon's vertices are ordered in a cycle).
- merge vertices.
- enlarge the hole with a deformation.

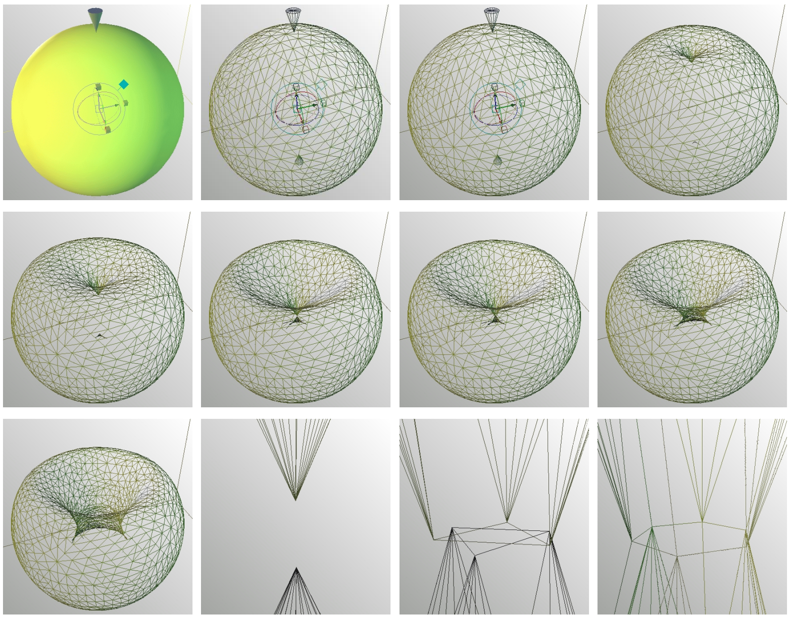


Figure 3.16: *Continuous pinching transformation, on a mesh. Close-up to the singular point.*

After the surface is pinched, the last deformation to apply will pull the shape apart. For this, we use the transformation $M(\mathrm{p})$, in which q is the normal projection of p on the line segment joining the needle tips:

$$M(\mathrm{p})\cdot\mathrm{p} = \mathrm{p} + \phi(\mathrm{p})\frac{\mathrm{p}-\mathrm{q}}{\|\mathrm{p}-\mathrm{q}\|} \qquad (3.41)$$

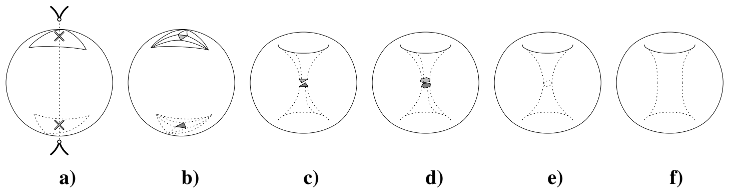


Figure 3.17: *Steps for pinching a mesh.*

with the field:

$$\phi(\mathrm{p}) = \begin{cases} 1 & \text{if } d = 0 \\ 0 & \text{if } d_{\mathrm{out}} \leq d \\ \beta(\frac{d_{\mathrm{out}}-d}{d_{\mathrm{out}}}) & \text{if } d \in (0, d_{\mathrm{out}}) \end{cases} \tag{3.42}$$

where d is the distance from p to the segment joining the needle tips before pinching.

3.5.2 Removing a hole

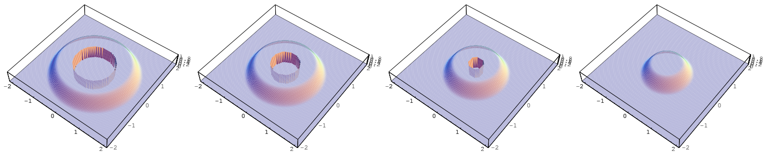


Figure 3.18: *2D plot of the strangling tool's scalar field. The deformation is discontinuous along the circle that contracts on the surface of a disk.*

As with the pinch-tool deformation, the strangle-tool deformation is also made of two main steps: strangling and pulling. The strangling deformation is only discontinuous on a disc. Elsewhere, the deformation is smooth. Let us first focus on the strangling deformation. More precisely, the strangling deformation is discontinuous along a circle that shrinks into a point, at which stage the surface snaps. A 2D slice of the moving scalar field is shown in Figure 3.18. In a local coordinate set, the discontinuous scalar field ϕ used to weight a scale in a discontinuous manner is the following:

$$\phi(\mathrm{p}) = \alpha(\mathrm{p})\rho(\mathrm{p}) \tag{3.43}$$

where α is a cone-like field rotated around z which is discontinuous along a disk, and ρ is a sphere-like field which bounds the effect of ϕ:

$$\alpha(\mathrm{p}) = \beta(\tfrac{a(\mathrm{p})-a_{out}}{a_{in}-a_{out}}) \qquad \rho(\mathrm{p}) = \beta(\tfrac{r(\mathrm{p})-r_{out}}{r_{in}-r_{out}}) \tag{3.44}$$

$$\begin{array}{rcl} \text{where} \quad r(\mathrm{p}) & = & \sqrt{\mathrm{p}^2} \\ a(\mathrm{p}) & = & \arccos((\mathrm{p}-(1-t)\vec{\mathrm{n}})\cdot\vec{\mathrm{n}}) \\ \vec{\mathrm{n}} & = & \frac{\mathrm{p}-(\mathrm{p}\cdot\vec{z})\vec{z}}{\|\mathrm{p}-(\mathrm{p}\cdot\vec{z})\vec{z}\|} \\ \beta(v) & = & \begin{cases} 0 & \text{if } v \leq 0 \\ 1 & \text{if } 1 \leq v \\ v^3(10-v(15-6v)) & \text{otherwise} \end{cases} \end{array} \tag{3.45}$$

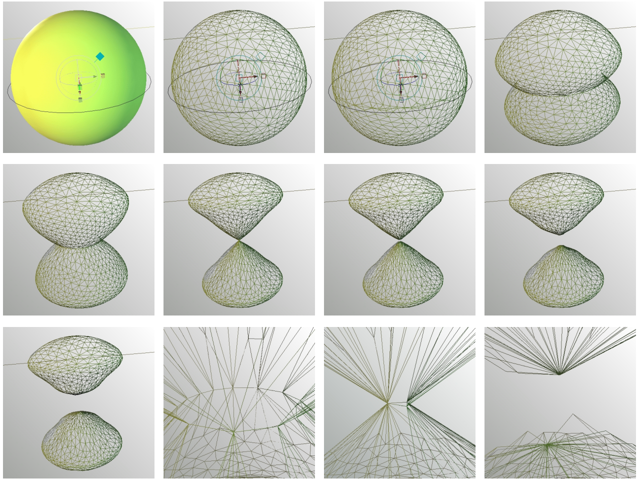
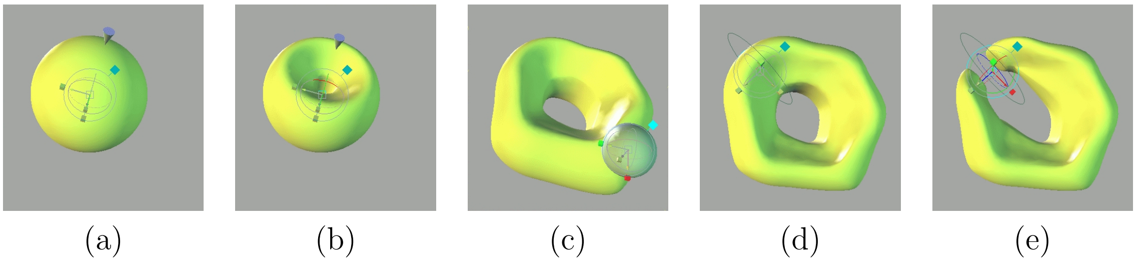


Figure 3.19: *Continuous pinching transformation, on a mesh. Close-up to the singular point.*

3.5.3 Results and limitations

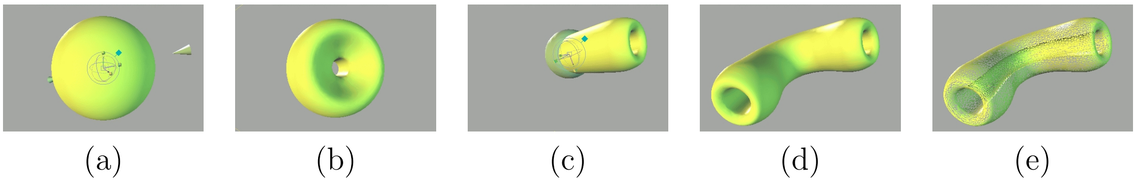
Snapshots of two simple shapes being modeled are shown in Figure 3.20 and Figure 3.21. Note that once a hole is created, it is guaranteed to stay since sweepers are foldover-free.

In the case of a shape represented by an updated mesh, the tools we have defined here are tedious for a programmer to implement, since the change of topology of the singular shape has to be handled explicitly in both pinch and strangle cases. This issue should be solved with further research to find an adequate shape description: this is the topic of Chapter 5. Further research could also be done on a more general topology changing tool, with more general discontinuous features.



(a) (b) (c) (d) (e)

Figure 3.20: *(a, b) Pinch-tool. (c) Single tool sweeper. (d, e) Strangle-tool.*



(a) (b) (c) (d) (e)

Figure 3.21: *Making a hollow tube with a Pinch-tool.*

3.6 Animation

The beginning of this chapter focuses on techniques for modeling shapes using space deformation. Space deformation is also a suitable framework for animating shapes, as done for example in [Bar84, CJ91, MMT97, SF98]. Parameters of the deformation are controlled by time curves. These methods can achieve complex deformations but at a cost: either they are expensive to compute, or a lot of effort is required from the user to specify them.

In this section, we show three possible ways to adapt sweepers to the context of animation. We are primarily interested in animating the modeling of a shape, in order to present a spectator with the artist's view of the modeling process. This is valuable for teaching modeling skills and also as an art form in its own right. The key feature of our approach is the ability to keyframe a deformation with large distortions, to edit the deformation, and finally to render it in high quality.

Metamorphosis, or the animation of a shape undergoing high distortions, has applications ranging from special effects to medical imaging and scientific visualization. The most popular generic technique is mesh morphing [Ale02b], which transforms one mesh into another. Given two shapes, mesh morphing finds some path joining them. The quality of the result is influenced not only by the chosen path, but also by the starting shapes. The motivation of this section is to give to the user more explicit control of the path, which he/she will specify for achieving the desired shape. Space deformation provides a convenient formalism for defining the deformation of a shape, although this technique has not yet been used for animating a shape undergoing great stretching and twisting.

3.6.1 Three techniques for animation

In this section, we reformulate *sweepers* in a way that is suitable for animation. We recall that a sweeper is a geometric tool together with a motion path. The basic idea is that the tool is placed somewhere in the region of a shape to be deformed and moved along the path. The motion drags a part of space with the tool subject to rules that prevent space foldover.

Space deformation provides a formalism for specifying any modeling operation by successively deforming the space in which an initial shape, $S(k_0)$, is embedded. In this section, the reader can interpret k as time. A deformed shape is given by the *modeling equation*, that is Equation (3.1). We rewrite it using the parameter k instead of t, since for the purpose of animating, the deformation parameter is not necessarily a monotonic function of time:

$$S(k_n) = \left\{ \mathop{\Omega}_{i=0}^{n-1} f_{k_i \mapsto k_{i+1}}(\mathrm{p}) \mid \mathrm{p} \in S(k_0) \right\} \tag{3.46}$$

where $f_{k_i \mapsto k_{i+1}} : \mathbb{R}^3 \to \mathbb{R}^3$ is the space deformation that deforms a point p of the shape $S(k_i)$ into a point of the shape $S(k_{i+1})$. Since we want to animate the modeling process, we need the deformation to be continuous not only in space, but also in k, over the interval $[k_0, k_n]$. For this we reformulate Equation (3.46) into the *animation equation*:

$$S(k) \quad = \quad \left\{ \overset{n-1}{\underset{i=0}{\Omega}} f^k_{k_i \mapsto k_{i+1}}(\mathrm{p}) \mid \mathrm{p} \in S(k_0) \right\} \tag{3.47}$$

$$\text{where} \quad f^k_{k_i \mapsto k_{i+1}}(\mathrm{p}) \quad = \quad \begin{cases} \mathrm{p} & \text{if } k \leq k_i \\ f_{k_i \mapsto k_{i+1}}(\mathrm{p}) & \text{if } k > k_{i+1} \\ f_{k_i \mapsto k}(\mathrm{p}) & \text{otherwise} \end{cases}$$

where $f_{k_i \mapsto k} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is the space deformation that deforms the shape $S(k_i)$ into the shape $S(k)$. Each deformation $f^k_{k_i \mapsto k_{i+1}}$ has to be continuous in $k \in [k_i, k_{i+1}]$. This can be achieved conveniently with sweepers, as we show below.

The deformation defined by a tool moving from M_{k_i} to $M_{k_{i+1}}$ is swept using s steps, as given by Equation (3.19), that subdivide the interval $(k_i, k_{i+1}]$ into "small enough" steps $\{\kappa_0, \ldots \kappa_s\}$. Let us denote $M = M_{k_{i+1}} M^{-1}_{k_i}$ the transformation matrix. The sweeper modeling-deformation is thus the composition of s functions:

$$f_{k_i \mapsto k}(\mathrm{p}) = \overset{s-1}{\underset{j=0}{\Omega}} f^k_{\kappa_j \mapsto \kappa_{j+1}}(\mathrm{p}) \tag{3.48}$$

$$\text{where} \quad f^k_{\kappa_j \mapsto \kappa_{j+1}}(\mathrm{p}) = \begin{cases} \mathrm{p} & \text{if } k \leq \kappa_j \\ f_{\kappa_j \rightarrow \kappa_{j+1}}(\mathrm{p}) & \text{if } k > \kappa_{j+1} \\ f_{\kappa_j \rightarrow k}(\mathrm{p}) & \text{otherwise} \end{cases}$$

In order to use sweepers for animation, we have to define $f^k_{\kappa_i \mapsto k}$ for $k \leq (\kappa_j, \kappa_{j+1})$. Since M_k can be assumed to be defined for all k, adapting sweepers to animation is as simple as substituting $M_{k_{i+1}}$ for M_k in Equation (3.48). Figure 3.22 illustrates the substitution of M_{k_1} for M_k.

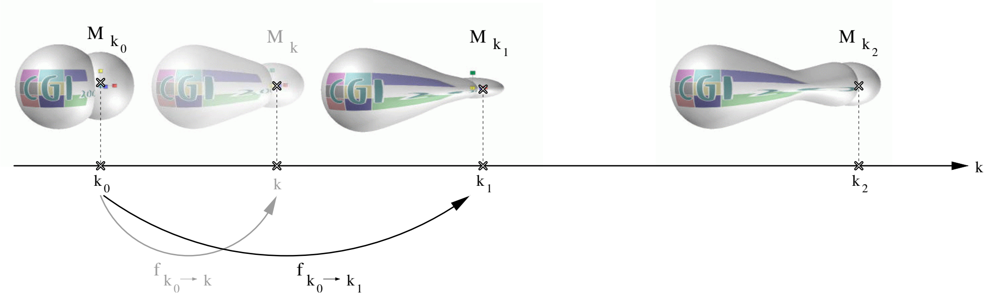


Figure 3.22: *Since the movement of the tool is continuous in k, so is the deformation. In this example, the tool's translation and scaling are animated.*

Thus, the sweeping of a tool defines a deformation that is smooth in both space and time; smooth meaning the field ϕ_k is C^2 in space, and the matrix M_k is C^1 in k.

The movements of tools have parameters of their own, which has been assimilated to the time parameter in the case of modeling. However, it can be used independently for animation, providing several ways of animating a shape with a sweeping deformation, as we show in the next sections.

Straight

By leaving the tool at a fixed position in the modeling space and animating only its destination through time, the animator only has to specify a curve (see Figure 3.23), the effect being to grab a portion of an object and to displace it straight from its original position to an animated target position. In Figure 3.24, a tool grabs a head smoothly, while the neck stays in place.

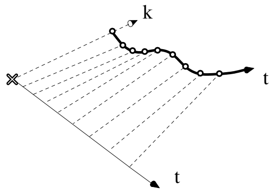


Figure 3.23: *The dark curve represents the animated portion of the modeling space, controlled by the user. The straight lines aligned with k bring the shape straight from modeling space to animated space.*

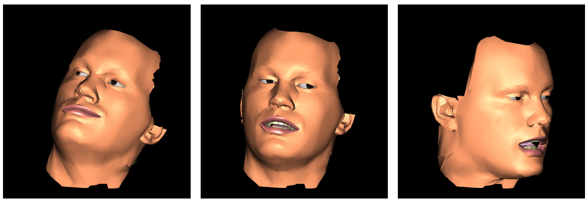


Figure 3.24: *Deformation of a neck using a straight deformation. A rotation turns the head, while the neck stays still. The facial expressions are procedural [KKM03].*

Sliding-straight

By specifying a deformation in a local coordinate set, and animating the coordinate set, the animator can slide a deformation through the scene, as shown in Figure 3.25. For instance, if the deformation is a scale, it can be used for animating a bulge moving inside an object. This approach has been used for instance by [CJ91], where instead of moving the deformation relative to the object, it is the object that is moved relative to the deformation. Also, we have implemented it with sweepers as a Maya plugin, which was used in a short animation as shown in Figure 3.26.

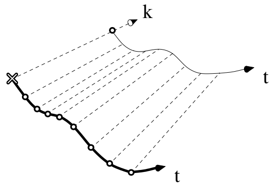


Figure 3.25: *Sliding a deformation through the scene. The curve controlled by the user is symbolized in thick black.*

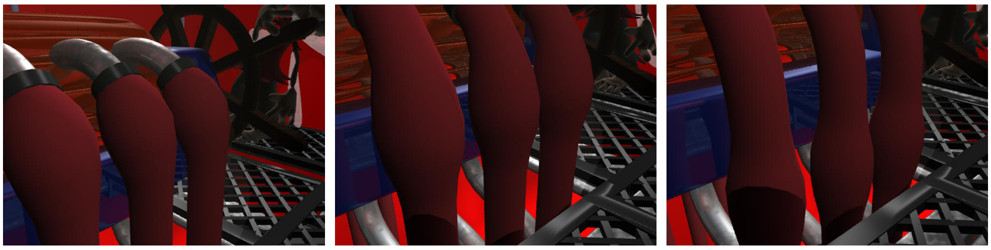


Figure 3.26: *Screenshots from a short animation using a simple deformation plugin under Maya. The bulge is animated by moving its locator inside the pipe.*

Modeling

By identifying the deformation parameter k to the time parameter t, a user can easily specify and control an *animated sweeper*. Every new deformation is composed with the previous one, thus building up an increasingly complex function. As shown in Figure 3.27, each function is not animated and is constant in t. The control-points on the line $k = t$ in this figure correspond to key positions M_{t_i}.

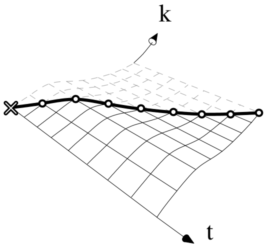


Figure 3.27: *Animating the modeling deformation. To each control-point corresponds a constant function, parallel with axis t. The only curve controlled by the user is symbolized in thick black.*

As time increases, the complexity of the deformation increases too. To provide quick feedback to the user, we propose in the next sections a data flow structure illustrated by an implementation.

3.6.2 Focus on animating a modeling

Let us place ourselves in the context of a user working on an animation. The sweeping tool is controlled through eleven curves:

- 3 translation curves along X, Y, Z (float)
- 3 rotation curves around X, Y, Z (float)
- 3 scale curves along X, Y, Z (float)
- Thickness curve (float): the constant λ in Equation (3.3)

- Active curve (boolean): if false, $f_{t_i \mapsto t} = I$ regardless of the ten other curves, otherwise the tool behaves normally. This is useful for moving the tool around the scene without interfering with the shape.

By modifying the control points of a curve driving the sweeping tool, the user is transparently modifying the animation Equation (3.47). An example is shown in Figure 3.28.

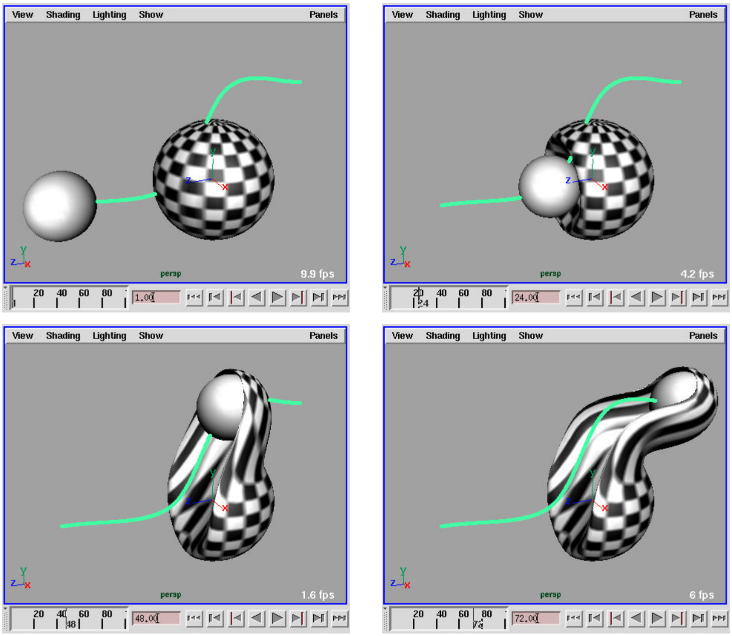


Figure 3.28: *Control of a deformation by keyframing the position of a tool. The green curve is the tool's motion trail.*

It would be rather inefficient if the shape $S(t)$ was to be recomputed from the initial shape $S(t_0)$ every time a key between t_0 and t has been modified, or every time the user selects a different time in the time slider. To prevent this, we propose an efficient architecture that caches intermediate shapes so that the user can work in a reasonable amount of time, real-time in most cases.

We suggest placing a cached shape at every time where at least one key has been set. As shown in Figure 3.29, the list of cached shapes does not necessarily correspond to the control points of a curve.

Graph nodes

The cached shapes are handled by nodes in a scene graph, and are updated only when necessary through the connections between the nodes. In fact, since our system has been implemented in Maya 5.0, the nodes we describe in the following sections are customized Dependency Graph (DG) nodes. The reader is referred to [Gou03] for a description on how data is pulled through the connections only when required.

Shape node: The shape node is responsible for displaying the current shape $S(t)$. It possesses an array of connections to the cached nodes' output shapes, sorted by increasing time, and a connection to the current time. When the current time changes,

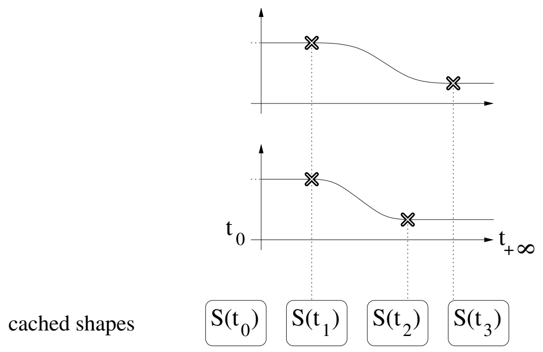


Figure 3.29: *Example of a configuration of cached shapes relative to the curves.*

the shape node determines which interval $(t_{i-1}, t_i]$ the value t belongs to, and asks cache node i for its output shape.

Cache nodes: A cache node has, as input, the cached shape of the previous cache node. If there is no previous cache node, it generates shape $S(t_0)$. A cache node is responsible for computing and storing the cached shape $S(t_i)$ when requested by the next cache node, and is also responsible for computing output shape $S(t)$ when requested by the shape node.

Tool node: Draws the tool in the real time modeling view, and will not be rendered by default.

Tool transform node: The tool transform node connects all the animation curves, and provides the matrix positions M_t to the cache nodes, so that they can deform the shape. The tool transform node also provides this matrix to the tool node so that it can be drawn at the right position.

The list of connected cache nodes represents a $4D$ buffer, and holds deformed shapes along time. This buffer is used to compute a single shape: the one displayed in the interface, at the current time.

Initialization

At initialization, there are two cache nodes at time t_0 and $t_{+\infty}$. The cache node at time t_0 contains the initial shape, before deformation. The cache node at time $t_{+\infty}$ contains the current final shape. The curves are clear of any key. By themselves, the two extremity caches define an inbetween function $f_{t_0 \mapsto t_{+\infty}}$ which is the identity: $S(t) = S(t_0)$. Keys can then be inserted on the curves while the buffer structure updates.

Operations

The operations are performed on the tool as if it were a classic animated object, through the eleven curves. These operations are performed on the scene graph via curve editing callbacks.

Key insertion: When a control point is inserted on a curve at time t_i, we first check whether a cache node already exists. If it does, we mark the node dirty. Otherwise, we insert a node in the node list.

Key deletion: When a control point is deleted on a curve at time t_i, we first check whether another curve has a control point at that time. If it does, we mark the node dirty. If not, we remove the node from the list.

Key modification: This is done by deletion followed by insertion.

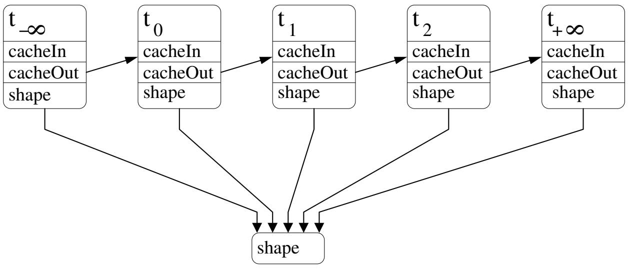


Figure 3.30: *Simplified data graph.*

Sampling the matrix curves

Maya provides a mechanism for C^1 interpolation of matrices, different from the one in [Ale02a], which we use for sweeping transformations. Thus, using the foldover-free condition with Maya's interpolation is incorrect. To solve this, we sample the matrix at every frame using Maya's interpolation scheme, and we sweep the transformation within that interval using the foldover-free condition. We have observed that, most of the time, the number of substeps is equal to one.

Results

We have implemented the structure in C++ using the Maya 5.0 API [Gou03]. The only difficulty of the implementation in Maya is the absence of a callback for detecting which control points have been modified. However, since it is possible to set a callback to know which curve has been modified, it is possible to circumvent this issue by duplicating the curve nodes (eleven, for each parameter). By counting the number of points, it is possible to know if a key has been modified, inserted or deleted, and by comparing the control points one by one, the modified control points of the curve can be found. Also, for rendering a custom shape, it is necessary to pass the custom mesh to a built-in Maya node.

Regarding interactivity, the system is fast and allows the user to specify and modify an animated sweeper with ease. Once all the cache nodes have computed their cached shape, playing the animation in real time is just a matter of deforming the shape within small time intervals $(t_i, t_{i+1}]$. Also, every cache is computed or updated only when required, that is when the current time in the time slider is greater than t_{i+1}. In order to precompute all the cached shapes, or update all the cached shapes when the first key has been modified, the user can select the last time t_n on the time slider. This operation can be slow if there are many deformations. However most of the time, the modeling is done locally both in space and time, allowing the computation to be done in real time.

The title page animation of a sphere morphing into an Anubis statue has been done conveniently with the technique described, and then rendered in high quality. The first step flattens the bottom of the sphere using a vertical scaling. The tool is then disabled and placed on another part to pull out the body of the statue. The rest of the morphing is also done by pulling, scaling, twisting, disabling and enabling the tool.

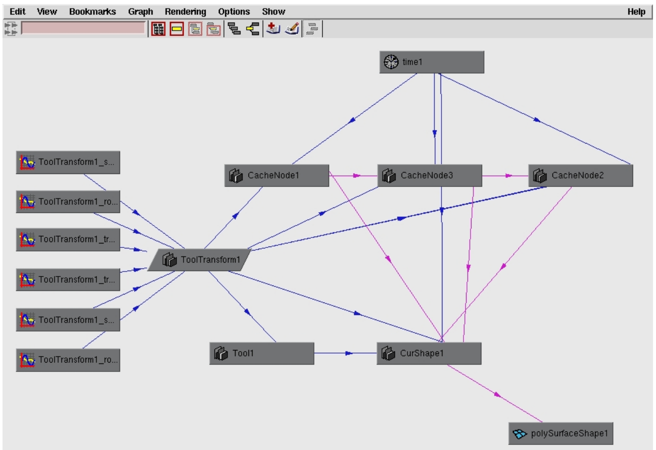


Figure 3.31: *Snapshot of the Maya dependency graph. Three keys have been set on translation X, rotation X and scale X, at the same time parameter. To this key corresponds a single cache node. Each curve has a duplicate used to modify the graph when a curve is edited.*

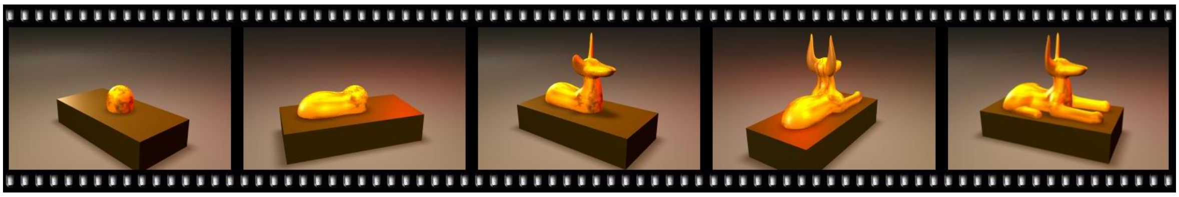


Figure 3.32: *Snapshots of the Anubis animation. The 2D texture (a noise) has been stretched with the deformation.*

3.7 Conclusion

Sweepers is a new class of smooth and normalized space deformations that are predictable and preserve the shape's coherency: deformation operations are defined by combining transformations non-linearly in matrix logarithmic space, which allows us to parametrize and decompose the deformations. In the case of simple transformations for single tools, fast expressions can be derived to be used for real time modeling. We have shown that a space deformation can both prevent foldovers and change topology: these are not contradictory properties. Sweepers can also be extended to animation. Because of the inherent motion parameter of sweepers, this extension is straightforward.

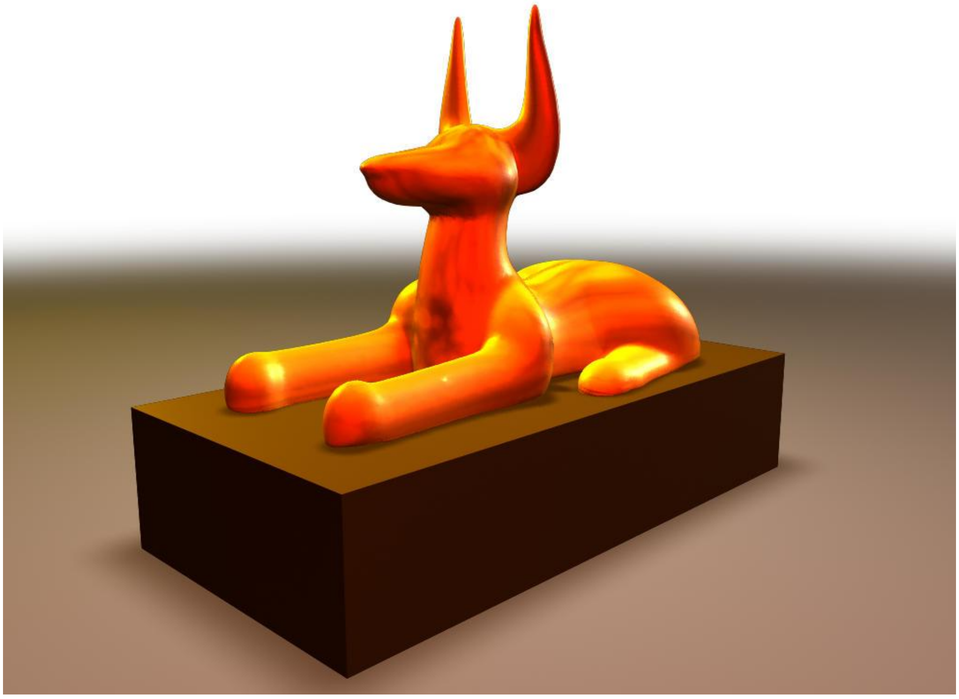
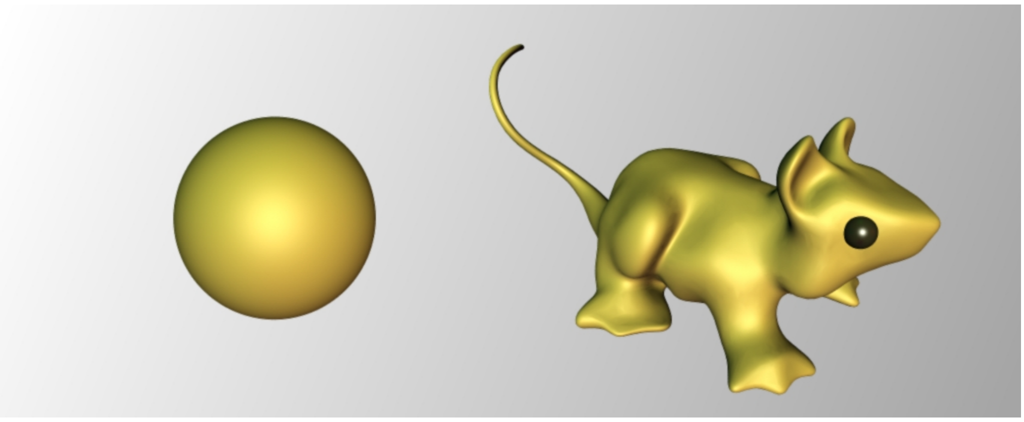


Figure 3.33: *Close-up: last frame of the Anubis animation.*

Chapter 4

Volume Preserving Modeling



The work presented in Section 4.1 was presented as a technical sketch at SIGGRAPH 2004 [ACWK04b], and extends the work published in Pacific Graphics 2004 [ACWK04a], acknowledged with a best paper award.

In a non-virtual context, when an artist models with a soft modeling compound, one of the most important factors which affects the artist's technique is the amount of available material. This aspect was ignored in the previous chapters. The notion of an amount of material is not only familiar to professional artists, but also to children who play with Play-Doh®at kindergarten and adults through everyday life experience.

In the context of modeling virtual shapes, in order to take advantage of people's familiarity with the concept of an amount of material, a challenge for Computer Graphics is to provide a technique that convinces the artist of the existence of a *virtual material.* This perfects the illusion of a virtual shape behaving in accordance with a clay-modeling metaphor. Also, we have observed that modeling by preserving the available amount of material produces shapes with a style, i.e. fluid-like features, that other virtual modeling methods can only achieve with more effort.

Volume preservation has been recognized for a long time in animation as a desirable property for the animation of believable animal and human characters [TJ81]. [PB88] use constrained optimization methods for objects discretized into lattices. [DC95] use controllers for maintaining the implicit surface that coats a set of particles to a constant volume during deformation. [FF01] achieve incompressibility in water simulation by maintaining a divergence-free velocity field, thanks to the Poisson equation. [TBHF03] rely on finite volume methods to simulate quasi-incompressible materials such as muscular tissue.

Volume preservation has also been considered as a very useful constraint in the intuitive modeling of shapes. [RSB95] propose an optimization method to adjust the control points of the popular free form deformations (FFD) [SP86], but the technique is restricted to shapes represented with tensor-solids. [HML99] also adjust FFD control points, but their method does not allow local editing. [AB97] propose a volume preserving space deformation based on a model called DOGME. The deformation does not have a local support, and requires the computation of the shape's volume. [BK03] preserve a volume only between the surface and a base surface. [DC03] introduce mass-preserving local and global deformations for shapes represented by a mass-density field sampled in a grid.

The limitation of existing methods is either that they only apply to a specific type of geometric representation, or they only apply to shapes whose volume can be computed. This chapter introduces two different approaches for defining a volume-preserving deformation that preserve volume independently from the shape description. In Section 4.1, our first method is based on the observation that a rotation modulated by a scalar decreasing isotropically from the axis of rotation is implicitly volume-preserving. In Section 4.2, our second method is based on equations borrowed from incompressible fluid dynamics, for which incompressibility is expressed explicitly.

Limit of scope: In Computer Graphics, the preservation of volume is limited by three main factors: memory capacity, computation time and numerical accuracy. Although properly speaking there is only one way to preserve volume, which is when the volume remains constant, in this chapter we may refer to the accuracy of a volume preservation technique to quantify how closely the volume is preserved.

4.1 Swirling-sweepers

Swirling-sweepers is our first method that preserves volume: it has local support, prevents local and global self-intersection of the surface and does not require any volume computation. Most importantly, using the method is simple: the artist only has to provide the trajectory of a point, for instance with a mouse. All of these properties are necessary for interactive modeling if the user is to have the impression that he or she is shaping a real material. Our method is the first to implement all four. Swirling-sweepers is based on closed-forms, and is much more accurate than the technique presented in Section 4.2.

4.1.1 A basic deformation

We define a particular case of sweeper, a *swirl*, by using a point tool, c, together with a rotation of angle, θ, around an axis $\vec{\mathrm{v}}$ (see Figure 4.1). A scalar function, ϕ, and a deformation are defined as before. Informally, a swirl twists space locally around the axis, $\vec{\mathrm{v}}$, without compression or dilation. We prove in Appendix B that a swirl preserves volume.

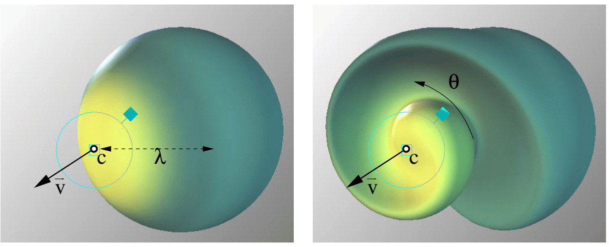


Figure 4.1: *The effect on a sphere of a swirl centered at* c, *with a rotation angle* θ *around* $\vec{\mathrm{v}}$. *The two shapes have the same volume.*

4.1.2 Combining for complexity

Many deformations of the above kind can be naively combined to create a more complex deformation

$$\begin{aligned} f(\mathrm{p}) &= \left(\bigoplus_{i=0}^{n-1} (\phi_i(\mathrm{p}) \odot M_i)\right) \cdot \mathrm{p} \\ &= \exp\left(\sum_{i=0}^{n-1} (\phi_i(\mathrm{p}) \log M_i)\right) \cdot \mathrm{p} \end{aligned} \tag{4.1}$$

It is important here to remark that the above blending is *not* the blending formula of simultaneous tools defined in Equation (3.16), and only uses simple weights. The reason for using the above simple blending equation as opposed to Equation (3.16) is that the latter modulates the amount of the individual transformations locally, and attempting to control the volume with it would be inappropriate. We provide a convenient way for the artist to input n rotations, by specification of a single translation $\vec{\mathrm{t}}$. Let us consider n points, c_i, on the circle of center h, and radius r lying in a plane perpendicular to $\vec{\mathrm{t}}$. To these points correspond n consistently-oriented unit tangent vectors $\vec{\mathrm{v}}_i$ (see Figure 4.2). Each pair $(\mathrm{c}_i, \vec{\mathrm{v}}_i)$, together with an angle θ_i, define a rotation. Along with radii of influence $\lambda_i = 2r$, we can define n swirls. The radius of the circle, r, is left to the user to choose. The following value for θ_i will transform h exactly into $\mathrm{h} + \vec{\mathrm{t}}$ (see Appendix B.2).

$$\theta_i = \frac{2\|\vec{\mathrm{t}}\,\|}{nr} \tag{4.2}$$

With this information, the deformation of Equation (4.1) is now a tool capable of transforming a point into a desired target. We show in Figure 4.2 the effect of the tool for different values of n; in practice, we use 8 swirls.

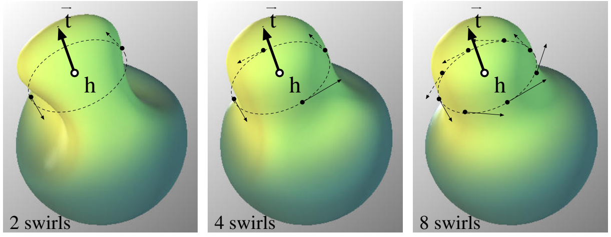


Figure 4.2: *By arranging n basic swirls in a circle, a more complex deformation is achieved. In the rightmost image: with 8 swirls, there are no visible artifacts due to the discrete number of swirls.*

Preserving coherency and volume: If the magnitude of the input vector $\vec{t}$ is too large, the deformation of Equation (4.1) will produce a self-intersecting surface, and will not preserve volume. The reason for self-intersection is explained in detail in Chapter 3. The volume is not preserved because the blending operator, $\oplus$, blends the transformation matrices, and not the deformations. To correct this, it is necessary to subdivide $\vec{t}$ into smaller vectors. Ideally the number of steps should be infinite, but this remark could be made about any animation techniques where time is integrated in finite steps. To make computation cost reasonable, we propose a lower bound to the number of steps proportional to the velocity and inversely proportional to the size of the tool:

$$s = \max(1, \lceil 4\|\vec{t}\,\|/r \rceil) \tag{4.3}$$

As the circle sweeps space, it defines a cylinder. Thus the swirling-sweeper is made of ns basic deformations. Figure 4.3 illustrates this decomposition applied to a shape.

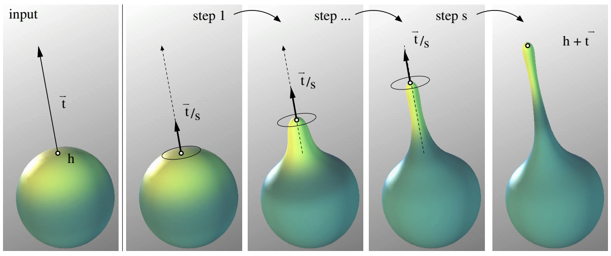


Figure 4.3: *A volume preserving deformation is obtained by decomposing a translation into circles of swirls. Three steps have been used for this illustration. As the artist pulls the surface, the shape gets thinner. The selected point's transformation is precisely controlled.*

4.1.3 Swirling-sweepers algorithm

We summarize here the swirling-sweepers algorithm, in which the function μ is the piecewise polynomial defined in Equation (3.2):

```
Input point, $h$, translation, $\vec{\mathrm{t}}$, and radius, $r$
Compute the number of required steps, $s$
Compute the angle of each step, $\theta_i = \frac{2\|\vec{\mathrm{t}}\|}{nrs}$
Precompute the matrices $M_{i,j}$
for each step $j$ from 0 to $s-1$ do
  for each point p in the tool's bounding box do
    $M = 0$
    for each swirl $i$ from 0 to $n-1$ do
      $M += \mu_{2r}(\|\mathrm{p} - \mathrm{c}_{ij}\|) \log M_{i,j}$
    end for
    $\mathrm{p} = (\exp M) \cdot \mathrm{p}$
  end for
end for
```

The point c_{ij} denotes the center of the i^{th} swirl of the j^{th} ring of swirls. For efficiency, a table of the basic-swirl centers, c_{ij}, and a table of the rotation matrices, $\log M_{i,j}$, are precomputed. We have a closed-form for the logarithm of the matrix, saving an otherwise expensive numerical approximation:

$$\vec{\omega} \;=\; \theta_i \vec{\mathrm{v}}_i \qquad \vec{\mathrm{m}} \;=\; c_{i,j} \times \vec{\omega} \tag{4.4}$$

$$\log M_{i,j} \;=\; \begin{pmatrix} 0 & -\omega_z & \omega_y & m_x \\ \omega_z & 0 & -\omega_x & m_y \\ -\omega_y & \omega_x & 0 & m_z \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.5}$$

Note that for the sake of efficiency, since $\log M_{i,j}$ is sparse and mostly anti-symmetric, we handle these matrices as pairs of vectors, $\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$. Once M is computed, we use a closed-form for computing $\exp M$. Since the matrix M is a weighted sum of matrices $\log M_{i,j}$, the matrix M is as sparse as the matrix in Equation (4.5). We show in Chapter 7 that M is in fact the logarithm of a twist, and can be represented with a pair, $(\vec{\omega}_M, \vec{\mathrm{m}}_M)$. If $\vec{\omega}_M = 0$, then $\exp M$ is a translation by the vector $\vec{\mathrm{m}}_M$. Else, if the dot product $\vec{\mathrm{m}}_M \cdot \vec{\omega}_M = 0$, then $\exp M$ is a rotation of center $\vec{\mathrm{c}}$, angle θ and axis $\vec{\mathrm{v}}$, as given by the following:

$$\mathrm{c} \;=\; \frac{\vec{\omega}_M \times \vec{\mathrm{m}}_M}{\vec{\omega}_M^2} \qquad \theta \;=\; \|\vec{\omega}_M\| \qquad \vec{\mathrm{v}} \;=\; \vec{\omega}_M / \theta \tag{4.6}$$

Finally, in the remaining cases, we let $l = \|\vec{\omega}_M\|$, and we use:

$$\exp M = \mathrm{I} + M + \tfrac{1-\cos l}{l^2} M^2 + \tfrac{l - \sin l}{l^3} M^3 \tag{4.7}$$

Efficiency

Applying the exponential of the matrix to a point does not require us to compute explicitly the exponential of the matrix. The formulas are given in Section 7.4.3. Also, symmetrical objects can be easily modeled by introducing a plane of symmetry about which the swirls are reflected (see Figure 4.9).

4.1.4 Convolution swirling-sweepers

The influence function μ_λ used to define a swirl in Equation (3.3) can be chosen from a set which is infinite. It is therefore possible to define families of swirling-sweepers based on the choice of this function. In this section, we show that one loop in the procedure proposed in the previous subsection may be removed, by making infinite the number of swirls in Equation (4.1). This is achieved mathematically by integrating swirls around the circle. In order for this to be possible, instead of defining the influence function ϕ_t in Equation (3.3) using a piecewise polynomial, $\mu_\lambda \circ d_t$, the influence function ϕ_t must be defined as $\gamma_\lambda \circ d_t$, where γ_λ is an alternative function whose integral over a circle has a closed-form. We use the following:

$$\gamma_\lambda(d) = \frac{1}{(1 + (d/\lambda)^2)^2} \tag{4.8}$$

The coefficient λ is a user-defined parameter, which is a cheap means of giving "properties" to the material, shown in Figure 4.4. λ is proportional to the gradient of the amount of rotation around the ring. The function of Equation (4.8) is numerically stable, since it is continuous and bounded at the origin. Let us parameterize the circle with α. The center of the swirls are $\mathrm{c}(\alpha) = \mathrm{c} + r\cos\alpha\vec{\mathrm{x}} + r\sin\alpha\vec{\mathrm{y}}$, while the axes of the swirls are $\vec{\mathrm{v}}(\alpha) = -\sin\alpha\vec{\mathrm{x}} + \cos\alpha\vec{\mathrm{y}}$. The following expression gives the deformation of a point:

$$f(\mathrm{p}) = \exp\left(\frac{1}{2\pi r}\oint_\alpha (\gamma_\lambda(\|\mathrm{p}-\mathrm{c}_\alpha\|)\ \log M_\alpha\ \mathrm{d}\alpha)\right) \cdot \mathrm{p} \tag{4.9}$$

where M_α denotes the 4×4 rotation matrix of center $\mathrm{c}(\alpha)$, axis $\vec{\mathrm{v}}(\alpha)$ and angle θ. The matrix $\log M_\alpha$ is represented by a pair of vectors:

$$\begin{array}{rcl} \log M_\alpha & = & \langle \theta\vec{\mathrm{v}}(\alpha), \theta\mathrm{c}(\alpha) \times \vec{\mathrm{v}}(\alpha)\rangle \\ & = & \langle \theta\vec{\mathrm{v}}(\alpha), \theta r\vec{\mathrm{z}}\rangle \end{array} \tag{4.10}$$

Using the symbolic computational engine Mathematica®, the integral in Equation (4.9) on the circle gives the following expression:

$$\Phi_{\text{circle}}(\mathrm{p}, \mathrm{c}) = \frac{1}{2\pi r}\int_\alpha \Phi_{\text{swirl}}(\mathrm{p}, \alpha)\ \mathrm{d}\alpha = \theta b\langle\vec{\eta}, \mathrm{c} \times \vec{\eta} + \vec{\mathrm{z}}\ a\rangle \tag{4.11}$$

$$\begin{array}{rrcl} \text{where} & a & = & (\mathrm{p}-\mathrm{c})^2 + \lambda^2 + r^2 \\ & b & = & \frac{2\lambda^4 r}{(a^2 - r^2\vec{\eta}^2)^{3/2}} \\ & \vec{\eta} & = & 2\vec{\mathrm{z}} \times (\mathrm{p}-\mathrm{c}) \end{array}$$

This integration is very similar to the one used in a different area of shape modeling, i.e. convolution implicit surfaces [MS98]. The following is the deformation of a point by a ring of swirls:

$$f(\mathrm{p}) = \exp(\Phi_{\text{circle}}(\mathrm{p}, \mathrm{c})) \cdot \mathrm{p} \tag{4.12}$$

The angle θ required to transform the point c into the point $c + \vec{\mathrm{t}}$ is the following:

$$\theta = \frac{\|\vec{\mathrm{t}}\|(\lambda^2 + r^2)^2}{2\lambda^4 r} \tag{4.13}$$

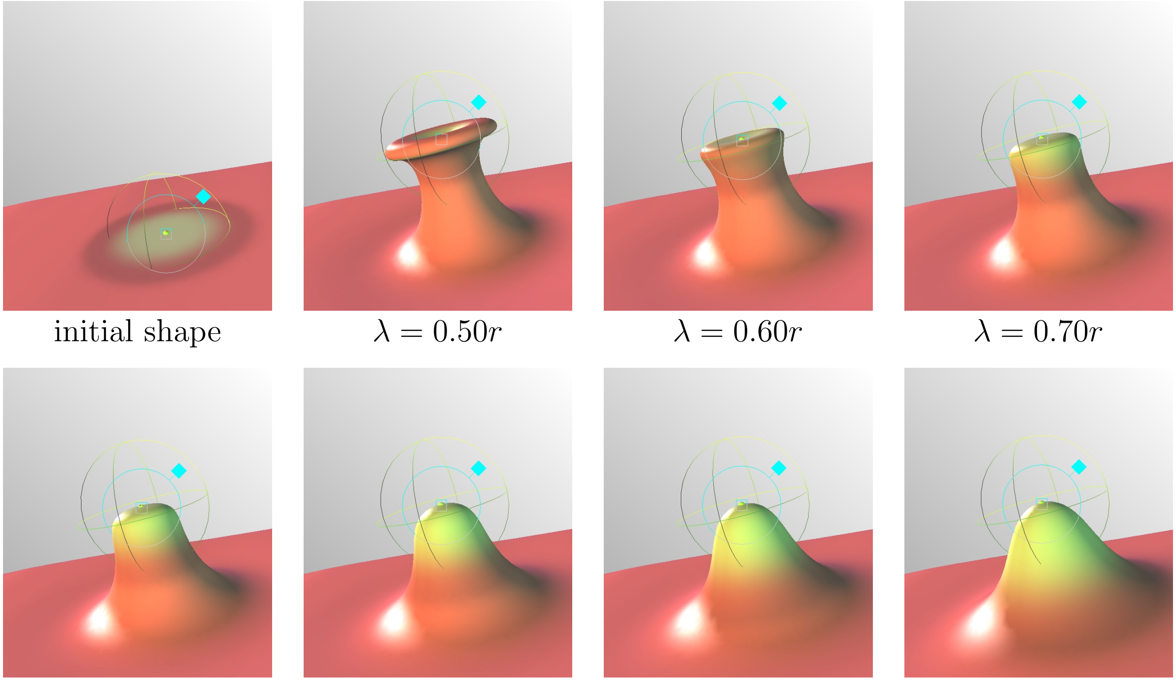
For the number of steps, Equation (4.3) may be used.

Bounding box: Since the function γ_λ does not have a local support, there is no volume outside which the deformation has no influence. For speeding up the deformation, it is however useful to define a bounding box. This bounding box can be specified by considering a volume outside which the deformation can be neglected. If we assume the ring is small compared to the shape, we use the center of the ring c, and approximate the influence of the ring at a point p by the following:

$$\iiint_{\mathrm{p}} \gamma_\lambda(\mathrm{p}-\mathrm{c})\ \mathrm{dp} \approx \frac{2\pi r}{(1+(\|\mathrm{p}-\mathrm{c}\|/r)^2)^2} \tag{4.14}$$

The radius for which this scalar function is smaller than a threshold, ϵ, specifies a bounding sphere around the ring. The following gives that radius:

$$r\sqrt{\sqrt{\frac{2\pi r}{\epsilon}}-1} \tag{4.15}$$



initial shape

$\lambda = 0.50r$

$\lambda = 0.60r$

$\lambda = 0.70r$

$\lambda = 0.80r$

$\lambda = 1.00r$

$\lambda = 1.20r$

$\lambda = 1.50r$

Figure 4.4: *Convolution swirling-sweepers: effects of control coefficient* r *for a movement that pulls the shape. A value of* $1.20r$ *is a good practical choice. Values below* $0.80r$ *are not interesting from a clay-like modeling point of view. Note that the control of the size of the tool,* r*, is left to the artist.*

4.1.5 Convolution swirling-sweepers algorithm

With a closed-form solution to the integration over a ring, one loop has been removed when compared to the algorithm in Section 4.1.3:

Input point, h, translation, $\vec{\mathrm{t}}$, and radius, r

```
Compute the number of required steps, $s$
Compute the angle of each step, $\theta = \frac{\|\vec{\mathrm{t}}\|(\lambda^2+r^2)^2}{2\lambda^4 r}$
for each step $j$ from 0 to $s-1$ do
    for each point p in the tool's bounding box do
        $\mathrm{p} = \exp(\Phi_{\mathrm{circle}}(\mathrm{p}, \mathrm{c}_j)) \cdot \mathrm{p}$
    end for
end for
```

A possible direction for future work might be to look for a closed-form for the composition of rings, in order to get rid of the step loop.

4.1.6 Star-shaped swirling-sweepers

Swirling-sweepers allow the artist to control precisely the path of a selected point in the scene. It is of great interest to the artist to be able to control a volumetric region of space, defined for instance by the shape of a tool. To achieve this, we first need to define an inverse to the volume-controlled space-deformation of P. Decaudin [Dec96], that is presented in Chapter 2. The function and its inverse are:

$$f_{\mathrm{D}} : \mathrm{p} \rightarrow \mathrm{c} + \left(1 + \frac{\rho^3}{\|\mathrm{p}-\mathrm{c}\|^3}\right)^{1/3} (\mathrm{p}-\mathrm{c}) \tag{4.16}$$

$$f_{\mathrm{D}}^{-1} : \mathrm{p} \rightarrow \mathrm{c} + \left(1 - \frac{\rho^3}{\|\mathrm{p}-\mathrm{c}\|^3}\right)^{1/3} (\mathrm{p}-\mathrm{c}) \tag{4.17}$$

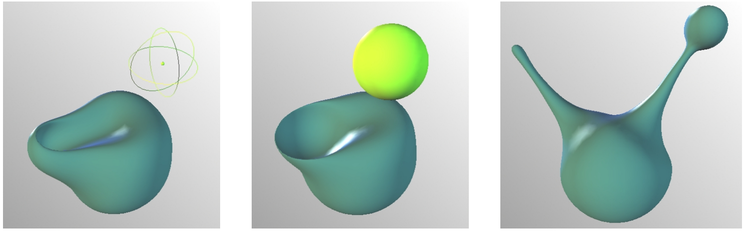
This inverse function satisfies the simple relation $f_{\mathrm{D}}^{-1} \circ f_{\mathrm{D}} = \mathrm{I}$. The scalar function $\rho(\mathrm{p}-\mathrm{c})^3$ defines a star-shape around the point c . We can use this function to define a star-shaped swirling-sweepers step:

$$f^*_{\tau_k \rightarrow \tau_{k+1}} = f_{\mathrm{D},\tau_{k+1}} \circ f_{\tau_k \rightarrow \tau_{k+1}} \circ f^{-1}_{\mathrm{D},\tau_k} \tag{4.18}$$

Loosely speaking, this function removes the star-shaped tool from the scene, applies a simple swirling-sweeper and inserts the star-shaped tool back again in the scene. Because the insertion and removal of the star-shape does not change the volume of the scene outside of the tool, the function $f^*_{\tau_k \rightarrow \tau_{k+1}}$ is volume preserving. Note that for multiple steps, the composition of the sub-function simplifies using the standard property of conjugation:

$$\begin{aligned} f^*_{t_i \rightarrow t_{i+1}} &= \underset{k=0}{\overset{s-1}{\Omega}} (f_{\mathrm{D},\tau_{k+1}} \circ f_{\tau_k \rightarrow \tau_{k+1}} \circ f^{-1}_{\mathrm{D},\tau_k}) \\ &= f_{\mathrm{D},\tau_s} \circ f_{\tau_{s-1} \rightarrow \tau_s} \circ f^{-1}_{\mathrm{D},\tau_{s-1}} \circ f_{\mathrm{D},\tau_{s-1}} \circ f_{\tau_{s-2} \rightarrow \tau_{s-1}} \circ f^{-1}_{\mathrm{D},\tau_{s-2}} \circ \cdots \circ f^{-1}_{\mathrm{D},\tau_0} \\ &= f_{\mathrm{D},t_{i+1}} \circ \left(\underset{k=0}{\overset{s-1}{\Omega}} f_{\tau_k \rightarrow \tau_{k+1}} \right) \circ f^{-1}_{\mathrm{D},t_i} \end{aligned} \tag{4.19}$$

Interestingly, the star-shape function does not have to be applied at each step of the deformation. The drawback of using a star-shaped tool is that current formula does not allow simultaneous star-shaped swirling-sweepers. Also, this deformation is discontinuous on the boundary of the star-shape, thus the tool cannot intersect the surface of the modeled shape: the tool has to be completely inside or completely outside of the shape. Some examples are shown in Figure 4.5.



(a) (b) (c)

Figure 4.5: *(a) With swirling-sweepers, the artist has control over a point. (b) With Star-shaped swirling-sweepers, the artist has control over a volume, in this example a ball. Notice the imprint of the ball in the shape. (c) Left: swirling-sweepers. Right: Star-shaped swirling-sweepers.*

4.1.7 Results

Swirling-sweepers is a space deformation technique that aims at taking advantage of the artist's familiarity with the notion of an amount of material. Swirling-sweepers use the sweepers formulation: a combination of matrices raised to powers of scalar functions. Combined with the original sweepers, the volume of a shape can be increased, preserved or decreased.

We have implemented swirling-sweepers in C++ using OpenGL®, on a Pentium® 2400Mhz with 1GB of RAM. This implementation works in real-time. The computational time is a function of the magnitude of the input vector, because this determines the number of sub-steps. Small vectors will produce extremely fast deformations. In order to preserve the sampling of the deformed surface, we use the mesh update algorithm proposed in Chapter 5, adapted for sweeping space deformations. Simple scenarios are shown in Figure 4.7, and more elaborate results are shown in Figure 4.9. Swirling-sweepers can also be used for the purpose of doing volume preserving animation, an example being shown in Figure 4.6.

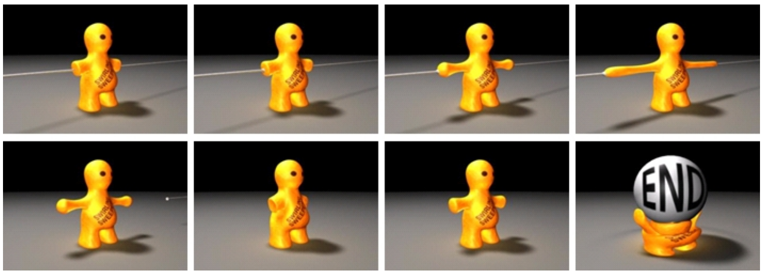


Figure 4.6: *Swirling-sweepers applied to animation.*

Limitations In our implementation, the tool cannot be too small compared to the density of the mesh, i.e. the radius of a swirl should be comparable to the length of an edge. In Figure 4.9, we compare the shapes' volumes with unit spheres on the right. For the mouse, goblin, alien and tree, the shapes volumes are respectively 101.422%, 99.993%, 101.158% and 103.633% of the initial sphere. This error is the result of accumulating smaller errors from each deformation. For instance 80 swirling-sweepers have been used to model the alien in Figure 4.9. The small errors are due to the finite number of steps, and to our choice of shape representation, which is described in Chapter 5.

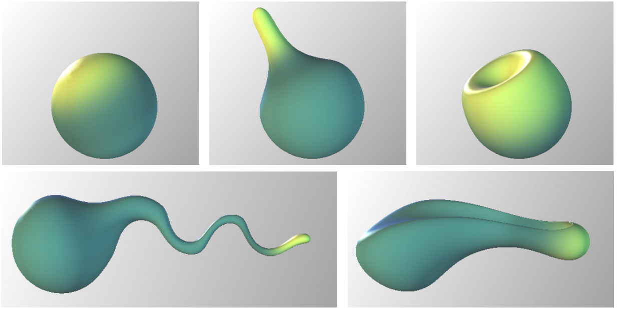


Figure 4.7: *When pushed or pulled, a sphere will inflate or deflate elsewhere.*

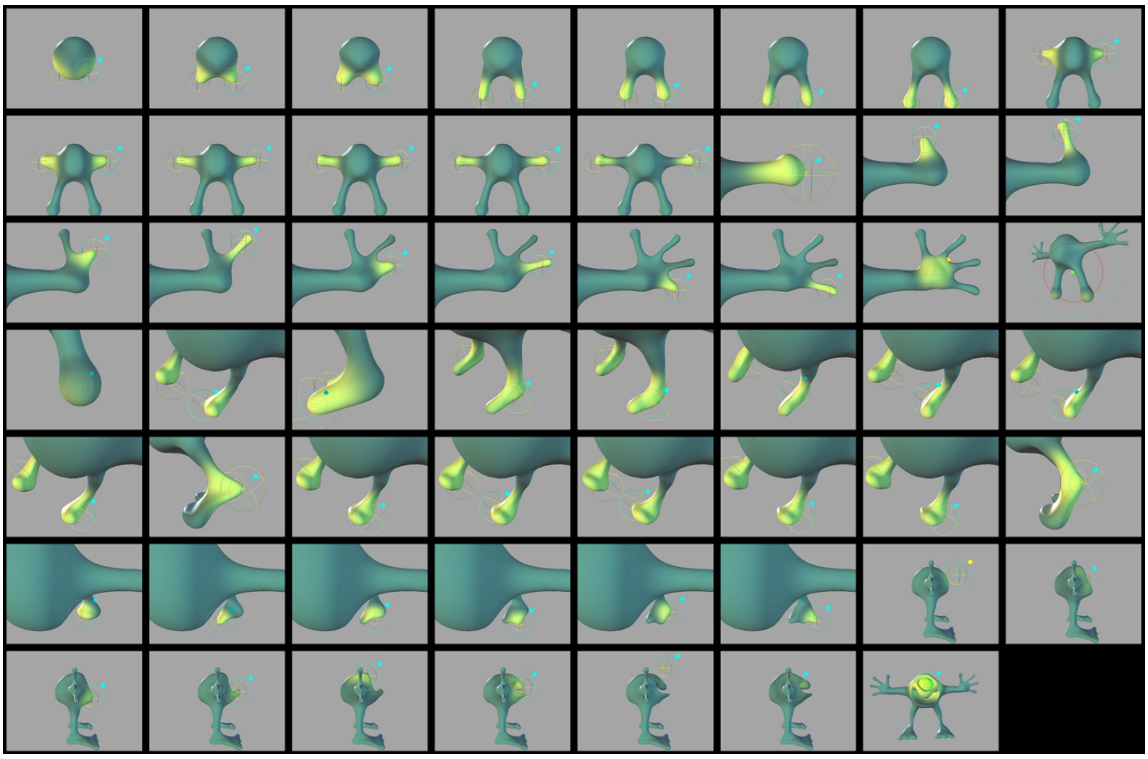


Figure 4.8: *Snapshots of the modeling process of the alien character of Figure 4.9.*

4.2 Swept-fluid

Swept-fluid is a volume-preserving space deformation technique that allows the artist to grab a portion of the shape and move it while the shape's volume is maintained. Since our intuition and observations tell us that clay-like materials behave essentially like

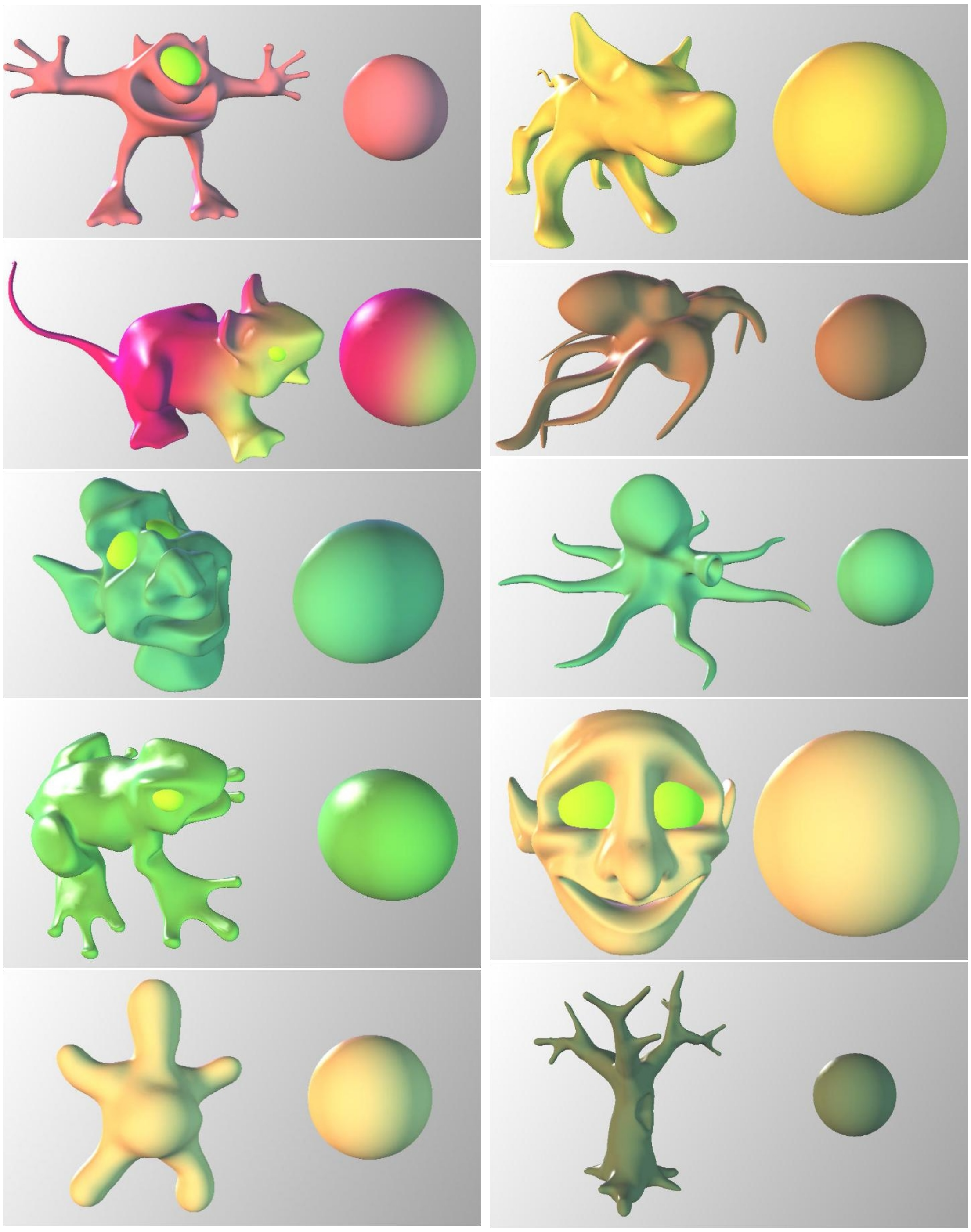


Figure 4.9: *Examples of models created with swirling-sweepers.*

fluids, we explicitly base our space deformation on a simplified version of the incompressible Navier-Stokes equations. Because with modern computing devices, we have observed that a fine enough solutions to the Navier-Stokes equation could not be computed at a high enough frequency for our application, we propose to compute the fluid's solution only once, and re-use the result multiple times. Our deformation is therefore fast, and maintains to some extent the volume of a shape being modeled. Since it is a space deformation, it can be applied to a wide range of geometric representations.

4.2.1 Basic equation

In CG, the incompressible Navier-Stokes equations are popular for animating fluids. These differential equations describe the evolution in time of the fluid's velocity. They have a compact form, and are to some extent solvable with modern computing devices:

$$\left\{ \begin{array}{rcl} \frac{\partial \vec{\mathrm{v}}}{\partial t} & = & -(\vec{\mathrm{v}} \cdot \nabla)\vec{\mathrm{v}} + \nu \nabla^2 \vec{\mathrm{v}} - \frac{1}{\rho}\nabla p + \vec{\mathrm{f}} \\ \nabla \cdot \vec{\mathrm{v}} & = & 0 \end{array} \right. \tag{4.20}$$

where $\vec{\mathrm{v}}$ is the velocity, p is the pressure, ρ is the density, ν is the kinematic viscosity and $\vec{\mathrm{f}}$ includes all the external body forces. One way to understand how to obtain the first equation is to apply Newton's Principle of Mechanics to a continuum, i.e. the acceleration of a point is equal to the sum of the forces per unit mass [FLS89]. The second equation simply means that the fluid is incompressible, i.e. the inflow entering a point has to be equal to the outflow leaving that point. We refer the reader to [Rut90] for a thorough insight into fluid mechanics, which is beyond the scope of this thesis. Some specialization of these equations can be made. First, the quantities ρ and ν are constants for the entire fluid. Second, external body forces are cumbersome vector fields for the purpose of shape modeling. We therefore remove them:

$$\left\{ \begin{array}{rcl} \frac{\partial \vec{\mathrm{v}}}{\partial t} & = & -(\vec{\mathrm{v}} \cdot \nabla)\vec{\mathrm{v}} + \nu \nabla^2 \vec{\mathrm{v}} - \frac{1}{\rho}\nabla p \\ \nabla \cdot \vec{\mathrm{v}} & = & 0 \end{array} \right. \tag{4.21}$$

In CG, these equations are usually solved in a discrete way, by sampling the velocity in a grid. The most straightforward way of applying the solution of these equations to a shape would be to embed the entire scene in a discrete grid and solve the equation for a tool moving in the scene, in a similar way that liquid has successfully been animated in CG [FF01, EMF02]. Techniques for animating fluids visually are capable of simulating behaviors too rich for our purpose, and are far from real-time. We propose in the following section a solution for precomputing the flow only around the tool. We will assume that the inside and outside of the tool are filled with the same fluid, and therefore both can be handled as a single fluid [Sta99].

4.2.2 Precomputed solution

Our objective is to allow the artist to interact with the shape via a tool, by applying translations to the tool. With current computing devices, we cannot solve the Navier-Stoke equations at a high enough rate over the entire scene, or in a neighborhood around the tool, even using a fast technique [FSJ01]. Let us consider in the remainder of this section a single translation, $\vec{t}$. We propose to precompute a solution around the tool in a canonic situation, and re-use the solution as many times as required along the trajectory of the tool.

Solving Navier-Stokes

To compute the fluid velocity, we use a Lagrangian method along the lines of [Sta99], i.e. we solve for the velocity with respect to a static frame as given by Equation (4.21). The velocity field of the flow is simply represented with a uniform discretization of space into N^3 cells of size 1. The velocity is defined at the cell centers

$$\vec{v}_{ijk}, \quad i, j, k = 0 \dots N-1 \tag{4.22}$$

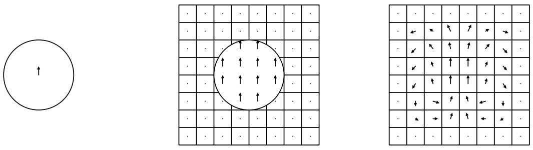


Figure 4.10: *The Navier-Stokes equations are solved only once. First, the tool is placed in the grid. Second, cells intersecting the tool are set to a default velocity. Third, the equations are solved.*

In the usual scenario of objects moving in a fluid, cells intersecting a moving object should be assigned the object's velocity. Since we want to precompute a solution, we do not know in advance the trajectory or the velocity that the artist will give to the tool. We therefore choose some initial velocity for the tool $\vec{v}_{\text{INIT}} = (1, 0, 0)^\top$, and any other velocity is set to zero (see Figure 4.10). Once the fluid is solved, we will see in the next subsection how to adapt the obtained solution to any trajectory input by the artist. The tool's shape used in our examples is simply a ball. To solve Equation (4.21), we proceed in steps exactly as described in [Sta99]. In the context of animating a fluid, the computation of the velocity at each time step is decomposed into four steps. Since we do not animate the fluid, we only apply these steps once to obtain the solution of the first time step Δ_t. Let us denote by $\vec{v}_0$ initial velocity:

First, we solve the convection term using the *method of characteristics* to get $\vec{v}_1$. To make this step stable, we use the monotonic interpolant described in Appendix C.

Second, we solve the diffusion term and the incompressible term. The two systems are of the form $A \cdot x = b$. The linear equations proposed by [Sta99] are symmetric and positive definite. As remarked by [FSJ01] and [FF01], an efficient solver is the conjugate gradient method, with an incomplete Choleski preconditioning. For solving the diffuse term, we solve a system of equations, i.e. the following for $\vec{v}_2$ defined at each node of the grid:

$$(1 - \Delta_t \ \nu \ \nabla^2)\vec{v}_2 = \vec{v}_1 \tag{4.23}$$

For solving the incompressible term, we solve for the pressure p the system to get $\vec{v}_2$:

$$(-\nabla^2)p = (-\nabla \cdot \vec{v}_2) \tag{4.24}$$

where ∇ and ∇^2 are discretized with centered derivatives, as described in Appendix C.2. Once this is solved in p, the new velocity $\vec{v}_3$ in the grid cells is

$$\vec{v}_3 = \vec{v}_2 - \frac{1}{\rho}\nabla p \tag{4.25}$$

Smoothing the solution

Note that the velocity obtained by the above solver $\vec{v}_2$ is not defined everywhere in $\mathbb{R}^3$, but only at the regularly spaced cells of the grid. A grid of vectors may be used in a similar way to an FFD lattice [SP86]: but in our method the vectors are obtained by solving a differential equation rather than specified manually one by one. Once the grid of velocity has been computed, it has to be smoothed in order to be defined continuously in space. We choose to use the trivariate cubic uniform B-Spline basis functions described in Appendix C, because they have relatively local support, and are second order continuous, which is an important property for smooth deformation of the normals of shapes.

4.2.3 Sweeping the solution

To apply the precomputed solution to the scene, it is necessary to decompose the input tool translation $\vec{t}$ into s small enough steps to prevent foldovers. Since the flow was solved for some tool velocity $\vec{v}_{\text{INIT}}$, it needs to be modulated by an adequate constant. The flow is then smoothed as described above and applied to the shape s times. At each step, the grid is also moved by a fraction of the translation vector, $\frac{1}{s}\vec{t}$.

Precomputed grid

We have used a grid of size 72^3. The tool must be contained in the core of the grid in order to compute an acceptable solution around it. We used a ball of radius 12. Our choice for the kinematic viscosity ν and the density ρ is empirical, based on visually pleasing quality and speed.

The density per grid cell, ρ, controls the locality of the effect of a tool. If the density is too high, a tool centered in the middle of the grid may affect the velocity on the faces of the cube. We do not want this to happen, since the velocity is 0 outside the cube. At the other end, if the density is too low, there may not be enough cells to represent accurately the velocity around the tool. In our examples, $\rho = 1.0 \times 10^{-6}$.

The viscosity is a way of blurring turbulence in the vector field, as a function of the time step. Setting ν to zero would produce serious artifacts on the shape. On the other hand, if ν is too high, the effect of the tool on the shape would barely be noticeable. In our examples, $\nu = 2$.

Adapting the solution

The solution is precomputed for a tool moving with velocity $\vec{v}_{\text{INIT}}$, defined in the solver's frame set. However, the user inputs a vector $\vec{t}$, in world coordinates. We

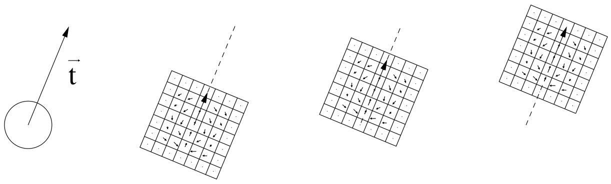


Figure 4.11: *A deformation of arbitrary length is obtained by composition of steps. Each step uses the precomputed fluid grid. Three steps have been used here.*

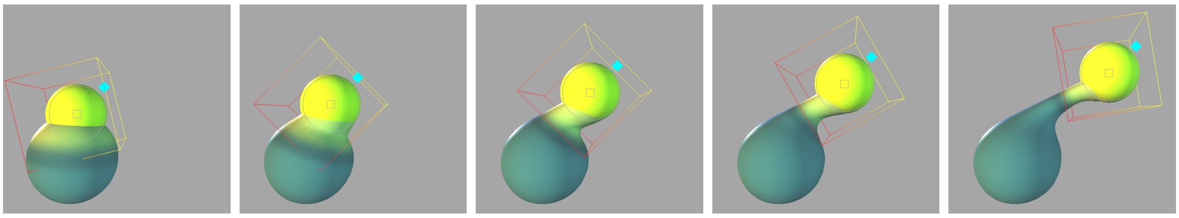


Figure 4.12: *The wireframe box contains the grid of the precomputed solution. It is oriented with the direction of the tool.*

describe a simple way to create a deformation of length $\vec{t}$ by composing the precomputed deformation many times. First, let us call $\vec{v}_{\text{CENTER}}$ the vector evaluated at the center of the grid, once the solution has been computed. It is important to notice that after computation of the solution, the value of the velocity at the center $\vec{v}_{\text{CENTER}}$ is not guaranteed to be equal to its initial value $\vec{v}_{\text{INIT}}$, although it is expected to be very close. Thus the deformation is decomposed into steps, each of length bounded in local coordinates by $\vec{v}_{\text{CENTER}}$. Thus if we denote by W the 4x4 matrix transforming a point in local grid frame set to the point in world coordinates, the number of required steps is:

$$s = \max(1, \lceil \frac{||W^{-1}\vec{t}||}{||\vec{v}_{\text{CENTER}}||} \rceil) \tag{4.26}$$

The velocity in the grid needs to be modulated at each step to be applied to points, otherwise the deformation would allow points to overshoot. The factor modulating the velocity of each step is proportional to the input vector $\vec{t}$ and inversely proportional to the number of steps and precomputed velocity:

$$l = \frac{||W^{-1}\vec{t}||}{s||\vec{v}_{\text{CENTER}}||} \tag{4.27}$$

It can be quickly verified that in the particular case where the vector $W^{-1}\vec{t}$ is equal to $\vec{v}_{\text{CENTER}}$, then $l = 1$. The decomposition of one translation $\vec{t}$ into steps is illustrated in Figure 4.11, and the decomposition of a trajectory is shown in Figure 4.12.

4.2.4 Results

In the examples shown, the shape is represented using a mesh, whose edges are collapsed or split to maintain a minimum surface sampling density (see Section 5.1). Note that our deformation is independent of the shape's geometric representation.

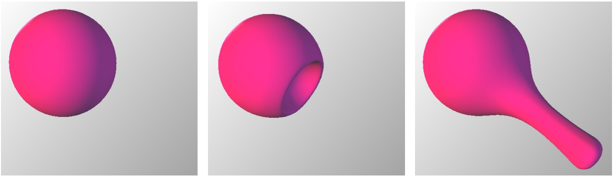


Figure 4.13: *Examples of push (middle) and pull (right). The initial object is on the left.*

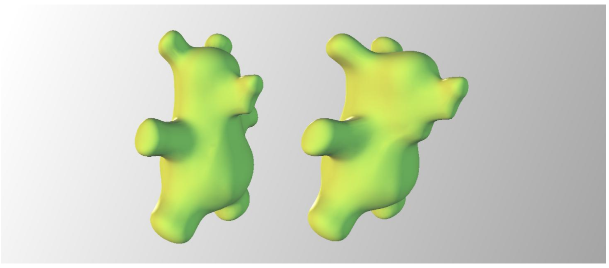


Figure 4.14: *Example of a shape modeled by deformation. Our tool can grab a portion of the shape (the nose), and translate it, while the volume variation is limited: the teddy bear's volume increase is* 109.301% *of the initial teddy bear. Note that although accumulation of error can be significant, the shape's local behaviour in time observed by the artist appears to preserve volume.*

The locality of the effect of the tool is controlled by applying a uniform scale[1] to the transformation matrix from local to world coordinates, W. The parameters ρ and ν cannot be changed at will to create a rich set of deformations, since their values are limited by the size of the grid; 72^3 in our examples. Also, the volume is not extremely well preserved due to the discrete grid or slight numerical deviations in the iterative solver. Simple effects resulting from applying a tool once are shown in Figure 4.13. In Figure 4.14, the volume of the teddy-bear has increased by a factor of 166.348% compared to the initial sphere. Although this volume variation has to be split among the number of operations the artist applied to the shape, this result is somehow disappointing. The technique presented in the previous section preserves volume much more accurately.

Note that one motivation for using differential equations is to create a rich set of material behaviors, by letting the artist play with parameters such as ρ and ν. However, the discrete solution to the equations imposes severe restrictions on these parameters.

4.3 Conclusion

Two volume-preserving methods have been developed: swirling-sweepers presented in Section 4.1 and Swept-fluid presented in Section 4.2. The former technique is not only faster, but preserves volume more accurately than the latter.

In our experience, modeling by volume-preserving space deformation is much more agreeable than using the techniques presented in Chapter 3, especially using swirling-

---

[1] Applying a non-uniform scale would affect the volume preservation.

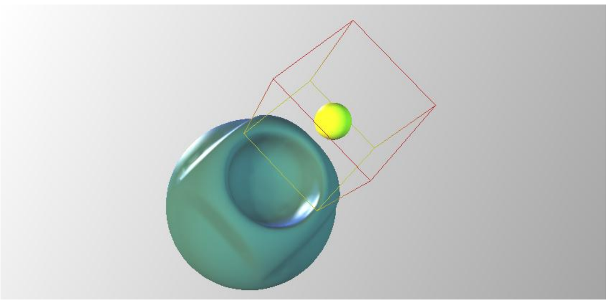


Figure 4.15: *Exaggerated artifact, by choosing a too high value for the density* ρ*. The boundary conditions have an effect on the flow.*

sweepers. The notion of volume is familiar to most artists, and our experience shows that using it increases the modeling speed: the models of Figure 4.9 were faster and easier to produce than those of Figures 3.10 and 3.11.

Chapter 5

Rendering

The common theme of Chapters 3 and 4 is to define techniques for deforming a shape, regardless of the shape's mathematical description. Since our deformations apply to space, they can already be applied to the control points of explicit shape descriptions, e.g. meshes, NURBS or subdivision surfaces. The application we are aiming at is however shape modeling, in which the number of deformations is possibly very large, and issues related to excessive deformation rise when the shape needs to be rendered.

We identify four possible ways to display a shape during or after transformation by space deformation: the first method consists of applying the deformation to the vertices of an updated mesh, the second method consists of applying the deformation to deformable particles, the third method consists of applying the inverse deformation to the nodes of a discrete implicit surface and is similar to advection methods used in fluid animation [Sta99], and the fourth one consists of applying the inverse deformation to the rays of light, and was identified early by A. Barr [Bar84]. The four methods are presented in the four following sections, and only the three first methods proved sufficiently fast to be used in an interactive modeling system. We conclude this chapter with a method that would be a good candidate for texturing without stretching the texture applied to our shapes in future research.

The majority of examples shown in this thesis were made using the method we are about to describe, since it was implemented early, and proved to be sufficiently fast and accurate.

5.1 Updated mesh

The objective of this section is to provide a shape description for interactive modeling which supports high deformation and does not break when highly stretched. A simple way of representing a deformable shape is to place a set of samples on the surface of the shape: this makes the task of deforming the shape as straightforward as deforming the points on its surface. Points are discrete surface samples, and need to be *smoothed* with a splatting, approximation or interpolation scheme in order to display a continuous surface or perform any other post-processing operation requiring surface continuity.

Our method will use some kind of surface patch: connectivity provides convenient $2D$-boundary information for rendering the surface as well as surface neighborhood information, which enables the artist to define very thin membranes without having them vanish, as shown in Figure 3.11(c). The reader however should be aware that point-sampled geometry has recently ignited a lot of interest among researchers [PKKG03], and we propose a point-sampled shape description in the next section.

The possibly large number of deformations applied by an artist requires some minimum surface sampling density. Although our deformations can be applied to the control points of any existing high-order parametric surface, e.g. NURBS or subdivision surfaces, we represent the modeled shape by a triangle mesh, locally refined and simplified in order to maintain an adequate sampling density. The restriction to triangular "C^0 patches" also circumvents issues related to non-regular vertices and smoothness maintenance across the boundaries that join patches. Also, polygons are handled very efficiently by current hardware, which is relevant to us since interactivity is among our objectives.

Thus, a scene is initialized with a polygonal model, e.g. a sphere with a homogeneous density of nearly equilateral triangles [1]. In order to quickly fetch the vertices to be deformed and the edges that require splitting or collapsing, these are inserted into a 3D grid. Note that this spatial limitation is not too restrictive for the artist, as our deformations allow us to translate the entire model rigidly and scale it uniformly.

To fetch the vertices that are deformed, a query is done with the tool's bounding box. Conveniently, this bounding box is also used in Equation (3.14). Since the principle of our swept deformations is to subdivide a gesture into a series of smaller ones, all the transformations applied to the vertices are bounded. To take advantage of this step decomposition, we apply a modified version of the generic algorithm in [GD99]. Our method requires keeping two vertices and two normals per vertex, corresponding to the current time τ_i and next time τ_{i+1} of some small step $f_{\tau_i \mapsto \tau_{i+1}}$. Loosely speaking, our surface-updating algorithm assumes that smooth curves run on the surface, and that the available information, namely vertices and normals, should be able to represent them. If this is not the case after deformation, then it means the surface is undersampled. On the other hand, if an edge is well enough represented by a single sample, then it is collapsed.

Let us consider an edge e defined by two vertices $(\mathrm{v}_0, \mathrm{v}_1)$ with normals $(\vec{\mathrm{n}}_0, \vec{\mathrm{n}}_1)$, and the deformed edge e' defined by vertices $(\mathrm{v}'_0, \mathrm{v}'_1)$ with normals $(\vec{\mathrm{n}}'_0, \vec{\mathrm{n}}'_1)$. In addition to the conditions in [GD99] based on edge length and angle between normals, we also base the choice of splitting edge e' on the error between the edge and a fictitious vertex, which belongs to a smooth curve on the surface. The fictitious vertex is used only for measuring the error, and is not a means of interpolating the vertices. If the error

---

[1] A simple way to obtain an homogeneous sphere polygonization consists of starting with an icosahedron, putting all its edges longer than h in a queue, splitting them and putting the pieces longer than h back in the queue. Each time a split is performed, the new edges are flipped to maximize the smallest angle.

between the fictitious vertex and the edge is too large, the edge e is split, and the new vertex and normal are deformed. On the other hand if the fictitious vertex represents the edge e' well enough, then edge e is collapsed, and the new vertex is deformed. We define the fictitious vertex as the mid-vertex of a C^1 curve, since vertices and normals only provide first order information about the surface. The following cubic polynomial curve interpolates the vertices v'_0 and v'_1 with corresponding tangents $\vec{\mathrm{t}}_0$ and $\vec{\mathrm{t}}_1$ defined below:

$$\mathrm{c}(u) = (\mathrm{v}'_0(1+2u) - \vec{\mathrm{t}}_0 u)(1-u)^2 + (\mathrm{v}'_1(1+2(1-u)) + \vec{\mathrm{t}}_1(1-u))u^2 \tag{5.1}$$

The only constraint on the tangent $\vec{\mathrm{t}}_i$ is to be perpendicular to the corresponding normal $\vec{\mathrm{n}}_i$. The following choice defines tangents of magnitude proportional to the distance between the vertices:

$$\begin{array}{rcl} \vec{\mathrm{t}}_0 & = & \vec{\mathrm{g}} - \vec{\mathrm{g}} \cdot \vec{\mathrm{n}}'_1 \\ \vec{\mathrm{t}}_1 & = & \vec{\mathrm{g}} - \vec{\mathrm{g}} \cdot \vec{\mathrm{n}}'_0 \\ \text{where } \vec{\mathrm{g}} & = & \mathrm{v}'_1 - \mathrm{v}'_0 \end{array} \tag{5.2}$$

With the above tangents, the expression of the middle vertex simplifies:

$$\mathrm{c}(0.5) = (\ \mathrm{v}'_0 + \mathrm{v}'_1 + (\vec{\mathrm{g}} \cdot \vec{\mathrm{n}}'_0 - \vec{\mathrm{g}} \cdot \vec{\mathrm{n}}'_1)/4\ )\ /\ 2 \tag{5.3}$$

With the fictitious vertex c(0.5), the tests to decide whether an edge should be split or collapsed can be defined.

Too-long edge: An edge e' is too long if at least one of the following conditions is met:

- The edge is longer than $L_{\max}$, the size of a grid-cell. This condition keeps a minimum surface density, so that the deformation can be caught by the net of vertices if the coating thickness λ_j is greater than $L_{\max}$.
- The distance between the fictitious vertex and the mid-vertex of e' is too large (we used $L_{\max}/20$). This condition prevents the sampling from folding on itself, which would produce multiple sampling layers of the same surface.
- The angle between the normals $\vec{\mathrm{n}}'_0$ and $\vec{\mathrm{n}}'_1$ is larger than a constant $\theta_{\max}$. This condition keeps a minimum curvature sampling.

Too-short edge: An edge e' is too short if all of the following conditions are met:

- The edge's length is shorter than $L_{\min}$ (we used $L_{\max}/2$).
- The angle between the normals $\vec{\mathrm{n}}'_0$ and $\vec{\mathrm{n}}'_1$ is smaller than a constant $\theta_{\min}$.
- The distance between the fictitious vertex and the mid-vertex of e' is too small (we used $L_{\min}/20$).

Also, to avoid excessively small edges, an edge is merged regardless of previous conditions if it is too small (we used $L_{\min}/20$).

We stress that the procedure for updating the mesh is applied at each small step, rather than after the user's deformation function has been applied. Because vertex displacements are bounded by the foldover-free conditions, the update of our shape description does not suffer from problems related to updating a greatly distorted triangulation. Figure 5.1 shows a twist on a simple U-shape. Figure 5.2 shows the algorithm preserving a fine triangulation only where required. Figure 5.3 shows the algorithm at work in a more practical situation. The procedure outline is:

```
Compute the number of steps required, $s$.
for each step $\tau_i \mapsto \tau_{i+1}$ do
  Deform the points, and hold their previous values
  for each too-long edge do
    split the edge and deform the new point.
  end for
  for each too-short edge do
    collapse the edge and deform the new point.
  end for
end for
```

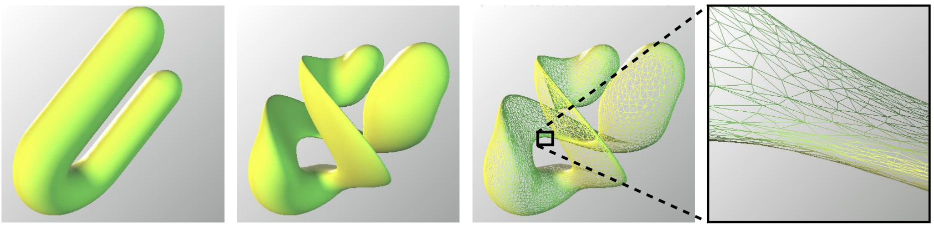


Figure 5.1: *Example of our mesh-updating algorithm on a highly twisted U-shape. The close-up shows a sharp feature, with finer elongated triangles.*

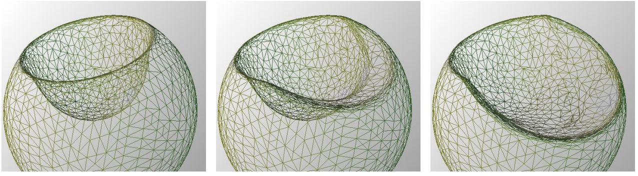


Figure 5.2: *Behavior of our mesh-updating algorithm on an already punched sphere. The decimation accompanying the second punch simplifies the small triangles of the first punch. The tool has been removed for better visualization.*

5.1.1 Results and limitation

With the updated mesh method, we choose to ignore the history of functions applied to the shape by the artist. We rather "collapse" the history by freezing it in the current shape. In order to explain the major consequence of this assumption, let us suppose the scene at a time t_k, such that the shape $S(t_k)$ is shown to the user. The

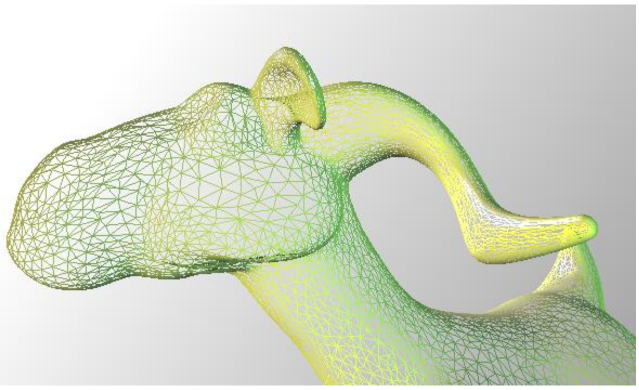


Figure 5.3: *Close-up of the goat. Notice the large triangles on the cheek and the fine ones on the ear. The initial shape is a sphere.*

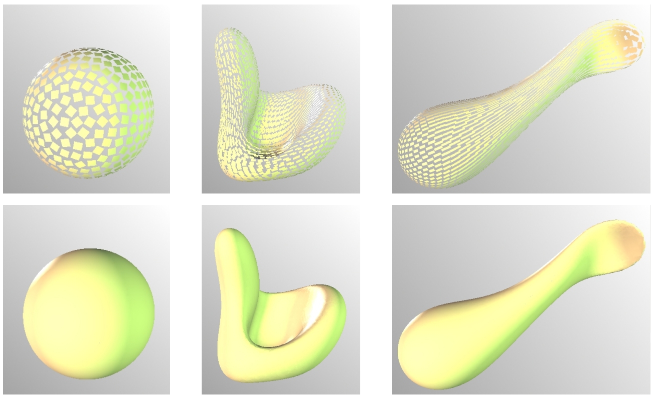
next deformation produced by the artist with the mouse is function $f_{t_k \mapsto t_{k+1}}$, and all the mesh refinements and simplifications are performed in $S(t_k)$. This is however an approximation: ideally the last operation should be concatenated to the history of deformations, and the whole series should be applied to the initial shape $S(t_0)$, i.e. $\overset{k}{\underset{i=0}{\Omega}} f_{t_i \mapsto t_{i+1}}$ should be applied to each new vertex. This would however become more and more time consuming as the sequence of deformations gets longer (k gets larger) until the modeling software becomes unusable. Measuring the deviation of our shape from the ideal shape is beyond the scope of this work.

The approximation consisting of deforming $S(t_k)$ rather than $S(t_0)$ works well enough in practice, and is fast enough for a highly interactive application. This method proved quite successful, and most results of previous chapters were obtained using it. This includes Figures 3.10, 3.11, 3.1, 3.8, 3.16, 3.19, 3.20, 3.21, 3.32, 4.1, 4.2, 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, 4.14, 4.13.

5.2 Point-based representation

For representing shapes, point-sampled surfaces (e.g. extremal surfaces or Moving Least Square (MSL) surfaces) are becoming popular in Computer Graphics because connectivity can be ignored between surface elements [AK04]. This last statement is almost true: the connectivity may be ignored for the purpose of purely editing the surface; but when the surface needs to be displayed, neighborhood information has to be reconstructed at a relatively high cost. This section introduces SOAP surfaces (Second Order Adaptive Particle surfaces), a clean point-based surface description in which neighborhood information is not required in order to display the surface. Our technique is thus fast in displaying shapes and easy to implement. Moreover, with space deformation editing tools, the surface can be edited interactively while maintaining a displayable shape.

The SOAP surface representation is based on particles, i.e. surface patches that bend and deform under strain, and that split if the strain is too high. Because there is little surface data inter-dependence, the method is relatively short and simple to implement. The definition of a patch is based on Differential Geometry [Lip69].



(a) (b) (c)

Figure 5.4: *Top: sample particles on the surface. Bottom: if the patches of the particles are large enough, local overlapping fills the gaps.*

5.2.1 Local surface parameterization

We show that to represent a surface with second order accuracy at a point, the required data is: a point, two vectors and two scalars denoted with the n-tuple $(\mathrm{s}, \vec{\mathrm{s}}_u, \vec{\mathrm{s}}_v, \lambda_{uu}, \lambda_{vv})$. Let us consider a surface defined locally for a point $s(u, v)$, where u and v are local arc-length parameters. The following expression is a second order Taylor expansion of the surface:

$$\mathrm{s}(u,v) = \mathrm{s}(0,0) + u\frac{\partial \mathrm{s}}{\partial u} + v\frac{\partial \mathrm{s}}{\partial v} + \frac{1}{2}(u^2\frac{\partial^2 \mathrm{s}}{\partial u^2} + 2uv\frac{\partial^2 \mathrm{s}}{\partial u \partial v} + v^2\frac{\partial^2 \mathrm{s}}{\partial v^2})$$

Let us define the two unit orthogonal tangents $\vec{\mathrm{s}}_u = \frac{\partial \mathrm{s}}{\partial u}$, $\vec{\mathrm{s}}_v = \frac{\partial \mathrm{s}}{\partial v}$ aligned with the surface's principal curvature directions and the surface normal $\vec{\mathrm{n}} = \vec{\mathrm{s}}_u \times \vec{\mathrm{s}}_v$. A result of differential geometry states that if the tangents are aligned with the principal curvature directions, then $\frac{\partial^2 \mathrm{s}}{\partial u \partial v} = 0$. Thus we may simplify the above as follows:

$$\mathrm{s}(u,v) = \mathrm{s} + u\ \vec{\mathrm{s}}_u + v\ \vec{\mathrm{s}}_v + \frac{1}{2}(u^2\lambda_{uu} + v^2\lambda_{vv})\vec{\mathrm{n}} \tag{5.4}$$

The above defines a surface patch locally. We define a surface as the union of such patches.

Initialization: Our initial shape is a sphere, since it has well known differential properties. Let us consider the sphere of radius r centered at the origin. Let us consider a point on the surface $\mathrm{s}(0, 0)$. Since points on the sphere are *umbilical*[2], $\lambda_{uv} = 0$. The other curvature coefficients are $\lambda_{uu} = \lambda_{vv} = \frac{1}{r}$.

5.2.2 Parameterization under deformation

Under the operations applied by the artist, the surface deforms, and the coefficients of Equation 5.4 need to be updated. We show how this can be done. Let us consider a continuous space deformation function $f : \mathbb{R}^3 \mapsto \mathbb{R}^3$ applied to the shape. Let us

---

[2] An umbilical point is a point at which the curvature of the surface is the same in every direction.

denote by $\vec{\mathrm{n}}'$ the normal of the new deformed surface. A second order approximation of the deformed surface is simply obtained with:

$$\begin{array}{rcl} \mathrm{s}'(u,v) & = & f(\mathrm{s}) + u\vec{\sigma}_u + v\vec{\sigma}_v + \frac{1}{2}(u^2 l_{uu} + 2uv l_{uv} + v^2 l_{vv})\vec{\mathrm{n}}' \\ \text{where} \quad \vec{\sigma}_u & = & \frac{\partial f(\mathrm{s})}{\partial u} \quad \text{and} \quad l_{uu} = \frac{\partial^2 f(\mathrm{s})}{\partial u^2} \cdot \vec{\mathrm{n}}' \\ \vec{\sigma}_v & = & \frac{\partial f(\mathrm{s})}{\partial v} \quad\quad\quad\; l_{vv} = \frac{\partial^2 f(\mathrm{s})}{\partial v^2} \cdot \vec{\mathrm{n}}' \\ & & \quad\quad\quad\quad\quad\; l_{uv} = \frac{\partial^2 f(\mathrm{s})}{\partial u \partial v} \cdot \vec{\mathrm{n}}' \end{array} \tag{5.5}$$

Note that the new tangents $\vec{\sigma}_u, \vec{\sigma}_v$ are not necessarily unit or orthogonal, like the original tangents. Thus it is necessary to rearrange the above expression of a deformed surface to obtain a similar form to Equation (5.4), in order to reduce the information describing the surface. Let us reparameterize the surface such that the new tangents $\vec{\mathrm{s}}'_u$ and $\vec{\mathrm{s}}'_v$ are unit and orthogonal:

$$\left\{ \begin{array}{rcl} \vec{\mathrm{s}}'_u & = & \alpha_u\vec{\sigma}_u + \alpha_v\vec{\sigma}_v \\ \vec{\mathrm{s}}'_v & = & \beta_u\vec{\sigma}_u + \beta_v\vec{\sigma}_v \end{array} \right. \quad \left\{ \begin{array}{rcl} u & = & u'\alpha_u + v'\beta_u \\ v & = & u'\alpha_v + v'\beta_v \end{array} \right. \tag{5.6}$$

Let us rewrite the deformed surface (with dummy parameters u, v):

$$\mathrm{s}'(u,v) = \mathrm{s} + u\ \vec{\mathrm{s}}'_u + v\ \vec{\mathrm{s}}'_v + \frac{1}{2} \left(\begin{array}{l} (u\alpha_u + v\beta_u)^2 l_{uu} + \\ 2(u\alpha_u + v\beta_u)(u\alpha_v + v\beta_v) l_{uv} + \\ (u\alpha_v + v\beta_v)^2 l_{vv} \end{array} \right) \vec{\mathrm{n}} \tag{5.7}$$

Thus locally, a second order approximation of the new surface is:

$$\begin{array}{rcl} \mathrm{s}'(u,v) = \mathrm{s}' + u\ \vec{\mathrm{s}}'_u + v\ \vec{\mathrm{s}}'_v + \frac{1}{2}(u^2\lambda'_{uu} + 2uv\lambda'_{uv} + v^2\lambda'_{vv})\vec{\mathrm{n}}' \\ \text{where} \quad \lambda'_{uu} & = & \alpha_u^2 l_{uu} + 2\alpha_u\alpha_v l_{uv} + \alpha_v^2 l_{vv} \\ \lambda'_{vv} & = & \beta_u^2 l_{uu} + 2\beta_u\beta_v l_{uv} + \beta_v^2 l_{vv} \\ \lambda'_{uv} & = & \alpha_u\beta_u l_{uu} + 2(\alpha_u\alpha_v + \beta_u\beta_v) l_{uv} + \alpha_v\beta_v^2 l_{vv} \end{array} \tag{5.8}$$

In the above equation, the term in uv can be removed by choosing an adequate pair of unit tangent vectors. The tangents are given by the two unit eigenvectors of the following matrix, and are the two principal directions of the surface:

$$\left(\begin{array}{cc} \lambda'_{uu} & \lambda'_{uv} \\ \lambda'_{uv} & \lambda'_{vv} \end{array} \right) \tag{5.9}$$

Therefore, the n-tuple $(\mathrm{s}, \vec{\mathrm{s}}_u, \vec{\mathrm{s}}_v, \lambda_{uu}, \lambda_{vv})$ suffices to represent the surface locally, and can be recomputed at every particle each time the surface is deformed by the artist.

5.2.3 Local particle strain

In order to measure the strain that applies to a particle, we represent the accumulated distortion with an ellipse embedded in the 3D space, whose shape is deformed by the strain. We show that this ellipse is represented by the matrix of covariance of the tangents, which constitutes enough information for splitting a surface particle. Let us define the unit tangents of a particle in circular coordinates:

$$\vec{\tau}(\theta) = \cos(\theta)\vec{\mathrm{s}}_u + \sin(\theta)\vec{\mathrm{s}}_v \tag{5.10}$$

Let us define the deformed tangents, where J is the Jacobian of the deformation introduced in Section 3.1.2:

$$\vec{\tau}'(\theta) = J \cdot (\cos(\theta)\vec{\mathrm{s}}_u + \sin(\theta)\vec{\mathrm{s}}_v) \tag{5.11}$$

Let us study the matrix of covariance of the deformed unit tangents [GW92]:

$$\begin{array}{rcl} C & = & \int_\theta \vec{\tau}'(\theta) \cdot \vec{\tau}'(\theta)^\top \mathrm{d}\theta \\ & = & \int_\theta J \cdot (\vec{\tau}(\theta) \cdot \vec{\tau}(\theta)^\top) \cdot J^\top \mathrm{d}\theta \\ & = & J \cdot (\frac{1}{2}(\vec{\mathrm{s}}_u \cdot \vec{\mathrm{s}}_u^\top + \vec{\mathrm{s}}_v \cdot \vec{\mathrm{s}}_v^\top)) \cdot J^\top \end{array} \tag{5.12}$$

Since the Jacobian matrices of a series of deformations can be obtained by multiplying by the Jacobian matrix of each deformation, the *matrix of covariance* may be initialized with $\frac{1}{2}(\vec{\mathrm{s}}_u \cdot \vec{\mathrm{s}}_u^\top + \vec{\mathrm{s}}_v \cdot \vec{\mathrm{s}}_v^\top)$, and updated as follows:

$$C' = J' \cdot C \cdot J'^\top \tag{5.13}$$

Since the tangents represent a 2D space, then $\det C = 0$. Therefore one of the eigenvalues of C equals 0, let us say λ_3. Since the characteristic polynomial is of the form $\lambda(a_3\lambda^2 + a_2\lambda + a_1) = 0$, the two other eigenvalues λ_1 and λ_2 of C are found easily. Let us call $\vec{\mathrm{v}}_1$ and $\vec{\mathrm{v}}_2$ their eigenvectors. Thus at any time the stretching can be computed and a surface particle split along the eigenvector associated with the largest eigenvalue λ_1:

$$C_{\frac{1}{2}} = (\vec{\mathrm{v}}_1 \ , \ \vec{\mathrm{v}}_2 \ , \ \vec{\mathrm{v}}_1 \times \vec{\mathrm{v}}_2) \cdot \begin{pmatrix} \lambda_1/2 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot (\vec{\mathrm{v}}_1 \ , \ \vec{\mathrm{v}}_2 \ , \ \vec{\mathrm{v}}_1 \times \vec{\mathrm{v}}_2)^{-1} \tag{5.14}$$

Particle split

Under excessive strain, we will split the particles. Note that the split threshold should be proportional to the curvature. The eigenvalue λ_1 of C gives the principal strain. The vector $\frac{\lambda_1}{2}\vec{\mathrm{v}}_1$ projected on the unit tangents gives the parameters of the offspring particles, that is:

$$\begin{array}{c} (\frac{\lambda_1}{2}\vec{\mathrm{v}}_1 \cdot \vec{\mathrm{s}}_u, \frac{\lambda_1}{2}\vec{\mathrm{v}}_1 \cdot \vec{\mathrm{s}}_v)^\top \\ (-\frac{\lambda_1}{2}\vec{\mathrm{v}}_1 \cdot \vec{\mathrm{s}}_u, -\frac{\lambda_1}{2}\vec{\mathrm{v}}_1 \cdot \vec{\mathrm{s}}_v)^\top \end{array} \tag{5.15}$$

These coordinates can be used for defining the equation of the two new patches. Note that the new tangents are not necessarily orthogonal, that the second derivatives are not necessarily perpendicular to the new tangents, and that the equation must be renormalized in a similar way to the procedure described in Section 5.2.2. Figure 5.4(c) shows that the particles split along the direction of surface stretch.

5.2.4 Algorithm

Deforming a SOAP-surface is straightforward, since each surface sample is handled individually:

Input: n tuple $S_i = (\mathrm{s}_i, \vec{\mathrm{s}}_{u,i}, \vec{\mathrm{s}}_{v,i}, \lambda_{uu,i}, \lambda_{vv,i}, C_i)$

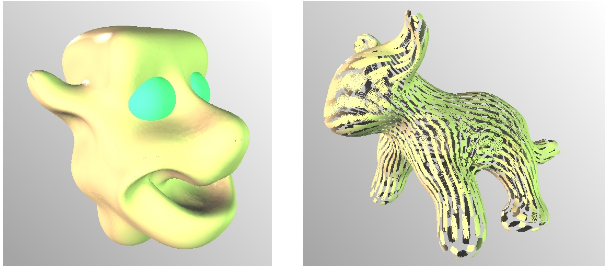


Figure 5.5: *Examples using a SOAP surface description.*

```
for each tuple $S_i$ do
   Deform the local parameterization, i.e $\mathrm{s}_i$, $\vec{\mathrm{s}}_{u,i}$, $\vec{\mathrm{s}}_{v,i}$, $\lambda_{uu,i}$ and $\lambda_{vv,i}$
   Accumulate the strain in $C_i$
   if the strain is too large then
      split the particle
   end if
end for
```

5.2.5 Results and limitations

Some results are shown in Figure 5.5. Although the method is quick enough for interactive modeling, there are still issues to be solved: we do not have a procedure to merge the particles when the surface is over-sampled, nor do we have a procedure to make sure no holes are left. However, this technique may benefit from existing active research in point-sampled geometry; there exists for example a technique for simplifying point-sampled surfaces [PGK02] and a technique for raytracing point-sampled surfaces [AA03].

5.3 Discrete implicit

One of the properties of the shape description proposed in the Section 5.1 is that it does not break when stretched. This property combined with the topology changing deformations proposed in Chapter 3 enables the artist to alter and maintain the topology of his shape in an explicit manner. But in a scenario where the artist would want to experiment on his shape by changing the topology very often, this explicit topology control can be a burden. The motivation of this section is to propose a shape description which easily changes topology, i.e. fuses portions of the shape that are close enough and breaks the shape when it is too thin. Note that although in the shape description proposed in Section 5.2 holes may appear, such holes are considered an artifacts since their appearance is not controlled,

The shape description we propose in this section relies on a discrete scalar field. This shape description is ideal for uncontrolled topology changes. The scalar field $d(\mathrm{p})$ holds the signed distance from the point $\mathrm{p} \in \mathbb{R}^3$ to the surface of the shape, and describes the shape as follows:

$$\begin{cases} d(\mathrm{p}) > 0 & \text{if p is outside the shape} \\ d(\mathrm{p}) < 0 & \text{if p is inside the shape} \\ d(\mathrm{p}) = 0 & \text{if p is on the surface} \end{cases} \tag{5.16}$$

To display the surface, the zero set of the scalar field d must be extracted. In the next section we propose to do this with a marching cubes algorithm [WMW86]. Then we present a method to deform this surface description.

5.3.1 Surface location

The information about the surface is discrete, located at the nodes of a regular grid in the form of signed distances to the surface. To define a continuous scalar field in $\mathbb{R}^3$ whose zero set defines the surface, the discrete signed distance must be interpolated or approximated within all cubic cells defined by eight nodes. The union of all the pieces constitutes a shape that can be displayed with any polygon rendering engine. Extracting the piecewise polygons constitutes the marching cube algorithm. Since a cube contains eight vertices that are inside or outside the shape, each possible configuration can be encoded on 8 bits, and a static table translates each of the 256 configurations into one out of 19 canonic configurations [NB93]. The location of the surface can sometimes be ambiguous, thus we use the *preferred polarity* disambiguation strategy, encoded in the table: for instance although Figure 5.6(III) and Figure 5.6(XV) are the dual of each other, their corresponding triangulation is different in order to avoid holes in the surface. We advise generating this correspondence table automatically to reduce human errors, for instance using an algorithm that transforms each of the 256 configurations, and tests it against the canonic configurations until matching. Table 5.6 shows the nineteen canonic configurations with their corresponding triangulations. Eleven extra figures are shown in light gray, and are the duals of some of the nineteen canonic configurations. To be displayed, the surface needs to be shaded, and the normals to the surface are required.

5.3.2 Surface normal

Since the scalar $d(\mathrm{p})$ is the signed distance from p to the shape, the gradient of the scalar field d measured on the surface is perpendicular to the surface, and points outwards. Thus in the following, normal and gradient on the surface are synonyms. Note that the gradient does not need to be unitized since the magnitude of the gradient of a signed distance field is equal to 1 at every point: this property defines a distance field.

Surface gradient along an edge

The vertices of the polyhedron produced by marching cube lie on the edges of the regular grid. Because normals are required to shade a surface, it is important to have an accurate estimate of the normal since the normals quality will dramatically affect the visual quality of the shape. Let us consider a pair of vertices defining a cube edge $(\mathrm{v}_0, \mathrm{v}_1)$ aligned with axis $\vec{e}_x$, parameterized in u as follows: $\{(1-u)\mathrm{v}_0 + u\mathrm{v}_1, u \in [0,1]\}$.

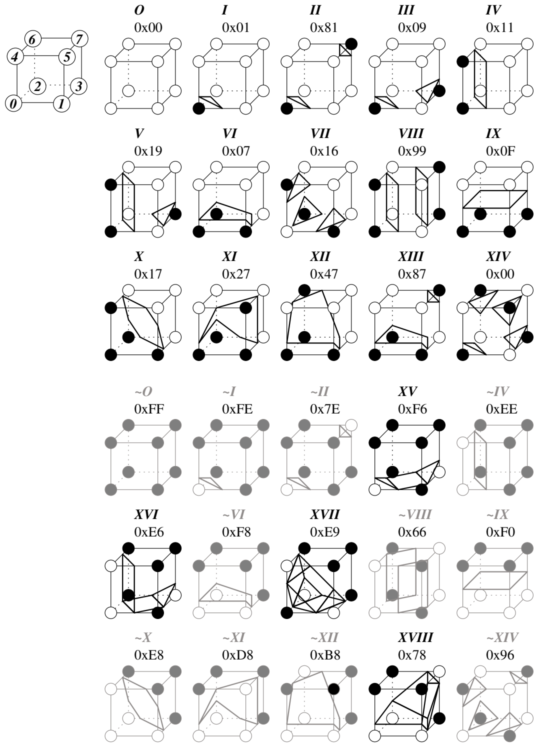


Figure 5.6: *The 19 canonic configurations with their corresponding triangulations. Black symbolizes a vertex inside the shape while white symbolizes a vertex outside the shape. The 11 extra light gray figures shown are the duals of some of the 19 canonic configurations. Between two dual configurations, the orientation of the surface swaps.*

We approximate the gradient $\vec{\gamma} = (\gamma_x, \gamma_y, \gamma_z)^\top$ at parameter u along this edge with the following expressions that were obtained by differentiating along x, y and z simple polynomials that interpolate the distance:

$$\begin{aligned} \gamma_x &= d_1 - d_0 \\ \gamma_y &= 0.5((1-u)(d_4 - d_2) + u(d_5 - d_3)) \\ \gamma_z &= 0.5((1-u)(d_8 - d_6) + u(d_9 - d_7)) \end{aligned} \tag{5.17}$$

The correspondence between the signed distances d_i and the nodes nearby the edge are shown in Figure 5.7. The gradient along an edge aligned with axes $\vec{e}_y$ or $\vec{e}_z$ is simply obtained by rotating the subscripts consistently.

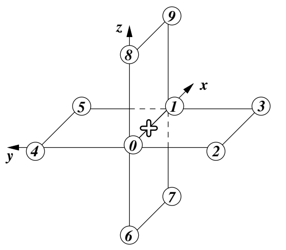


Figure 5.7: *Gradient of a discrete scalar field along an edge. The position along the edge parameterized in u, where the gradient is evaluated, is symbolized by a cross.*

5.3.3 Deforming the surface

To deform the shape, a naive approach would be to deform the node locations since they hold the signed distance to the shape. This approach would suffer the same limitation as an explicit non-updated shape: in stretched areas the shape may become under-sampled, and further modeling would not be possible. It is however possible to use a *semi-Lagrangian* method, which preserves the sampling resolution and that is used for advecting fluids [Sta99]. This method requires two grids swapped at every time step.

In the context of a fluid, although the animated scalar value is a density rather than a distance, the principle is the same. To find the new fluid density at each node, the technique consists of tracing backward with a particle the inverse of the flow and finding the location that would have landed exactly on the node. Since by tracing the flow backward, the location found is not necessarily another grid-point, an interpolation is used to get a value for the density between nodes (trilinear interpolation or higher order interpolation schemes).

In the context of a deformed shape, the technique is similar. For each node, the deformation's inverse is applied to its location: the scalar value at the location obtained gives the new value of the point. Since only the inverse deformation is required to deform a discrete implicit shape, the function of a deformation can be defined only by its inverse, as opposed to the function itself.

An inverse deformation

For a single tool, the deformation's inverse function is simply obtained based on Equation (3.11) where the matrix $M_{\tau_k \mapsto \tau_{k+1}}$ of a sub-function is replaced with $M^{-1}_{\tau_k \mapsto \tau_{k+1}}$, and the sub-functions $f_{\tau_k \mapsto \tau_{k+1}}$ are applied in reverse order:

$$\begin{array}{rcl} f^{-1}_{t_i \mapsto t_{i+1}}(\mathrm{p}) & = & \overset{0}{\underset{k=s-1}{\Omega}} f^{-1}_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) \\ \text{where } f^{-1}_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) & = & \left(\phi_{\tau_{k+1}}(\mathrm{p}) \odot M^{-1}_{\tau_k \mapsto \tau_{k+1}}\right) \cdot \mathrm{p} \end{array} \tag{5.18}$$

For multiple tools, the inverse sub-functions are obtained in a similar manner, and are applied in reverse order as well. The following gives the sub-functions:

$$f^{-1}_{\tau_k \mapsto \tau_{k+1}}(\mathrm{p}) = \begin{cases} \mathrm{p} & \text{if } \sum \phi_{j,\tau_{k+1}} = 0 \\ \exp\left(\frac{1-\prod_j (1-\phi_{j,\tau_{k+1}})}{\sum_j \phi_{j,\tau_{k+1}}} \sum_j \left(\phi_{j,\tau_{k+1}} \log(M_j^{-1})\right)\right) \cdot \mathrm{p} & \text{otherwise} \end{cases} \tag{5.19}$$

Maintaining a coherent structure

The signed distance field $d(\mathrm{p})$ satisfies the following property: $\|\nabla d(\mathrm{p})\| = 1$, a special case of the Eikonal equation [Bae01]. After the artist performs a deformation on the shape and hence on the field, this property may not hold, and the scalar field must be corrected. Fortunately there exist propagation procedures called the Fast Marching Method (FMM) and High Accuracy Fast Marching Method (HAFMM) that can update a field to make it satisfy the Eikonal equation within a bounded error. Presenting these algorithms is beyond our scope, and practical information on their implementation can be found in [Bae01]. These methods require a forward and backward approximation of the gradient at the nodes to extrapolate the distance value. Given a regular grid, one can compute the expression of the gradient at the grid-points by defining along each axis an n^{th} order polynomial that interpolates the grid nodes, and differentiate it. Hence a first order approximation of the gradient of d projected on an axis as shown in Figure 5.8 is:

- Backward: $d_0 - d_{n1}$
- Forward: $d_{p1} - d_0$

A second order approximation of the gradient is:

- Backward: $\frac{1}{2}(d_{n2} - 4d_{n1} + 3d_0)$
- Centered: $\frac{1}{2}(d_{p1} - d_{n1})$
- Forward: $\frac{1}{2}(-3d_0 + 4d_{p1} - d_{p2})$

The FMM and HAFMM methods can be restrained to update the discrete distance field within a neighborhood of the surface, thus using them does not require updating the entire grid.

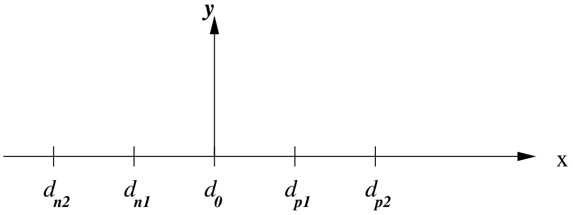


Figure 5.8: *Node positions for computing the gradient of* d *along axis* x.

5.3.4 Algorithm

```
Compute the number of required steps, $s$
for each step $k$ from 0 to $s-1$ do
    Compute the tool's bounding box, $B_k$
    Advect the scalar field of cells intersecting $B_k$, using an inverse
    Update the distance with an FMM in a neighborhood of the surface
    Update polygonization with a marching cube in cells intersecting $B_k$
    Place the tool at next position $\left(\frac{k+1}{s} \odot M\right) \cdot M_{t_i}$
end for
```

5.3.5 Results and limitations

With a discrete implicit surface, the detail at which an artist can model a shape is limited by the resolution of the grid. Since shape modeling needs to be decently interactive, this limits the grid size to $\approx 512^3$ on current computing devices, and the results are of much lesser quality than the ones achieved with the updated mesh method. However, the limited grid resolution presents an advantage since it enables the artist to change the topology easily by blurring the detail of the geometry. Note that with this approach, if we stored surface details such as color in the grid, they would be subject to smearing artifacts for the same reason. Also, the frame rate achieved with this technique on current computing devices is lower than with the updated mesh. A possible direction for future work is to investigate the possibility of using an adaptive grid, for instance inspired by [FCG02].

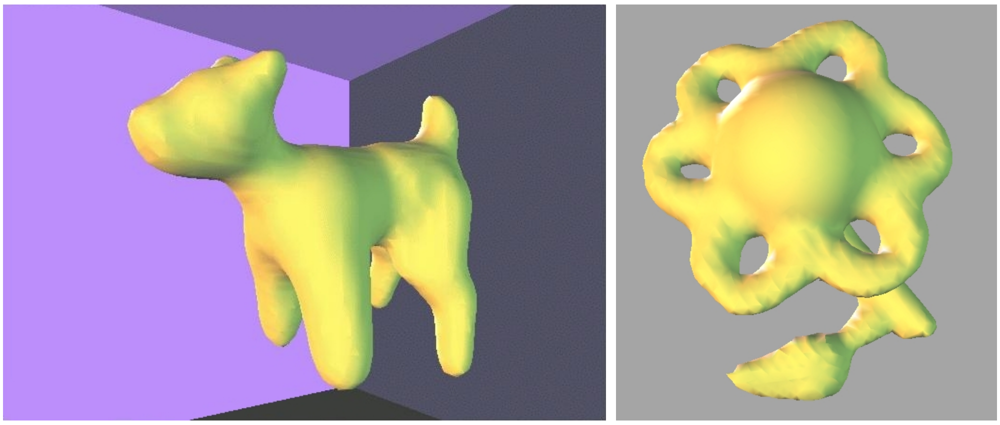


Figure 5.9: *Examples of simple shapes modeled with a discrete implicit shape representation. Left: dog. Right: flower.*

5.4 Ray-tracing

In the cases where the deformations $f_{t_i \mapsto t_{i+1}}$ can be inverted, authors have proposed an alternative way to do rendering [Bar84, KY97]: by deforming the visualization instead of deforming the shape description, i.e. by rendering the shape S_{t_0} in a deformed space in which rays of light are not lines but curves:

$$\left(\mathop{\Omega}_{i=0}^{n-1} f_{t_i \mapsto t_{i+1}}\right)^{-1}(\mathbb{R}^3) = \mathop{\Omega}_{i=0}^{n-1} f^{-1}_{t_{n-i-1} \mapsto t_{n-i}}(\mathbb{R}^3) \tag{5.20}$$

With this method, the visualized shape does not suffer from problems related to surface sampling with polygons, hence the motivation here is a method that is capable of producing high quality results at a low memory cost. The trade-off however is that the shape is slow to display because this requires efficient methods for rendering with curves of light [Bar84, KY97, MW01]. The context of use for this fourth method is that the artist first models the shape interactively with one of the shape descriptions of Section 5.1, 5.3 or 5.2, and then this method is used for a final rendering. Note that for rendering a shape in this manner, both the deformation function and inverse functions may be required. Figure 5.10 shows this principle applied to a simple deformation.

5.4.1 Deformation inverses

We propose three different ways of computing the inverse of the function of our space deformations for the simple case of a single tool. Let us denote by p a point and by

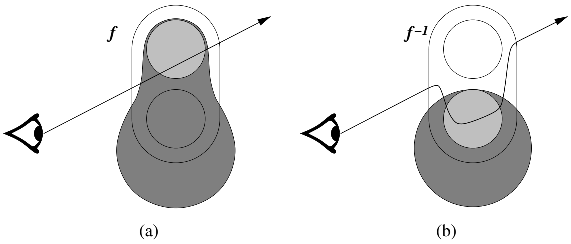


Figure 5.10: *(a) Deformed scene, visualized with rays of light. (b) Equivalent undeformed scene, visualized with curves of light.*

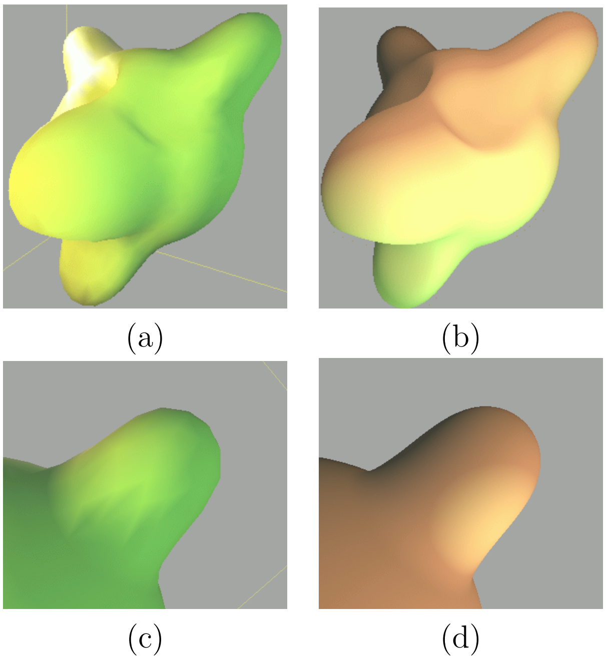

(a) (b)
(c) (d)

Figure 5.11: *(a) Object visualized in the interactive modeler. (b) Object produced by a ray-casting algorithm. (c) and (d) show the advantage of ray-casting over polygons: the shape's shading does not suffer from a poorly sampled surface.*

p′ its image under the deformation. The forward relation between these points is the following:

$$\mathrm{p}' = (\phi(\mathrm{p}) \odot M_{t_i \mapsto t_{i+1}}) \cdot \mathrm{p} \tag{5.21}$$

In order to invert this function, it must be solved for $\mathrm{p} \in \mathbb{R}^3$. Thus one must compute:

$$\mathrm{p} = (\phi(\mathrm{p}) \odot M_{t_i \mapsto t_{i+1}})^{-1} \cdot \mathrm{p}' \tag{5.22}$$

Using the right hand side of Equation 5.22 to directly compute p is not possible, since the amount of deformation $\phi(\mathrm{p})$ at p is needed on the right hand side. However, the matrix transformation $M_{t_i \mapsto t_{i+1}}$ defines streamlines, that we can parameterize with $h \in \mathbb{R}$ as follows: $h \odot M_{t_i \mapsto t_{i+1}}$. If a point p is mapped onto a point p′, then they must belong to the same streamline. Thus we can conveniently reduce the dimensions of the solution space. Let us rearrange this equation to make the streamline appear, by applying ϕ to both sides:

$$\phi(\mathrm{p}) = \phi((\phi(\mathrm{p}) \odot M_{t_i \mapsto t_{i+1}})^{-1} \cdot \mathrm{p}') \tag{5.23}$$

If we replace $\phi(\mathrm{p})$ by h, the equation can be solved along a streamline as follows:

$$\phi((h \odot M_{t_i \mapsto t_{i+1}})^{-1} \cdot \mathrm{p}') - h = 0 \ , \quad h \in [0,1] \tag{5.24}$$

Once h is found, the inverse image of p' can be obtained: $\mathrm{p} = (h \odot M_{t_i \mapsto t_{i+1}})^{-1} \cdot \mathrm{p}'$. Since there corresponds a scalar h to each point p', we can interpret the solution as a scalar field $h(\mathrm{p}')$ associated with the inverse deformation. To find $h(\mathrm{p}')$, we propose to use a Newton-Raphson iteration, i.e. at each iteration, a new temporary value for h is found by intersecting the tangent line to the above energy function with the abscissa. Since in Section 3.3 the scalar function $\phi(\mathrm{p})$ is defined as $\mu \circ d(\mathrm{p})$, the following is a particular form of the Newton-Raphson procedure:

```
**INVERTNEWTON**()
$h = 0$
**repeat**
    $h_0 = h$
    $d = d((h_0 \odot M_{t_i \mapsto t_{i+1}})^{-1} \cdot \mathrm{p}')$
    $h_1 = \mu(d)$
    $h \mathrel{-}= \frac{(h_0 - h_1)}{1 - \frac{\mathrm{d}d}{\mathrm{d}h}(h_0)\mu'(d)}$
**until** $|h_1 - h_0| \leq \epsilon$
$\mathrm{p} = ((-h) \odot M_{t_i \mapsto t_{i+1}}) \cdot \mathrm{p}'$
```

In the above, the derivative terms are expensive to compute. It appears however that h_1 and h are very close to each other while iterating since $\phi(M^{-1}_{t_i \mapsto t_{i+1}} \cdot \mathrm{p}')$, the scalar field transformed by $M_{t_i \mapsto t_{i+1}}$, is spatially very close to the unknown scalar field $h(\mathrm{p}')$ that defines the inverse transformation. Thus we propose a faster iteration that converges in practice. If non-convergence is detected, the previous procedure can be used. Table 5.12 shows that the following procedure is more efficient, since each loop is cheaper.

```
**INVERTFASTER**()
$h = 0$
**repeat**
    $h_0 = h$
    $h = \phi((h_0 \odot M_{t_i \mapsto t_{i+1}})^{-1} \cdot \mathrm{p}')$
**until** $|h - h_0| \leq \epsilon$
$\mathrm{p} = ((-h) \odot M_{t_i \mapsto t_{i+1}}) \cdot \mathrm{p}'$
```

	Newton-Raphson	Faster N.-R.
translation	14	13
scale	17	16
rotation	74	76

Figure 5.12: *Average number of iterations for inverse function procedures.*

In **INVERTNEWTON** or **INVERTFASTER**, the numerical expressions are expensive to compute; we propose a closed-form for an approximate inverse, already used in Section 5.3. The advantage of using this procedure is that it can be generalized to multiple tools. Most importantly, if the function itself never needs to be evaluated, we can decide to define deformation functions by their inverse, and consider the following expression to be exact:

```
INVERTCLOSEDFORM()
$\mathrm{p} = ((\phi(\mathrm{p}')) \odot M^{-1}_{t_i \mapsto t_{i+1}}) \cdot \mathrm{p}'$
```

5.4.2 Ray-casting algorithms

We have implemented a naive method that uses **INVERTFASTER**() for rendering a deformed shape using ray-marching: it consists of taking steps along a curve of light starting at the position of the camera, until intersection with the undeformed shape is found.

A ray of light L intersecting the final shape $S(t_n)$ corresponds to a curve of light $(M^{\beta}_{t_0 \mapsto t_n})^{-1}(L)$ intersecting the initial shape $S(t_0)$, as shown in figure 5.10. Finding the intersection with the sphere along the curve is delicate, since equally spaced samples l_j on L may not be equally spaced once undeformed into $(M^{\beta}_{t_0 \mapsto t_n})^{-1}(l_j)$. However, with an upper bound on the expansion factor of each small transformation $\frac{1}{n} \odot (M^{\beta}_{t_i \mapsto t_{i+1}})^{-1}$, the length of a step can be bounded. Let us denote by $\mathrm{p}(t) = \mathrm{s} + \vec{\mathrm{d}}t$ a ray. The bound is:

$$\max(\|\frac{\partial f(\mathrm{s} + \vec{\mathrm{d}}t)}{\partial t}\|) \tag{5.25}$$

Let us define a ray $\mathrm{p}(t) = \mathrm{s} + t\vec{\mathrm{d}}$, and consider a simple translation deformation $f(\mathrm{p}) = \mathrm{p} + \phi(\mathrm{p})\vec{\mathrm{t}}$. Using the definition of ϕ in Equation (3.3) and the bound on μ of Equation (3.4), the following bound on the expansion is obtained:

$$\begin{array}{rcccc} \|\frac{\partial}{\partial t}(f(\mathrm{s} + \vec{\mathrm{d}}t))\| = & \|\frac{\partial}{\partial t}(\mathrm{p} + \phi(\mathrm{p})\vec{\mathrm{t}})\| & = & \|\vec{\mathrm{d}} + \vec{\mathrm{t}}\frac{\partial}{\partial t}(\phi(\mathrm{p}))\| \\ < & 1 + \|\vec{\mathrm{t}}\|\frac{\partial}{\partial t}(\phi(\mathrm{p})) & < & 1 + \frac{15\|\vec{\mathrm{t}}\|}{8\lambda} \end{array} \tag{5.26}$$

Thus with the expansion factors $e_i = 1 + \frac{15\|\vec{\mathrm{t}}\|}{8\lambda}$, the following is a naive algorithm, illustrated in Figure 5.13:

1. undeform p_i
2. while deforming p_i, accumulate compression factor by multiplying them with $\prod_j e_j$. The result gives the radius of a safety region, within which a step can be taken for the next point p_{i+1} on the ray.

5.4.3 Results and improvements

The results shown in Figure 5.4 illustrate the advantage of using an inverse raytracing as opposed to deforming the shape explicitly: computational effort is spent to generate

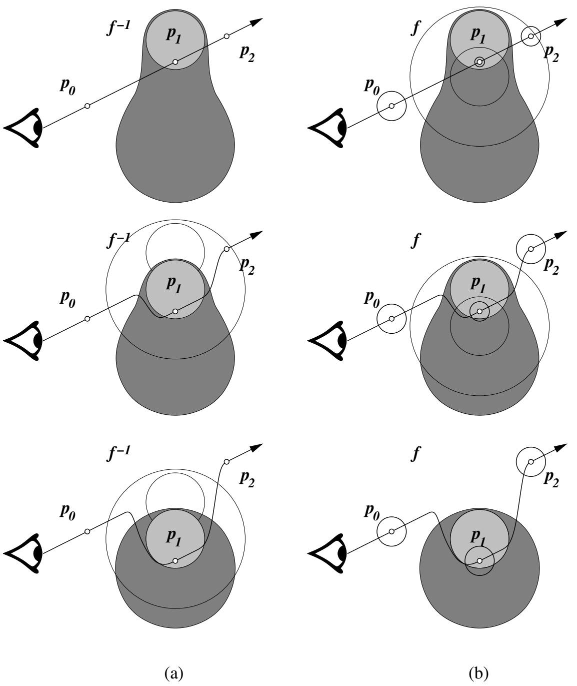


Figure 5.13: *(a) Inverse deformation applied to three points, from top to bottom. (b) Deformation of the points, while accumulating compression factors from bottom to top. All neighborhoods do not decrease equally.*

a particular view of the shape. The major drawback of the method presented is that it does not take into account the fact that space may be deformed anywhere, as opposed to just near the shape's surface. Thus light curves may take an incredibly complicated path before reaching the shape, and make rendering unnecessarily slow. Although we have not pursued further investigation, we propose possible improvements. Firstly, the above method spends a lot of time deforming points that are far from the surface. Deforming the portion of rays that are near the surface would reduce the cost, and these portions may be found using the forward deformation of the shape presented in Section 5.1. Secondly, our bound on the expansion defines a spherical neighborhood at each point: using differential properties such as the displacement gradient tensor would permit the definition of ellipsoidal neighborhoods whose principal directions are aligned with the directions of strain. Thirdly, instead of marching along the ray until the shape is found, rays may be deformed using an analogous algorithm to the one we have used for updating a mesh describing a 2D surface in Section 5.1.

5.5 Texturing with spherical springs

In previous sections and chapters, the surface color of our shapes is uniform, and is controlled only with global parameters, as illustrated for instance in Figure 5.14. Computer Graphics applications often require the user to describe local surface properties. It is however inappropriate to describe surface properties with geometry, and a popular and efficient approach is to apply 2D textures onto the surface using a mapping technique. The aim of this section is to show that mapping textures onto our shapes is possible, and can be done in a controlled manner.

If we ignore topology changes and only consider foldover-free continuous space deformation, our shapes are homeomorphic to a sphere. Thus to each point on the surface of our shapes s_i corresponds a point on the surface of the unit sphere x_i . We use this property to define a texture mapping on our shapes that minimizes a user-defined energy, e.g. stretching. In the following we call *parametric space* the unit sphere, and *parametric coordinates* its elements.

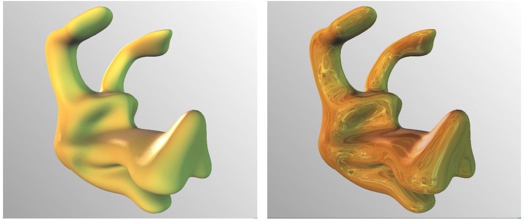


Figure 5.14: The surface property can be controlled globally. Left: plastic surface. Right: reflective surface.

The unit sphere is a space where geodesic distances can be defined easily, as the length of the shortest arc connecting two points. Dynamic elements can therefore be efficiently animated in the sphere, provided that they are represented adequately. We present in this section the basic elements for dynamic simulation in the unit sphere. The formulas apply to quaternions, and are very similar to formulas for animating particles in Euclidean space [WB01]. Spherical springs can be used for texturing our shapes or imported shapes, and for performing any other task requiring some minimization process in the unit sphere. In this section, we focus on performing relaxation of the parametric coordinates of a shape homeomorphic to a sphere. The first step is to introduce simple springs in the spherical texture space. Because of its bent nature, it is undesirable to use linear distance springs between two points in texture space: the points would have to be reprojected onto the sphere very often, and the behavior of long springs would be unpredictable. We believe that spherical springs is the most natural way to define springs in the unit sphere.

5.5.1 Quaternions

Quaternions are introduced in the preliminary section on notation. An excellent reference on quaternions is [DKL98]. The operations we perform on quaternions are inspired from the formalism of [Ale02a], similarly to sweepers in Chapter 3. We use the logarithmic space to perform a linear combination of unit quaternions. The advantage of using unit quaternions instead of matrices is the availability of closed-form formulas for the log and exp. In the following sections, we use the fact that $\exp^{\log \mathbf{p} + \log \mathbf{q}}$ is a commutative combination of quaternions $\mathbf{p}$ and $\mathbf{q}$.

5.5.2 Spherical forces

Spherical forces are defined between point-quaternions $\mathbf{x}_{i\in[1,n]}$. Each point-quaternion corresponds to a vertex on the surface of the shape s_i. By acting on the quaternions $\mathbf{x}_i$, the mapping from the shape's surface to the sphere can be modified. By defining springs between $\mathbf{x}_i$, this mapping can be made to minimize some criteria.

Given a scalar condition $C(\mathbf{x}_i)$ which we want to be zero, C gives rise to a force quaternion $\mathbf{f}$, whose expression parallels the one for Euclidean forces [WB01], and is obtained from the expression:

$$\mathbf{f} = (\dot{\nabla} C)^{-kC} \tag{5.27}$$

where $\dot{\nabla}$ denotes the spherical gradient quaternion defined in the following subsection, and where k is a stiffness constant of our choice. The expression of the logarithm of the force is closer to the expression given by [WB01]:

$$\log \mathbf{f} = -k\ C \log(\dot{\nabla} C) \tag{5.28}$$

From now on, all forces will be expressed with their logarithm: this yields more efficient formulas since the log of a quaternion can be handled as a mere three-dimensional vector. Moreover it simplifies the writing and makes more obvious the parallelism with Euclidean physics.

Spherical gradient

The spherical gradient is the equivalent of the straight vector Cartesian gradient, but constrained to the sphere; thus it is a unit quaternion.

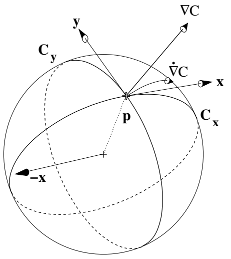


Figure 5.15: Spherical gradient.

Let us consider a scalar function $C(\mathrm{x})$, where x belongs to the unit sphere. As shown in Figure 5.15, let us consider two great circles S_x and S_y that are orthogonal, defined by the tangent vectors $\vec{\mathrm{v}}_x$ and $\vec{\mathrm{v}}_y$, and the point x at which the great circles intersect, such that $\mathrm{p} = \vec{\mathrm{v}}_x \times \vec{\mathrm{v}}_y$. The quaternion $(0, \vec{\mathrm{v}}_y)$ defines a rotation of angle π of p in direction $\vec{\mathrm{v}}_x$ along great circle S_x, and the quaternion $(0, -\vec{\mathrm{v}}_x)$ defines a rotation of angle π of p in direction $\vec{\mathrm{v}}_y$ along great circle S_y. In those terms, the spherical gradient of a function C is the quaternion:

$$\dot{\nabla} C = \exp(\frac{1}{2}((\nabla C \cdot \vec{\mathrm{v}}_x)\vec{\mathrm{v}}_y - (\nabla C \cdot \vec{\mathrm{v}}_y)\vec{\mathrm{v}}_x)) \tag{5.29}$$

In logarithm space:

$$\begin{aligned} 2 \log \dot{\nabla} C &= (\nabla C \cdot \vec{\mathrm{v}}_x)\vec{\mathrm{v}}_y - (\nabla C \cdot \vec{\mathrm{v}}_y)\vec{\mathrm{v}}_x \\ &= \mathrm{p} \times \nabla C \end{aligned} \tag{5.30}$$

It can be observed that the spherical gradient has no component along p. In spherical coordinates, the reader can verify that it also disappears for the parametric gradient: the spherical gradient at p does not make the point p spin on itself.

Angle spring

The angle spring aims at keeping a constant angle between two point-quaternions. Although we do not use it for our examples, it is the most simple spring, and demonstrates simply how a spherical force can be specified. Consider two mass points $\mathbf{x}_i$ and $\mathbf{x}_j$, connected with a spherical spring with rest angle θ_{rest} and stiffness coefficient $k_s \in \mathbb{R}^+$ as shown in Figure 5.16. The condition we use is:

$$C(\mathbf{x}_i, \mathbf{x}_j) = \theta_{ij} - \theta_{\text{rest}} \tag{5.31}$$

To evaluate θ_{ij}, we use arccos $\in [0, \pi]$:

$$\theta_{ij}(\mathbf{x}_i, \mathbf{x}_j) = \arccos(\vec{\mathrm{x}}_i \cdot \vec{\mathrm{x}}_j) \tag{5.32}$$

Thus the Cartesian gradient in $\vec{\mathrm{x}}_i$ of the condition C is:

$$\nabla C = \nabla \theta_{ij} = \frac{-\vec{\mathrm{x}}_j}{\sqrt{1 - (\vec{\mathrm{x}}_i \cdot \vec{\mathrm{x}}_j)}} \tag{5.33}$$

The logarithm of the spherical gradient in $\vec{\mathrm{x}}_i$, derived using Equation (5.30), is:

$$\log(\nabla \dot{C}) = \frac{\vec{\mathrm{x}}_i \times \vec{\mathrm{x}}_j}{\sqrt{1 - (\vec{\mathrm{x}}_i \cdot \vec{\mathrm{x}}_j)}} \tag{5.34}$$

Substituting $\log(\nabla \dot{C})$ for the above in Equation (5.28), the log of the force $\mathbf{f}_{j \to i} \in H_1$ that applies on $\mathbf{x}_i$, is simply the quaternion:

$$\log \mathbf{f}_{j \to i} = -k_s \arccos(\vec{\mathrm{x}}_i \cdot \vec{\mathrm{x}}_j) \frac{\vec{\mathrm{x}}_i \times \vec{\mathrm{x}}_j}{\sqrt{1 - (\vec{\mathrm{x}}_i \cdot \vec{\mathrm{x}}_j)}} \tag{5.35}$$

Note that by using the log instead of quaternion directly, the stiffness k_s won't suffer from the periodicity of cos and sin, thus large forces are handled naturally. Although a large stiffness may cause the physical integration to be unstable, the particles will always stay on the sphere.

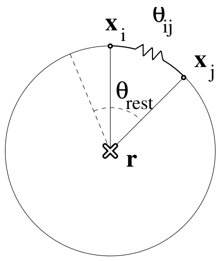


Figure 5.16: *Spherical angle spring.*

Solid angle spring

The solid angle spring aims at keeping the area of a spherical triangle $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k$ proportional to the area of a triangle on the surface of shape $\mathrm{s}_i, \mathrm{s}_j, \mathrm{s}_k$. The area of a spherical triangle is called the solid angle, and is equal to the sum of the angles between the arcs joining the quaternions. Since three arcs define two complementary spherical triangles whose union cover the entire sphere, 4π, this force is defined to keep constant the area of the spherical triangle whose vertices are clockwise oriented when observed from outside the sphere (see Figure (5.17)). If we define the arctan with values within $[-\pi, \pi]$, the effect is to flip the triangle smoothly if the vertices are wrongly oriented. Consider three point-quaternions $\mathbf{x}_i$, $\mathbf{x}_j$ and $\mathbf{x}_k$. Let Θ_{ijk} be the solid angle of the triangle ijk, and let Θ_{rest} be the solid angle at rest. The condition we use is:

$$C(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) = \Theta_{ijk} - \Theta_{\text{rest}} \tag{5.36}$$

An efficient expression of Θ_{ijk} is given by [vOS83], which we can further simplify for points on the unit sphere:

$$\Theta_{ijk} = 2\arctan\left(\frac{\mathrm{x}_i \cdot (\mathrm{x}_j \times \mathrm{x}_k)}{1 + \mathrm{x}_i \cdot \mathrm{x}_j + \mathrm{x}_j \cdot \mathrm{x}_k + \mathrm{x}_k \cdot \mathrm{x}_i}\right) \tag{5.37}$$

Let us call α and β the numerator and denominator of the fraction inside the arctan. As in [vOS83], we use the C++ function `atan2` with α and β to compute the arctan. The Cartesian gradient in x_i of the solid angle is:

$$\frac{1}{2}\nabla C = \frac{1}{2}\nabla\Theta_{ijk} = \frac{(\mathrm{x}_j \times \mathrm{x}_k)\beta - (\mathrm{x}_j + \mathrm{x}_k)\alpha}{\alpha^2 + \beta^2} \tag{5.38}$$

The following is the logarithm of the spherical gradient in $\mathbf{x}_i$ (see the definition of the spherical gradient in Equation (5.30)):

$$\log \dot{\nabla}\Theta_{ijk} = \frac{1}{2}\mathrm{x}_i \times \nabla\Theta_{ijk} \tag{5.39}$$

Similar formulas are obtained for the gradient in $\mathbf{x}_j$ and $\mathbf{x}_k$ by rotating the indices. The logarithm of the force is obtained by replacing C and $\log \dot{\nabla}\Theta_{ijk}$ in Equation (5.28) by the above values. In our examples, we have used a value Θ_{rest} proportional to the area of the triangle joining the position vertices of the textured triangle in Euclidean space a_{ijk}, and inversely proportional to the area of the shape $A = \sum a_{ijk}$:

$$\Theta_{\text{rest}} = 4\pi\frac{a_{ijk}}{A} \tag{5.40}$$

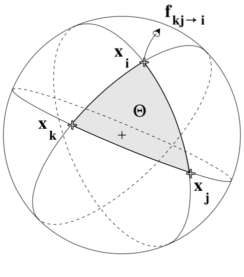


Figure 5.17: *Solid angle spring.*

Triangle angle spring

The triangle angle spring aims at keeping the angles of a spherical triangle $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k$ proportional to the angles of a triangle $\mathrm{s}_i, \mathrm{s}_j, \mathrm{s}_k$. Let us denote by ζ_{ijk} the angle $\widehat{\mathbf{x}_k, \mathbf{x}_i, \mathbf{x}_k}$.

$$C(\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k) = \zeta_{ijk} - \zeta_{\text{rest}} \tag{5.41}$$

To evaluate ζ_{ijk}:

$$\zeta_{ijk} = \arccos\left(\frac{(\mathrm{x}_j - \mathrm{x}_i) \cdot (\mathrm{x}_k - \mathrm{x}_i)}{\|\mathrm{x}_j - \mathrm{x}_i\| \|\mathrm{x}_k - \mathrm{x}_i\|}\right) \tag{5.42}$$

The following is the logarithm of the spherical gradient in $\mathbf{x}_i$

$$\begin{array}{rcl} \log \dot{\nabla} C & = & \log \dot{\nabla} \zeta_{ijk} \\ & = & \frac{\mathrm{x}_i \times (\mathrm{x}_k((\mathrm{x}_k - \mathrm{x}_i) \cdot (\mathrm{x}_k - \mathrm{x}_j))/l_{ik}^2 - \mathrm{x}_j((\mathrm{x}_j - \mathrm{x}_i) \cdot (\mathrm{x}_k - \mathrm{x}_j))/l_{ij}^2)}{l_{ij} l_{ik} \sqrt{1 - \zeta_{ijk}^2}} \end{array} \tag{5.43}$$

The logarithm of the force is obtained by replacing C and $\log \dot{\nabla} C$ in Equation (5.28) by the above values. In our examples, we have used a value ζ_{rest} proportional to the angle of the vertices of triangle in Euclidean space $\widehat{\mathrm{s}_k, \mathrm{s}_i, \mathrm{s}_k}$.

$$\zeta_{\text{rest}} = \arccos\left(\frac{(\mathrm{s}_j - \mathrm{s}_i) \cdot (\mathrm{s}_k - \mathrm{s}_i)}{\|\mathrm{s}_j - \mathrm{s}_i\| \|\mathrm{s}_k - \mathrm{s}_i\|}\right) \tag{5.44}$$

Particle damping

The damping of a particle $\mathbf{p}_i$ is defined using its velocity $\mathbf{v}_i \in H_1$, which is defined in Section 5.5.3. Its effect is to keep the particle in place. Let $k_d \in \mathbb{R}^+$ be the damping coefficient. The damping is:

$$\log \mathbf{d}_i = -k_d \log \mathbf{v}_i \tag{5.45}$$

where $k_d \in [0, \frac{m}{\delta t}]$ (mass m and time step δt are defined later on). However, as remarked [BW98], this damping is a simple viscous function that dissipates kinetic energy independently of the type of motion.

Spring damping

To define the damping function for the force induced by a condition C, we propose similarly to [BW98] to project the velocity on the force:

$$\log \mathbf{d}_i = -k_d (\log \dot{\nabla} C \cdot \log \mathbf{v}_i) \log \dot{\nabla} C \tag{5.46}$$

Combining forces

To compute the overall force applied on $\mathbf{p}_i$, we combine all the forces in logarithmic space:

$$\mathbf{F}_i = \exp\left(\sum_j (\log \mathbf{f}_{j \to i} - k_d \log \mathbf{v}_i)\right) \tag{5.47}$$

5.5.3 Physical integration

For computing efficiency purposes, all the computations are done in logarithmic space, thus the exponential is computed only once for each particle at each time step. Also, combining quaternion logarithms is a natural way of expressing their simultaneous effect. Note that in the following, the use of quaternions has to be done carefully, depending on whether they represent vectors or positions.

Acceleration: To use an explicit Euler integration, we propose an adaptation of the second law of motion for quaternions, i.e. in the spherical space a force is the logarithm of a rotation:

$$\log \mathbf{a}_i = \frac{1}{m} \sum_j \log \mathbf{f}_{j \to i} \tag{5.48}$$

Velocity: At the beginning of the simulation, the logarithm of the velocity is initialized to $(0,0,0)^\top$. Given a previous velocity $\log \mathbf{v}_i$, the logarithm of the new velocity at next time step δt adds up a portion δt of the acceleration. We choose the following formula for computing the new velocity:

$$\log \mathbf{v}'_i = \log \mathbf{v}_i + \delta t \log \mathbf{a}_i \tag{5.49}$$

Another possibility for computing $\mathbf{v}'_i$ would be to use $\mathbf{v}'_i = \mathbf{a}_i^{\delta t} * \mathbf{v}_i$, but this is not necessarily better and would yield more time consuming formula.

Position: The new position is obtained by rotating the point using the velocity:

$$\mathbf{p}'_i = \mathbf{v}_i^{\delta t} * \mathbf{p}_i * \mathbf{v}_i^{-\delta t} \tag{5.50}$$

where

$$\begin{array}{rcl} \mathbf{v}_i^{\delta t} & = & \exp^{\delta t \log \mathbf{v}_i} \\ \mathbf{v}_i^{-\delta t} & = & \bar{\mathbf{v}}_i^{\delta t} \text{ , the conjugate.} \end{array} \tag{5.51}$$

5.5.4 Summary

All the computations are done in logarithmic space, and thus are quite efficient. For each particle:

- compute the current acceleration (see Section 5.5.2 for computing forces):

$$\log \mathbf{a}_i = \frac{1}{m} \sum_j \log \mathbf{f}_{j \to i} \tag{5.52}$$

- compute the new velocity:

$$\log \mathbf{v}'_i = \log \mathbf{v}_i + \delta t \log \mathbf{a}_i \tag{5.53}$$

- update the position:

$$\begin{array}{rcl} \mathbf{d}_v & = & \exp^{\delta t \log \mathbf{v}'_i} \\ \mathbf{p}(t + \delta t) & = & \mathbf{d}_v * \mathbf{p}(t) * \bar{\mathbf{d}}_v \end{array} \tag{5.54}$$

5.5.5 Results and limitations

A simple example of the angle spring is shown in Figure 5.18. The motion integration is done with a simple explicit Euler integration scheme described above [WB01]. One of the advantages of using spherical springs on the unit sphere instead of using Euclidean springs is apparent when the system is unstable: instead of sending the particles to infinity, they stay on the sphere.

In Figure 5.19, we have used spherical springs on the parametric coordinates of the vertices of a shape while it is being deformed interactively: the texture expands in over-covered areas and contracts in under-covered areas. Since the spherical springs energy minimization is done while the shape is deformed, the state of minimum energy of the texture coordinates is quickly reached. Since a sphere does not map to a flat 2D texture without distortion, we have used a cube mapping to apply 6 images onto the unit sphere.

In Figure 5.20, we have used spherical springs to define texture coordinates of an imported model. In this example, we have used spherical coordinates to apply the chessboard image to the sphere. Note that since the spring relaxation algorithm is performed in the sphere, it is independent of the type of texture mapping applied to the sphere.

Note that we did not use arc springs in our examples. If we did, a problem would occur when a triangle flips with the solid angle spring: at the transition state, the vertices would gain extremely large velocities under the force of the arc spring. This may be corrected by multiplying the arc spring condition with some function of the solid angle.

Note that spherical springs do not generalize to topology changes. For instance after insertion of a hole in the shape, the sphere presents two discontinuities: two points that were originally distant become connected in the parametric space. With one hole, handling a torus is probably better than using a sphere with two marked discontinuities. At this stage, we do not have a solution to the general case.

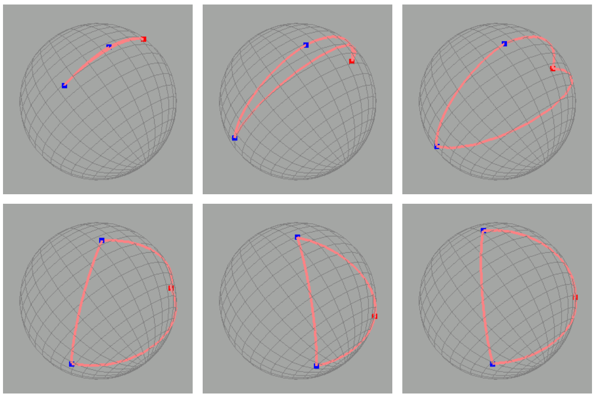
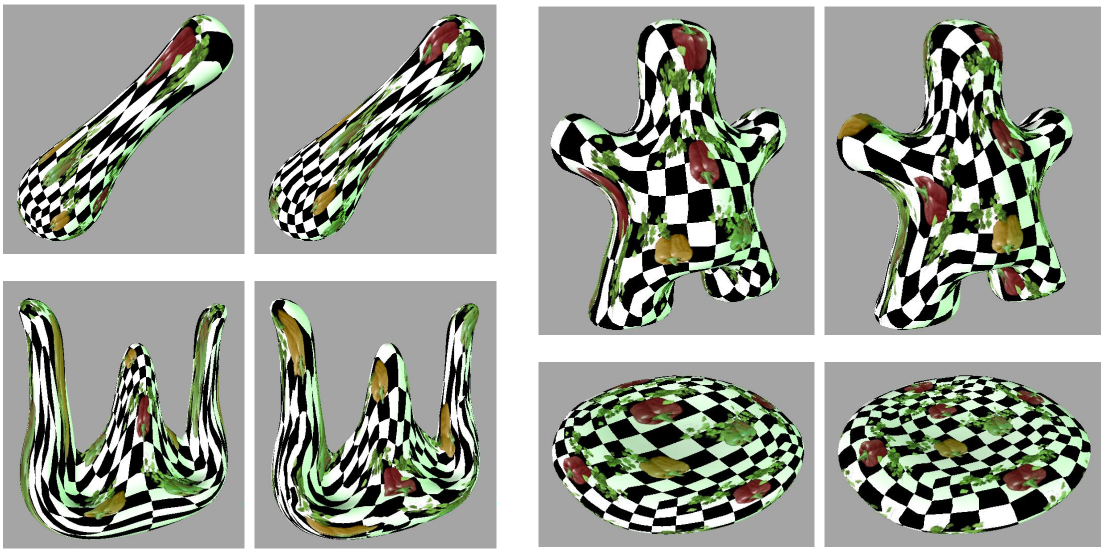


Figure 5.18: *Animation of three spherical springs with rest angles equal to $\pi/2$. At rest (sixth picture), the triangle is an octant of the sphere.*



(left) (right)

Figure 5.19: *Comparison without (left) and with (right) spherical spring. Notice how the capsicums are less stretched on the right pictures, and how they are of comparable size overall.*

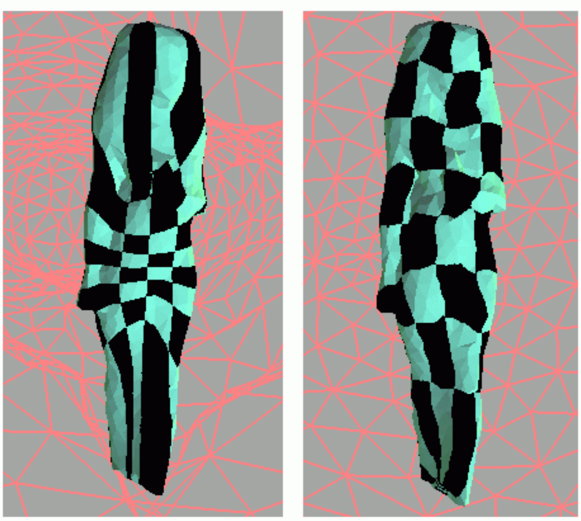


Figure 5.20: *Spherical springs applied to an imported mesh (right). The initial texture coordinates (left) were simply obtained by projecting the vertices of the mesh onto a sphere placed at the center of the mesh.*

Chapter 6

Simulation of Smoke based on Vortex Filament Primitives

The work presented in this chapter was done in collaboration with Fabrice Neyret[1], and was presented as a paper at the Symposium on Computer Animation 2005 [AN05].

Chapter 4 introduces two space deformation techniques: swept-fluid based on a physically plausible fluid model and swirling-sweepers, a technique which is in appearance procedurally-based. The motivation of this chapter is to show that there is in fact a link between swirling-sweepers and a physically plausible fluid model. We define in this chapter a fluid model as a solid basis for defining future space deformation techniques that intend to mimic fluid behaviour. We illustrate our model in the more demanding context of animating an incompressible gas.

Various paths have been followed in order to adapt the simulation of gaseous phenomena to the peculiar requirements of CG applications: Eulerian [KM90, FM96], Lagrangian [MP89], semi-Lagrangian [Sta99, FSJ01] or spectral [Sta01]. A common challenge is to obtain the *fastest computation time* for the *maximum possible fluid resolution*. Knowing that graphics applications often trade accuracy for efficiency can help in choosing a scheme: e.g. the unconditional stability of [Sta99] permits using large time steps. Constraints due to the grids in Eulerian methods are released by [SCP$^+$04]. Mixed models can increase the apparent resolution by relying on simpler models at small scales (carried by high level primitives), such as noise [Ney03, SSEH03] or procedural models [WH91], or by combining such high-resolution simple 3D models to interpolated 2D simulations [RNGF03].

A general problem (especially important for CG) is to obtain a *living fluid*: most methods suffer numerical dissipation (intrinsic to Eulerian approaches, and due to

---

[1] CNRS Researcher in Computer Graphics, Member of the Evasion team from the GRAVIR-IMAG laboratory, INRIA Rhone-Alpes project.

resampling for Lagrangian approaches) in which small scale eddies die too quickly. To counter this, vorticity confinement was introduced in CG by [FSJ01], and sub-grid analytical models can be used [Ney03].

Another challenge is to ease the *control of the fluid* by an artist. The high-level primitives of the mixed models mentioned above are naturally adapted for this. More recently, techniques have been proposed to target specific states of the fluid by controlling the whole field [FL04, MTPS04], or by controlling particles [REN$^+$04, PCS04].

In this chapter, we introduce a new path to CG fluids: simulation in the *3D vorticity space*[2]. The vorticity space is dual to the velocity space as explained in Section 6.1. But numerous fluid features appear more structured in vorticity space, as a multi-scale combination of *vortex filaments* (like swirls, tornadoes) and *vortex rings* (like smoke rings, explosion plumes, mushroom clouds). In numerous interesting situations the flow is characterized by a few such primitives, which are tightly connected to the visible features of the fluid: these primitives are thus interesting handles for user control. We represent them as 1D curves, i.e. connected particles. Our model permits the user to interactively create and modify such primitives. Procedural generation can also be used (e.g. to introduce turbulent fluctuations, as a physically more reasonable feature than the usual noise functions).

The velocity field can be reconstructed at any time from the vorticity filaments thanks to the Biot-Savart law, Equation (6.3), i.e. the animated flow is totally defined by a few animated curves. Thus, the motion can be simulated quickly. Moreover, these curves can easily be edited, replayed for tuning, keyframed, interpolated, or even stored, in the spirit of [PCS04]. Costly rendering can be done later at arbitrary resolution, and this will not modify the animation contrary to Eulerian or semi-Lagrangian methods (as mentioned in [LF02]).

Inconveniently, each vorticity element induces motion in the whole field so that computing the Biot-Savart integral can be time consuming. Moreover, local self-induction can cause numerical instabilities. In this chapter, we introduce a new formalization which enables a higher order scheme, thus larger time steps. We also introduce an approximation which permits analytical integration and also stabilizes the simulation. Moreover, we propose a filament LOD (level of detail) scheme. Thus, our model allows us to efficiently compute the velocity induced at any given location by all the filaments.

Our contributions are:

- The introduction of 3D vortex methods to CG fluids, using 1D-Lagrangian filaments (well adapted to user control), with dynamic LOD.
- An original formalization of Biot-Savart based on twists, permitting a higher-order solver, and a modified kernel weight yielding a fast and stable simulation.
- An original adaptive scheme for density particles permitting a high quality rendering.

Section 6.1 reviews the concepts, equations and properties related to the vortical aspects of fluids. In Section 6.2 we revisit these equations in order to permit a higher

---

[2] Note that this path has already been introduced in 2D by [GLG95]. But 3D vorticity is very different since it is vectorial and highly spatially structured (see Section 6.1).

order solver, and we detail our filament representation. Practical adaptations are proposed in Section 6.3. In Section 6.4, we describe the representation and the simulation of our vortex primitives, comprising an adaptive scheme, an LOD hierarchy and a noise function. Smoke particles are treated in Section 6.5. We present our interactive application in Section 6.6 and discuss results in Section 6.7.

6.1 The physics of vorticity and filaments

6.1.1 The Lagrangian vorticity expression of fluids

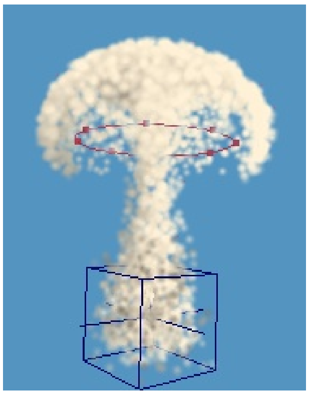


Figure 6.1: *Flow induced by a vortex ring (in red). Smoke particles are generated in the box.*

Vorticity-based approaches – called *Vortex methods* – are already used in CFD (Computational Fluid Dynamics) [CK00]. They are especially suitable for turbulent fields and simulation of eddies since they track thin features better. Because the thickness of these features can be far smaller than any reasonable grid cell step, they are also more accurate [CMOV02]. As a Lagrangian approach, they do not suffer from the numerical dissipation which tends to kill small eddy structures when using Eulerian approaches.

The vorticity $\vec{\omega}$ is defined as $\nabla \times \vec{\mathrm{v}}$ where $\vec{\mathrm{v}}$ is the velocity. Assuming incompressibility, mass conservation can be expressed as $\nabla \cdot \vec{\mathrm{v}} = 0$. The Lagrangian formalism follows the properties of fluid parcels represented by *particles* and advected along the flow with velocity $\vec{\mathrm{v}}$. The Lagrangian formalism is especially adapted to CFD since the non-zero vorticity is generally concentrated in loci (the vortices) which follow the flow.

In 2D, the Lagrangian vorticity form of the Navier-Stokes equation for inviscid fluids is simply $\frac{\mathrm{d}\vec{\omega}}{\mathrm{d}t} = 0$. It means that once created, vorticity never dies and is simply advected along the field. Handling the 2D case is simple since it only requires vortices placed at isolated particles. It has been used in CG by [GLG95]. In 3D, the equation is:

$$\frac{\mathrm{d}\vec{\omega}}{\mathrm{d}t} = (\vec{\omega} \cdot \nabla)\vec{\mathrm{v}} \tag{6.1}$$

The above means that while following the flow, vortices are stretched by its local deformation. The 3D case is far more complicated since the vorticity is a vector and spatially structured in filaments, often rings (i.e. closed loops). The *strength* of a filament is defined as the circulation Γ, a scalar:

$$\Gamma = \oint_L \vec{\mathrm{v}} \cdot \vec{\mathrm{t}}\, \mathrm{d}l = \iint_S \vec{\omega} \cdot \vec{\mathrm{n}}\, \mathrm{d}s \tag{6.2}$$

where S is a cross section of the tube and $\vec{\mathrm{n}}$ is the normal to the surface of the section, and L the border closed-curve of S and $\vec{\mathrm{t}}$ is the tangent to the curve. This vortical structure has several consequences [Bat67, Mar97]:

- Firstly, since vorticity in a location induces a rotational motion everywhere in the fluid, parts of the same filament induce each-other: filaments induce self-deformation (e.g. oscillation modes) and global motion (e.g. a smoke ring moves straight due to its self-induction). And of course, filaments interact with each other. For example, two close parallel rings leapfrog through each other (i.e. one sucks the other which will suck the first right after, and so on).

- Secondly, filaments never die[3] and behave in a peculiar way when stretched: as stated by Kelvin's theorem [Rut90], the circulation Γ is constant both along the filament and in time, which means that vorticity *increases* when the radius of the filament decreases due to the stretching[4]. This behavior can be interpreted as the conservation of the angular momentum. As the fluid motion creates ubiquitous stretching, vortical areas quickly tend to concentrate into tubes, then to increasingly thinner filaments. This complex structure of turbulent fluids is what makes them so complicated to simulate, and explains why the classical methods lose important features.

Due to stretching, vortex tubes are often assumed to have a small core, hence the name *filament*. Thus they can be conveniently represented by a 1D curve together with the circulation Γ, rather than a very concentrated explicit $\vec{\omega}$ field. Since the circulation is preserved over time, no equation is needed for the evolution of the vortex strength.

Lagrangian primitives used in vortex methods can be 0D (regular particles), 1D (curves made of connected particles) and even 2D (since vorticity often starts as a 2D stretched layer between two fluid areas before degenerating into vortex tubes then filaments). Note that 0D particles [Gha01] lose the filament coherency, and have to explicitly track the effects of stretching.

6.1.2 Reconstructing the velocity field

Recovering $\vec{\mathrm{v}}$ from $\vec{\omega}$, i.e. inverting $\vec{\omega} = \nabla \times \vec{\mathrm{v}}$, is not easy. A solution evaluated at a point p is given by the Biot-Savart law:

$$\vec{\mathrm{v}}(\mathrm{p}) = \frac{1}{4\pi} \iiint_{\mathrm{x}} \frac{\vec{\omega}(\mathrm{x}) \times (\mathrm{p} - \mathrm{x})}{\|\mathrm{p} - \mathrm{x}\|^3} \mathrm{dx} \qquad (6.3)$$

Three comments can be made about this formula:

- Firstly, this solution is not unique: Equation (6.3) only provides one divergence-free solution, to which we can add any velocity field $\vec{\mathrm{v}}_h$ satisfying both $\nabla \times \vec{\mathrm{v}}_h = 0$ and the fluid hypotheses (mass conservation and boundary conditions). This harmonic field $\vec{\mathrm{v}}_h$ corresponds to a solution of the flow using the simplest assumptions. It is solved separately by vorticity-based physical methods and in our case we will assume it is given by the user or simply zero, so that in the following we do not handle it explicitly and we only consider the Biot-Savart solution.

---

[3] As long as the inviscid hypothesis is valid. In practice filaments are dissipated when they become too thin.

[4] In particular, turbulence is made of a dense soup of very thin very rapidly rotating filaments.

- Secondly, evaluating Equation (6.3) at p using numerical integration is expensive, since it is performed with x over the entire space. To avoid this cost, *Vortex-In-Cell* methods rely instead on a finite difference solver on a grid to invert $\vec{\omega} = \nabla \times \vec{\mathrm{v}}$, with the various drawbacks associated with grid sampling (including dissipation). We introduce analytical integration and LODs to avoid this cost.

- Thirdly, the integrand diverges at p = x, which corresponds to the evaluation of local self-induction. This can lead to a singularity[5], or at least to numerical instabilities. We will introduce a modified Biot-Savart kernel weight to avoid this.

6.1.3 Boundary conditions

Incoming and outgoing flux are typically accounted for by the harmonic component of the velocity field (i.e. by solving with the divergence-free irrotational assumption). Regarding contact with borders, it has been shown [LNC91] that the interaction of a ring filament with a border with slip condition (i.e. only the normal component of velocity cancels) is equivalent to the interaction of the filament with its mirror image. Note that a no-slip condition can be obtained by inserting vorticity near the boundary so that the tangent component of the velocity cancels.

6.2 Our choice of representation and solver

In Vortex Methods, the Biot-Savart law is used to compute the velocity of particles at any position p. We propose a new formalization of this law (detailed in Section 6.2.1), introducing a *whirl* operator which lets us recover higher order information concerning the trajectory of particles. This enhances the precision of our solver (presented in Section 6.2.3) and thus permits larger time steps.

6.2.1 Our Biot-Savart reformulation

Let us denote by the scalar function $\beta_{\mathrm{BS}}(\mathrm{x}) = \frac{1}{4\pi\|\mathrm{x}\|^3}$ the *Biot-Savart kernel weight.* Let us denote by R the 4×4 matrix whose multiplication with a point p rotates p around the center x, axis $\vec{\omega}$ with an angle $\|\vec{\omega}\|$.

The matrix $\log R$ has a simple expression, which we give in Equation (6.4). We show in Chapter 7 how to compute the exponential of it to obtain a rotation matrix. It conveniently satisfies $(\log R) \cdot \mathrm{p} = \vec{\omega} \times \mathrm{p} - \vec{\omega} \times \mathrm{x} = \vec{\omega} \times (\mathrm{p} - \mathrm{x})$.

$$\log R = \langle \omega,\ \mathrm{x} \times \omega \rangle = \begin{pmatrix} 0 & -\omega_z & \omega_y & x_y\omega_z - x_z\omega_y \\ \omega_z & 0 & -\omega_x & x_z\omega_x - x_x\omega_z \\ -\omega_y & \omega_x & 0 & x_x\omega_y - x_y\omega_x \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{6.4}$$

---

[5] And it does for several theoretical filament models, which makes their theoretical study so complicated.

This matrix is sparse and it has only 6 degrees of freedom: it can be represented by two 3D vectors $\vec{\omega}$ and $\mathrm{x} \times \vec{\omega}$. We denote by $\langle \vec{\omega},\ \mathrm{x} \times \vec{\omega} \rangle$ the operator producing the matrix (6.4) from these two parameters. Note that this operator is linear. It satisfies:

$$\langle \mathrm{a}_1 + \mathrm{a}_2,\ \mathrm{b}_1 + \mathrm{b}_2 \rangle = \langle \mathrm{a}_1,\ \mathrm{b}_1 \rangle + \langle \mathrm{a}_2,\ \mathrm{b}_2 \rangle \quad , \qquad \langle \alpha \mathrm{a},\ \alpha \mathrm{b} \rangle = \alpha \langle \mathrm{a},\ \mathrm{b} \rangle$$

Thus, the Biot-Savart law can be re-written:

$$\vec{\mathrm{v}}(\mathrm{p}) = \left(\iiint_{\mathrm{x}} \beta_{\mathrm{BS}}(\mathrm{p} - \mathrm{x})\ \langle \vec{\omega},\ \mathrm{x} \times \vec{\omega} \rangle\ \mathrm{dx} \right) \cdot \mathrm{p} \tag{6.5}$$

We call the integrand in Equation (6.5) the *whirl of a vortex*. The expression of this matrix is:

$$\varphi(\mathrm{p}, \mathrm{x}, \vec{\omega}) \quad = \quad \beta_{\mathrm{BS}}(\mathrm{p} - \mathrm{x})\ \langle \vec{\omega},\ \mathrm{x} \times \vec{\omega} \rangle$$

Note that φ is linear in $\vec{\omega}$: $\quad \varphi(\mathrm{p}, \mathrm{x}, \vec{\omega}) = \|\vec{\omega}\| \varphi(\mathrm{p}, \mathrm{x}, \frac{\vec{\omega}}{\|\vec{\omega}\|})$. φ represents a rotation of center x, axis $\vec{\omega}$, with an angle proportional to $\|\vec{\omega}\|$ and decreasing away from the center. Stated another way, it represents the effect of an atomic *vortex element*. The angle of rotation is maximal at the vortex center x. The scalar $\beta_{\mathrm{BS}}(\mathrm{p} - \mathrm{x})$ describes how the rotation magnitude decreases with distance. Note that the rotation matrix can be recovered by $\exp(\varphi)$. We call the integral of φ the *whirl of a fluid*:

$$\Phi(\mathrm{p}) \quad = \quad \textstyle\iiint_{\mathrm{x}}\ \varphi(\mathrm{p}, \mathrm{x}, \vec{\omega})\ \mathrm{dx}$$

Thus, the Biot-Savart expression becomes:

$$\vec{\mathrm{v}}(\mathrm{p}) = \Phi(\mathrm{p}) \cdot \mathrm{p} \tag{6.6}$$

Due to the linearity of $\langle ,\ \rangle$ the matrix $\Phi(\mathrm{p})$ is as sparse as $\log R$, thus only 6 scalar integrals have to be calculated to obtain it. We will see in Section 6.4 that the expression can be reduced down to 3 or even 1 scalar integral in some situations. $\Phi(\mathrm{p})$ encodes a *twist*, i.e. a translation along a straight line together with a rotation about that line.

6.2.2 Spatial integral and field representation

$\Phi(\mathrm{p})$ has to be calculated at every point p to get the velocity induced by the vorticity field $\{\vec{\omega}(\mathrm{x}), \mathrm{x} \in \text{space}\}$. A numerical integration over the entire space would be very expensive.

We draw on the classical *vortex filaments* assumption mentioned in Section 6.1.1: we consider that the vorticity is concentrated in thin tubes (i.e. filaments) C_i and null elsewhere. Note that $\vec{\omega} = 0$ at some location does not mean that there is no motion there, since vorticity induces motion at a distance. The flow is thus entirely defined by the set $C = \{C_i, i \in [1, n]\}$. We will consider curves parameterized in arc-length.

The filaments are considered as differential elements, i.e. 1D curves with a formal radius $r(u)$. The vorticity is locally tangent to the curves. Various analytical profiles of the vorticity through a tube section are considered in the literature: e.g. constant or Gaussian. Physics regards the circulation Γ as the only meaningful notion, which is the integral of the vorticity projected on a section (see Equation (6.2)). Thus, introducing the notation $\vec{\Gamma}$ with $\|\vec{\Gamma}\| = \Gamma$ and $\frac{\vec{\Gamma}}{\|\vec{\Gamma}\|} = \frac{\vec{\omega}}{\|\vec{\omega}\|}$, we can call the expression $\varphi(\mathrm{p}, \mathrm{x}, \vec{\Gamma})$ the *whirl of a section* and forget about $\vec{\omega}$ and r. Kelvin's theorem states that Γ is constant along a filament and over time for inviscid fluids, even when considering stretching (see Section 6.1.1). As circulation represents the intuitive notion of the *strength* of a vortex filament, we treat it as a user-defined parameter.

However, we explain in Section 6.3.2 that it is still necessary to store and maintain the filament thickness $r(u)$ – which decreases with stretching – if we want to take viscosity into account, since its effect is highly dependent on r. Viscosity will affect the strength locally so that we also need $\vec{\Gamma}(u)$. Thus, each filament is defined by a parametric curve holding positions, circulation and radius:

$$C_i = \{\{\mathrm{x}_i(u), \vec{\Gamma}_i(u), r_i(u)\}, u \in [0, L_i]\} \tag{6.7}$$

Thus we now simply have to compute 1D integrals representing the *whirl of the filaments*:

$$\Phi(\mathrm{p}, C) = \sum_i \int_0^{L_i} \varphi(\mathrm{p}, \mathrm{x}_i(u), \vec{\Gamma}_i(u)) \, \mathrm{d}u \tag{6.8}$$

The set of filaments C is a parameter of Φ since the support of integration moves in time.

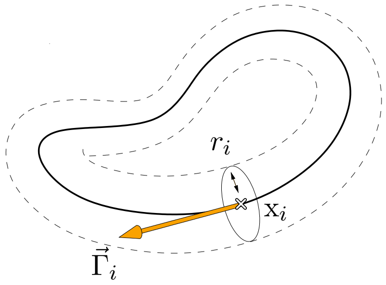


Figure 6.2: *Parameters defining a element of the filament: positions* x_i*, circulation along a vector* $\vec{\Gamma}_i$ *and radius* r_i*.*

6.2.3 Time integration scheme

Let us consider a particle at location p in the fluid. The matrix $\Phi(\mathrm{p}, C)$ in Equation (6.8) gives us access to the velocity at p through $\vec{\mathrm{v}}(\mathrm{p}) = \Phi(\mathrm{p}) \cdot \mathrm{p}$, and thus to an estimate of the trajectory of p during the time step $\delta\ t$: $\tilde{\mathrm{p}}' = \mathrm{p} + \tau\ \vec{\mathrm{v}}(\mathrm{p})\ ,\ \tau \in [0, \delta\ ]$. But the matrix Φ can provide more information than just the velocity. As we have already mentioned in Section 6.2.1, Φ encodes a *twist*, i.e. a rotation combined with a translation, whose matrix can be recovered with $\exp(\Phi)$ (see Appendix 7). This provides us with higher order information about the trajectory of p. Thus we compute the new location p′ after a time step δt as:

$$\mathrm{p}' = f(\mathrm{p}) = \exp(\delta t\ \Phi(\mathrm{p}, C)) \cdot \mathrm{p} \tag{6.9}$$

Since $f(\mathrm{p})$ is of higher order than a translation, the estimate p' is more accurate than $\tilde{\mathrm{p}}'$. This allows us to make larger time steps and therefore gain speed. Moreover, if the flow is a pure rotation, translation or twist, the reconstructed trajectory of p will exactly follow it regardless of the length of time step δt.

Note that the two first terms of the series expansion of exp give the linear trajectory $(\mathrm{I} + \delta t \Phi) \cdot \mathrm{p}$, so our scheme is asymptotically equivalent to a simple Euler integration step.

We use the scheme based on Equations (6.8) and (6.9) for animating marker particles in the fluid as well as the points x_i defining the filaments. The evolution of the set of filaments C after a time step δt is thus simply defined by $C' = f(C)$. This is simply a restatement of Equation (6.1).

6.3 Practical approximations and extensions

To gain even more efficiency, we want to avoid costly numerical integrations by obtaining closed forms as much as possible for the Biot-Savart integrals along the vortex primitives that we will define in Section 6.4.

For this, we replace the Biot-Savart kernel weight with another (see Section 6.3.1) which eases analytical integration and which is more stable.

At this stage we will have a working incompressible inviscid fluid in an unbounded space. We show how viscosity and boundary conditions can be introduced in Section 6.3.2.

6.3.1 Changing the Biot-Savart kernel weight

The Biot-Savart kernel weight β_{BS} has two drawbacks: it diverges at 0 ($\beta_{\mathrm{BS}}(0) = \infty$), leading to numerical instabilities for particles that are very close to a vortex center (typically, the neighbor nodes on the filament), and it generally disables closed forms for integrals.

We propose to replace the Biot-Savart kernel weight β_{BS} with another radial basis function β_{MS} which is defined and smooth around 0 and which eases the analytical integration:

$$\beta_{\mathrm{MS}}(\mathrm{x}) = \frac{1}{\pi(1 + \mathrm{x}^2/s^2)^2} \tag{6.10}$$

This kernel weight β_{MS} is proportional to the one introduced in [MS98] in the context of convolution surfaces. The coefficient s is a user controllable parameter related to the apparent thickness of the filament: the region closer than s to the curve tends to rotate like a solid core. During simulation s should be roughly proportional to r, but for stability it should not decrease below a threshold s_1. Thus we model $s(r)$ as $s_0 r + s_1 k(r)$ where $k(r)$ is a function decreasing from 1 to 0 and s_0, s_1 are such that $s(r)$ is monotonic. We chose $s_0 = 1$, $s_1 = \frac{2}{3}$ and $k(r) = e^{-\frac{3}{2}r}$.

This kernel weight smooths the local self-induction, but it also slightly underestimates the induction on distant particles. The resulting animation is still visually satisfactory. Note that changing the kernel weight β does not alter the incompressibility property.

6.3.2 Viscosity, stretching and boundaries

Viscosity: It has two effects on fluids. Firstly, it spatially smears quantities (velocity, vorticity, markers). As is often done, in this chapter we consider that this effect is negligible at visible scales by assuming the fluid is inviscid, which yields the simple equations we use. Secondly, it dampens filaments, and prevents them from becoming infinitely thin with infinite vorticity, which makes real fluids free of singularities. This is also the very mechanism which *dissipates* the vortical energy transmitted from higher scales. This effect occurs at very small scales, but it is important to take it into account in order to avoid singularities, endless accumulation of filaments and infinite growth of energy. We model this as a radius-dependent damping of the filament strength, done at each time step:

$$\Gamma_i'(u) = (1 - \nu(r_i(u)))^{\delta t}\, \Gamma_i(u) \tag{6.11}$$

Where $\nu(r)$ is a damping function decreasing from 1 to 0 with a characteristic viscous scale r_0. In our implementation we use $\nu(r) = \exp^{-\frac{r}{r_0}}$. Weak filaments are faded out to zero then destroyed. Note that independently from this physical decay, it is useful to allow the user to decide when to fade and kill a filament as mentioned in Section 6.6.

Radius Stretching: In order to know $r_i(u)$, we need to compute the vortex stretching during the animation of the filament. Let λ be the lengthening rate measured at a given filament location. The volume conservation of a small cylindrical portion of filament tells us that when its length multiplies by λ, its radius divides by $\sqrt{\lambda}$. Thus, we simply compute at each time step: $r_i'(u) = \frac{r_i(u)}{\sqrt{\lambda}}$.

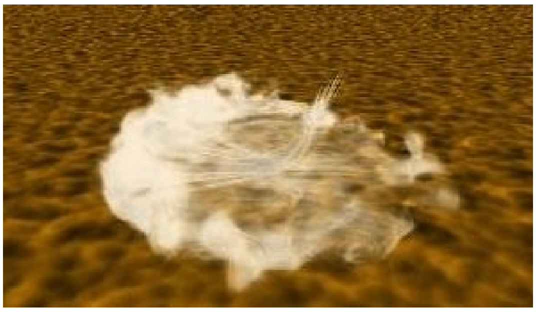


Figure 6.3: *A plume falling on the floor.*

Boundary Conditions: In this chapter, we only deal with flat motionless boundaries with a slip condition. As explained in Section 6.1.3, in the vortex formalism, one has simply to simulate the interaction of filaments with their mirror images through the border plane. Thus, we need to compute the whirl $\Phi'(\mathrm{p})$ of the mirrored filaments and its influence on a given point p. Conveniently, it is equivalent to compute the whirl $\Phi(\mathrm{p}')$ of the regular flow at point p' which is the mirror of p relative to the plane. Let us denote by S the mirroring operator relative to the plane (i.e. $\mathrm{p}' = S \cdot \mathrm{p}$). Then the mirrored flow is $\Phi'(\mathrm{p}) = S \cdot \Phi(S \cdot \mathrm{p}) \cdot S$. The total flow is simply $\Phi_{\mathrm{TOT}} = \Phi + \Phi'$. If p is on the plane with normal $\vec{\mathrm{n}}$ then $\exp(\Phi_{\mathrm{TOT}})$ is a transformation tangent to the plane, since $\vec{\mathrm{n}} \cdot (\Phi_{\mathrm{TOT}} \cdot \mathrm{p}) = 0$.

In practice, we only consider Φ' for filaments and p close enough to a boundary. Moreover, for moderately curved boundaries a tangent plane approximation can be made.

6.4 Our primitives of vorticity

The whirl Φ of a fluid is defined by Equation (6.8) as the sum of the whirl of each filament. The purpose of this section is to describe how we represent the filaments and how we compute their whirl, taking advantage of the adapted kernel weight β_{MS} defined in Equation (6.10). As we have seen in Section 6.2.3, from this whirl we can compute the displacement and the velocity at every point p in the flow. This is used to advect all the particles, including the filaments.

Every vortex element in the flow influences every particle. To save computations, we introduce a hierarchy of models to represent filaments, and LODs for the finest model.

- The finest filament model consists of a set of connected particles. The computation of the filament whirl is based on the integration of φ on its segments. We detail it in Section 6.4.2, as well as its LOD structure.
- The coarser level consists of a circular ring, treated in Section 6.4.1. A circle is an approximation which makes sense since flow perturbations often start as simple vorticity rings, which can remain circular for a while depending on the environment. It also makes sense to approximate small rings by circles since the extra detail would have little effect at distance. Conveniently, the whirl of a circle can be computed analytically which makes it especially efficient.
- Similarly, a coarse model should be handled for straight filaments. In fact, this case can be handled directly as the coarsest LOD level of the regular filament model.
- For the coarsest level we introduce a vortex noise model consisting of isolated vortex primitives. We detail it in Section 6.4.3.

For each of these models we describe how to evaluate their whirl and how to update their structure through simulation.

6.4.1 Circular ring

Circular Ring Whirl: A circle is defined at each time by a center c, a radius r_c, and a vector $\vec{\mathrm{z}}$ perpendicular to the circle plane. The symbolic integration of the section whirl along the circle gives the following whirl, similar to Equation 4.11:

$$\begin{array}{rclcrcl} \Phi_{\text{circle}}(\mathrm{p}) & = & r_c \int_u \varphi(\mathrm{p},u)\,\mathrm{d}u & = & \Gamma\, b\, \langle \vec{\eta},\ \mathrm{c} \times \vec{\eta} + a r_c \vec{\mathrm{z}} \rangle \\ & \text{where} & & a & = & (\mathrm{p}-\mathrm{c})^2 + s^2 + r_c^2 \\ \vec{\eta} & = & 2\vec{\mathrm{z}} \times (\mathrm{p}-\mathrm{c}) & b & = & \frac{2s^4 r_c}{\pi(a^2 - r_c^2 \vec{\eta}^2)^{3/2}} \end{array} \tag{6.12}$$

Circular Ring Advection: The advection of our circular ring is done in three steps: taking samples on that circle, advecting the samples, and fitting a circle to the newly obtained positions. When the circle fitting error is too high according to a user-defined criterion, it can be swapped with a closed deformable filament.

6.4.2 Deformable filament

A deformable filament C_i is represented with a polygonal curve, i.e. vertices connected by segments. The filament is simply deformed by advecting the vertices. The total whirl $\Phi_i(\mathrm{p}, C_i)$ generated at a point p by the polygonal curve is computed by summing the whirls generated by each segment. In the next subsection we describe how to compute the whirl of a segment. The polygonal curve to be used is determined according to our LOD scheme and an error criterion, described in the two following subsections. The result in p is reasonably valid for a neighborhood around p, and thus the LOD can be computed only once for a cluster of many particles. These clusters are defined using a floating grid which adapts to clouds of particles (typically, the smoke particles described in Section 6.5).

For defining the LODs of a deformable filament we build a binary-tree of polygonal curves, whose nodes are segments. The leaves represent the segments of the detailed filament, and each internal node represents a segment which is the *average* of its two children.

Segment Whirl: Let us define a segment $(\mathrm{p}_0, \mathrm{p}_1)$ parameterized in $u \in [0, 1]$. Let us denote by l the length of the segment. The symbolic integration of the section whirl along the segment gives:

$$\Phi_{\text{segment}}(\mathrm{p}) \quad = \quad l \textstyle\int \varphi(\mathrm{p}, u)\ \mathrm{d}u \quad = \quad \Gamma\ h(\mathrm{p})\ \langle \mathrm{p}_1 - \mathrm{p}_0,\ \mathrm{p}_0 \times \mathrm{p}_1 \rangle \tag{6.13}$$

where the scalar function $h = \int \beta_{\text{MS}}(\mathrm{p}, u)\ \mathrm{d}u$ is:

$$h(\mathrm{p}) = \frac{s^4}{2a^2\pi^2}\left(\frac{a_0}{d_0 + s^2} + \frac{a_1}{d_1 + s^2} + \frac{l^2}{a}(\arctan\frac{a_0}{a} + \arctan\frac{a_1}{a})\right)$$

$$\begin{array}{lll} \text{in which} & d_0 = \|\mathrm{p} - \mathrm{p}_0\|^2 & d_1 = \|\mathrm{p} - \mathrm{p}_1\|^2 \\ & d = \frac{1}{2}(d_0 + d_1 - l^2) & a^2 = d_0 d_1 - d^2 + l^2 s^2 \\ & a_0 = d_0 - d & a_1 = d_1 - d \end{array}$$

Since a repeated evaluation of the above expression is expensive, an accurate approximation is useful. If we denote by p_{min} and p_{max} the closest and farthest points from point p on the segment, we can minimize and maximize terms in the integrand of Equation (6.13): if $\|\beta_{\text{MS}}(\mathrm{p}_{\text{min}}) - \beta_{\text{MS}}(\mathrm{p}_{\text{max}})\| < \epsilon$, the following is a good approximation, i.e. it is a Riemann sum with two intervals [Weic]:

$$\begin{array}{rcl} \tilde{\Phi}_{\text{segment}}(\mathrm{p}) & = & \Gamma\ \tilde{h}(\mathrm{p})\ \langle \mathrm{p}_1 - \mathrm{p}_0,\ \mathrm{p}_0 \times \mathrm{p}_1 \rangle \\ \text{where}\quad \tilde{h}(\mathrm{p}) & = & \frac{1}{2}\left(\beta_{\text{MS}}(\mathrm{p}_{\text{min}}) + \beta_{\text{MS}}(\mathrm{p}_{\text{max}})\right) \end{array} \tag{6.14}$$

Building the LOD Tree: The filaments deform during the simulation, so their LOD tree has to be reconstructed at each time step. To build the tree bottom-up, all that is required is a method for averaging pairs of neighbor segments. Our criterion is to best preserve the whirl, i.e. the whirl of each level of the tree is as close as possible to the whirl of the levels below, for any point where it will be evaluated later.

Finding a polygonal curve whose whirl best matches the whirl of a polygonal curve with twice as many segments is an expensive minimization problem, that we cannot afford to solve interactively. We propose the following simple scheme which works well. Other schemes could be used, such as an inverse-subdivision scheme [SNBW03].

- The starting point is a detailed filament with 2^l segments. These correspond to the tree leaves.
- For each pair of neighbor segments $\{2i, 2i+1\}$, we define the parent segment i with a length equal to the sum of the lengths of its children, and intersecting the children at mid-length. For its circulation, we simply take the average.
- We repeat this step until the root is reached, i.e. a single segment for open filaments, and at least three segments for closed filaments.

Choosing the LOD of a Whirl: Determining the LOD of the whirl to be evaluated for a point p (and its neighborhood) is done top-down by subdividing the segments in an adaptive non-uniform manner. The segment subdivision criterion is based on an estimate of the error produced when using the whirl of a single segment Φ_{e_0} instead of the sum of the whirls of its two children Φ_{e_1} and Φ_{e_2}. The exact geometric error is the distance between the transform of p by the twist encoded by Φ_{e_0} and the transform of p by the twist encoded by $\Phi_{e_1} + \Phi_{e_2}$ (applying Equation (6.9)):

$$\epsilon(\mathrm{p}) = \| \ \exp(\delta t(\Phi_{e_1} + \Phi_{e_2})) \cdot \mathrm{p} - \exp(\delta t \Phi_{e_0}) \cdot \mathrm{p} \ \|$$

In order to save costly computations, we rely on two approximations to estimate this error. Firstly, we approximate twists with translations, i.e. a first order approximation of the exponential: $\exp(M) \approx \mathrm{I} + M$. Secondly, for computing the matrices Φ_{e_i} we estimate the costly integration of Equation (6.13) by bounding the kernel weight β_{MS} with a higher and lower bounds for each segment, as shown in Figure 6.4. Thus, an approximation of the error is:

$$\tilde{\epsilon}(\mathrm{p}) = \max_{ijk} \left\| \beta_{\mathrm{MS}}(\mathrm{q}_1^j) M_{e_1} \cdot \mathrm{p} + \beta_{\mathrm{MS}}(\mathrm{q}_2^k) M_{e_2} \cdot \mathrm{p} - \beta_{\mathrm{MS}}(\mathrm{q}_0^i) M_{e_0} \cdot \mathrm{p} \right\|^2$$

where

- M_{e_i} is the matrix $\delta t \ \Gamma_i \ \langle \mathrm{p}_1 + \mathrm{p}_0, \ \mathrm{p}_0 \times \mathrm{p}_1 \rangle$, associated with edge i
- $\mathrm{q}_i^0, \mathrm{q}_i^1$ are the closest and farthest points from p on e_i

Deformable Filament Advection: The leaf vertices of a filament are simply advected like particles, and the binary-tree is updated at each time step. Whenever the leaf segments themselves are too stretched and become undersampled, several solutions are available:

- Add an extra LOD level by splitting all the segments.
- Resample the curve evenly.
- Wait for the filament to naturally vanish, since the over-stretching weakens it (see Section 6.3.2).
- Let the user decide when to fade out the curve, e.g. by keyframing $\nu(t)$.

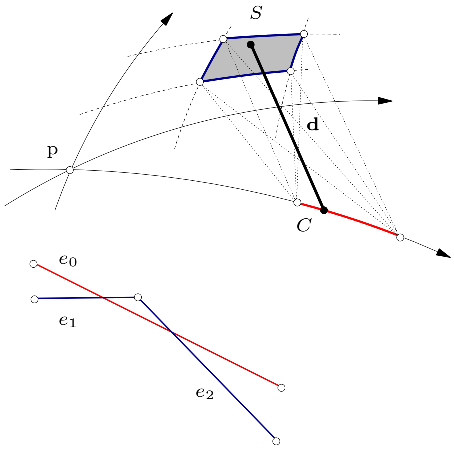


Figure 6.4: *The extremities of curve C (red) are bounds of the transformation of* p *by the flow of segment* e_0*. The edges of patch S (blue) are bounds to the transformation of* p *by the flow of segments* e_1 *and* e_2*. Using C and S, a higher bound to the distance d between the image of point* p *by a segment and its children can be found.*

6.4.3 Noise vortices

The amount of detail that can be simulated with CFD methods is limited, since an increase of resolution requires a significant increase of computing resources. In order to circumvent this limitation, tricks can be used in CG for amplifying realism: various kinds of noise functions have been proposed in the literature, such as Perlin Noise [Per85], flownoise [PN01] and stochastic divergence-free fields [SF93, RNGF03]. But their visual quality suffers from the fact that the noise does not satisfy the fluid properties: only the last kind is divergence-free, and all of them lack the temporal coherence of eddies. In our formalism, a user can model an efficient and high quality noise, by spawning *noise vortices* in areas where turbulence is wanted. A noise-vortex consists of a position c_i, an axis of rotation a_i and a rotation amplitude Γ_i. It only influences marker particles, within a radius of influence r_i. It is advected in the flow like the other particles.

An advected axis cannot be simply transformed using the Jacobian of the displacement $J(f)$ (where f is defined in Equation (6.9)) like a material tangent would: the stretching of the flow would tend to align neighboring axes along the axis of the local stretch. In order to keep unorganized noise axes, the eigenvectors of J could be used; but cases where they are undetermined arise. We propose a simple scheme in which these eigenvectors are attractors (when they exist): we attach a sort of local Frenet frame $(\vec{\mathrm{t}}, \vec{\mathrm{n}}, \vec{\mathrm{b}})$ to particles, composed of *tangent*, *normal* and *binormal* axes, updated as follows:

$$\vec{\mathrm{t}}' = J \cdot \vec{\mathrm{t}} \qquad \vec{\mathrm{n}}' = J^c \cdot \vec{\mathrm{n}} \qquad \vec{\mathrm{b}}' = \vec{\mathrm{t}} \times \vec{\mathrm{n}} \tag{6.15}$$

where J^c is the cofactors matrix[6] of J; see [Bar84] for justifications. Then we define

---

[6] If we denote by $\{\vec{\mathrm{j}}_0, \vec{\mathrm{j}}_1, \vec{\mathrm{j}}_2\}$ the columns of J, then the columns of J^c are $\{\vec{\mathrm{j}}_1 \times \vec{\mathrm{j}}_2, \vec{\mathrm{j}}_2 \times \vec{\mathrm{j}}_0, \vec{\mathrm{j}}_0 \times \vec{\mathrm{j}}_1\}$.

noise vortices among three categories, *tangent-vortices*, *normal-vortices* and *binormal-vortices*, whose rotation axes are defined by one of the frame axes.

6.5 Smoke particles

In CG applications the visual fluid features consist of interfaces (water surface), distribution of markers (smoke, cloud droplets, colors), or advection of objects (e.g., leaves). Advection of passive[7] objects is done easily with our method by simply evaluating the whirl at the object location: this provides the new object position as well as its rotation.

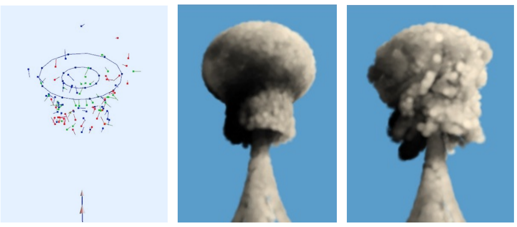


Figure 6.5: Left: *two leapfrogging filament and noise particles.* Middle: *flow simulated without the noise.* Right: *with the noise particles.*

The purpose of this section is to describe our representation of marker densities. Eulerian methods can either treat this density[8] as an extra quantity to be updated at grid nodes, or rely on particles spawned in the simulated flow [FM97]. Naturally, Lagrangian methods rely on particles, i.e. floating markers whose position p_i is carried by the flow.

Sizeless particles make it difficult to maintain a correct sampling of the visible features through simulation, and complex heuristics must be provided to generate new particles in undersampled dense areas. This can result in visual artifacts, especially for highly stretched flows. Instead, we consider *blob* particles [SF93] to which a reference size s_i and a density ρ_i are associated. The fluid parcel corresponding to this volume will distort in a complicated manner through time. [SF93] reproduced this effect on large blobs using backwarped rays, but this technique does not easily apply to real-time rendering.

Assuming small particles and that the magnitude of their strain is tiny compared to the large scale motion of the fluid, we handle linear anisotropic distortion of blobs, i.e. we simulate *ellipsoid* blobs. This enables long smooth particles, which gives a high quality result at low cost (see Figure 6.6). When the stretching becomes too high for the linear assumption, we split the particle.

The ellipsoid shape of our blobs is represented by a quadratic form Q_i, whose eigenvectors $\{\vec{e}_0, \vec{e}_1, \vec{e}_2\}$ and eigenvalues $\{\lambda_0, \lambda_1, \lambda_2\}$ give the ellipsoid principal axes and radii. They can be recovered by diagonalizing the matrix Q_i.

Stretching Smoke Particles: The strain added during a time step is given by the Jacobian[9] of the displacement $J = \nabla(f)$ where f is defined in Equation (6.9). The ellipsoid shape of our blobs directly represents the accumulated distortion. Starting

---

[7] Cross-interaction between large objects and the fluid is a complex problem that is beyond of the scope of this thesis.

[8] It is important to note that this is the *smoke* density, not the *fluid* density.

[9] Note that [Ney03] only considers the norm of the strain and thus does not capture the directional information.

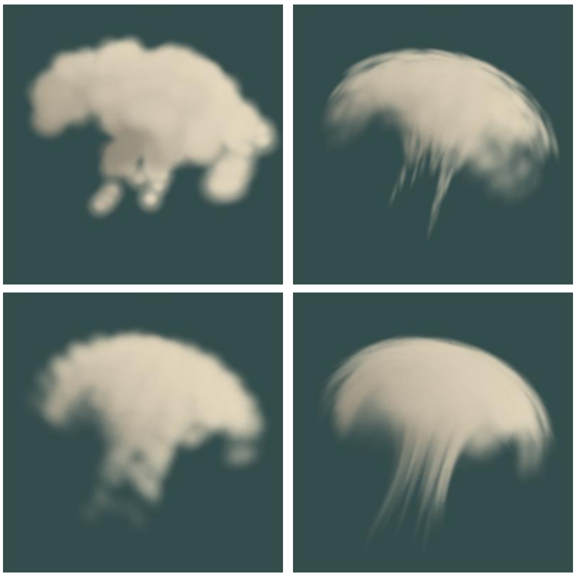


Figure 6.6: *Without (left column) and with (right column) particle distortion. Without (top row) and with (bottom row) particle splitting.*

with $Q_i = s_i \mathrm{I}$, at each time step we compute how the ellipsoid is deformed with $Q'_i = J(\mathrm{p}) \cdot Q_i \cdot J^\top(\mathrm{p})$. Note that incompressibility yields $\det(J) = 1$, so the volume of the blob is preserved (aside from numerical errors).

Let us show that the matrix Q_i gives the shape of an ellipsoid located at the particle's position. Let us define a parameterization of the unit sphere around the particle:

$$\mathrm{x} = (\cos\psi, \sin\psi\cos\theta, \sin\psi\sin\theta)^\top \quad , \quad (\psi,\theta) \in [0,\pi] \times [0, 2\pi] \tag{6.16}$$

By definition of J, the vectors $J \cdot \mathrm{x}$ are located on an ellipsoid. We can use this parameterization to build the covariance matrix of all the points on the ellipsoid, by integrating over the sphere using the solid angle $\mathrm{d}\omega$:

$$\begin{array}{rcl} C & = & \frac{3}{4\pi} \iint_\omega J \cdot \mathrm{x} \cdot (J \cdot \mathrm{x})^\top \mathrm{d}\omega \\ & = & J \cdot (\frac{3}{4\pi} \int_{\psi=0}^{2\pi} \int_{\theta=0}^{\pi} \mathrm{x} \cdot \mathrm{x}^\top \sin\theta \; \mathrm{d}\theta \; \mathrm{d}\psi) \cdot J^\top \\ & = & J \cdot J^\top \end{array} \tag{6.17}$$

Therefore, the covariance matrix of the vectors $J \cdot \mathrm{x}$ is the symmetric matrix $J \cdot J^\top$. The eigenvalues and eigenvectors of C give the axes and dimensions of the ellipsoid [GW92].

For clarity, we stress that the covariance of the Jacobian is not the rate of strain tensor, often denoted e_{ij} in fluid mechanics. The rate of strain tensor describes an ellipsoid at an instant in time, while the covariance of the Jacobian describes the ellipsoid after an interval of time.

Splitting Smoke Particles: Let us denote by λ_0 the largest eigenvalue of the matrix Q_i, corresponding to the main axis $\vec{\mathrm{e}}_0$ of the ellipsoid. When λ_0/s_i exceeds a threshold the particle is too stretched, so it is split across the stretching direction $\vec{\mathrm{e}}_0$. Two children particles are generated in place of the parent particle with the same axes, and radii $(\lambda_0/2, \lambda_1, \lambda_2)$, using a decomposition:

$$Q'_i = \begin{pmatrix} \vec{\mathrm{e}}_0 & \vec{\mathrm{e}}_1 & \vec{\mathrm{e}}_2 \end{pmatrix} \begin{pmatrix} \lambda_0/2 & 0 & 0 \\ 0 & \lambda_1 & 0 \\ 0 & 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \vec{\mathrm{e}}_0 & \vec{\mathrm{e}}_1 & \vec{\mathrm{e}}_2 \end{pmatrix}^{-1} \tag{6.18}$$

Each child keeps the same reference size s_i, inherits half of the density ρ_i, and is placed at point $\mathrm{p}_i \pm \frac{\lambda_0}{2}\vec{\mathrm{e}}_0$ ($\vec{\mathrm{e}}_0$ is assumed to be unitary).

Drawing Particles: An ellipsoid particle is easy to render since its projection in screen space can be obtained analytically. Given two orthogonal 3D unit vectors $\vec{\mathrm{x}}$, $\vec{\mathrm{y}}$ contained in the viewing plane, the 2×2 projected matrix Q_{2D} is:

$$Q_{2D} = \begin{pmatrix} \vec{\mathrm{x}}^\top \\ \vec{\mathrm{y}}^\top \end{pmatrix} \cdot Q_i \cdot \begin{pmatrix} \vec{\mathrm{x}} & \vec{\mathrm{y}} \end{pmatrix} \tag{6.19}$$

Thus, to render a particle ellipsoid we simply render a 2D ellipse on a billboard facing the camera: the size and orientation of the billboard are determined by Q_{2D} eigenvalues and eigenvectors. We only need a small texture containing a circular gradient of opacity, which is shared by all the splatted particles.

6.6 Interactive design of flows

For our tests we have implemented an interactive editor allowing users to specify and edit a flow. While simple enough, it illustrates how our representation allows a user to design, edit and control a flow.

Geometric objects are of two kinds: vortex filaments, and smoke particles (plus obstacles). Vortex filaments consist of curves that can be interactively inserted in the scene, and loaded or saved on disk. Smoke particles are treated as in particle systems editors: their initial position is spread interactively or procedurally within simple volumes or on surfaces. Both types of objects have various associated attributes controlling their behavior or their appearance.

The framework of our scene editor is similar to the one of a classical CG animation system: the user can select the current time with a slider, then add or edit the geometrical content, tune the attributes, keyframe geometrical or attribute data. When playing part of an animation in the editor, the keyframed data is treated in a standard way, while the non-keyframed data is simulated in real-time. Both kinds are thus naturally integrated. Combinations are easy to manage: the user can keyframe the extinction of a simulated filament, or let a filament interpolate between a simulation and a keyframed curve, or switch from one mode to the other for a period of time.

At any time, the flow can be rendered either in fast or high quality rendering. The quality/efficiency ratio can be controlled in two ways. Firstly, by selecting the visual effects: e.g., shadows, complex lighting. Secondly, by tuning the smoke particles' global attributes: sampling density, self-subdivision enabling, ellipsoidal distortion enabling, and associated thresholds. As shown in the results, reasonable renderings can be obtained in real-time.

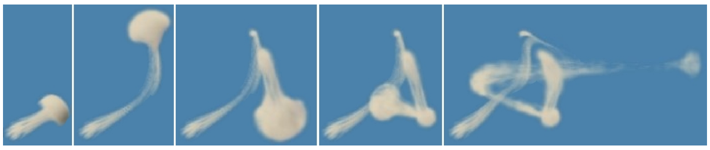


Figure 6.7: *A vortex ring following a curve.*

Thus, unlike usual fluid simulators – whose various limitations from the CG point of view are mentioned in [LF02] – the framework in our fluid editor is similar to that of a geometric modeler. In particular, the flow features are represented as *compact vector data*.

This has several consequences:

- It is easy to store the entire animated scene and to edit it interactively, going back in time to change a detail.
- Features are meaningful and easy to handle for the artist.
- The simulation is resolution-independent, and deterministic in practice, as opposed to grid-based fluid simulations where results change when the grid size changes.
- It is easy to play several minutes of animation before reaching the relevant time range to be rendered.
- It is easy to re-render a given simulation with new rendering attributes, or to add new frames later.

6.7 Results

Features: The effects of vortex-induced motion, noise, collision on the floor, distortion of smoke particles, and keyframed vortices, are illustrated on Figure 6.1, 6.3, 6.5, 6.6 and 6.7. Some snapshots are shown in Figure 6.8 and 6.10. The smoke sheet in a 3D flow (Figure 6.8(c)) deserves some comments. As is done in real-world wind tunnels, we have placed sources of smoke such that a thin sheet of markers interacts with the 3D flow.

Complexity and Performances: Let n_f be the number of filaments, k_f the total number of filament segments at the finest level, $\tilde{k}_f$ the average number of filament segments considered taking LOD into account, n_n the number of noise particles, n_s the number of smoke particles, $\tilde{n}_n$ the average number of noise particles acting on a smoke particle.

The *simulation cost* can be estimated from the number of evaluations of $\Phi_{segment}$ as $(k_f + n_n + n_s)\tilde{k}_f + n_s\tilde{n}_n$. Its most significant component is $n_s\tilde{k}_f$. This means that the simulation of filaments alone is almost free, which makes the interactive modeling of flow features easy. All the simulation time is spent on smoke particles. Accounting for particle distortion multiplies the cost by 4 due to the finite-difference estimation of the Jacobian. The grid LOD factorization yields a 15% saving. See below for possible improvements.

The *rendering cost* decomposes into the splatting of smoke particles (comprising the calculation of the ellipse shapes) and the shadow calculation (self and cast). The shadows represent the main part of the rendering cost. We measured that the rendering cost was roughly independent of the resolution (about 2% of overhead for 12 times more pixels).

Benchmarks: The following performance measurements were done on a `Pentium 4` processor at 2.8 GHz with a nVIDIA GeForce FX 5200 graphics board. Note that we measure simulation and rendering with smoke particles.

Simulation and visualization of filaments alone in the flow editor is real-time.

- *Heavy explosion* (Figure 6.5). The field is defined by 2 circle rings and 100 noise vortices. The smoke consists of about 5,000 non-deformable particles. The animation is rendered (without shadows) at 16 fps.

- *Smoke sheet* (Figure 6.8(b)). The field is defined by 10 circle rings. The smoke consists of about 30,000 deformable particles. The animation is rendered at 0.7 fps (or 3 fps using simple particles).

- *Train smoke* (Figure 6.8(a)). The field is defined by 20 circle rings and 20 filaments made of 64 segments each. The smoke consists of about 30,000 deformable particles. The animation is rendered (with shadows) at 12 seconds per frame, 29% of which is due to shadows, and 70% to advection ($\frac{1}{4}$) and distortion ($\frac{3}{4}$) of smoke particles.

- *Field of plumes* (Figure 6.8(c) and (d)). The field is defined by 6 filaments starting as circles then turning to deformable filaments made of 64 segments. 96 noise particles per filament were used for e, and no noise for d. The smoke consists of about 50,000 particles. The animation is rendered at 7 seconds per frame for d and 23 seconds per frame for e.

- *Comparison with [FSJ01]*, a reference for smoke simulation. It is not really possible to compare Eulerian and Lagrangian methods fairly, because there exists no standard by which we can compare the two complexities. Moreover, what is an easy case for one method corresponds to the difficult case for the other and vice versa (e.g. resolved detail vs crowded volume). Still, we tried to produce two flows where apparent complexity was roughly comparable to the examples of [FSJ01]. In the following we have upgraded their timings according to our CPU clock.

  - The field shown on Figure 6.9(left) is simulated at 0.9 fps. It resembles their Figure 3 which would play at 0.1 fps.

  - The field shown on Figure 6.9(right) is simulated at 24 fps. It resembles their Figure 8 which would play at 1.6 fps.

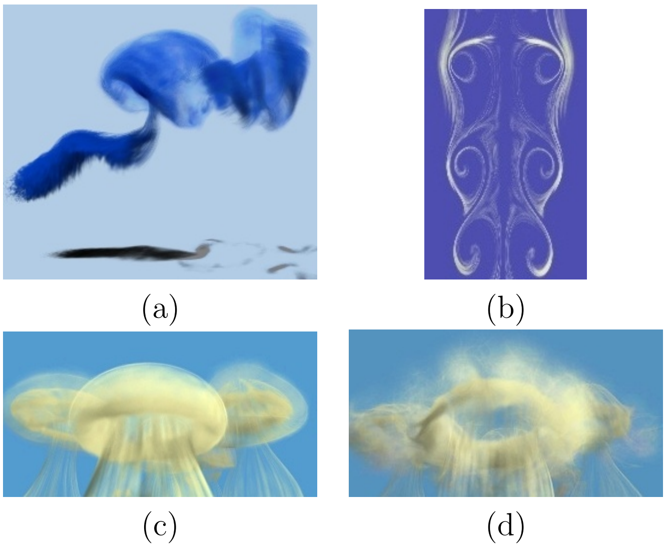

(a)
(b)
(c)
(d)

Figure 6.8: *Various examples of animated flows.*

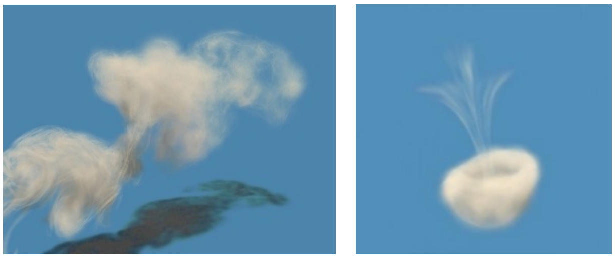


Figure 6.9: *Benchmarking with scenes close to [FSJ01] Figure 3 and Figure 8.*

Possible Enhancements to Improve Performances:

- As we have seen, deformation of smoke particles is very costly. The deformation of a particle could probably be estimated only once in a while. Moreover, the Jacobian could be calculated analytically rather than requiring 3 extra evaluations of Φ for each particle.

- Φ is evaluated billions of times. The floating grid only saves LOD estimations. It should be possible to save a lot more by interpolating in grid cells the components of Φ corresponding to distant vortices. In our formalism the results of the integration Φ is a whirl *operator* and not directly a new point or velocity, so a good quality interpolation can be expected.

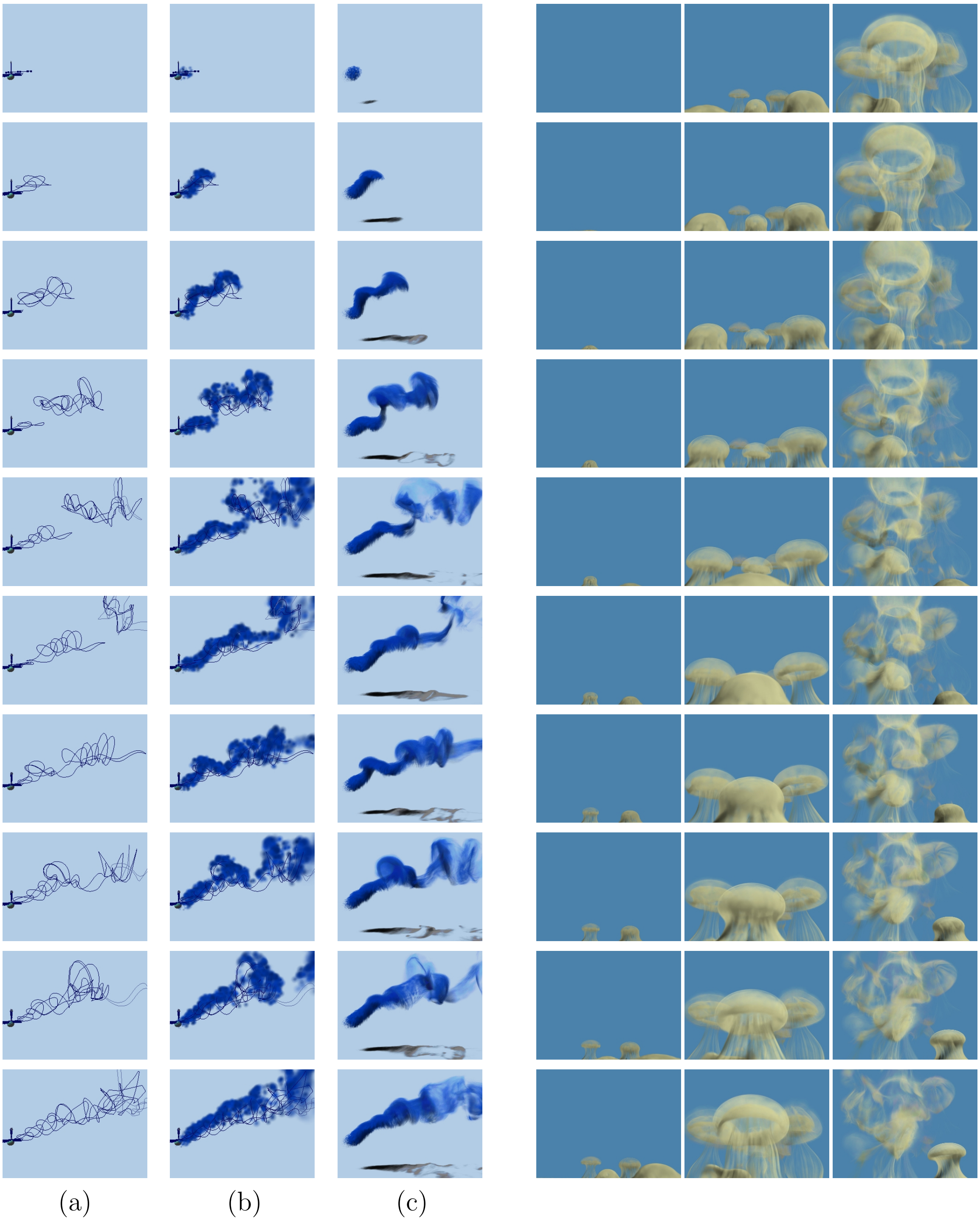
- Smoke particles keep splitting with stretching, thus numerous diluted particles tend to appear. In our application we remove particles under a chosen density threshold, but this can lead to visual artifacts (vanishing smoke) since the accumulation of numerous diluted particles may be visible. Neighboring diluted particles should be resampled and combined into bigger particles.

6.8 Conclusion

Our method allows an artist quickly and easily to design and simulate flows such as turbulent smoke by relying on a compact high-level primitive, the *vortex filament*, which induces a velocity field. Filaments are geometric objects easy to edit and animate in a modeling system. We also presented a rendering scheme based on deformable particles to represent and render the smoke advected in this field. Our Lagrangian vorticity scheme does not suffer from numerical dissipation and is not spatially bounded by a static grid. The simulation is independent of the rendering, and the smoke resolution can be chosen and changed afterwards without affecting the simulation. Our animated examples show that very detailed results can be generated efficiently.

The issues faced by Vortex Methods in fluid engineering are also of interest for CG, even if in our domain we can circumvent most of the constraints. These issues concern complex environments and long simulations. For the former, complex boundary conditions rather should be considered, and LOD should be extended to account for clusters of filaments. For the latter, the complex interaction on filaments should be modeled, especially their reconnections and collapses.

The presented fluid model shows how a space deformation is related to a visually acceptable fluid. The next step in shape modeling by space deformation research would be to define a set of useful operations that mimic fluid behaviour.



(a) (b) (c)

Figure 6.10: *Snapshots of sequences of smoke animation. Left: train smoke, (a) filaments of vorticity, (b) rendering with simple particles, (c) rendering with deformable particles. Right: field of plumes.*

Chapter 7

Hexanions

Some of the popular representations for rigid object motion include 4×4 real matrix or a combination of vector and quaternion [Ale02a, DKL98]. But a real matrix is capable of representing a too large set of transformations, including scaling and shear. Another drawback with matrices is the accumulation of numerical errors which make it deviate from a rigid transformation, and requires us to renormalize the matrix once in a while. On the other hand, a drawback of using a combination of vector and quaternion is the separation of the motion into two components that have to be separately. This component separation can make implementation on computing devices unclear or unnecessarily long. While there are surely other alternatives for representing numerically rigid motion, we propose a formalism called *hexanion.* Hexanion is an algebraic structure for a compact and natural representation of the rigid transformations: translations, rotations and twists. The number of dimensions of hexanion space is six, which is also the number of degrees of freedom for rigid transformations. We have recently discovered that this 6D representation of twists was already known [BM98] and is in fact a Lie algebra [Gal01], although we have not found evidence of the existence of the closed-form twist exponential or multiplication operator with points and vectors [Gal01].

7.1 Motivation

The motivation behind the formalism of hexanions is the necessity to compute fractions of a rigid transformation and combine multiple rigid transformations. For a transformation represented by a 4×4 real matrix, taking a fraction h can be done with the following:

$$M^h = \exp(h \log M) \tag{7.1}$$

$$\text{where} \quad \begin{array}{rcl} \exp M & = & \sum_{k=0}^{\infty} \frac{M^k}{k!} \\ \log M & = & -\sum_{k=1}^{\infty} \frac{(\mathrm{I}-M)^k}{k} \end{array}$$

M. Alexa [Ale02a] proposes to use numerical methods to achieve this in the general case. In the case of a rigid transformation, a matrix is however a poor representation. A better choice is to use a hexanion, which can be interpreted as a shorthand for $\log M$ when the matrix M is a twist (this is inaccurate, since log is undefined). A hexanion is a 6D element, written with a pair of vectors:

$$\langle \vec{\omega}, \vec{\mathrm{m}} \rangle \tag{7.2}$$

The relation between a hexanion and the logarithm of a twist matrix is:

$$\log M = \begin{pmatrix} 0 & -\omega_z & \omega_y & m_x \\ \omega_z & 0 & -\omega_x & m_y \\ -\omega_y & \omega_x & 0 & m_z \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{7.3}$$

We called the above the hexanion's matrix form, and we denote it with round brackets, $(\vec{\omega}, \vec{\mathrm{m}})$. Note that in a hexanion, the part $\vec{\omega}$ is in fact the logarithm of a quaternion.

7.2 Overview

The hexanion space is a *real vector space*, whose elements represent twists, also known as screws. A twist is a translation of magnitude t in a direction along a line defined by a point c and a unit vector $\vec{\mathrm{n}}$ together with a rotation of angle θ about axis $\vec{\mathrm{n}}$. Mozzi and Cauchy have proved that any motion of a rigid body in space at every instant is a twist motion [Weib]. The following expression is the hexanion of the twist transformation defined as above, illustrated by Figure 7.1:

$$\langle \theta\vec{\mathrm{n}}, \theta c \times \vec{\mathrm{n}} + t\vec{\mathrm{n}} \rangle \tag{7.4}$$

Pure translation or rotation are obtained by setting either θ or t to zero. Combining or interpolating twists with hexanions is as simple as vector algebra. For instance if $\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$ denotes the new position, a motion from the origin to $\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$ is described as follows:

$$\langle u\vec{\omega}, u\vec{\mathrm{m}} \rangle, \quad u \in [0, 1] \tag{7.5}$$

With the definition of operator $*$ described in the following section, the trajectory parameterized in u of the point of a rigid object is simply described as follows:

$$\langle u\vec{\omega}, u\vec{\mathrm{m}} \rangle * \mathrm{p}, \quad u \in [0, 1] \tag{7.6}$$

7.3 Algebraic properties

In this section, algebraic properties of the hexanion are reviewed.

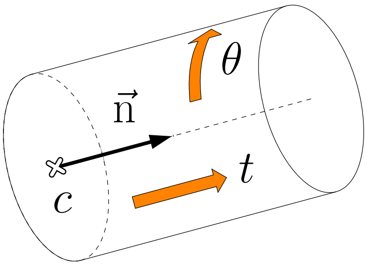


Figure 7.1: *A twist is a simultaneous translation and rotation about the same axis. Although the point* c *is not unique, the hexanion of a twist is, since it features* $\mathrm{c} \times \vec{\mathrm{n}}$.

Algebraic structure: Define the hexanion space, as the *real vector space* $(\mathbb{R}^6, +, \cdot)$. A pair of 3D vectors is a convenient notation for manipulating the elements of hexanion space, denoted with angle brackets:

$$\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$$

Consider two typical hexanions $A = \langle \vec{\omega}_A, \vec{\mathrm{m}}_A \rangle$ and $B = \langle \vec{\omega}_B, \vec{\mathrm{m}}_B \rangle$. Then $\lambda \cdot A$ is the hexanion whose components are those of A multiplied by λ, i.e. $\langle \lambda\vec{\omega}_A, \lambda\vec{\mathrm{m}}_A \rangle$ and $A+B$ is the hexanion whose components are the sum of respective components in A and B, i.e. $\langle \vec{\omega}_A + \vec{\omega}_B, \vec{\mathrm{m}}_A + \vec{\mathrm{m}}_B \rangle$.

Exponential: In its matrix form, a hexanion can be raised to integer powers. The integer powers of hexanion matrices are used for defining the exponential of a hexanion matrix, as follows:

$$\exp(\vec{\omega}, \vec{\mathrm{m}}) = \sum_{k=0}^{\infty} \frac{(\vec{\omega}, \vec{\mathrm{m}})^k}{k!} \tag{7.7}$$

The exponential of a hexanion is the 4×4 projective matrix of a rigid transformation. Conveniently, the exponential of a hexanion has a closed-form, proved in Section 7.5:

$$\exp(\vec{\omega}, \vec{\mathrm{m}}) = \begin{cases} \mathrm{I} + (\vec{\omega}, \vec{\mathrm{m}}) & \text{if } \|\vec{\omega}\| = 0 \\ \mathrm{I} + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}(\vec{\omega}, \vec{\mathrm{m}})^2 + \frac{\sin\|\vec{\omega}\|}{\|\vec{\omega}\|}(\vec{\omega}, \vec{\mathrm{m}}) & \text{if } \vec{\omega} \cdot \vec{\mathrm{m}} = 0 \\ \mathrm{I} + (\vec{\omega}, \vec{\mathrm{m}}) + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}(\vec{\omega}, \vec{\mathrm{m}})^2 + \frac{\|\vec{\omega}\|-\sin\|\vec{\omega}\|}{\|\vec{\omega}\|^3}(\vec{\omega}, \vec{\mathrm{m}})^3 & \text{otherwise} \end{cases} \tag{7.8}$$

Note that the above is not a piecewise definition of the exponential, but it is a convenient formulation of the cases where the exponential simplifies, since we aim at numerical applications. In some applications it can be useful to use a first order approximation of the exponential, which conveniently corresponds to the translation part of the hexanion :

$$\exp(\vec{\omega}, \vec{\mathrm{m}}) \approx \mathrm{I} + (\vec{\omega}, \vec{\mathrm{m}}) \tag{7.9}$$

Hexanion-point multiplication: We define the multiplication operator, $*$, of a point, $\mathrm{p} = (p_x, p_y, p_z)^\top$, by a hexanion $\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$, as follows:

$$\langle \vec{\omega}, \vec{\mathrm{m}} \rangle * \mathrm{p} = \exp(\vec{\omega}, \vec{\mathrm{m}}) \cdot \mathrm{p} \tag{7.10}$$

By using the approximation of the exponential, the above can be approximated if necessary:

$$\langle \vec{\omega}, \vec{\mathrm{m}} \rangle * \mathrm{p} \approx \mathrm{p} + \vec{\mathrm{m}} + \vec{\omega} \times \mathrm{p} \tag{7.11}$$

Hexanion-vector multiplication: We define the multiplication operator, $*$, of a vector, $\vec{\mathrm{v}} = (v_x, v_y, v_z)^\top$, by a hexanion $\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$, as follows:

$$\langle \vec{\omega}, \vec{\mathrm{m}} \rangle * \vec{\mathrm{v}} \;\; = \;\; \exp(\vec{\omega}, \vec{\mathrm{m}}) \cdot \vec{\mathrm{v}} \tag{7.12}$$

By using the approximation of the exponential, the above can be approximated. If the reader decides to use the approximation, he must realize that it does not preserve the length of $\vec{\mathrm{v}}$, as opposed to the closed-form of Equation 7.12:

$$\langle \vec{\omega}, \vec{\mathrm{m}} \rangle * \vec{\mathrm{v}} \approx \vec{\mathrm{v}} + \vec{\omega} \times \vec{\mathrm{v}} \tag{7.13}$$

Neutral: The neutral hexanion element is $\langle 0, 0 \rangle$. It also satisfies:

$$\langle 0, 0 \rangle * \mathrm{p} = \mathrm{p} \tag{7.14}$$

$$\langle 0, 0 \rangle * \vec{\mathrm{n}} = \vec{\mathrm{n}} \tag{7.15}$$

Hexanion inverse: The inverse of $\langle \vec{\omega}, \vec{\mathrm{m}} \rangle$ is $\langle -\vec{\omega}, -\vec{\mathrm{m}} \rangle$. The inverse describes the reverse motion. The inverse satisfies:

$$\langle -\vec{\omega}, -\vec{\mathrm{m}} \rangle * (\langle \vec{\omega}, \vec{\mathrm{m}} \rangle * \mathrm{p}) = \mathrm{p} \tag{7.16}$$

7.4 Hexanion subgroups

This section presents the hexanion-to-matrix conversion, the hexanion-point multiplication and the hexanion-vector multiplication. The hexanion-to-matrix conversion is also referred to as the exponential. Hexanions can be classified in three sets: translation, rotation and twists. Knowing the type of a hexanion can simplify computations. In the general case, fast expressions can be derived for particular uses of hexanions.

7.4.1 Translation-hexanion:

The set of translations is a subset of the hexanion space. The hexanion of a translation of vector $\vec{\mathrm{m}}$ is:

$$\langle 0, \vec{\mathrm{m}} \rangle \tag{7.17}$$

Conversely, a hexanion $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$ is a translation if $\|\vec{\omega}\| = 0$. The exponential of a translation-hexanion is much simpler than in the general case, and is defined as follows:

$$\exp(0,\vec{\mathrm{m}}) = \begin{pmatrix} 1 & 0 & 0 & m_x \\ 0 & 1 & 0 & m_y \\ 0 & 0 & 1 & m_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.18}$$

The multiplication of a point, p, and a normal $\vec{\mathrm{v}}$ by a translation-hexanion, $\langle 0,\vec{\mathrm{m}}\rangle$, produce simple expressions:

$$\begin{array}{rcl} \langle 0,\vec{\mathrm{m}}\rangle * \mathrm{p} & = & \mathrm{p}+\vec{\mathrm{m}} \\ \langle 0,\vec{\mathrm{m}}\rangle * \vec{\mathrm{v}} & = & \vec{\mathrm{v}} \end{array} \tag{7.19}$$

7.4.2 Rotation-hexanion:

The set of rotations is a subset of the hexanion space. Note that we refer to the set of rotations about a point, more general than unit quaternions which are rotations about the origin. The hexanion of a rotation defined by axis $\vec{\omega}$, center c, and angle $\|\vec{\omega}\|$ is:

$$\langle\vec{\omega}, \mathrm{c}\times\vec{\omega}\rangle. \tag{7.20}$$

Conversely an element $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$ is a rotation if $\vec{\omega}\cdot\vec{\mathrm{m}} = 0$. The center, angle and unit axis of rotation are given by the following equations:

$$\begin{array}{rcl} \mathrm{c} & = & \frac{\vec{\omega}\times\vec{\mathrm{m}}}{\|\vec{\omega}\|^2} \\ \theta & = & \|\vec{\omega}\| \\ \vec{\mathrm{n}} & = & \frac{\vec{\omega}}{\|\vec{\omega}\|} \end{array} \tag{7.21}$$

The exponential of a rotation-hexanion is simpler than in the general case, and is defined as follows:

$$\exp(\vec{\omega},\vec{\mathrm{m}}) = \mathrm{I} + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}(\vec{\omega},\vec{\mathrm{m}})^2 + \frac{\sin\|\vec{\omega}\|}{\|\vec{\omega}\|}(\vec{\omega},\vec{\mathrm{m}}) \tag{7.22}$$

The multiplication of a point, p, by a rotation $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$, has a closed-form expression:

$$\begin{array}{rcl} \langle\vec{\omega},\vec{\mathrm{m}}\rangle * \mathrm{p} & = & \mathrm{p} + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}\vec{\omega}\times\vec{\xi} + \frac{\sin\|\vec{\omega}\|}{\|\vec{\omega}\|}\vec{\xi} \\ \text{where } \vec{\xi} & = & \vec{\omega}\times\mathrm{p}+\vec{\mathrm{m}} \end{array} \tag{7.23}$$

The multiplication of a vector, $\vec{\mathrm{v}}$, by a rotation $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$, has a closed-form expression:

$$\langle\vec{\omega},\vec{\mathrm{m}}\rangle * \vec{\mathrm{v}} \;=\; \vec{\mathrm{v}} + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}\vec{\omega}\times(\vec{\omega}\times\vec{\mathrm{v}}) + \frac{\sin\|\vec{\omega}\|}{\|\vec{\omega}\|}\vec{\omega}\times\vec{\mathrm{v}} \tag{7.24}$$

7.4.3 Twist-hexanion:

Translations and rotations are particular cases of twists. Hexanion space is in fact the set of twists. The hexanion of a twist is defined by a rotation angle θ around the axis defined by direction $\vec{\mathrm{n}}$ and center c, and a translation of magnitude t along $\vec{\mathrm{n}}$:

$$\langle\theta\vec{\mathrm{n}}, \theta c\times\vec{\mathrm{n}} + t\vec{\mathrm{n}}\rangle \tag{7.25}$$

Conversely, if we denote twists with the general 6D vector $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$, the rotation part and the translation part can be identified. We denote these parts Rot and Trs :

$$\begin{array}{rcl} \mathrm{Rot}\langle\vec{\omega},\vec{\mathrm{m}}\rangle & = & \langle\vec{\omega},\frac{1}{\|\vec{\omega}\|^2}\vec{\omega}\times\vec{\mathrm{m}}\times\vec{\omega}\rangle \\ \mathrm{Trs}\langle\vec{\omega},\vec{\mathrm{m}}\rangle & = & \langle 0,\frac{\vec{\mathrm{m}}\cdot\vec{\omega}}{\|\vec{\omega}\|^2}\vec{\omega}\rangle \end{array} \tag{7.26}$$

$$\langle\vec{\omega},\vec{\mathrm{m}}\rangle = \mathrm{Rot}\langle\vec{\omega},\vec{\mathrm{m}}\rangle + \mathrm{Trs}\langle\vec{\omega},\vec{\mathrm{m}}\rangle \tag{7.27}$$

The exponential of a rotation-hexanion is given by the general case, as follows:

$$\exp(\vec{\omega},\vec{\mathrm{m}}) = \mathrm{I} + (\vec{\omega},\vec{\mathrm{m}}) + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}(\vec{\omega},\vec{\mathrm{m}})^2 + \frac{\|\vec{\omega}\|-\sin\|\vec{\omega}\|}{\|\vec{\omega}\|^3}(\vec{\omega},\vec{\mathrm{m}})^3 \tag{7.28}$$

The multiplication of a point p by a twist-hexanion $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$ has a closed-form expression:

$$\begin{array}{rcl} \langle\vec{\omega},\vec{\mathrm{m}}\rangle * p & = & p + \vec{\mathrm{m}} + \vec{\omega}\times(\vec{\omega}\times(ap + \frac{1-b}{\|\vec{\omega}\|^2}\vec{\mathrm{m}}) + a\vec{\mathrm{m}} + bp) \\ \text{where } a & = & \frac{1-\cos(\|\vec{\omega}\|)}{\|\vec{\omega}\|^2} \\ b & = & \frac{\sin(\|\vec{\omega}\|)}{\|\vec{\omega}\|} \end{array} \tag{7.29}$$

The multiplication of a vector, $\vec{\mathrm{n}}$, by a twist-hexanion $\langle\vec{\omega},\vec{\mathrm{m}}\rangle$, has a closed-form expression. Notice that this expression is exactly the same as in the case of the rotation-hexanion, since a vector is a direction and is unaffected by translations:

$$\langle\vec{\omega},\vec{\mathrm{m}}\rangle * \vec{\mathrm{v}} = \vec{\mathrm{v}} + \frac{1-\cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}\vec{\omega}\times(\vec{\omega}\times\vec{\mathrm{v}}) + \frac{\sin\|\vec{\omega}\|}{\|\vec{\omega}\|}\vec{\omega}\times\vec{\mathrm{v}} \tag{7.30}$$

7.5 Closed-form exponential

We show here that the exponential of a hexanion's matrix form has a closed-form. Let us denote the matrix form of a hexanion by $M = (\vec{\omega},\vec{\mathrm{m}})$. The following is its exponential:

$$\begin{array}{rcl} \exp M & = & \sum_{k=0}^{\infty}\frac{M^k}{k!} \\ & = & \mathrm{I} + M + \sum_{k=1}^{\infty}\frac{M^{2k}}{(2k)!} + \sum_{k=1}^{\infty}\frac{M^{2k+1}}{(2k+1)!} \end{array} \tag{7.31}$$

If M is a translation, then the solution is $\exp M = \mathrm{I} + M$. Now let us assume in the following that M is not a translation. Using the relation between a hexanion and its rotation component, the following equalities can be obtained:

$$\begin{array}{rcl} M^{2k+1} & = & (-\|\vec{\omega}\|^2)^k\ \mathrm{Rot}M \\ M^{2k} & = & (-\|\vec{\omega}\|^2)^{k-1}M^2 \end{array} \tag{7.32}$$

In the exponential, the above can then be substituted for their values:

$$\exp M = \mathrm{I} + M + \sum_{k=1}^{\infty}\frac{(-\|\vec{\omega}\|^2)^{k-1}}{(2k)!}M^2 + \sum_{k=1}^{\infty}\frac{(-\|\vec{\omega}\|^2)^k}{(2k+1)!}\mathrm{Rot}M \tag{7.33}$$

Since we assume that M is not a translation, $\vec{\omega}$ is different from zero, and terms can be factored out from the sums:

$$\exp M = \mathrm{I} + M - \frac{1}{\|\vec{\omega}\|^2}\sum_{k=1}^{\infty}\frac{(-\|\vec{\omega}\|^2)^k}{(2k)!}M^2 + \frac{1}{\|\vec{\omega}\|}\sum_{k=1}^{\infty}\frac{\|\vec{\omega}\|(-\|\vec{\omega}\|^2)^k}{(2k+1)!}\mathrm{Rot}M \tag{7.34}$$

In the first and second sums, parts of the cosine and sine of $\|\vec{\omega}\|$ can be identified:

$$\exp M = \mathrm{I} + M - \frac{1}{\|\vec{\omega}\|^2}(\cos\|\vec{\omega}\| - 1)M^2 + \frac{1}{\|\vec{\omega}\|}(\sin\|\vec{\omega}\| - \|\vec{\omega}\|)\mathrm{Rot}M \tag{7.35}$$

Using the relation between $\mathrm{Rot}M$ and M to replace $\mathrm{Rot}M$, we obtain the final formula:

$$\exp M = \mathrm{I} + M + \frac{1 - \cos\|\vec{\omega}\|}{\|\vec{\omega}\|^2}M^2 + \frac{\|\vec{\omega}\| - \sin\|\vec{\omega}\|}{\|\vec{\omega}\|^3}M^3 \tag{7.36}$$

In the case where M is a rotation, the formula further simplifies using the equality $M^3 = -\|\vec{\omega}\|^2 M$.

7.6 Hexanion product

Let us define two hexanions $h_0 = \langle\vec{\omega}_0, \vec{\mathrm{m}}_0\rangle$ and $h_1 = \langle\vec{\omega}_1, \vec{\mathrm{m}}_1\rangle$. They can be applied one after the other to a point p as follows:

$$h_1 * (h_0 * \mathrm{p}) \tag{7.37}$$

In this section, we will define cases of the hexanion product operator $h_1 * h_0$, whose result is a hexanion, i.e. useful for collapsing a series of hexanions into a single one. Each hexanion h_i carries the point it multiplies along a trajectory. Since they are arbitrary hexanions, there is a discontinuity along the trajectory followed by p, when $h_0 * \mathrm{p}$ is suddenly multiplied by h_1. By collapsing hexanions, some information about the trajectory of p is inevitably lost. In the following section we propose convenient formulas for the product of specific hexanions. We have not yet developed the complete set of products. This may acknowledge some of the problems with hexanions.

7.6.1 Translation-translation

Let us define two translation hexanions $h_0 = \langle 0, \vec{\mathrm{m}}_0\rangle$ and $h_1 = \langle 0, \vec{\mathrm{m}}_1\rangle$. Their product $h_S = h_0 * h_1 = \langle 0, \vec{\mathrm{m}}_0 + \vec{\mathrm{m}}_1\rangle$ is commutative and is a translation.

7.6.2 Translation-rotation

Let us define two hexanions: a translation $h_T = \langle 0, \vec{\mathrm{m}}_T\rangle$ and a rotation $h_R = \langle\vec{\omega}_R, c_R \times \vec{\omega}_R\rangle$. Their product $h_S = h_T * h_R$ is a twist (called also screw). The hexanion $h_S = \langle\vec{\omega}_S, \vec{\mathrm{m}}_S\rangle$ is given by

$$\begin{array}{rcl} \vec{\omega}_S & = & \vec{\omega}_R \\ \vec{\mathrm{m}}_S & = & \vec{\mathrm{m}}_R + \vec{\mathrm{m}}_T + \frac{\vec{\mathrm{m}}_T \times \vec{\omega}_R}{2} + (\frac{\|\vec{\omega}_R\|}{2\tan(\|\vec{\omega}_R\|/2)} - 1)\frac{\vec{\omega}_R \times \vec{\mathrm{m}}_T \times \vec{\omega}_R}{\|\vec{\omega}_R\|^2} \end{array} \tag{7.38}$$

7.6.3 Rotation-translation

Their product $h_S = h_R * h_T$ is a twist (called also screw). The hexanion $h_S = \langle\vec{\omega}_S, \vec{\mathrm{m}}_S\rangle$ is given by

$$\begin{array}{rcl} \vec{\omega}_S & = & \vec{\omega}_R \\ \vec{\mathrm{m}}_S & = & \vec{\mathrm{m}}_R + \vec{\mathrm{m}}_T - \frac{\vec{\mathrm{m}}_T \times \vec{\omega}_R}{2} + (\frac{\|\vec{\omega}_R\|}{2\tan(\|\vec{\omega}_R\|/2)} - 1)\frac{\vec{\omega}_R \times \vec{\mathrm{m}}_T \times \vec{\omega}_R}{\|\vec{\omega}_R\|^2} \end{array} \tag{7.39}$$

7.7 Properties

The cube of a hexanion matrix is a rotation-hexanion matrix:

$$(\vec{\omega}, \vec{\mathrm{m}})^3 = -\|\vec{\omega}\|^2 \mathrm{Rot}(\vec{\omega}, \vec{\mathrm{m}}) \tag{7.40}$$

The cube of a hexanion is related to the rotation component of a hexanion through the following:

$$\mathrm{Rot}(\vec{\omega}, \vec{\mathrm{m}}) = \frac{-1}{\|\vec{\omega}\|^2} (\vec{\omega}, \vec{\mathrm{m}})^3 \tag{7.41}$$

The square of a hexanion matrix is not a hexanion matrix, but satisfies the following:

$$(\mathrm{Rot}(\vec{\omega}, \vec{\mathrm{m}}))^2 = (\vec{\omega}, \vec{\mathrm{m}})^2 \tag{7.42}$$

In general, the operator $*$ is non-commutative

$$\langle \vec{\omega}_0, \vec{\mathrm{m}}_0 \rangle * \langle \vec{\omega}_1, \vec{\mathrm{m}}_1 \rangle * \mathrm{p} \neq \langle \vec{\omega}_1, \vec{\mathrm{m}}_1 \rangle * \langle \vec{\omega}_0, \vec{\mathrm{m}}_0 \rangle * \mathrm{p} \tag{7.43}$$

The following is a relation between the exponential of a hexanion and the matrix form of that hexanion :

$$\left. \frac{\partial}{\partial h} M^h \right|_{h=0} = (\vec{\omega}, \vec{\mathrm{m}}) \tag{7.44}$$

The multiplications of a point or a vector by the matrix form have simple expressions. The following expressions involve a matrix-point multiplication, and are useful for developing the closed-form formulas for transforming a point or a normal. They should not be mis-interpreted by the reader as the transform of a point or a normal:

$$\begin{array}{rcl} (\vec{\omega}, \vec{\mathrm{m}}) \cdot \mathrm{p} & = & \vec{\omega} \times \mathrm{p} + \vec{\mathrm{m}} \\ (\vec{\omega}, \vec{\mathrm{m}}) \cdot \vec{\mathrm{v}} & = & \vec{\omega} \times \vec{\mathrm{v}} \end{array} \tag{7.45}$$

In fact, $(\vec{\omega}, \vec{\mathrm{m}}) \cdot \mathrm{p}$ is the velocity of the twist at p. Note that the multiplication of a point is a normal, thus $(\vec{\omega}, \vec{\mathrm{m}})^2 \cdot \mathrm{p} = \vec{\omega} \times (\vec{\omega} \times \mathrm{p} + \vec{\mathrm{m}})$.

7.8 Matrix to hexanion conversion:

This conversion is not meant to be optimal, and is expected to be done only very few times, for instance when importing a rigid transformation. As a guideline, we recommend representing rigid transformations with hexanions and avoid this conversion. Let us define three unit vectors $\vec{\mathrm{x}}$, $\vec{\mathrm{y}}$ and $\vec{\mathrm{z}}$, forming a right-handed orthogonal system. Let us define a position o. These define a 4×4 matrix of a rigid transformation from the origin to that position.

$$M = \begin{pmatrix} x_x & y_x & y_x & o_x \\ x_y & y_y & y_y & o_y \\ x_z & y_z & y_z & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.46}$$

We know that this matrix can be written with only six components, in the form of a hexanion:

$$\langle \theta\vec{\mathrm{n}}, \theta\vec{\mathrm{c}} \times \vec{\mathrm{n}} + t\vec{\mathrm{n}} \rangle \tag{7.47}$$

Vector $\vec{\mathrm{n}}$ is the unit eigenvector of matrix M associated with eigenvalue 1:

$$M \cdot \vec{\mathrm{n}} = \vec{\mathrm{n}} \tag{7.48}$$

The solution for $\vec{\mathrm{n}}$ is:

$$\vec{\mathrm{n}} = \frac{\vec{\mathrm{n}}_0}{\|\vec{\mathrm{n}}_0\|} \tag{7.49}$$

$$\text{where } \vec{\mathrm{n}}_0 = \begin{cases} \vec{\mathrm{x}} + (1 - y_y - z_z, y_x, z_x)^\top & \text{if } \ 1 + x_x - y_y - z_z \neq 0 \\ \vec{\mathrm{y}} + (x_y, 1 - x_x - z_z, z_y)^\top & \text{if } \ 1 - x_x + y_y - z_z \neq 0 \\ \vec{\mathrm{z}} + (x_z, y_z, 1 - x_x - y_y)^\top & \text{if } \ 1 - x_x - y_y + z_z \neq 0 \end{cases} \tag{7.50}$$

Once unit vector $\vec{\mathrm{n}}$ is found, point c and translation are found by solving the following:

$$M \cdot \mathrm{c} = \mathrm{c} + t\vec{\mathrm{n}} \tag{7.51}$$

The solution for the magnitude of the translation is:

$$t = o \cdot \vec{\mathrm{n}} \tag{7.52}$$

The solution for the center, c, is the following:

$$c = \begin{cases} \frac{(0, s_z z_y + s_y(1 - z_z), s_y y_z + s_z(1 - y_y))}{1 + x_x - y_y - z_z} & \text{if } \ 1 + x_x - y_y - z_z \neq 0 \\ \frac{(s_z z_x + s_x(1 - z_z), 0, s_x x_z + s_z(1 - x_x))}{1 - x_x + y_y - z_z} & \text{if } \ 1 - x_x + y_y - z_z \neq 0 \\ \frac{(s_y y_x + s_x(1 - y_y), s_x x_y + s_y(1 - x_x), 0)}{1 - x_x - y_y + z_z} & \text{if } \ 1 - x_x - y_y + z_z \neq 0 \end{cases} \tag{7.53}$$

$$\text{where } s = o - t\vec{\mathrm{n}} \tag{7.54}$$

For any vector $\vec{\mathrm{v}}$ non-aligned with vector $\vec{\mathrm{n}}$, the angle of rotation, θ, is given by the following:

$$\begin{aligned} \cos\theta &= \vec{\mathrm{u}} \cdot M\vec{\mathrm{u}} \\ \sin\theta &= (\vec{\mathrm{n}} \times \vec{\mathrm{u}}) \cdot M\vec{\mathrm{u}} \end{aligned} \tag{7.55}$$

$$\text{where } \vec{\mathrm{u}} = \frac{\vec{\mathrm{v}} - (\vec{\mathrm{v}} \cdot \vec{\mathrm{n}})\vec{\mathrm{n}}}{\|\vec{\mathrm{v}} - (\vec{\mathrm{v}} \cdot \vec{\mathrm{n}})\vec{\mathrm{n}}\|} \tag{7.56}$$

There are many solutions for the angle θ, given by $\theta + 2k\pi, k \in \mathbb{Z}$. With the matrix form, the original value of θ is lost. The hexanion that describes a straight path between between position matrix M_0 and position matrix M_1 is computed as above, with M defined as follows:

$$M = M_1 M_0^{-1} \tag{7.57}$$

7.9 Conclusion

Hexanions are useful for computing positions of a rigid object along its trajectory:

$$u\langle \vec{\omega}, \vec{\mathrm{m}} \rangle, \quad u \in [0,1] \tag{7.58}$$

Hexanions can be used to interpolate two positions, where $\langle \vec{\omega}_0, \vec{\mathrm{m}}_0 \rangle$ and $\langle \vec{\omega}_1, \vec{\mathrm{m}}_1 \rangle$ may be obtained using the matrix-to-hexanion conversion:

$$(1-u)\langle \vec{\omega}_0, \vec{\mathrm{m}}_0 \rangle + u\langle \vec{\omega}_1, \vec{\mathrm{m}}_1 \rangle, \quad u \in [0,1] \tag{7.59}$$

Hexanions are very useful for weighted sums of rigid transformations:

$$\frac{\sum_i \phi_i \langle \vec{\omega}_i, \vec{\mathrm{m}}_i \rangle}{\sum_i \phi_i} \tag{7.60}$$

On a computing device, the most expensive part of hexanions is the computation of the exponential. Depending on the scenario, this computation can be done only once to obtain a 4 × 4 matrix, or does not need to be computed explicitly: to deform a point or a vector, a closed-form formula can be used.

Limitations: At the beginning, we argued that having the translation and rotation components merged in a single formalism was an advantage in terms of simplifying the handling of a twist. From the point of view of a user who wants precise control, this can be seen as an inconvenience: it prevents the choice of individual schemes to handle the interpolation of the translation and rotation part of the motion. It is however possible to express this separation in the hexanion formalism. The following describes a motion straight from a point, c, to another point, p + $\vec{\mathrm{m}}$, accompanied by a rotation about a center of rotation c:

$$\langle \vec{\omega}, (\langle 0, \vec{\mathrm{m}} \rangle * \mathrm{c}) \times \vec{\omega} \rangle * \langle 0, \vec{\mathrm{m}} \rangle \tag{7.61}$$

Given the transformation, the description of the motion is straightforward:

$$\langle u\vec{\omega}, (\langle 0, u\vec{\mathrm{m}} \rangle * \mathrm{c}) \times (u\vec{\omega}) \rangle * \langle 0, u\vec{\mathrm{m}} \rangle, \quad u \in [0,1] \tag{7.62}$$

From this and the above formulas we can develop a closed-form trajectory $C(\mathrm{p})$, of a point, p:

$$\begin{array}{rcl} C(\mathrm{p}, u) & = & \mathrm{p} + u\vec{\mathrm{m}} + \frac{1-\cos\|u\vec{\omega}\|}{\|\vec{\omega}\|^2}\vec{\omega} \times \vec{\xi} + \frac{\sin\|u\vec{\omega}\|}{\|\vec{\omega}\|}\vec{\xi} \\ \text{where } \vec{\xi} & = & \vec{\omega} \times (\mathrm{p} - \mathrm{c}) \end{array} \tag{7.63}$$

Conclusion

Sweepers, our main contribution introduced in Chapter 3, is a framework for defining swept deformation operations for shape modeling. It permits the description of a family of shape interfaces based on gesture, between the artist and the mathematics describing the shape. Hence sweepers enables an artist to handle shapes in a more efficient way rather than directly modifying a shape's mathematical description. Because sweepers are foldover-free, they easily maintain a shape coherency. We have found that this is not in contradiction to changing the topology of a shape, which can be done in a foldover-free yet discontinuous manner.

We also propose swirling-sweepers and swept-fluid in Chapter 4, two types of swept-deformation for describing shape modeling interfaces that implicitly preserve the shape's volume. Subjectively, swirling-sweepers is the most effective modeling technique defined in the sweepers framework.

The separation of the shape's interface and the shape's description leads us to explore four alternative ways to describe a shape's surface or volume for rendering: an updated mesh, a point-sampled surface, a discrete implicit surface and inverse ray-tracing. While our proposed methods are sufficient in a wide range of situations, more research should be done in this area.

We have realized that sweepers can be used for the more general purpose of describing high order trajectories in continuums, and this has stimulated our interest in other areas of Computer Graphics: we have derived from swirling-sweepers a technique for animating visual fluids in Chapter 6, and have used sweepers in the context of keyframed animation in Section 3.6.

With modeling by space deformation, the separation of the shape's description from the set of operations performed on the shape identifies clearly two branches for future research.

Firstly, the fluid model presented in Chapter 6 shows how a space deformation may be related to a visually acceptable fluid. With the conclusion of Chapter 4 suggesting that an operation on the shape mimicking an incompressible fluid provides an efficient tool to the artist, future space deformation may investigate further fluid dynamics and, more importantly, techniques for specifying small scale features on the surface of the shape.

Secondly, the shape descriptions proposed in Chapter 5 possess both advantages and inconveniences. To our knowledge, there is no satisfying shape description that can be used as a "universal material": simple to implement as well as capable of enduring any operation desired by the artist, for example controlled and uncontrolled topology changes, sharp features, small and large scale editing and texturing. We suggest that future research should focus on defining the requirement of a shape description for shape modeling, and solve the related issues to produce a versatile shape description suitable for shape modeling.

APPENDIX A

Affine Transformations

This section describes the construction of 4 × 4 simple matrices. Visualizing these matrices is of relevance for understanding and building more complex deformations. The transformations of a point, $\mathrm{p} = (p_x, p_y, p_z)^\top$, or a vector, $\vec{\mathrm{v}} = (v_x, v_y, v_z)^\top$, do not involve matrix algebra, and their closed-forms are given. For some applications, the logarithm of the matrix may be required, and their closed-forms are provided: the product $\log M \cdot \mathrm{p}$ gives the velocity of p under the transformation.

A.1 Translation

Let us define a translation vector $\vec{\mathrm{t}} = (t_x, t_y, t_z)^\top$. The fourth homogeneous coordinate of $\vec{\mathrm{t}}$ is ignored. The translation matrix is:

$$T_{\vec{\mathrm{t}}} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{A.1}$$

The deformation of a point and a normal are respectively:

$$\begin{aligned} T_{\vec{\mathrm{t}}} \cdot \mathrm{p} &= \mathrm{p} + \vec{\mathrm{t}} \\ T_{\vec{\mathrm{t}}} \cdot \vec{\mathrm{v}} &= \vec{\mathrm{v}} \end{aligned} \tag{A.2}$$

The logarithm of a translation matrix is:

$$\log T_{\vec{\mathrm{t}}} = \begin{pmatrix} 0 & 0 & 0 & t_x \\ 0 & 0 & 0 & t_y \\ 0 & 0 & 0 & t_z \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{A.3}$$

The logarithm of a translation at a point and a normal are respectively:

$$\begin{aligned} \log T_{\vec{\mathrm{t}}} \cdot \mathrm{p} &= \vec{\mathrm{t}} \\ \log T_{\vec{\mathrm{t}}} \cdot \vec{\mathrm{v}} &= 0 \end{aligned} \tag{A.4}$$

A.2 Scale

Let us define a scale factor $s \in \mathbb{R}$. The matrix of a scale centered at the origin is much more simple than that of a scale centered at a point, c. We describe the two cases.

A.2.1 Scale centered at the origin

$$S_s = \begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} S_s \cdot \mathrm{p} &= s\mathrm{p} \\ S_s \cdot \vec{\mathrm{v}} &= s\vec{\mathrm{v}} \end{aligned}$$

$$\log S_s = \begin{pmatrix} \log s & 0 & 0 & 0 \\ 0 & \log s & 0 & 0 \\ 0 & 0 & \log s & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} \log S_s \cdot \mathrm{p} &= \log(s)\mathrm{p} \\ \log S_s \cdot \vec{\mathrm{v}} &= \log(s)\vec{\mathrm{v}} \end{aligned}$$

A.2.2 Scale centered at a point c

Scale center $=(c_x, c_y, c_z)^\top$

$$S_{s,\mathrm{c}} = T_{\mathrm{c}} \cdot S_s \cdot T_{-\mathrm{c}} = \begin{pmatrix} s & 0 & 0 & (1-s)c_x \\ 0 & s & 0 & (1-s)c_y \\ 0 & 0 & s & (1-s)c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} S_{s,\mathrm{c}} \cdot \mathrm{p} &= \mathrm{p} + (s-1)(\mathrm{p} - \mathrm{c}) \\ S_{s,\mathrm{c}} \cdot \vec{\mathrm{v}} &= s\vec{\mathrm{v}} \end{aligned}$$

$$\log S_{s,\mathrm{c}} = \begin{pmatrix} \log s & 0 & 0 & -c_x \log s \\ 0 & \log s & 0 & -c_y \log s \\ 0 & 0 & \log s & -c_z \log s \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

A.3 Non-uniform scale

Let us define a unit scale vector $\vec{\mathrm{n}} = (n_x, n_y, n_z)^\top$, along which the scale is performed. The matrix of a non-uniform scale centered at the origin is much more simple than that of a non-uniform scale centered at a point, c. We describe the two cases.

A.3.1 Centered at the origin

$$S_{s,\vec{\mathrm{n}}} = I + (s-1)\vec{\mathrm{n}}\cdot\vec{\mathrm{n}}^\top = \begin{pmatrix} 1+(s-1)n_x^2 & (s-1)n_xn_y & (s-1)n_xn_z & 0 \\ (s-1)n_xn_y & 1+(s-1)n_y^2 & (s-1)n_yn_z & 0 \\ (s-1)n_xn_z & (s-1)n_yn_z & 1+(s-1)n_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{aligned} S_{s,\vec{\mathrm{n}}}\cdot\mathrm{p} &= \mathrm{p} + (s-1)(\mathrm{p}\cdot\vec{\mathrm{n}})\vec{\mathrm{n}} \\ S_{s,\vec{\mathrm{n}}}\cdot\vec{\mathrm{v}} &= \vec{\mathrm{v}} + (1/s-1)(\vec{\mathrm{v}}\cdot\vec{\mathrm{n}})\vec{\mathrm{n}} \end{aligned}$$

$$\log S_{s,\vec{\mathrm{n}}} = \log(s)\vec{\mathrm{n}}\cdot\vec{\mathrm{n}}^\top = \begin{pmatrix} \log(s)n_x^2 & \log(s)n_xn_y & \log(s)n_xn_z & 0 \\ \log(s)n_xn_y & \log(s)n_y^2 & \log(s)n_yn_z & 0 \\ \log(s)n_xn_z & \log(s)n_yn_z & \log(s)n_z^2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

A.3.2 Centered at a point

$$\begin{aligned} S_{s,\vec{\mathrm{n}},\mathrm{c}} &= T_\mathrm{c}\cdot S_{s,\vec{\mathrm{n}}}\cdot T_{-\mathrm{c}} \\ &= \begin{pmatrix} 1+(s-1)n_x^2 & (s-1)n_xn_y & (s-1)n_xn_z & (1-s)(\mathrm{c}\cdot\vec{\mathrm{n}})n_x \\ (s-1)n_xn_y & 1+(s-1)n_y^2 & (s-1)n_yn_z & (1-s)(\mathrm{c}\cdot\vec{\mathrm{n}})n_y \\ (s-1)n_xn_z & (s-1)n_yn_z & 1+(s-1)n_z^2 & (1-s)(\mathrm{c}\cdot\vec{\mathrm{n}})n_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} S_{s,\vec{\mathrm{n}},\mathrm{c}}\cdot\mathrm{p} &= \mathrm{p} + (s-1)((\mathrm{p}-\mathrm{c})\cdot\vec{\mathrm{n}})\vec{\mathrm{n}} \\ S_{s,\vec{\mathrm{n}},\mathrm{c}}\cdot\vec{\mathrm{v}} &= \vec{\mathrm{v}} + (1/s-1)(\vec{\mathrm{v}}\cdot\vec{\mathrm{n}})\vec{\mathrm{n}} \end{aligned}$$

$$\log S_{s,\vec{\mathrm{n}},\mathrm{c}} = T_\mathrm{c}\cdot\log S_{s,\vec{\mathrm{n}}}\cdot T_{-\mathrm{c}}$$

A.4 Rotation

Let us define a rotation of angle θ around unit axis $\vec{\mathrm{n}} = (n_x, n_y, n_z, 0)^\top$. We define positive rotations to be clockwise in the viewing direction $\vec{\mathrm{n}}$. The reader should be aware that it may is more convenient to use quaternions [DKL98]. The formulas below may be obtained by factorizing the matrix of a rotation with a quaternion:

A.4.1 Centered at the origin

$$\begin{aligned} R_{\theta,\vec{\mathrm{n}}} &= (1-\cos(\theta))(\vec{\mathrm{n}}\cdot\vec{\mathrm{n}}^\top) \\ &+ \begin{pmatrix} \cos(\theta) & -\sin(\theta)n_z & \sin(\theta)n_y & 0 \\ \sin(\theta)n_z & \cos(\theta) & -\sin(\theta)n_x & 0 \\ -\sin(\theta)n_y & \sin(\theta)n_x & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} R_{\theta,\vec{\mathrm{n}}}\cdot\mathrm{p} &= (1-\cos(\theta))(\vec{\mathrm{n}}\cdot\mathrm{p})\vec{\mathrm{n}} + \cos(\theta)\mathrm{p} + \sin(\theta)\vec{\mathrm{n}}\times\mathrm{p} \\ R_{\theta,\vec{\mathrm{n}}}\cdot\vec{\mathrm{v}} &= (1-\cos(\theta))(\vec{\mathrm{n}}\cdot\vec{\mathrm{v}})\vec{\mathrm{n}} + \cos(\theta)\vec{\mathrm{v}} + \sin(\theta)\vec{\mathrm{n}}\times\vec{\mathrm{v}} \end{aligned}$$

$$\log R_{\theta,\vec{\mathrm{n}}} = \begin{pmatrix} 0 & -\theta n_z & \theta n_y & 0 \\ \theta n_z & 0 & -\theta n_x & 0 \\ -\theta n_y & \theta n_x & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} \log R_{\theta,\vec{\mathrm{n}}} \cdot \mathrm{p} &= \theta \vec{\mathrm{n}} \times \mathrm{p} \\ \log R_{\theta,\vec{\mathrm{n}}} \cdot \vec{\mathrm{v}} &= \theta \vec{\mathrm{n}} \times \vec{\mathrm{v}} \end{aligned}$$

A.4.2 Centered at a point

$$R_{\theta,\vec{\mathrm{n}},\mathrm{c}} = T_{\mathrm{c}} \cdot R_{\theta,\vec{\mathrm{n}}} \cdot T_{-\mathrm{c}}$$

$$\begin{aligned} R_{\theta,\vec{\mathrm{n}},\mathrm{c}} \cdot \mathrm{p} &= (1-\cos(\theta))(\vec{\mathrm{n}} \cdot (\mathrm{p}-\mathrm{c}))\vec{\mathrm{n}} + \sin(\theta)\vec{\mathrm{n}} \times (\mathrm{p}-\mathrm{c}) + \cos(\theta)(\mathrm{p}-\mathrm{c}) + \mathrm{c} \\ R_{\theta,\vec{\mathrm{n}},\mathrm{c}} \cdot \vec{\mathrm{v}} &= (1-\cos(\theta))(\vec{\mathrm{n}} \cdot \vec{\mathrm{v}})\vec{\mathrm{n}} + \sin(\theta)\vec{\mathrm{n}} \times \vec{\mathrm{v}} + \cos(\theta)\vec{\mathrm{v}} + \mathrm{c} \end{aligned}$$

$$\begin{aligned} \vec{\mathrm{a}} &= \theta_i \vec{\mathrm{v}}_i \\ \vec{\mathrm{b}} &= \mathrm{c} \times \vec{\mathrm{a}} \\ \log R_{\theta,\vec{\mathrm{n}},\mathrm{c}} &= \begin{pmatrix} 0 & -a_z & a_y & b_x \\ a_z & 0 & -a_x & b_y \\ -a_y & a_x & 0 & b_z \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{aligned} \tag{A.5}$$

Appendix B

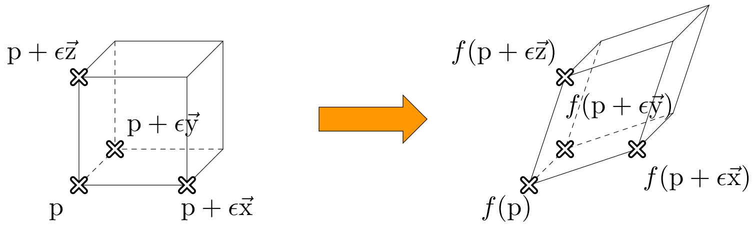
Constant-Volume Swirl

B.1 Constant volume basic swirl

Let us prove that a basic swirl $f(\mathrm{p})$, preserves the volume everywhere. The determinant of the Jacobian of f, a scalar denoted $\det J$ is the ratio between the deformed volume and the initial volume measured at a point (see Figure B.1):

$$\begin{array}{rcl} \det J & \underset{\epsilon\to 0}{=} & \frac{\det(f(\mathrm{p}+\epsilon\vec{\mathrm{x}})-f(\mathrm{p}),f(\mathrm{p}+\epsilon\vec{\mathrm{y}})-f(\mathrm{p}),f(\mathrm{p}+\epsilon\vec{\mathrm{z}})-f(\mathrm{p}))}{\epsilon^3} \\ & = & \det(\frac{\partial f}{\partial x},\frac{\partial f}{\partial y},\frac{\partial f}{\partial z}) \end{array} \tag{B.1}$$

Thus to show that a swirl preserves volume, we need to show that the determinant of the Jacobian, $\det J$, is equal to 1 everywhere.



$\mathrm{p}+\epsilon\vec{\mathrm{z}}$
$\mathrm{p}+\epsilon\vec{\mathrm{y}}$
p
$\mathrm{p}+\epsilon\vec{\mathrm{x}}$
$f(\mathrm{p}+\epsilon\vec{\mathrm{z}})$
$f(\mathrm{p}+\epsilon\vec{\mathrm{y}})$
$f(\mathrm{p}+\epsilon\vec{\mathrm{x}})$
$f(\mathrm{p})$

Figure B.1: *Left: an element of volume* ϵ^3. *Right: the volume of the deformed element is* $(f(\mathrm{p}+\epsilon\vec{\mathrm{x}})-f(\mathrm{p}))\cdot((f(\mathrm{p}+\epsilon\vec{\mathrm{y}})-f(\mathrm{p}))\times(f(\mathrm{p}+\epsilon\vec{\mathrm{z}})-f(\mathrm{p})))$. *With* $\epsilon\to 0$, *the volumes ratio is the determinant of the Jacobian of* f.

In this section, we denote by p a spatial element and by $\mathbf{p}$ it corresponding quaternion. Both are linked with the following relation: $\mathbf{p}=(0,\mathrm{p})$. A rotation of angle 2θ around $\vec{\mathrm{n}}$ and its powers in ϕ can be modeled with a quaternion:

$$\mathbf{q}^\phi=(\cos(\phi\theta),\sin(\phi\theta)\vec{\mathrm{n}}) \tag{B.2}$$

Hence the swirl deformation of a point $\mathrm{p} = (x, y, z)^\top$ is

$$f(\mathrm{p}) = \mathbf{q}^\phi * \mathbf{p} * \overline{\mathbf{q}^\phi} \tag{B.3}$$

We can assume without loss of generality that the rotation is centered at the origin. To express the Jacobian, we need the three partial derivatives of f. The first partial derivative of f is

$$\frac{\partial f(\mathrm{p})}{\partial x} = \frac{\partial \mathbf{q}^\phi}{\partial x} * \mathbf{p} * \overline{\mathbf{q}^\phi} + \mathbf{q}^\phi * \mathbf{e}_x * \overline{\mathbf{q}^\phi} + \mathbf{q}^\phi * \mathbf{p} * \frac{\partial \overline{\mathbf{q}^\phi}}{\partial x} \tag{B.4}$$

The reader can verify the quaternion equality:

$$\frac{\partial \mathbf{q}^\phi}{\partial x} = -\theta \frac{\partial \phi}{\partial x}(\sin(\phi\theta), -\cos(\phi\theta)\vec{\mathrm{n}}) \tag{B.5}$$

Using B.5, the leftmost term of the right side of Equation (B.4) is a quaternion:

$$\begin{array}{rl} & \frac{\partial \mathbf{q}^\phi}{\partial x} * \mathbf{p} * \overline{\mathbf{q}^\phi} \\ = & \theta \frac{\partial \phi}{\partial x}(-\sin(\phi\theta), \cos(\phi\theta)\vec{\mathrm{n}}) * \mathbf{p} * \overline{\mathbf{q}^\phi} \\ = & \theta \frac{\partial \phi}{\partial x}(-\cos(\phi\theta)\vec{\mathrm{n}} \cdot \mathrm{p}, -\sin(\phi\theta)\mathrm{p} + \cos(\phi\theta)\vec{\mathrm{n}} \times \mathrm{p}) * \overline{\mathbf{q}^\phi} \\ = & \theta \frac{\partial \phi}{\partial x}(-\vec{\mathrm{n}} \cdot \mathrm{p}, \cos(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) - \sin(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) \times \vec{\mathrm{n}}) \end{array}$$

The reader can verify similarly that the rightmost term of equation B.4 is also a quaternion:

$$\frac{\partial \mathbf{q}^\phi}{\partial x} * \mathbf{p} * \overline{\mathbf{q}^\phi} = \theta \frac{\partial \phi}{\partial x}(\vec{\mathrm{n}} \cdot \mathrm{p}, \cos(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) - \sin(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) \times \vec{\mathrm{n}})$$

The partial derivative of f in x is a vector:

$$\frac{\partial f(\mathrm{p})}{\partial x} = 2\theta \frac{\partial \phi}{\partial x}(\cos(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) - \sin(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) \times \vec{\mathrm{n}}) + \mathbf{q}^\phi * \mathbf{e}_x * \overline{\mathbf{q}^\phi}$$

Let us introduce $\vec{\mathrm{q}}_x$ and $\vec{\mathrm{u}}$ for the sake of simplicity:

$$\begin{array}{rcl} \mathbf{q}_x & = & \mathbf{q}^\phi * \mathbf{e}_x * \overline{\mathbf{q}^\phi} \\ \vec{\mathrm{u}} & = & 2\theta(\cos(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) - \sin(2\phi\theta)(\vec{\mathrm{n}} \times \mathrm{p}) \times \vec{\mathrm{n}}) \end{array}$$

The partial derivative in x shortens to:

$$\frac{\partial f(\mathrm{p})}{\partial x} = \frac{\partial \phi}{\partial x}\mathbf{u} + \mathbf{q}_x \tag{B.6}$$

The two other partial derivatives of f are obtained by substituting x for y or z. Since a rotation preserves lengths and angles, we can write:

$$\vec{\mathrm{q}}_x = \vec{\mathrm{q}}_y \times \vec{\mathrm{q}}_z$$

Let us develop the determinant of the Jacobian by replacing $\frac{\partial f(\mathrm{p})}{\partial x}$, $\frac{\partial f(\mathrm{p})}{\partial y}$ and $\frac{\partial f(\mathrm{p})}{\partial z}$ by their values:

$$\begin{array}{rcl} \det J & = & \frac{\partial f}{\partial x} \cdot (\frac{\partial f}{\partial y} \times \frac{\partial f}{\partial x}) \\ & = & 1 + \vec{\mathrm{u}} \cdot (\frac{\partial \phi}{\partial x} q_x + \frac{\partial \phi}{\partial y} q_y + \frac{\partial \phi}{\partial z} q_z) \end{array}$$

We can assume without loss of generality that $\vec{\mathrm{n}} = \vec{\mathrm{e}}_x$. This provides expressions for the rotated canonic set:

$$\begin{array}{rcl}\vec{\mathrm{q}}_x & = & \vec{\mathrm{e}}_x \\ \vec{\mathrm{q}}_y & = & \cos(2\phi\theta)\vec{\mathrm{e}}_y + \sin(2\phi\theta)\vec{\mathrm{e}}_z \\ \vec{\mathrm{q}}_z & = & \cos(2\phi\theta)\vec{\mathrm{e}}_z - \sin(2\phi\theta)\vec{\mathrm{e}}_y\end{array}$$

This assumption also provides a simple expression for the double cross product:

$$(\vec{\mathrm{n}} \times \mathrm{p}) \times \vec{\mathrm{n}} = y\vec{\mathrm{e}}_y + z\vec{\mathrm{e}}_z$$

We will now use the fact that the tool is spherical. We model the field function ϕ as a function of the distance to the origin, $d(p)$. The field can be partially derived:

$$\frac{\partial\phi(d(\mathrm{p}))}{\partial x} = \frac{\partial\phi}{\partial d}\frac{\partial d(\mathrm{p})}{\partial x} = \frac{\partial\phi}{\partial d}\frac{x}{d(\mathrm{p})}$$

With this, the determinant of the Jacobian becomes:

$$\begin{array}{rl} & \det J \\ = & 1 + \frac{\partial\phi}{\partial d}\frac{2\theta}{d(\mathrm{p})} \\ & (\cos()\vec{\mathrm{e}}_x \times \mathrm{p} - \sin()(y\vec{\mathrm{e}}_y + z\vec{\mathrm{e}}_z))\cdot \\ & (x\vec{\mathrm{e}}_x + (y\cos() - z\sin())\vec{\mathrm{e}}_y + (y\sin() + z\cos())\vec{\mathrm{e}}_z)\end{array}$$

The above dot product expands to zero, hence $\det J$ is equal to 1 everywhere. Therefore the deformation stretches space with no expansion or compression.

B.2 Swirl angle

The image of a point p in the center of a circle of swirls is given by Equation 4.1. Since the point is at the center, one can substitute ϕ_i for 1/2 (using the notation of Section 3.1.1):

$$f(\mathrm{p}) = \left(\bigoplus_{i=0}^{n-1}(\frac{1}{2} \odot M_i)\right) \cdot \mathrm{p} \tag{B.7}$$

The speed of this deformation at p is given by the logarithm:

$$\vec{\mathrm{v}} = \sum_{i=0}^{n-1} \frac{1}{2}\log(M_i) \cdot \mathrm{p} \tag{B.8}$$

Since M_i is a rotation matrix, this simplifies (see Equation A.5):

$$\vec{\mathrm{v}} = \frac{\theta}{2}\sum_{i=0}^{n-1}(\vec{\mathrm{v}}_i \times (\mathrm{p} - \mathrm{c}_i)) \tag{B.9}$$

By taking the Euclidean norm:

$$\|\vec{\mathrm{v}}\| = \frac{\theta}{2}\sum_{i=0}^{n-1}\|\mathrm{p} - \mathrm{c}_i\| \tag{B.10}$$

Since the centers are equidistant to p:

$$\|\vec{\mathrm{v}}\| = \frac{\theta}{2}nr \tag{B.11}$$

Therefore the angle is :

$$\theta = \frac{2\|\vec{\mathrm{v}}\|}{nr} \tag{B.12}$$

APPENDIX C

Monotonic Mostly-C^1 Interpolation

C.1 Monotonic Mostly-C^1 Interpolation

An easy way of interpolating data is to use piecewise polynomials. The data consists of a set of values f_k defined at the locations $k = 0, \dots N-1$. As done by [FSJ01], a value at a point $t \in [t_k, t_{k+1}]$ can be interpolated using a Hermite interpolant

$$f(t) = a_0 + a_1(t - t_k) + a_2(t - t_k)^2 + a_3(t - t_k)^3$$

where

$$\begin{aligned} a_3 &= d_k + d_{k+1} - 2\Delta_k \\ a_2 &= 3\Delta_k - 2d_k - d_{k+1} \\ a_1 &= d_k \\ a_0 &= f_k \end{aligned}$$

and

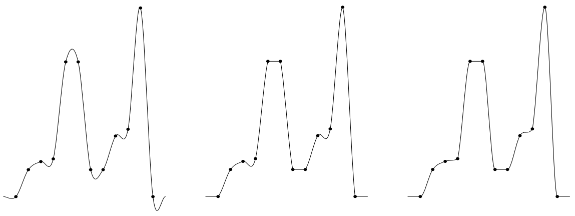
$$d_k = (f_{k+1} - f_{k-1})/2 \ , \quad \Delta_k = f_{k+1} - f_k$$

For convecting a field, it is important not to overshoot the data. [FSJ01] propose a necessary but not sufficient condition to control monotonicity: their strategy is to set the slopes to 0 whenever the slopes have a sign different from Δ_k. Their interpolant is not always monotonic and can overshoot the data, as shown in Figure C.1(b). Also, it loses C^1 continuity unnecessarily in cases where the slope of two joining polynomials could be set to 0.

We propose a sufficient condition, by clamping the slopes in a interval proportional to Δ_k. By doing this, our interpolant will always be monotonic, and will be C^1 continuous as long as it stays monotonic.

```
if ($\Delta_k < 0$)
  {
    if ($3\Delta_k > d_k$)
      $d_k = 3\Delta_k$;
    else if ($d_k > \Delta_k/2$)
      $d_k = 0$;
    if ($3\Delta_k > d_{k+1}$)
      $d_{k+1} = 3\Delta_k$;
    else if ($d_{k+1} > \Delta_k/2$)
      $d_{k+1} = 0$;
  }
```

```
else
  {
    if ($3\Delta_k < d_k$)
      $d_k = 3\Delta_k$;
    else if ($d_k < \Delta_k/2$)
      $d_k = 0$;
    if ($3\Delta_k < d_{k+1}$)
      $d_{k+1} = 3\Delta_k$;
    else if ($d_{k+1} < \Delta_k/2$)
      $d_{k+1} = 0$;
  }
```



(a) (b) (c)

Figure C.1: Standard cubic Hermite interpolation (left) and Fedkiw et al. interpolant (middle) produces overshoots, while our interpolation scheme (right) guarantees that no overshoots occur.

C.2 Discretization

We represent the vector field of the flow in a uniform discretization of space into N^3 voxels, with spacing 1. We define the divergence as:

$$\begin{aligned}(\nabla \cdot u)_{i,j,k} \;=\; & (u_{i+1,j,k} - u_{i-1,j,k} + \\ & u_{i,j+1,k} - u_{i,j-1,k} + \\ & u_{i,j,k+1} - u_{i,j,k-1}) \;/\; 2\end{aligned} \tag{C.1}$$

We define the discrete gradient as:

$$\begin{aligned}(\nabla p)_{i,j,k} \;=\; & (p_{i+1,j,k} - p_{i-1,j,k}, \\ & p_{i,j+1,k} - p_{i,j-1,k}, \\ & p_{i,j,k+1} - p_{i,j,k-1}) \;/\; 2\end{aligned} \tag{C.2}$$

We define the discrete Laplacian as:

$$\begin{aligned}(\nabla^2 p)_{i,j,k} \;=\; & p_{i+1,j,k} + p_{i-1,j,k} + \\ & p_{i,j+1,k} + p_{i,j-1,k} + \\ & p_{i,j,k+1} + p_{i,j,k-1} - 6p_{ijk}\end{aligned} \tag{C.3}$$

References

[AA03] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In *Proceedings of Shape Modeling International 2003*, pages 272–279, 2003.

[AB97] Fabrice Aubert and Dominique Bechmann. Volume-preserving space deformation. *Computers & Graphics*, 21(5):625–639, 1997.

[ACWK04a] Alexis Angelidis, Marie-Paule Cani, Geoff Wyvill, and Scott King. Swirling-sweepers: Constant-volume modeling. In *Pacific Graphics 2004*, pages 10–15. IEEE, October 2004. Best paper award at PG04.

[ACWK04b] Alexis Angelidis, Marie-Paule Cani, Geoff Wyvill, and Scott King. Swirling-sweepers: Constant-volume modeling (sketch). SIGGRAPH 2004 Sketch, August 2004.

[AJC02] Alexis Angelidis, Pauline Jepp, and Marie-Paule Cani. Implicit modeling with skeleton curves: Controlled blending in contact situations. In *Shape Modeling International*, pages 137–144. ACM, IEEE Computer Society Press, 2002.

[AK04] Nina Amenta and Yong Kil. Defining point-set surfaces. In *SIGGRAPH 2004*, volume 23(3) of *ACM Transactions on Graphics, Annual Conference Series*, pages 264–270. ACM, August 2004.

[Ale02a] Marc Alexa. Linear combination of transformations. In *Proceedings of SIGGRAPH'02*, volume 21(3) of *ACM Transactions on Graphics, Annual Conference Series*, pages 380–387. ACM, ACM Press / ACM SIGGRAPH, July 2002.

[Ale02b] Marc Alexa. Recent advances in mesh morphing. In *Computer Graphics Forum*, volume 21(2), pages 173–198. ACM, June 2002.

[AN05] Alexis Angelidis and Fabrice Neyret. Simulation of smoke based on vortex filament primitives. In *Symposium on Computer Animation'05*, pages 87–96, July 2005. http://www-evasion.imag.fr/Publications/2005/AN05.

[AW04] Alexis Angelidis and Geoff Wyvill. Animated sweepers: Keyframed swept deformations. In *CGI 2004*, pages 320–326. IEEE, July 2004.

[AWC04] Alexis Angelidis, Geoff Wyvill, and Marie-Paule Cani. Sweepers: Swept user-defined tools for modeling by deformation. In *Proceedings of Shape Modeling and Applications*, pages 63–73. IEEE, June 2004. Best paper award at SMI04.

[Bae01] J. A. Baerentzen. On the implementation of fast marching methods for 3d lattices. Technical Report IMM-REP-2001-13, Technical University of Denmark (DTU), 2001.

[Bar84] Allan H. Barr. Global and local deformations of solid primitives. In *Proceedings of SIGGRAPH'84*, volume 18(3) of *Computer Graphics Proceedings, Annual Conference Series*, pages 21–30. ACM, ACM Press / ACM SIGGRAPH, July 1984.

[Bat67] G. K. Batchelor. *An Introduction to Fluid Dynamics.* Cambridge Univ. Press, UK, 1967.

[BB91] P. Borrel and D. Bechmann. Deformation of n-dimensional objects. In *Proceedings of the first symposium on Solid modeling foundations and CAD/CAM applications*, pages 351–369, 1991.

[BBB$^+$97] Chandrajit Bajaj, Jim Blinn, Jules Bloomenthal, Marie-Paule Cani-Gascuel, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill. *Introduction to Implicit Surfaces.* Morgan-Kaufmann, 1997.

[BK03] Mario Botsch and Leif Kobbelt. Multiresolution surface representation based on displacement volumes. In *Proceedings of Eurographics*, pages 483–491. Eurographics, September 2003.

[Bla94] Carole Blanc. A generic implementation of axial procedural deformation techniques. In *Graphics Gems*, volume 5, pages 249–256, 1994. Academic Press.

[Blo90] Jules Bloomenthal. Calculation of reference frames along a space curve. *Graphics gems*, pages 567–571, 1990.

[BM98] C. Bregler and J. Malik. Tracking people with twists exponential maps. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, page 8. IEEE, 1998.

[BR94] P. Borrel and A. Rappoport. Simple constrained deformations for geometric modeling and interactive design. In *ACM Transactions on Graphics*, volume 13(2), pages 137–155, April 1994.

[BW98] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of SIGGRAPH'98*, Computer Graphics Proceedings, Annual Conference Series, pages 46–54. ACM, July 1998.

[CJ91] S. Coquillart and P. Jancène. Animated free-form deformation: An interactive animation technique. In *Proceedings of SIGGRAPH'91*, volume 25(4) of *Computer Graphics Proceedings, Annual Conference Series*, pages 23–26. ACM, ACM Press / ACM SIGGRAPH, July 1991.

[CK00] G.-H. Cottet and P. D. Koumoutsakos. *Vortex Methods: Theory and Practice*. Cambridge University Press, 2000. ISBN: 0521621860.

[CMOV02] Georges-Henri Cottet, Bertrand Michaux, Sepand Ossia, and Geoffroy VanderLinden. A comparison of spectral and vortex methods in three-dimensional incompressible flows. *J. Comput. Phys.*, 175(2):702–712, 2002.

[Coq90] Sabine Coquillart. Extended free-form deformation: A sculpturing tool for 3d geometric modeling. In *Proceedings of SIGGRAPH'90*, volume 24(4) of *Computer Graphics Proceedings, Annual Conference Series*, pages 187–195. ACM, ACM Press / ACM SIGGRAPH, July/August 1990.

[CR94] Y.-K. Chang and A. P. Rockwood. A generalized de Casteljau approach to 3d free-form deformation. In *Proceedings of SIGGRAPH'94*, Computer Graphics Proceedings, Annual Conference Series, pages 257–260. ACM, ACM Press / ACM SIGGRAPH, July 1994.

[Cre99] Benoit Crespin. Implicit free-form deformations. In *Proceedings of the Fourth International Workshop on Implicit Surfaces*, pages 17–24, 1999.

[DC95] Mathieu Desbrun and Marie-Paule Cani. Animating soft substances with implicit surfaces. In *Proceedings of SIGGRAPH'95*, pages 287–290. ACM, August 1995.

[DC03] Guillaume Dewaele and Marie-Paule Cani. Interactive global and local deformations for virtual clay. In *Proceedings of Pacific Graphics*, pages 131–140. IEEE, October 2003.

[Dec96] Philippe Decaudin. Geometric deformation by merging a 3d object with a simple shape. In *Graphics Interface*, pages 55–60, May 1996.

[DKL98] Erik B. Dam, Martin Koch, and Martin Lillholm. Quaternions, interpolation and animation. Technical report, Institute of Computer Science (DIKU) University of Copenhagen, 1998. http://www.diku.dk/forskning/image/teaching/Studentprojects/Quaternion/.

[Ebe01] David H. Eberly. *3D Game Engine Design*. Morgan Kaufmann, 2001.

[EMF02] Douglas Enright, Steve Marschners, and Ron Fedkiw. Animation and rendering of complex water surfaces. In *Proceedings of SIGGRAPH*, volume 21(3) of *ACM Transactions on Graphics, Annual Conference Series*, pages 736–744. ACM, July 2002.

[Far90] G. Farin. Surfaces over Dirichlet tessellations. *Computer Aided Geometric Design*, 7(1-4):281–292, June 1990.

[FCG02] Eric Ferley, Marie-Paule Cani, and Jean-Dominique Gascuel. Resolution adaptive volume sculpting. *Graphical Models*, 63:459–478, March 2002.

[FF01] Nick Foster and Ron Fedkiw. Practical animation of liquids. In *Proceedings of SIGGRAPH*, pages 23–29. ACM, August 2001.

[FL04] Raanan Fattal and Dani Lischinski. Target-driven smoke animation. In *Proceedings of SIGGRAPH'04*, volume 23(3), pages 441–448, August 2004.

[FLS89] Richard Feynman, Robert Leighton, and Matthew Sands. *The Feynman, Lectures on Physics*, volume III, chapter 40 and 41. Addison-Wesley, 1989.

[FM96] Nick Foster and Demitri Metaxas. Realistic animation of liquids. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 204–212. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996.

[FM97] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *SIGGRAPH 97 Proceedings*, pages 181–188. ACM SIGGRAPH, August 1997.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of SIGGRAPH*, ACM Transactions on Graphics, Annual Conference Series, pages 15–22. ACM, August 2001.

[FvDF$^+$94] James D. Foley, Andries van Dam, Steven K. Feiner, John Hughes, and Richard Phillips. *Introduction to Computer Graphics.* Addison-Wesley, 1994. p.392.

[Gal01] Jean Gallier. *Geometric Methods and Applications For Computer Science and Engineering*, chapter 14. Springer-Verlag, 2001.

[GD99] James E. Gain and Neil A. Dodgson. Adaptive refinement and decimation under free-form deformation. *Eurographics'99*, 7(4):13–15, April 1999.

[GD01] James E. Gain and Neil A. Dodgson. Preventing self-intersection under free-form deformation. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):289–298, October-December 2001.

[Gha01] A. Gharakhani. Grid-free simulation of 3-d vorticity diffusion by a high-order vorticity redistribution method. In *15th AIAA Computational Fluid Dynamics Conference*, pages 1–10, June 2001.

[Gla89] Andrew Glassner, editor. *An Introduction to Ray traycing.* Morgan Kaufmann, 1989.

[GLG95] Manuel Noronha Gamito, Pedro Faria Lopes, and Mário Rui Gomes. Two-dimensional simulation of gaseous phenomena using vortex particles. In *EG Computer Animation and Simulation '95*, pages 2–15. Eurographics, September 1995.

[Gou03] David A.D. Gould. *Complete Maya Programming: An Extensive Guide to MEL and the C++ API.* Morgan Kaufman, 2003.

[GP89] J. Griessmair and W. Purgathofer. Deformation of solids with trivariate b-splines. In *Eurographics Conference Proceedings*, pages 137–148. Elsevier Science, 1989.

[Gri99] David J. Griffiths. *Introduction to Electrodynamics.* Prentice Hall, third edition, 1999.

[Gué01] A. P. Guéziec. "Meshsweeper": Dynamic point-to-polygonal-mesh distance and applications. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):47–61, January/March 2001.

[GW92] R. Gonzalez and R. Woods. *Digital Image Processing*, chapter 3, pages 148–156. Addison-Wesley, 1992.

[HHK92] W. M. Hsu, J. F. Hughes, and H. Kaufman. Direct manipulation of free-form deformations. In *Proceedings of SIGGRAPH'92*, volume 26(2) of *Computer Graphics Proceedings, Annual Conference Series*, pages 177–184. ACM, ACM Press / ACM SIGGRAPH, July 1992.

[HML99] Gentaro Hirota, Renee Maheshwari, and Ming C. Lin. Fast volume-preserving free form deformation using multi-level optimization. In *Proceedings of the fifth ACM symposium on Solid modeling and applications*, pages 234–245. ACM, June 1999.

[HQ04] J. Hua and H. Qin. Scalar-field-guided adaptive shape deformation and animation. *The Visual Computer*, 1(1):47–66, April 2004.

[KKM03] Scott A. King, Alistair Knott, and Brendan McCane. Language-driven nonverbal communication in a bilingual conversational agent. In *16th International Conference on Computer Animation and Social Agents (CASA 2003)*, pages 17–22, May 2003.

[KM90] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *Proceedings of SIGGRAPH'90*, pages 49–57, August 1990.

[Koe90] Jan J. Koenderink. *Solid Shapes.* MIT Press, 1990.

[KY97] Y. Kurzion and R. Yagel. Interactive space deformation with hardware assisted rendering. *IEEE Computer Graphics and Applications*, 17(5):66–77, September/October 1997.

[LCJ94] Francis Lazarus, Sabine Coquillart, and Pierre Jancène. Axial deformations: an intuitive deformation technique. In *Computer-Aided Design*, volume 26(8), pages 607–613, 1994.

[LF02] Arnauld Lamorlette and Nick Foster. Structural modeling of natural flames. In *Proceedings of SIGGRAPH 02*, pages 729–735, July 2002.

[Lip69] Martin M. Lipschutz. *Differential Geometry.* McGraw-Hill, 1969.

[LKG$^+$03] I. Llamas, Byungmoon Kim, Joshua Gargus, Jarek Rossignac, and Chris D. Shaw. Twister: A space-warp operator for the two-handed editing of 3d shapes. In *SIGGRAPH*, volume 22(3) of *ACM Transactions on Graphics, Annual Conference Series*, pages 663–668. ACM, August 2003.

[LNC91] T. T. Lim, T. B. Nickels, and M. S. Chong. A note on the cause of rebound in the head-on collision of a vortex ring with a wall. *Expt. in Fluids*, 12(1/2):41–48, 1991.

[Mar97] Daniel Margerit. *Mouvement et dynamique des filaments et des anneaux tourbillons de faible épaisseur.* PhD thesis, INPL, 1997.

[MJ96] R. A. MacCracken and K. I. Joy. Free-form deformations with lattices of arbitrary topology. In *Proceedings of SIGGRAPH'96*, Computer Graphics Proceedings, Annual Conference Series, pages 181–188. ACM, ACM Press / ACM SIGGRAPH, August 1996.

[MMT97] L. Moccozet and N. Magnenat-Thalmann. Dirichlet free-form deformation and their application to hand simulation. In *Computer Animation'97*, pages 93–102, June 1997.

[MP89] Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers & Graphics*, 13(3):305–309, 1989.

[MS98] Jon McCormack and Andrei Sherstyuk. Creating and rendering convolution surfaces. In *Computer Graphics Forum*, volume 17(2), pages 113–120, June 1998.

[MTPS04] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. In *Proceedings of SIGGRAPH'04*, volume 23(3), pages 449–456, August 2004.

[MW01] David Mason and Geoff Wyvill. Blendeforming: Ray traceable localized foldover-free space deformation. In *Proceedings of Computer Graphics International (CGI)*, pages 183–190, July 2001.

[NB93] P. Ning and J. Bloomenthal. An evaluation of implicit surface tilers. In *IEEE Computer Graphics and Applications*, volume 13(6), pages 33–41. ACM, 1993.

[Ney03] Fabrice Neyret. Advected textures. In *Symposium on Computer Animation*, pages 147–153, August 2003.

[Par77] R. Parent. A system for sculpting 3d data. In *Proceedings of SIGGRAPH'77*, volume 11(2) of *Computer Graphics Proceedings, Annual Conference Series*, pages 138–147. ACM, ACM Press / ACM SIGGRAPH, July 1977.

[PB88] John C. Platt and Alan H. Barr. Constraint methods for flexible models. In *Proceedings of SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 279–288. ACM, August 1988.

[PCS04] Frédéric Pighin, Jonathan M. Cohen, and Maurya Shah. Modeling and editing flows using advected radial basis functions. In *Symposium on Computer Animation*, pages 223–232. ACM/Eurographics, August 2004.

[Per85] Ken Perlin. An image synthesizer. In *Proceedings of SIGGRAPH'85*, pages 287–296, July 1985.

[PGK02] Mark Pauly, Markus Gross, and Leif P. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization '02*, pages 163–170. IEEE Computer Society, 2002.

[PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. In *Proceedings of SIGGRAPH'03*, volume 22(3), pages 641–650. ACM, July 2003.

[PN01] Ken Perlin and Fabrice Neyret. Flow noise. *Siggraph Technical Sketches and Applications*, page 187, August 2001.

[REN$^+$04] N. Rasmussen, D. Enright, D. Nguyen, S. Marino, N. Sumner, W. Geiger, S. Hoon, and R. Fedkiw. Directable photorealistic liquids. In *Symposium on Computer Animation*, pages 193–202, July 2004.

[RNGF03] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald P. Fedkiw. Smoke simulation for large-scale phenomena. In *Proceedings of SIGGRAPH'03*, volume 22(3), pages 703–707, July 2003.

[RSB95] Ari Rappoport, Alla Sheffer, and Michel Bercovier. Volume-preseving free-form solids. In *Proceedings of Solid Modeling*, pages 361–372. ACM, May 1995.

[Rut90] Aris Rutherford. *Vectors, Tensors and the Basic Equations of Fluid Mechanics.* Dover, 1990.

[SCP$^+$04] Maurya Shah, Jonathan M. Cohen, Sanjit Patel, Penne Lee, and Frédéric Pighin. Extended galilean invariance for adaptive fluid simulation. In *Symposium on Computer Animation*, pages 213–221, July 2004.

[SE04] Sagi Schein and Gershon Elber. Discontinuous free form deformations. In *Proceedings of Pacific Graphics*, pages 227–236. IEEE, October 2004.

[SF93] Jos Stam and Eugene Fiume. Turbulent wind fields for gaseous phenomena. In *Proceedings of SIGGRAPH'93*, pages 369–376, August 1993.

[SF98] Karan Singh and Eugene Fiume. Wires: a geometric deformation technique. In *Computer graphics, Proceedings of SIGGRAPH'98*, Computer Graphics Proceedings, Annual Conference Series, pages 405–414. ACM, ACM Press / ACM SIGGRAPH, July 1998.

[SNBW03] F. F. Samavati, M. Ali Nur, R. Bartels, and B. Wyvill. Progressive curve representation based on reverse subdivision. In *the 2003 International Conference on Computational Science and Its Applications*, May 2003.

[SP86] T. Sederberg and S. Parry. Free-form deformation of solid geometric models. In *Proceedings of SIGGRAPH'86*, volume 20(4) of *Computer Graphics Proceedings, Annual Conference Series*, pages 151–160. ACM, ACM Press / ACM SIGGRAPH, August 1986.

[SSEH03] Joshua Schpok, Joseph Simons, David S. Ebert, and Charles Hansen. A real-time cloud modeling, rendering, and animation system. In *Symposium on Computer Animation*, pages 160–166, August 2003.

[Sta99] Jos Stam. Stable fluids. In *Proceedings of SIGGRAPH*, pages 121–128. ACM, August 1999.

[Sta01] Jos Stam. A simple fluid solver based on the FFT. *Journal of Graphics Tools*, 6(2):43–52, 2001.

[TBHF03] J. Teran, S. Blemker, V. Ng Thow Hing, and R. Fedkiw. Finite volume methods for the simulation of skeletal muscle. In *Eurographics/SIGGRAPH Symposium on Computer Animation'2003*, pages 68–74. ACM, 2003.

[TJ81] Frank Thomas and Ollie Johnston. *The illusion of life.* Hyperion, 1981.

[VF02] Anne Verroust and Matthieu Finiasz. A control of smooth deformations with topological change on a polyhedral mesh based on curves and loops. In *Shape Modelling International*, pages 191—198. ACM, IEEE Computer Society Press, May 2002. Banff, Alberta, Canada.

[vOS83] A. van Oosterom and J. Strackee. The solid angle of a plane triangle. In *IEEE Transactions on Biomedical Engineering*, volume 30(2), pages 125–126, February 1983.

[WB01] Andrew Witkin and David Baraff. Physically based modeling, online siggraph 2001 course notes, 2001. http://www.pixar.com/companyinfo/research/pbm2001/.

[Weia] Eric Weisstein. Lagrange multipliers. From Mathworld – A Wolfram Web Ressource http://mathworld.wolfram.com/LagrangeMultipliers.html.

[Weib] Eric Weisstein. Screw theorem. From Mathworld – A Wolfram Web Ressource http://mathworld.wolfram.com/ScrewTheorem.html.

[Weic] Eric W. Weisstein. Riemann sum. From MathWorld–A Wolfram Web Resource http://mathworld.wolfram.com/RiemannSum.html.

[WH91] Jakub Wejchert and David Haumann. Animation aerodynamics. In *Proceedings of SIGGRAPH'91*, pages 19–22, held in Las Vegas, Nevada; 28 July - 2 August 1991 1991.

[WMW86] G. Wyvill, G. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, August 1986.