A robust approach to interactive virtual cutting: geometry and color

by

Pietroni Nico

Theses Series **DISI-TH-2009-02**

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

A robust approach to interactive virtual cutting: geometry and color

by

Pietroni Nico

June, 2009

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by Nico Pietroni
DISI, Univ. di Genova
nico.pietroni@isti.cnr.it

Date of submission: February 2009

Title: A robust approach to interactive virtual cutting: geometry and color

Advisor: FABIO GANOVELLI
Visual Computing Laboratory
ISTI, Consiglio Nazionale delle Ricerche (C.N.R.)
fabio.ganovelli@isti.cnr.it

Supervisor: ENRICO PUPPO
Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
puppo@disi.unige.it

Ext. Reviewers: SUMANTA N. PATTANAIK
School of Electrical Engineering and Computer Science
University of Central Florida
sumant@cs.ucf.edu

RICCARDO SCATENI
Dipartimento di Matematica e Informatica Università di Cagliari
riccardo@unica.it

Abstract

Interactive simulation of deformable bodies has attracted growing interest in the course of the last decade and, while for a long time it has been limited to applicative domains such as virtual surgery, it is nowadays a fundamental part of almost every game engine. The reasons of this evolution may be found both in the continuous effort of the scientific community and in the technological improvement of computers performance that allowed to sustain such a calculation-intensive task even on commodity computers.

The simulation of a deforming object requires a physical model of the object behavior and an efficient and stable algorithm to simulate it. Generally speaking, the physical model must consider the phenomenon at the right scale (e.g. a ball will not be modeled as the interaction of its atoms) and capture the aspects of the simulation we are interested in (e.g. do not include the temperature when computing the bouncing of the ball). Concerning the algorithm, it must be able to update the state of the system in real-time and it must be stable. The latter is particularly critical because simulation includes resolution of Partial Differential Equations (PDEs) which easily could easily diverge if not handled with care.

Although many consolidated results in this field exist, there are still problems that need further investigation, for example how to model the cutting (or fracturing) of deformable objects.

A cut on a deformable object has two major implications: it changes its boundary by adding a new portion of surface (the part that is revealed by the cut) which means that the geometric description must be updated on-the-fly; new information (e.g. the color) is needed to render the newly generated surface portion; finally, it changes the physical behavior of the object, which translates in updating the boundary conditions of the physical model.

The contribution of this thesis to the problem stated above is twofold:

- A new algorithm to model interactive cuts or fractures on deforming objects, named Splitting Cubes. The Splitting Cubes can be considered as a tessellation algorithm for deformable surfaces. It is independent from the underlying physical model which defines the deformation functions. Due to its stability and efficiency, the Splitting Cubes is particularly suitable for interactive simulations, including virtual surgery and games.

- A new algorithm to derive the color of the interior of an object from few cross sections. To address this problem we propose a new appearance-modeling paradigm for synthesizing the internal structure of a 3D model from photographs of a few cross-sections of a real object. In our approach colors attributes (textures) of the surface are synthesized on demand during the simulation. We will demonstrate that our modeling paradigm reveal highly realistic internal surfaces in a variety of artistic flavors. Due to its efficiency, our approach is suitable for real-time simulations.

We finally present two collateral results that emerged during the research carried out in these years: a robust model for real-time simulation of knot-tying which is certainly useful in endoscopic surgical simulator; a technique for building a virtual model of a human head, developed in the framework of the approximation of individual Head Related Transfer Functions (HRTF) for the realistic binaural rendering of three-dimensional sound.

To my family

Creativity is more than just being different. Anybody can plan weird; that's easy. What's hard is to be as simple as Bach. Making the simple, awesomely simple, that's creativity. *(Charles Mingus)*

Acknowledgements

First, I thank Fabio Ganovelli for the solid scientific contribute he provides to this work. Another special thank goes to Miguel A. Otaduy for his precious help on texturing stuff. Thanks to Roberto Scopigno, Claudio Montani and Enrico Puppo for supporting me during the PHD.
I wish to thank all the people composing the Visual Computing Laboratory including Marco Tarini, Paolo Cignoni, Guido Ranzuglia, Marco Callieri, Matteo Delle Piane, Ricardo Marroquim, Marco Di Benedetto, Federico Ponchio, Massimiliano Corsini, together with past Lab members: Giuseppe Turini, Andrea Spinelli, Valentino Fiorin, Giuseppe Croccia, Rita Borgo and Paolo Pingi.
Thanks to Markus Gross for involving me in the Computer Graphics Laboratory in Zurich, and thus enriching my scientific skill.
I wish to thank all the other people involved in my papers: Blazej Kubiak, Marco Fratarcangeli, Bernd Bickel, Nicolas Tsingos and Manuel Asselot.
A special thanks goes to Paola for providing me every day a new fresh loving energy, together with Angela for her artistic sense of life. I am grateful to my parents and to my brothers, they everyday believed in me.
Finally, thanks to my secret advisor...

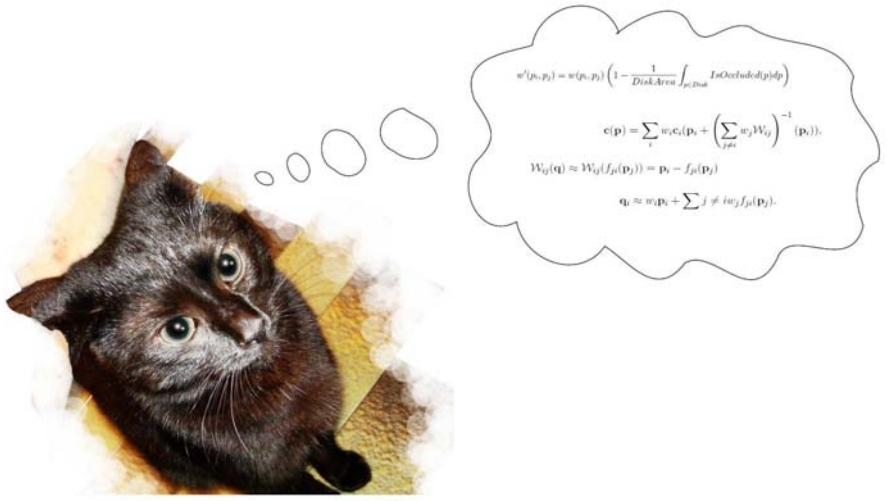

$w'(p_i,p_j) = w(p_i,p_j)\left(1 - \frac{1}{DiskArea}\int_{p\in Disk} IsOccluded(p)dp\right)$
$\mathbf{c}(\mathbf{p}) = \sum_i w_i \mathbf{c}_i(\mathbf{p}_i + \left(\sum_{j\neq i} w_j \mathcal{W}_{ij}\right)^{-1}(\mathbf{p}_i)).$
$\mathcal{W}_{ij}(\mathbf{q}) \approx \mathcal{W}_{ij}(f_{ji}(\mathbf{p}_j)) = \mathbf{p}_i - f_{ji}(\mathbf{p}_j)$
$\mathbf{q}_i \approx w_i\mathbf{p}_i + \sum j \neq i w_j f_{ji}(\mathbf{p}_j).$

Figure 1: My secret advisor..

Table of Contents

List of Figures **5**

List of Tables **9**

Chapter 1 Introduction **11**

1.1 Applications domains of deformable bodies simulation 12

1.2 Cutting or fracturing a deformable object 13

1.3 Scope of the thesis and results 14

1.3.1 Primary Contribution 14

1.3.2 Secondary Contribution 15

1.4 Outline of the thesis 15

Chapter 2 Interacting with deformable objects **17**

2.1 Background 17

2.1.1 Continuum Elasticity 17

2.1.2 Discretization 19

2.1.3 Time Integration 20

2.1.4 Mesh-based and Meshless methods 23

2.1.5 Mass-Spring Method 24

2.1.6 Finite Elements Method 25

2.1.7 Meshless methods 28

2.1.8 Meshless deformations Based on Shape Matching 31
2.1.9 Mesh-based vs Mesh-free methods 32
2.2 Encoding discontinuities on Mesh-based methods 33
2.3 Encoding discontinuities on Mesh-free methods 35
2.3.1 Adapting the physical simulation in mesh-free methods 36
2.3.2 Adapting the surface in mesh-free methods 37
2.3.3 An abstract representation of a Discontinuity 39
2.4 The Splitting Cubes Algorithm 40
2.4.1 Splitting Cubes: a 2D illustrative example 42
2.4.2 Splitting Cubes: from 2D to 3D 43
2.4.3 Position of the vertices 44
2.4.4 Interpolation inside a cell 46
2.5 Construction of the Look-Up-Table. 47
2.6 Physical response to cutting 49
2.6.1 Nodes-Phyxels bounds 49
2.6.2 Phyxels-Phyxels bounds: the Extended Visibility Method 49
2.6.3 Implementing the Extended Visibility Method. 51
2.7 Initial Setup 52
2.8 Results and discussion 53

Chapter 3 Modeling the interior of an object **59**
3.1 texture classification 60
3.2 A brief introduction to 2D Texture Synthesis 60
3.2.1 Procedural Methods 61
3.2.2 Statistical features-matching methods 62
3.2.3 Pixel Based 64
3.2.4 Patch Based Methods 66
3.2.5 Texture optimization method 68

3.3 Designing internal properties of meshes 70

3.3.1 Solid Textures 70

3.3.2 Solid Meshes 82

3.3.3 Solid Textures vs Solid Meshes 86

3.4 Texturing Internal surfaces from a few cross sections 89

3.4.1 Texture Synthesis Pipeline 90

3.4.2 Texture Interpolation Domain 92

3.4.3 Texture Morphing 95

3.5 Results 100

3.6 Conclusions and Future Work 104

Chapter 4 Additional Results **107**

4.1 A Robust method for Real-Time thread simulation 107

4.1.1 Previous Work 108

4.1.2 Our Approach 109

4.1.3 Collision Detection 113

4.1.4 Visual and Haptic feedback 114

4.1.5 Conclusions and future work 115

4.2 Reconstructing head models from photographs for individualized 3D-audio processing 116

4.2.1 individualized HRTF modeling 117

4.2.2 Reconstruction of head models 119

4.2.3 Results 125

4.3 Conclusions and future work 127

Chapter 5 Final Remarks **129**

5.1 Future work 130

5.2 List of Publications 132

Bibliography **133**

List of Figures

1 My secret advisor.. 1

2.1 Clothes modeled with mass spring systems using implicit time integration [BW98]. 25

2.2 FEM with Stiffness Warping 26

2.3 A sequence produced by using Invertible FEM [ITF04] 28

2.4 Point Based Animation [MKN$^+$04] 30

2.5 Meshless deformation based on shape matching [MHTG05] 32

2.6 Techniques to implement cuts in mesh based models 33

2.7 Comparison of between a real (Top) and a simulated (Bottom) fracture [OH99] 34

2.8 Visibility, Diffraction and Transparency kernel 36

2.9 A Comparison between visibility criterion, diffraction methods, transparency method and Extended Visibility criterion 38

2.10 An abstract representation of a discontinuity 39

2.11 Example of cut with multiple advancing fronts using the Splitting Cubes . 40

2.12 The idea underlying the Splitting Cubes 41

2.13 The six configurations for a face of the splitting cube 43

2.14 Two examples of cuts handled with the splitting cubes 44

2.15 Placement of vertices in the Splitting Cubes 45

2.16 Interpolation inside a Cell 46

2.17 Construction of Splitting Cubes Look-Up-Table 47

2.18 The occlusion Discs 50
2.19 Decrease kernel by using the Extended Visibility Method 51
2.20 Cutting of a deformable model of the liver 53
2.21 Performances of the Splitting Cubes 53
2.22 Scissors cutting a deformable pear. 54
2.23 A cut with a curved tool. 55
2.24 Another cut with a curved tool. 56

3.1 Classification of textures as proposed by [LLH04]. 60
3.2 Application of Perlin noise 61
3.3 Examples of textures generated by the generalized reaction-diffusion model [McG08]. 62
3.4 Example of conditioning kernel for Pixel-based methods 64
3.5 A comparison between 2D texture synthesis algorithms 66
3.6 some results of Parallel 2D texture synthesis [LH05, LH06] 67
3.7 Texture quilting algorithm [EF01] 68
3.8 Examples of textures synthesized by using global optimization [KEBK05] . 69
3.9 A 3D neighborhood composed by 3 orthogonal 3D slices. 71
3.10 Examples of solid textures produced by using Perlin noise 71
3.11 [Examples of solid textures synthesized by using statistical features-matching methods](a) Examples of solid textures produced by [HB95]. The textured model is carved from the synthesized texture block (b) Anisotropic solid textures generated by [DGF98] using multiple template images. 73
3.12 The synthesis pipeline of stereological technique for solid texture synthesis [JDR04] 74
3.13 Displacement configurations of [QY07] generated by a 3^2 kernel. 75
3.14 Examples of Aura 3D synthesis [QY07] 77
3.15 Examples of solid textures produced by [Wei02] 78
3.16 The candidate used by Lazy solid texture synthesis 79
3.17 Solid textures produced by Lazy Solid Texture Synthesis [DLTD08] 80

3.18 Comparison of different methods to authoring solid texture synthesis . . . 82
3.19 Layered textures created by using a procedural language [CDM+02] 83
3.20 Volumetric Illustrations [ONOI04] 84
3.21 2D Lapped textures [PFH00] 85
3.22 Classification of solid texture appearance according to [TOII08]. 86
3.23 Lapped Solid Textures [TOII08] 87
3.24 Our Paradigm for Digital Content Creation 90
3.25 BSP-Tree Produced by Exemplars 93
3.26 Interpolation Inside BSP Regions 94
3.27 Morphing Using Gaussian Stacks 96
3.28 Showing the internal appearance of an orange. 99
3.29 Feature enhancement using local histogram matching 101
3.30 Bunny with patterned textures 102
3.31 Versatile Texturing 103
3.32 New Generation of Transgenic Fruits 104

4.1 An example of knot tying performed by our algorithm. 108
4.2 Stiffness,Bending & Collision constraints formulation 110
4.3 Friction & Contact constraints formulation 111
4.4 Torsion constraints formulation 112
4.5 Simple pruning test for self collision detection based on angles 113
4.6 Haptic setup 114
4.7 Strangling the bunny 115
4.8 Example of knots 116
4.9 A scheme of the whole HRTF calculation system 118
4.10 Three elements of the 3D dummy library 120
4.11 Ear and Head selection and alignment 121
4.12 Lateral head deformation 123

4.13 Symmetrization 124
4.14 Example of ear morphing sequence. 124
4.15 Two results of processed heads. 125
4.16 Two examples of polar plots for measurements on couples of scanned-reconstructed 3D heads. 127

List of Tables

3.1 Comparison between solid textures and solid meshes. 87

4.1 Distance in mm between key-points of scanned and reconstructed model . . 126

4.2 Difference in mm between distances indicated in [Lar01] 126

Chapter 1

Introduction

The simulation of real-world phenomena is one of the major topics of Computer Graphics. Physics provides models for representing and simulate natural phenomena in a way they can be encoded and solved via computer simulation.

In computer graphics, physically-based models are exploited in several contexts with the goal of increasing the degree of realism of a simulated environment. For example, physically-based modeling of light transmission increases visual realism of a rendered scene, as well as accurate simulation of mechanical phenomena present in a virtual scene, like smoke, fire or fluids, co-occur to make a simulated environment more realistic.
Among the applications of physics to computer graphics, a growing interest concerns the simulation of deformable bodies. With the term deformable body we mean an object whose shape changes under the action of external forces, according to the mechanical properties of the materials it is made of. Most organic objects in the real world are deformable. The simulation of deformable objects is used to increase the realism of a simulated world by animating the objects in the scene. Moreover, deformable bodies simulation is a fundamental task for various application domains such as off-line animation or interactive applications like video-games or virtual surgery.
The models proposed by continuum mechanics are usually approximated and discretized to make them suitable for computer simulation. Intuitively, the more a simulation is physically accurate the more it is perceived as real by the user. Nevertheless, the majority of Computer Graphics applications require only that the produced simulation appears physically "plausible". That is not the case, for example, for mechanical engineering applications that generally needs highly accurate simulations for safety and functionality.

Although the early approaches to implement deformation where essentially borrowed from mechanical engineering, further research for interactive methods lead to solutions that not necessarily produce *plausible* but not necessarily accurate behavior, therefore trading accuracy for speed preserving realism.

1.1 Applications domains of deformable bodies simulation

We can split the Computer Graphics applications domain of deformable bodies modeling in two main categories:

Off-line simulation Simulation of deformable objects can be used to create realistic animations with a minimum user intervention.
Imagine we want to create a sequence of frames, which defines the animation of a particular physical phenomenon. In this case, we want to maximize the realism of the animation and at the same time to minimize the workload of the animator. Instead of animating manually each frame of the entire sequence, a physically-based method can be used to define implicitly the behavior of the different objects composing the scene. For some complex scenarios, such as fluid animation, defining manually the entire animation is a tedious work, and, in most of the cases, the final result will appear unrealistic. On the other hand, the quality produced by a physical simulation is almost indistinguishable with respect to a real-world sequence. Another application can be found in the design of clothes, to be able to see how they fit a particular virtual body before actually producing it.
This class of applications requires the simulation to be as accurate as possible, in terms of rendering quality and accuracy of the mechanical behavior, while efficiency is generally considered a secondary objective.

Real-time simulation A different group of applications requires the simulation of deformable objects in real-time, i.e. a sustained frame rate has to be achieved. Virtual surgery and games are the typical application domains for this class of systems. In both cases the visual feedback produced by the system has to be fast enough to give the user the impression of the real world. The accuracy of the simulation and the rendering quality are typically compromised to benefit the update frequency and the visual smoothness of the system. Another fundamental property that must be achieved by this class of systems is robustness, since the approximations due to model discretization may lead to numerical errors that make the system unstable.

1.2 Cutting or fracturing a deformable object

One of the main goal of this thesis is real-time simulation of cutting and fracturing of deformable objects. This task is fundamental, for example, in virtual surgery, where the surgeon usually cuts or lacerates human tissues by using a set of tools. In a similar way, a game may allow fracturing objects to increase both interactivity and degrees of realism. Modeling cuts is a complex operation, which involves physical and visual modifications in the deformable object:

Physical Modification When we cut a deformable objects we create a discontinuity inside its volume.
Most of the representations employ a subdivision of the object domain in finite elements that are used both for visual representation and for deformation computation. The problem is that the requirements of such subdivision to guarantee an accurate and stable simulation conflicts with the need to update it on-the-fly to implement a cut. This problem has been tackled in a number of research papers without actually providing a complete solution. Generally speaking, for off line simulation the subdivision can be carefully adapted step by step, trading speed for accuracy, while for interactive simulation the solutions tend to add simplificative assumptions and/or to limit the accuracy for keeping the simulation fast and stable.

Modification of object's appearance Some modifications have an impact on the object representation in terms of shape and color properties.
The external shape of the object has to be modified in order to show internal surfaces revealed by cuts or fractures. Such modifications are typically implemented by increasing the geometric primitives representing the object's boundary. Since the external surface of the object has to be animated as well, its complexity has to be limited in order ensure the minimum update rate required by the application. Furthermore, the modifications of the external shape may produce degeneration of the surface representation.
When an object is cut or fractured, it shows its internal appearance. For example if we split a fruit or a vegetable we expect to see its internal structure, in terms of color variation, according to the cut location and orientation. Then, in order to increase the realism of the simulation, it is important to provide a system that dynamically texturizes appropriately the new object's boundaries created by the cuts or the fractures.

1.3 Scope of the thesis and results

We define a set of efficient methodologies, which are integrated in a framework, to produce an interactive simulation of deformable objects, which allows the insertion of cuts or fractures. Our research is mainly motivated to ensure stability and efficiency in a real time framework. Furthermore, rendering quality is improved by synthesizing texture attribute of internal surfaces in real-time.

As previously stated, interactive simulation of deformable objects cut must be at the same time robust and realistic.
Robustness is the capacity of the system to remain stable even when cuts, fractures or large deformations occur. Methods previously proposed in literature keep the system stable by using complex re-meshing operations effecting both physical model and boundary representation.

1.3.1 Primary Contribution

Splitting Cubes We propose a new object's boundary representation called *Splitting Cubes*, which allows to define cuts and fractures while, at the same time, ensuring stability and efficiency. Splitting Cubes is an algorithm which provides a dynamic tessellation of an evolving surface embedded in a deforming space. Such deforming space could be dynamically updated by a physical simulation.
We decouple the physical simulation from the boundary shape representation. Although the cut resolution is fixed, we gain a considerable advantage in terms of efficiency and stability with respect to previous methods. That characteristic makes this algorithm particularly suitable for real time applications.

Extended Transparency Method We successfully integrate the Splitting Cubes with a meshless method which simulates the physical behavior of the deformable object. We propose the *Extended Transparency Method* to model discontinuities due to cuts or fractures for meshless methods. We demonstrate how our method improves stability with respect to previous methods proposed in literature. Furthermore thanks to its formulation, the Extended Transparency Method is particularly suitable for real-time applications, since it can be easily implemented on the GPU.

Synthesis of internal appearance We integrate our framework with a method that allows, with a minimum user intervention, to define internal appearance of the objects.

The main challenges relatives to these particular class of modeling paradigm are: to design of an intuitive interface allowing the user to specify the interior of the object with a minimum workload, to built a synthesis method that can capture and synthesize features at different scales of resolution. The modeling paradigm we propose is an advance in terms of intuitive modeling and expressing power (in terms of synthesis). We show how a naive user can easily model the internal appearance of an object to increase the visual realism of the simulation. Furthermore, being real-time, our method is particularly suitable for interactive simulations.

1.3.2 Secondary Contribution

Tying knots As additional result we first propose a novel method for interactive simulation of thread dynamics allowing knot tying, which is a key task for endoscopic surgery simulation. Our method proves to be more efficient and robust, with respect to previous methods.

Automatic calculation of HRTF Secondarily, we propose a method for automatic calculation of individualized Head Related Transfer Function (HRTF), which is indispensable for binaural rendering of three-dimensional sound. This function is strictly related to the peculiar features of ears and face of the listener. The system proves to be fast, automatic, robust and reliable: geometric validation and preliminary assessments show that it can be accurate enough for HRTF calculation.

1.4 Outline of the thesis

The remaining of this dissertation is organized as follows:

- In Chapter 2 we introduce the problem of virtual cutting. First we provide a brief introduction on continuum based mechanics, which is the basic theory underlying discretized models used in Computer Graphics. Next, we introduce the problem of time discretization, which is fundamental to represent body dynamics, showing and discussing existing methodologies. The models proposed in literature for deformable object simulation are split in two main groups: mesh-based and mesh-less methods. We finally conclude this introduction by making a comparison between this two class of methods, devoting particular attention to how they model discontinuities. Then we provide a detailed description of the splitting cubes algorithm, showing results and main advantages provided by our method.

- In Chapter 3 we analyze the problem of defining the internal appearance of an object. After an introduction to 2D texture synthesis, we illustrate the main approaches proposed in literature, which we group in two main classes: solid texture and solid meshes, describing advantages and drawbacks of each class of methods. Then we introduce a novel paradigm to capture and synthesize the property of an interior of an object in real-time. Among the possible advantages shown by this method, we show how it can be easily integrated with the interactive simulation framework we proposed in the previous chapter to provide the color of newly created surfaces.

- Finally, we provide in Chapter 4 our additional contributions. We first define a new robust and efficient method for real-time thread simulation allowing knot tying. Then, we define a completely automatic system for producing a 3D model of a head using uncalibrated photographs. The method has been used for sound scattering calculation.

The contributions of this thesis are summarized in chapter 5, also suggesting some way to improve our work and opportunities for future research

Chapter 2

Interacting with deformable objects

2.1 Background

2.1.1 Continuum Elasticity

A deformable object is a body whose shape may change dynamically under the action of a force field. The way its shape changes depends on a set of parameters K that represents its *mechanical properties*.
The *undeformed shape*, or *rest shape*, is the initial configuration in which the object is when no external force is applied.
An *equilibrium configuration* is a state where the energy due to deformation is at a local minimum. In particular, rest shape can be classified as an equilibrium configuration where that energy term is zero.
The rest shape is described by a continuous connected subset of $I\!R^3$ referred as *material coordinates* or M, while points belonging to such subset are usually called *material points.*
The parameters K characterize how the object changes its shape when external events perturbs its equilibrium configuration. Intuitively, such parameters determine the mechanical behavior of a deformable object, for example they differentiate a soft from a stiff object or fluid from a solid object.

Under the action of applied forces, the object deforms, according to K, to reach a new equilibrium position. In particular, each material point x originally located at its rest position $m(x) \in I\!R^3$, moves to a new coordinates $w(x) \in I\!R^3$, which are called *deformed coordinates.* We can express deformed coordinates as the sum of material coordinates $m(x)$ and a displacement vector $u(x)$:

$$w(x) = m(x) + u(x) \tag{2.1}$$

The displacement field $u(x), \forall x \in M$ encode the entire body's deformation. It is important to notice that not every possible displacement field produces a deformation; a rigid transformation for example, such as a rotation or uniform displacement, does not produce any deformation.
The "amount of deformation" is expressed in terms of spatial variations of the displacement fields, so that the contribute provided by rigid transformations is nullified.

The *Strain tensor* ε express the "amount of deformation" in terms of *gradient of the displacement field* ∇u, that is the *Jacobian* of the displacement field:

$$\nabla u = \begin{bmatrix} u_{xx} & u_{yx} & u_{zx} \\ u_{xy} & u_{yy} & u_{zy} \\ u_{xz} & u_{yz} & u_{zz} \end{bmatrix} \tag{2.2}$$

A popular choice consist of evaluating strain trough the *Green strain tensor* ϵ_g, or its linear approximation, the *Cauchy's strain tensor* ϵ_c:

$$\epsilon_g = \frac{1}{2}(\nabla u + \nabla u^T + \nabla u^T \nabla u) \tag{2.3}$$

$$\epsilon_c = \frac{1}{2}(\nabla u + \nabla u^T) \tag{2.4}$$

Cauchy's stress principle asserts that when a force acts on a continuum body, then internal reactions (coded as force vectors) rise between the material points. In order to simulate the dynamic of a deforming object, it is important to quantify such internal forces.
The *Stress* measures the amount of force applied per area-unit. It is a measure of the intensity of the total internal forces acting within M across imaginary internal surfaces.
The state of stress at a point in M is defined by the nine components of the Cauchy stress tensor, $\sigma \in \mathbb{R}^{3\times 3}$, that, for isotropic purely elastic materials, is linearly related to strain by the *Hook's law*:

$$\sigma = E\epsilon \tag{2.5}$$

The coefficients $E \in \mathbb{R}^{3\times 3}$ depend on intrinsic material characteristics K, and determine the stiffness of simulated object. If we assume that the material is isotropic, then E is univocally determined by two independent values, *Young's modulus* and *Poisson's ratio.* Young's modulus E is the ratio of stress to strain on the loading plane along the loading direction, while Poisson's express the ratio of lateral strain and axial strain. The *Strain Energy Density* $U(x)$ defines the amount of energy stored on material points:

$$U(x) = \frac{1}{2}\epsilon(x) \cdot \sigma(x) \tag{2.6}$$

consequently, the total elastic energy U is obtained by integrating $U(x)$ over the entire domain:

$$U = \int_M U(x) \tag{2.7}$$

Finally, the *Elastic Force*, $F(x)$, acting on a material point, is the negative gradient of elastic strain density with respect to material point's displacement.
In linear elasticity this relation can be expressed as:

$$F(x) = -\nabla_u U(x) \tag{2.8}$$

2.1.2 Discretization

The total potential energy Π of a deformable object is described by the following equation:

$$\Pi = U + W \tag{2.9}$$

Where W is the load due to external forces (gravity or contact constraints for example).
The potential energy reaches a local minimum (defining an equilibrium configuration of the deformable object), when the derivative of Π with respect to the material points displacements functions is zero.

The minimization process leads to the resolution of the following differential equation, commonly referred as *Equilibrium Equation*, which describes the dynamics of a material point x:

$$\rho \cdot \ddot{w}(x,t) = \nabla \cdot \sigma + f_{ext} \tag{2.10}$$

where ρ is the density of the material, f_{ext} represents an externally applied force The divergence operator turns the 3 by 3 stress tensor back into a 3 vector:

$$\nabla \cdot \sigma = \begin{bmatrix} \sigma_{xx,x} + \sigma_{xy,y} + \sigma_{xz,z} \\ \sigma_{yx,x} + \sigma_{yy,y} + \sigma_{yz,z} \\ \sigma_{zx,x} + \sigma_{zy,y} + \sigma_{zz,z} \end{bmatrix} \tag{2.11}$$

The first term of equation 2.10 represents the internal force acting on material point x. It is defined by multiplying the second derivative of it's world position, which represents the acceleration, by its local density. The second term of equation is the sum of internal forces (which are described in terms of stress tensor) and the external applied forces.
While is possible to solve the PDE expressed by Equation 2.10 directly for very simple cases which provides an analytic description of the domain M (such as a sphere or a bar), it not possible to do so for the cases where the shape is more complex. For the majority of the real objects we have no analytic description of the domain (the integral described by Equation 2.7 is analytically insoluble), then in order simulate their elastic behavior we need to approximate somehow their domain.

Following these considerations, it becomes essential to discretize the continuum-mechanic based model, (described in Section 2.1.1), in a way that the PDE of Equation 2.10 is locally soluble on each discrete sample.
There are two main classes of methods related to the discretization of deformable objects domain, *mesh-based* and *mesh-less*. The following paragraphs give an overview of the major advantages and drawback of both classes, along with a detailed description of the discrete models that are the most popular in computer animation.

The derivation of the forces, due to deformation, is clearly not sufficient to animate a deforming body. Animating its dynamics requires the knowledge of time-dependent world coordinates of material points $w(x, t)$.
To make the whole simulation suitable for computer animation, time must be discretized by sampling at fixed interval δ_t, usually called *time steps*.
Then the "time-step dependent" sequence of world coordinates:

$$w(x, t_0), w(x, t_1)..., w(x, t_{(n-1)}), w(x, t_{(n-1)}) \tag{2.12}$$

can be used to generate the frame sequence of the scene.
Next section introduces the different numerical methods to express time-dependent world coordinates during the simulation, focusing on advantages and drawbacks of each of them.

2.1.3 Time Integration

If we want to display the dynamics of deforming objects, it is fundamental to know, for each time step t_i, the world coordinates $w(x, t_i)$ of material points.
Unfortunately, given the internal forces acting on the material points x and its world coordinates $w(x, t_i)$ at time step t_i, to obtain world coordinates for the next time step $w(x, t_{(i+1)})$ is not straightforward.
The value of world coordinates at next time-step is implicitly defined by the solution of the differential equation resulting from *Newton's second law of motion*:

$$\ddot{w}(x,t) = D(\dot{w}(x,t), w(x,t), t) \tag{2.13}$$

Where the function D depends on the model used in the simulation. The formula 2.13 asserts that the acceleration of the material point (expressed as second derivative of position, $\ddot{w}(x,t)$) is a function of its velocity ($\dot{w}(x,t)$) and its current position $w(x,t)$. The function, indicated as D, is uniquely determined by the model used for the simulation.

The most basic scheme to solve this differential equation is called *Explicit Euler Integration*. Given the solution at time t, that is $w(x,t)$; the solution for time $t+\delta_t$, is approximated using the first two terms of the *Taylor expansion*:

$$w(x,t+\delta_t) = w(x,t) + \delta_t \cdot \dot{w}(x,t) + O(\delta_t^2) \tag{2.14}$$

If we associate the mass $Mass(x)$ to the material point x, by applying that principle to equation 2.13, then the motion of material point x can be rewritten as:

$$\dot{w}(x,t+\delta_t) = \dot{w}(x,t) + \ddot{w}(x,t+\delta_t) \cdot \delta_t \tag{2.15}$$

$$w(x,t+\delta_t) = w(x,t) + \dot{w}(x,t+\delta_t) \cdot \delta_t \tag{2.16}$$

By considering Taylor expansion, it is easy to see that the error, for small δ_t, is proportional to δ_t^2.

Runge-Kutta Integration is an extension of Euler integration that allows substantially improved accuracy. The basic idea to subdivide the time step in interval to get a better evaluation of the derivatives:

$$w(x,t+\delta_t) = w(x,t) + \delta_t \cdot \dot{w}(x,t+\frac{\delta_t}{2}) \tag{2.17}$$

Equation 2.17 shows a particular instance of Runge-Kutta method, in this case derivatives are approximated using an additional evaluation step located in the middle of the time interval. This method is referred in literature as *Mid-Point Method.*
It is important to notice that the increment of accuracy given by Runge-Kutta methods, implies a computational overhead. This better approximation of time derivatives increases the stability of the whole time integration process, nevertheless, since it requires an additional evaluation step, the computational complexity is increased as well.
Another popular choice in computer graphics is the *Verlet Method* (see [MHHR07] for

details).

Summarizing, we can schematize the typical animation loop as follows:

$$\underbrace{u(x,t_i) \rightarrow \epsilon(x,t_i) \rightarrow \sigma(x,t_i) \rightarrow F(x,t_i)}_{elastic\ forces\ derivation} \Longrightarrow \underbrace{\ddot{w}(x,t_i) \rightarrow \dot{w}(x,t_i) \rightarrow w(x,t_i)}_{integration\ phase} \Longrightarrow u(x,t_{(i+1)}) \rightarrow ... \tag{2.18}$$

The first part of the animation loop concern the derivation of elastic forces: by using displacements of material points it is possible to derive strains, stresses and consequently the resulting elastic forces. Then in the integration phase, by using elastic forces we can derive acceleration, velocity and finally the word coordinates of the material point x. Finally, world coordinates of material point x, is used to derive its displacement vector for the sequent time interval.

All the Integration methods cited above can be grouped as *explicit integration techniques*, in fact unknown values for time t_i are calculated *explicitly* as a function of values at time $t_{(i-1)}$.
Explicit time integration methods are easy to implement and computationally light to execute, unfortunately its stability is strictly related to the size of time-step. In case of Euler integration, Delingette [Del98] asserts that given system of n masses linked by linear springs, then stability is achieved if $k_c \leq \frac{M_{tot}}{n\pi^2(\delta_t)^2}$,where k_c is the elastic constant of the strings, and M_{tot} is the total mass. In general stability is achieved by reducing the time step, slowing the overall simulation. The degree of realism of an interactive simulation of deformable bodies is strictly related to the size of the time step. Indeed reducing the size of time-step, usually increases the gap between simulated time and real-time, reducing the realism perceived by the user.

This problems is resolved by *Implicit Integration Techniques*. While for explicit methods the solution at time $t + \delta_t$ is a function of values and derivatives at time t, $w(x, t + \delta_t) = Fun_{explicit}(w(x,t), \dot{w}(x,t))$, in implicit methods the solution at time $t + \delta_t$ is function of derivatives at the same time $t + \delta_t$, then $w(x, t + \delta_t) = Fun_{implicit}(w(x,t), \dot{w}(x, t + \delta_t))$.
Baraff et al. in [BW98] introduce the Computer Graphics community to implicit integration; they develop an implicit integration schema using Euler method for cloth simulation, showing a significant advance in terms of stability. The implicit instance of Euler integration is usually referred as *backward Euler step* ,while its explicit version as *forward Euler step*.
In backward euler step the unknowns for next time step $(t + \delta t)$ appear in both sides of equation:

$$w(x, \mathbf{t} + \delta_\mathbf{t}) = w(x,t) + \delta_t \dot{w}(x, \mathbf{t} + \delta_\mathbf{t}) \tag{2.19}$$

$$\dot{w}(x, \mathbf{t} + \delta_{\mathbf{t}}) = \dot{w}(x,t) + \delta_t Fun_{implicit}(\dot{w}(x, \mathbf{t} + \delta_{\mathbf{t}}), w(x, \mathbf{t} + \delta_{\mathbf{t}})) \tag{2.20}$$

Where $Fun_{implicit}$ is defined by the discretized model used for the simulation (please see [BW98] for details).

Implicit methods still generate some error (similarly to explicit method, it derives from a truncation of the Taylor-series), but, in practical cases, it is more robust than explicit methods. Implicit integration solves world coordinates of material points together at each time step, as a coupled system, while Explicit integration treat each point independently. Additionally, Implicit methods reduces the potential for instability.
Implicit methods are, in general, more difficult to implement with respect to explicit methods and require more computational resources. Indeed these methods comports the resolution of a large sparse linear system at each time step, which is solved by [BW98] using a modified conjugate gradient. That system of equation is determined by the function D of equation 2.20. Such function describes the relation between the material points belonging to the deformable model.
If we want to create a discontinuity (such as cuts or fractures) into the domain, then we have to change the formulation of D, with the consequent reassembly of the system of equations. For this reasons Explicit time integration is more suitable for the simulation of cuts and fractures in real-time, since it treat each material point independently.

2.1.4 Mesh-based and Meshless methods

Discrete methods for deformable bodies simulation can be grouped into two broad categories: *mesh-based* and *meshless* methods.

In mesh-based methods the domain is divided into a set of disjoint elements grouped into topological map called a mesh.
The topology of the mesh and the shape of each element is designed such that is possible to interpolate inside each cell quantities that are defined just on nodes.
So doing we can integrate functions inside each cell and, consequently, apply continuum-based mechanics formulation (as explained in Section 2.1.1). In other words, the mechanic behavior of the entire domain is described by this mesh of cells, each one of them considered as a continuous piece of material. That consideration summarizes the main philosophy constituting the *Finite Element Analysis.*
Section 2.1.5 shows the simpler and more intuitive among mesh-based models, the *mass-spring method*, then section 2.1.6 introduces the classical *Finite Element Method* or *FEM.*

In meshless methods the volume of the body is sampled with a set of *particles* without any constraints regarding their distribution or connections. Quantities are interpolated using *meshless shape functions* that requires only the knowledge of a set of neighbors. Connectivity between particles (neighboring relations) usually is not maintained explicitly but updated for each simulation step.
Section 2.1.7 reports a meshless method for the simulation of deformable bodies: The *Point Based Animation Method* [MKN+04], which can be considered the implementation of the Element-free Galerkin Method [BLG94].
Finally in Section 2.1.8 we briefly present a simulation framework based on a geometry-based energy formulation: the *Shape Matching Method* [MHTG05] for deformable object simulation, which does not rely on continuum mechanics.

The reader can find a wider collection of simulation models in [GM97] or [NMK+05]).

2.1.5 Mass-Spring Method

Mass-Springs systems can be considered the simplest and most intuitive of all deformable models. They were successfully used for facial ([PB81], [Wat87]) and cloth animation ([EWS96], [BW98], [CK02], [EGS03]). Figure 2.1 shows an example of clothes animated using an implicit formulation of a mass-spring system [BW98].
In Mass-Springs system the domain M is sampled with unitary particles linked by a net of springs. Usually, springs are considered to be linear, but non-linear springs can be also used to model tissues that exhibit inelastic behavior.
Elastic forces rise when springs are elongated or compressed; if springs are linear then they are modeled by Hook's law. The elastic force F acting on a mass m_0 generated by a spring connecting the two particles p_0 and p_1, is described by the following equation:

$$F(m_0) = K \cdot (L - |w(p_0) - w(p_1)|) \cdot sign(w(p_0) - w(p_1)) \tag{2.21}$$

Where K is spring elastic constant , $w(p_0)$ and $w(p_1)$ are respectively world position of particles p_0 and p_1, and L is the length of the spring at its rest shape.
Since physical bodies are not perfectly elastic, they lose energy during deformation. To simulate this feature, a viscosity force term F_V is added:

$$F_V(m_0) = K_d(v_1 - v_0) \tag{2.22}$$

where v_0 and v_1 are the velocity of the particles and K_d is the spring's damping constant.

Usually, to model the dynamics of a deformable object we redistribute its total mass among particles. Similarly, quantities representing object's current state (such as velocity and world position) are sampled by particles.

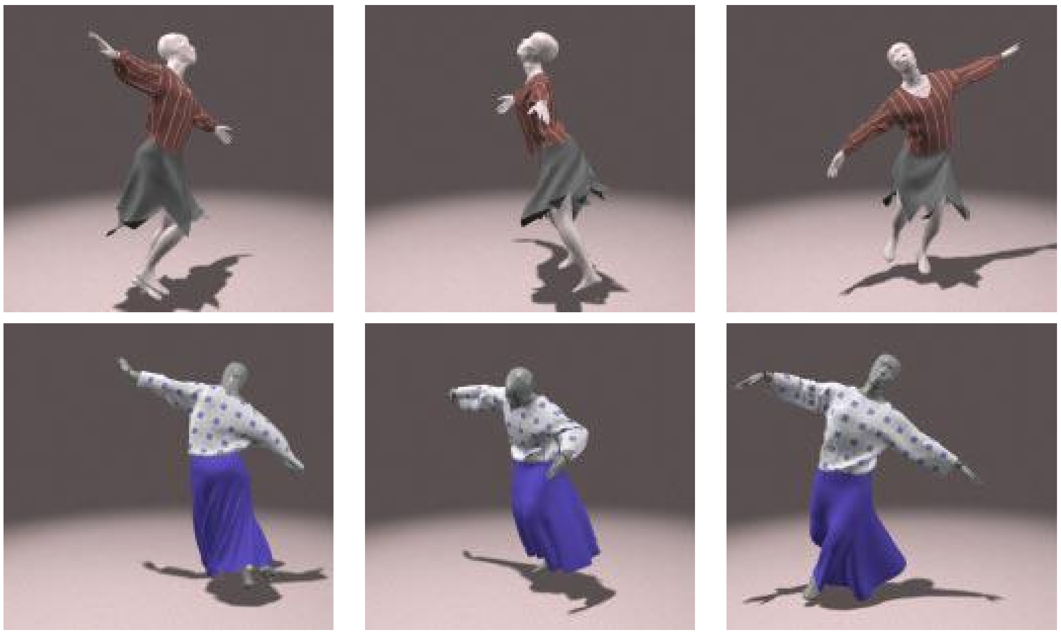


Figure 2.1: Clothes modeled with mass spring systems using implicit time integration [BW98].

Mass-springs systems are intuitive and simple to implement. However, to tune parameters of the springs that correspond to the known elastic properties of the material is not a trivial task [DKT95, Van98].

2.1.6 Finite Elements Method

In Finite Elements Methods (FEM) the volume is discretized into a set (mesh) of disjoint volumetric elements (also called cells) whose vertices are commonly referred to as nodes. Each element can be seen as continuously connected volume that can be modeled using continuum mechanics (as explained in section 2.1.1) .
Instead of solving the partial differential equation governing body dynamics (expressed in 2.10), over the entire volume, they are solved locally for each element.

Quantities that are defined for nodes $Q(x_i)$, are continuously interpolated inside each cell using nodal values:

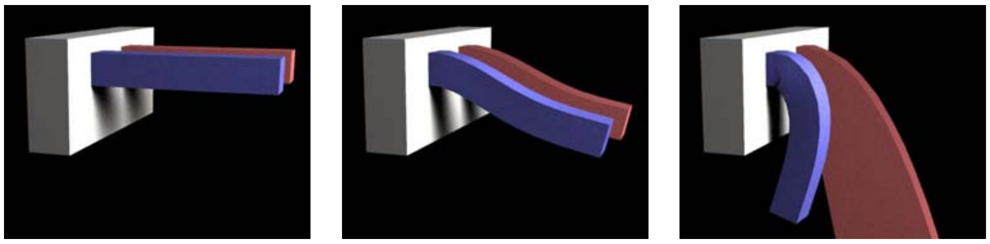


Figure 2.2: Two deformable bars simulated using Tetrahedral FEM with warped (blue) and linear (red) stress measures. Courtesy of [MMD+02].

$$\tilde{Q}(x) = \sum_i Q(x_i)B_i \tag{2.23}$$

where B_i are in general referred to as *shape functions* which are 1 at node i and zero at all other nodes (*Kronecker Delta* property). Because adjacent elements share nodes, then the entire mesh defines a piecewise interpolation function across the domain.
Through the shape functions it is possible to express PDE of 2.10, as a function of nodal values $w(x_i, t)$.

In computer animation is common to use a simplified version of FEM, which is called *Explicit Finite Element Method.* In Explicit FEM, for each cell i, a continuous 3D displacement vector field $u^i(x)$ is obtained trough linear interpolation of nodal displacements $U = u_0, u_1...u_N$, weighted by shape functions $B_k(x)$:

$$u^i(x) = \sum_{k<N} u(x_k)B_k(x) \tag{2.24}$$

Where N is the number of nodes composing the element.
Given $u^i(x)$, we can express the jacobian of displacement field $\nabla u^i(x)$ and, consequently, define a strain $\epsilon^i(x)$ and a stress $\sigma^i(x)$ field over each cell.

Then elastic energy is integrated for each cell:

$$U^i = \int_V \frac{1}{2}\epsilon^i(x) \cdot \sigma^i(x) \tag{2.25}$$

The total elastic energy can be seen as the sum of energy contributes coming from cells:

$$U = \sum_i U^i \tag{2.26}$$

Similarly to Equation , the elastic force $F(n_i)$ acting on the node n_i is defined as the negative gradient of U with respect to nodal displacements (the strain energy density of each point belonging to the cell can defined trough interpolation of shape functions).
Since in linear elasticity force is considered to be linearly related to energy, then it is possible to express the force field $F_e(x)$ produced by an element e as:

$$F_e(x) = K_e u_e(x) + O(\|u_e(x)\|^2) \tag{2.27}$$

where K_e is usually called the *stiffness matrix*. The linear algebraic equation governing the motion for a mesh composed by of N nodes is:

$$M\ddot{u} + D\dot{u} + Ku = f_{ext} \tag{2.28}$$

Where $M \in R^{N\times N}$ is the *Mass Matrix* expressing the mass of each node, $D \in R^{N\times N}$ is the *Damping Matrix*. The damping matrix reduces the force acting on a node proportionally to its velocity. The stiffness matrix $K \in R^{N\times N}$ is the obtained by cumulating the stiffness matrices coming from each elements, with respect to nodes (considering that adjacent elements shares nodes).Then, the simulation loop only requires the resolution of a system of equation. Since the stiffness matrix K depends on mesh topology and mesh's rest shape, then it can be precomputed. Note that equation can be considered as a discretization of equation 2.10, where the term $M\ddot{u}$ corresponds to the total elastic forces acting on a node, Ku models the elastic internal forces due to deformation, and the force term $D\dot{u}$ is added to model damping.

Linear elastic forces yield stable and accurate simulation in case of small deformations, while for large deformation it create visible artifacts.
Non-linear tensors overcome this problems ([DSB99]), unfortunately the use of non-linearity introduce stability problems and make the simulation loop more complex. To eliminate these artifacts while maintaining the main advantages of linear methods, Müller et al.([MMD$^+$02]) extracts the rotational part of the deformation for each element and compute the forces with respect to the non-rotated reference frame (see figure 2.2) . At each time step is computed a tensor field that describes the local rotations of all the vertices in the mesh. This field allows us to compute the elastic forces in a non-rotated reference frame (nullifying the strain contributes given by rotations).
An interesting development of FEM method is presented by Irving et al. ([ITF04]), where authors propose a FEM model that allows large stable deformations and volume inversion (as shown by figure 2.3).

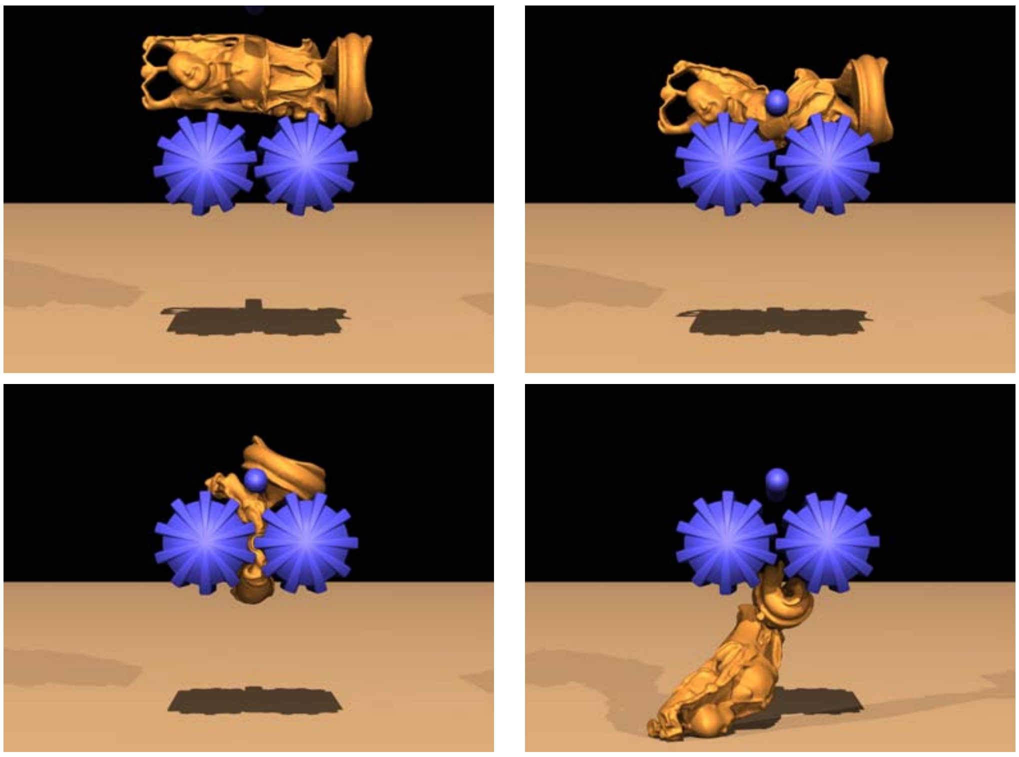


Figure 2.3: A sequence produced by using Invertible FEM [ITF04]

In computer graphics the most popular choice is to use tetrahedral cell to implement FEM; tetrahedrons are always convex and basis functions are derived from barycentric coordinates (see [OH99] and [OBH02] for details regarding tetrahedral FEM).
The current trend is to generalize FEM basis function to works on cells of arbitrary shapes. In [WBG07] using mean-value coordinates, tetrahedral barycentric basis functions are extended to spread into arbitrary convex polyhedra, while in [MKB$^+$08] FEM is generalized to non-convex shapes using harmonic coordinates.

2.1.7 Meshless methods

As we previously introduced in Section 2.1.4, *mesh-free* methods doesn't impose any constraints regarding volume decomposition.
Similarly to explicit FEM, meshless methods are also based on continuum mechanics. The main difference between them is the strategy used to extrapolate a continuous 3D values field from sampled values. Meshless methods requires the knowledge, for each sample i

(also referred to as *Phyxel*), of a set neighbors N_i for which the distance respect to i is lesser than a given support radius r_i. A continuous displacement fields is interpolated from values sampled by neighbors. Since the set of neighbors can be dynamically updated during the simulation, then the shape functions adapts dynamically to the current object's state. Thanks to their flexibility, meshless methods are particularly suitable for the simulation of large deformations and topology changes. They are often used for the simulation of fracturing [PKA$^+$05], melting [MKN$^+$04] of deforming objects. In ([KAG$^+$05]) authors explain how to extend this method to simulate fluids by merging mechanics equation with *Navier-Stokes* equations. The reader may refer to [SL04, BLG94, BKF$^+$96] for an in-depth examination of meshless models.

In ([MKN$^+$04]) authors introduce a mesh-free continuum mechanics-based framework to simulate elastic plastic and melting objects. The approach proposed in that paper is known as *Point-Based Animation* or *PBA*.
While in FEM values are interpolated through basis functions (that are usually designed on element's shape); in PBA values are approximated from neighbors using the first two terms of the Taylor series.
More in detail, a continuous displacement $u(x)$ in the neighborhood of phyxel i can be approximated by the first two terms of the Taylor series:

$$u(x_i + \Delta x) = u(x_i) + \nabla u(x_i)\Delta x + O(\|\Delta x\|^2) \tag{2.29}$$

where $u(x_i)$ refers to the displacement vector of phyxel i. Equation 2.29 should be modified in order to work over a discretized domain. Following that consideration, the displacement value of phyxel j is approximated by Taylor expansion at neighbor i as:

$$\tilde{u}(x_j^i) = u(x_i) + \nabla u(x_i) \tag{2.30}$$

Considering that the displacements approximations $\tilde{u}(x_j^i)$ is estimated at its neighbors, then the introduced error can be estimated as:

$$err_j = \sum_{i \in Neigh_j} (u(x_j) - \tilde{u}(x_j^i))^2 * w_{ji} \tag{2.31}$$

Interpolation weights w_{ji} are defined by continuous shape functions, which are functions of the distance between the phyxel's i and j.

Displacement derivatives on phyxels are approximated trough *Moving Least Squares Method* (see [LS81] for details). For each phyxel x_j, the derivatives of displacement field $\nabla u(x_j) \in I\!R^3$ with respect to each component x, y, z are estimated by minimizing the error err_j (see for the complete formulation [MKN$^+$04]).

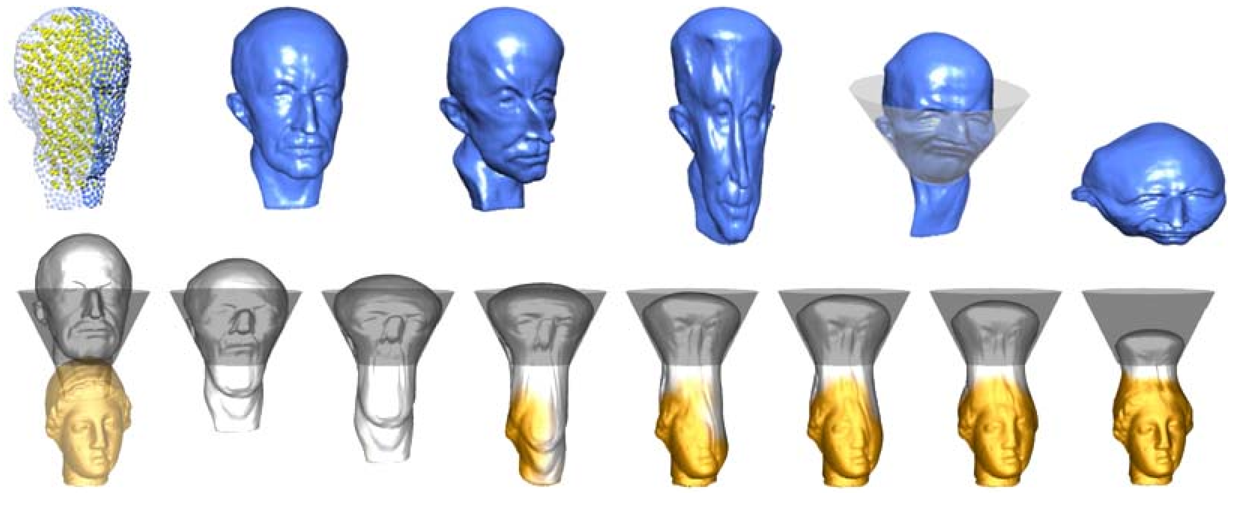


Figure 2.4: Point Based Animation [MKN+04] allows to animate elastic, plastic, melting and solidifying objects

These derivatives are reassembled to obtain the Jacobian of the displacement field $\nabla u(x_j) \in \mathbb{R}^{3\times 3}$.
Once we estimated the Jacobian of displacement field, then strain ϵ is evaluated by *Green strain tensor* (equation 2.3), stress σ by applying Hook's linear material law (equation 2.5), and finally strain energy density $U(x_j)$ is evaluated using equation 2.6:

$$\begin{aligned} \epsilon_j &= \frac{1}{2}(\nabla u(x_j) + \nabla u(x_j)^T + \nabla u(x_j)^T \nabla u(x_j)) \\ \sigma_j &= E\epsilon_j \\ U(x_j) &= \frac{1}{2}(\epsilon_j \cdot \sigma_j) \end{aligned} \tag{2.32}$$

Finally, elastic forces ar computed via the derivation of U_j.
Point-based animation simulates the behavior of a wide range of material properties (See figure 2.4), furthermore they can be adapted to simulate non-linear characteristic like plasticity.

Since PBA does not naturally provide a representation of the object's boundary, a detailed surface representation is used to render the external boundary of the object.
In the case that any topological change occurs during the simulation that surface can be simply dragged along with phyxels displacements. Each vertex, or *surfel*, of the external surface stores a kernel of neighboring phyxels. The position of the surfels is dynamically

updated using a smooth displacement vector field which is invariant under linear transformation. The displacement vector is evaluated by using the displacement's derivative $\nabla u(x_i)$ of neighboring phyxels:

$$u_{surf} = \frac{1}{\sum w_{i,surf}} \sum w_{i,surf} \Big(u(x_i) + \nabla u(x_i)^T (x_i - x_{surf}) \Big) \qquad (2.33)$$

In a more complex scenario, like in fluids or viscoelastic objects simulation, topology can arbitrary changes during the simulation. In that case the external boundary is defined as an implicit surface which can be rendered as a triangle mesh, extracted by *Marching Cubes Algorithm* [LC87, MS94], or using *Point-Based* rendering techniques [PKKG03, ZPvBG02].

2.1.8 Meshless deformations Based on Shape Matching

In [PW89] the authors introduce the use *modal analysis* for computer animation purposes. Modal analysis is used to find a linear approximation of the PDE governing the dynamics of a deforming body (equation 2.10 or its discretization for FEM in Equation 2.1.6). This system of nonlinear equation is turned into a simple set of decoupled linear equations that may be individually solved analytically. The main benefit of modal analysis is the gain in term of efficiency and stability. Nevertheless the realism of the simulation is compromised by the fact that the linearization leads to a first order approximation of the true solution. Some interesting application of modal analysis on FEM are [SHGO02, HSO03].

In [MHTG05] the authors propose a non physically based animation technique based on modal analysis.
The main difference of this model with respect to [PW89] is that the classic continuum-mechanics based formulation of energies is replaced with a purely by geometric formulation. Solving a set of geometric constraints it is possible to find a set of goal positions defined for each particle.
These goal positions are determined via a generalized shape matching of an undeformed rest state with the current deformed state of the point cloud.
Basing on the knowledge of the goal positions, it is possible to define an integration scheme that avoid overshooting (see Section 2.1.3 for details on integration methods), making the whole system unconditionally stable (see figure 2.5).
The performance of this approach in terms memory consumption and computational efficiency together with the unconditional stability of the dynamic simulation make the approach particularly interesting for games.
This method is unconditionally stable even under large deformation, but it does not produce realistic deformation since it is not physically based.

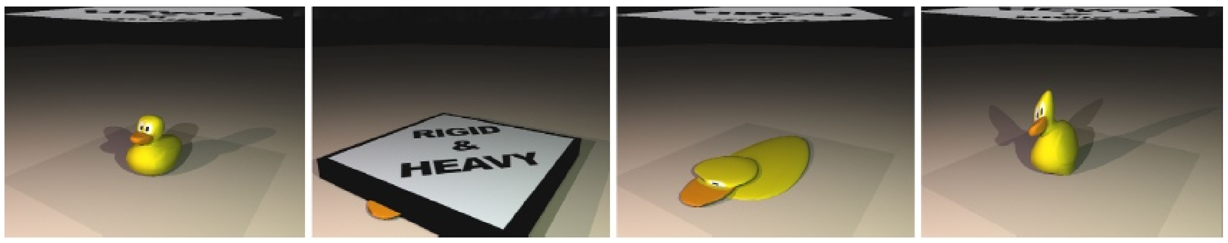


Figure 2.5: Excessive deformation handled by [MHTG05]. The stability of this approach makes it the possible for the object to recover to its original shape.

2.1.9 Mesh-based vs Mesh-free methods

Most of the methods for real time interaction with deformable objects deal with mesh-based models.
Mesh-based methods usually show their limits when large deformations occurs, in those cases an excessive element's change of shape can make the simulation inaccurate or unstable. On the other hand, due to their flexibility, mesh-free methods are suitable for the simulation of a wide range of material, from deformable bodies to viscoelastic fluids.
When a topology change occurs (such as cuts or fractures), mesh-based methods require remeshing operations to preserve a conforming mesh, that operations can introduces some instabilities. Since in mesh-free methods there is no explicit connectivity, topology changes come for free.
Topological changes entail shape functions to code a discontinuity. Unfortunately, while in mesh-based methods discontinuities are implicitly defined by mesh connectivity, mesh-less methods require to update the interpolation weights.

Mesh-Based methods provide a natural representation of the boundary which is described by the external surface of the polyhedral elements composing the volumetric mesh. Since cuts or fractures are usually realized by modifying mesh connectivity, then the external boundary is updated while elements defines new portion of external boundary.
In meshless method, as explained in Section 2.1.7, a detailed surface is dynamically updated according to the deformation field expressed by phyxels. That surface has to be modified to show internal features revealed by cuts or fractures.
In the following sections we present a discussion about cuts and fractures representation for mesh-based and mesh-less models.

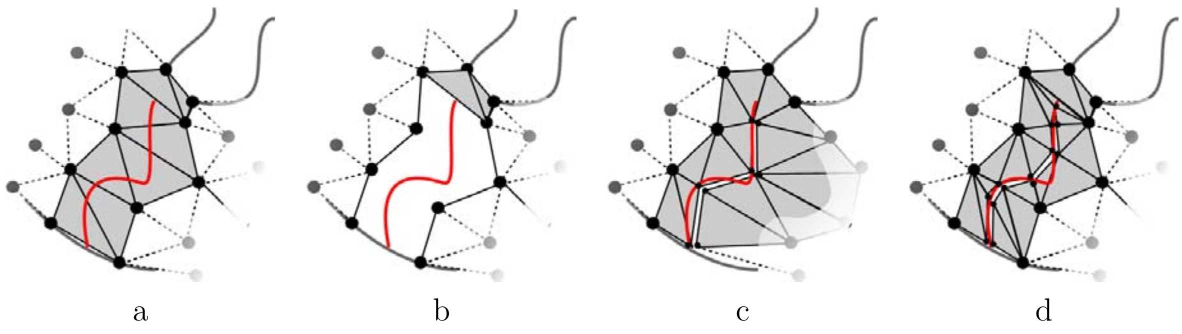


Figure 2.6: Techniques to implement cuts in mesh based models: (a) Portion of a triangulation with a cut surface (in red) (b) Removing elements (c) Snapping vertices on the cut surface (d) Remeshing.

2.2 Encoding discontinuities on Mesh-based methods

In Mesh-based models, the refresh rate of the physical system is linearly related to the number of primitives of the mesh. Moreover, the stability of dynamic solvers is strongly influenced by the quality of the elements composing the mesh. Therefore, mesh-based methods for modeling discontinuities focuses on how to produce an accurate cut representation with the minimum number of primitives, taking into account of their geometric quality.
Because of Kronecker delta property, shape functions becomes zero over element's boundary. Exploiting the advantage given by that property it is possible to create discontinuities inside the mesh by simply changing its connectivity.

Delingette et. al. [DCA99] uses *Tensor Mass Model* to simulate the mechanics of elements involved in the cut, while the rest of the mesh is modeled with a more accurate FEM.
Tensor Mass Model can be considered as a continuum extension of the mass spring system. As in FEM, nodal displacements are continuously interpolated inside each tetrahedron, then forces are evaluated on each tetrahedron independently, as an isolated system. Finally, the force contributions, coming from tetrahedrons, are accumulated on their shared vertices.
Due to the fact that Tensor Mass Model treat each tetrahedron independently (instead of precomputing a global stiffness matrix as in the FEM), it is particularly suitable for topological modifications.
Cuts are realized by removing the tetrahedra touched by the cutting tool. This method avoids the creation of new primitives, unfortunately, because of element removal operations, it produce a poor visual feedback , along with the loss of volume (see Figure 2.6.(b)).

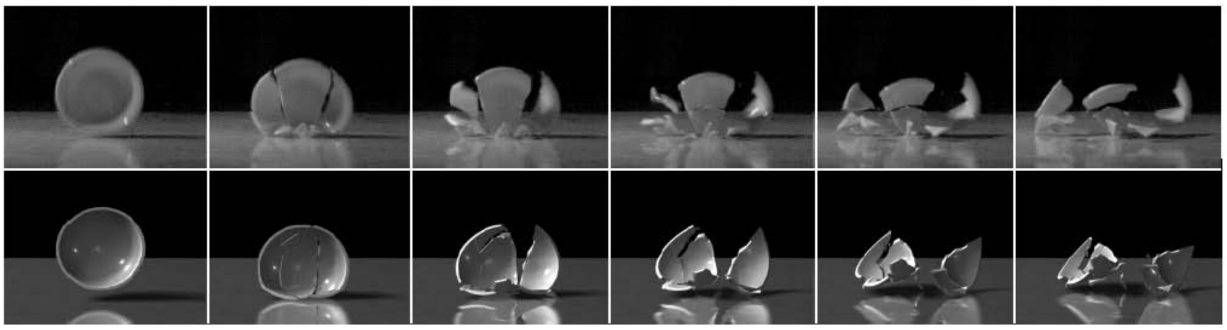


Figure 2.7: Comparison of between a real (Top) and a simulated (Bottom) fracture [OH99]

In the solution proposed by Nienhuys et al. [Nie03], the nodes closer to cut's path are snapped onto the cut surface and duplicated to open the cut (see Figure 2.6.(c)). This method does not create new tetrahedra and can be coupled with a FEM simulation, since the updating of the stiffness matrix can be done on-the-fly.
Several authors use re-meshing operations to adapt mesh tessellation to cut surface. In [THK98, BS01, LD04] the deformable object is modeled using a mesh of triangles. Triangles are split in correspondence of the intersection between the object and the cut surface. Instead of using a mesh of triangles, Bielser et al. [BMG99, BG00] and Ganovelli et al.[GCMS00] represent the deformable object using a tetrahedral mesh. The set of tetrahedrons intersected by the cut surface are substituted with a new set of tetrahedra such that the new object's boundary represent the cut surface. Each of the tetrahedrons involved in the cut is re-meshed by a new set of tetrahedrons, whose connectivity is usually determined by the combination of cut edges (see Figure 2.6.(d)). A pre-computed *Look-Up-Table* is used to store the connectivity deriving from each possible combination of cut edges. Remeshing provides an accurate representation of the cut surface, although it produces mesh fragmentation that can be only partially alleviated by enhancing the re-meshing strategy with on-the-fly edge collapse operations [Gan01]. In [SHGS06], the authors reduce the number of inserted primitives by limiting re-meshing operations to the cases where the two sides of the cut edge are longer than a prefixed threshold; otherwise, they proceed as in [Nie03] (nodes belonging to the cut edge are projected onto the cut surface and duplicated).

O'Brien et al. proposed a solution for modeling brittle and ductile fractures [OH99, OBH02] in off-line simulations (see Figure 2.7). The authors proposed a formulation of Finite Element Method on tetrahedral meshes that has become very popular in the Computer Graphics community. Shape functions are realized trough barycentric coordinates while mass is lumped on nodes. They used continuum mechanics equations to derive the crack surface. In their method re-meshing is used to accurately represent the crack surface, since "approximating it with the existing element boundaries would create undesirable

artifacts" [OH99].
Other solutions decouple the simulation from the representation. In [MBF04], similarly to [BMG99, BG00], each tetrahedron is re-meshed to show the details revealed by cuts or fractures. Such decomposition effects only the rendering, since the original tetrahedron is still used for the physical simulation; only in the case that the cut generates disconnected components, such tetrahedron is duplicated.
While in [MBF04] the tetrahedron can be decomposed at most in 4 components (one for each node), in [SDF07] this idea extended by allowing the tetrahedra to be split any number of time, always considering the intersection of the crack surface with the current decomposition, and not only with the 6 edges of the original tetrahedron. In this manner the objects can be cut in pieces arbitrarily small, at the price of generating polyhedra with any number of faces (which all need to be tested for intersection and collision detection.
A more recent approach [PK08] uses *Discontinuous Galerkin Finite Element Method.* This method allows to define discontinuities inside the shape functions used by classic FEM formulation, and is used to perform robust cutting. However the computational complexity required by this method is too high for real-time simulation.

2.3 Encoding discontinuities on Mesh-free methods

Since mesh-free methods do not satisfy the Kronecker delta property, special care needs to be taken to model discontinuities.
As explained in section 2.1.7, phyxels are interconnected between them by a set of continuous shape functions. Therefore, a discontinuity in mesh-free model, is realized by changing the shape functions such that phyxels on opposite sides of the cut surface become disconnected. This is usually realized by enriching the shape function or, more easily, by reducing the weight function to loosen the mutual influence between phyxels separated by cut's surface.

Contrary to mesh-based methods, mesh-free methods do not naturally provide a representation of the boundary surface. That characteristic has to be taken in account in the case we have to model discontinuities. A mesh of surfels, as explained in section 2.1.7, encodes the external boundary of a mesh-less animated deformable body. This mesh must be dynamically modified in order to visualize new details revealed when cuts or fractures occurs. That modifications involves only the appearance of the deformable object, since the updating of the physical characteristics is a separate task.
Indeed, while in mesh-based method modifications of external surface is direct consequence of modifications of the physical model, in mesh-less methods physical model and rendered surface can be considered as two disjoint entities that have to be treated separately.
In the following paragraphs we'll give an overview of techniques to model discontinuities

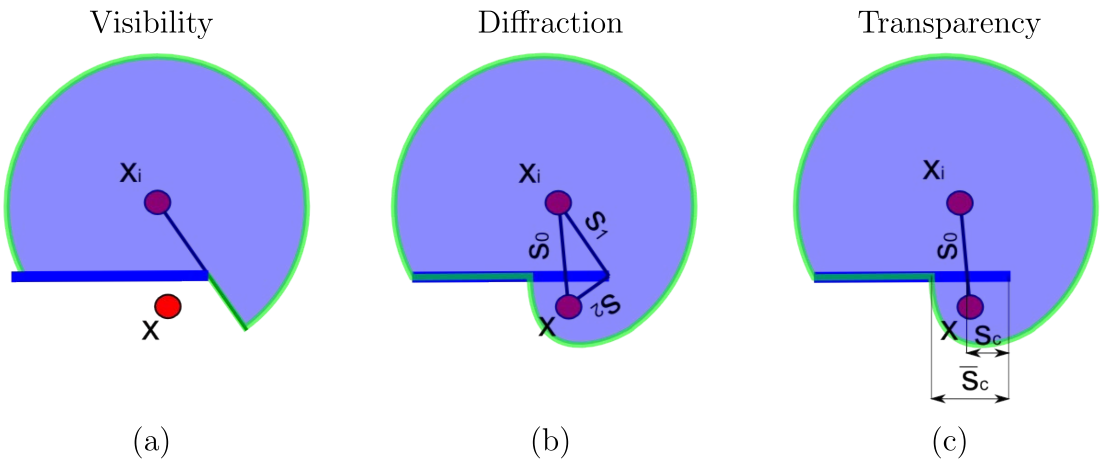


Figure 2.8: (a) The Visibility Criterion [BLG94]: If the segment $\overline{x\ x_i}$ crosses the newly created surface the kernel value is set to zero. (b) The Diffraction Method [BKF+96, OFTB96]: the value of shape function between two points is function of the shortest path not intersecting the discontinuity line. (c)The Transparency Method [OFTB96, BKF+96] uses the the length of $\overline{x\ x_i}$ and the distance between the intersection point and the the crack tip x_c.

on mesh-free models, where the physical model and external surface are treated as two disjoint entities.

2.3.1 Adapting the physical simulation in mesh-free methods

As previously stated (see section 2.3.3), a cut introduces a discontinuity in the deformation function. In mesh-less methods this discontinuity can be encoded by altering the weights of the shape functions, which can be done in various ways. Those methods has to satisfy some requirements.
If a discontinuity separate the domain in two or more disjoint partitions, then such partitions must be two "physically independent" objects, i.e. weights bindings phyxels coming from different partitions are equal to zero.
During the simulation, while a discontinuity progressively appear in the domain, weights are progressively modified. Moreover, while those weights are modified, the framework continue the simulation loop. In order to keep the whole simulation stable, weights must be updated as smoothly as possible. An abrupt change of weight reflects on forces evaluation, possibly making the system diverge.

One straightforward method to encode discontinuities is the *Visibility Criterion* proposed by Belytschko et al. in [BLG94]. It consists in zeroing the value of the shape function in those points from which the phyxel i is not visible, i.e. if the segment $\overline{x\ x_i}$ crosses the newly created surface (see figure 2.8.a). Although very simple to implement, this method introduces also an undesired discontinuity at the horizon line (see Figure 2.9.a) that affects convergence and stability.

In the *Diffraction Method* [BKF+96, OFTB96] the value of shape function between two points is function of the shortest path not intersecting the discontinuity line. If x_c identifies the crack tip through which passes the shortest path not intersecting crack surface, then the distance d_l between a point x with respect to a phyxel x_i is described by the following equation:

$$d_l = \left\{ \frac{s_1 + s_2}{s_0} \right\}^{\gamma} s_0 \tag{2.34}$$

Where, γ is usually equal to 1 or 2 [FTP03, DMTB96], and $s_0 = \|x_i - x\|$, $s_1 = \|x_i - x_c\|$ and $s_1 = \|x - x_c\|$, see Figure 2.8.b. Although the diffraction method produce a smooth weight decrease in most of the cases, one potential divergence situation still exists. In fact, when the discontinuity obscures the visibility completely, weight suddenly collapses to zero, producing a discontinuity between two consecutive time steps (see Figure 2.9.b. for details). Furthermore, due to complexity of the distance approximation, this method is, in most the cases, not suitable for real-time simulations.

An approximated but interactive version of this approach is also used by Steinemann et al. in [SOG06]. In a preprocessing phase, they build a connectivity graph on the phyxels by adding an edge for each couple of neighboring phyxels, i.e. phyxels connected with a weight value greater than zero. The distance between two phyxels is always taken as the shortest path in the connectivity graph. When a discontinuity surface is defined, the arcs intersecting the discontinuity surface are removed and the shortest paths between the phyxels in the neighborhood are recomputed.

The *Transparency Method* [OFTB96, BKF+96] uses the the length of $\overline{x\ x_i}$ and the distance between the intersection point and the crack tip x_c (see figure 2.8.c):

$$d_l = s0 + \rho_l \left\{ \frac{s_c}{\bar{s}_c} \right\}^{\gamma}, \gamma \geq 2 \tag{2.35}$$

This method, similarly to the Diffraction Method, can produce a temporal discontinuity (see Figure 2.8.a).

Transparency method is successfully used in [PKA+05] for non-interactive simulation of fracturing solids.

2.3.2 Adapting the surface in mesh-free methods

In the solution proposed by Pauly et al. [PKA+05] the surface of the deformable object is dynamically sampled with surfels rendered as oriented elliptical splats.

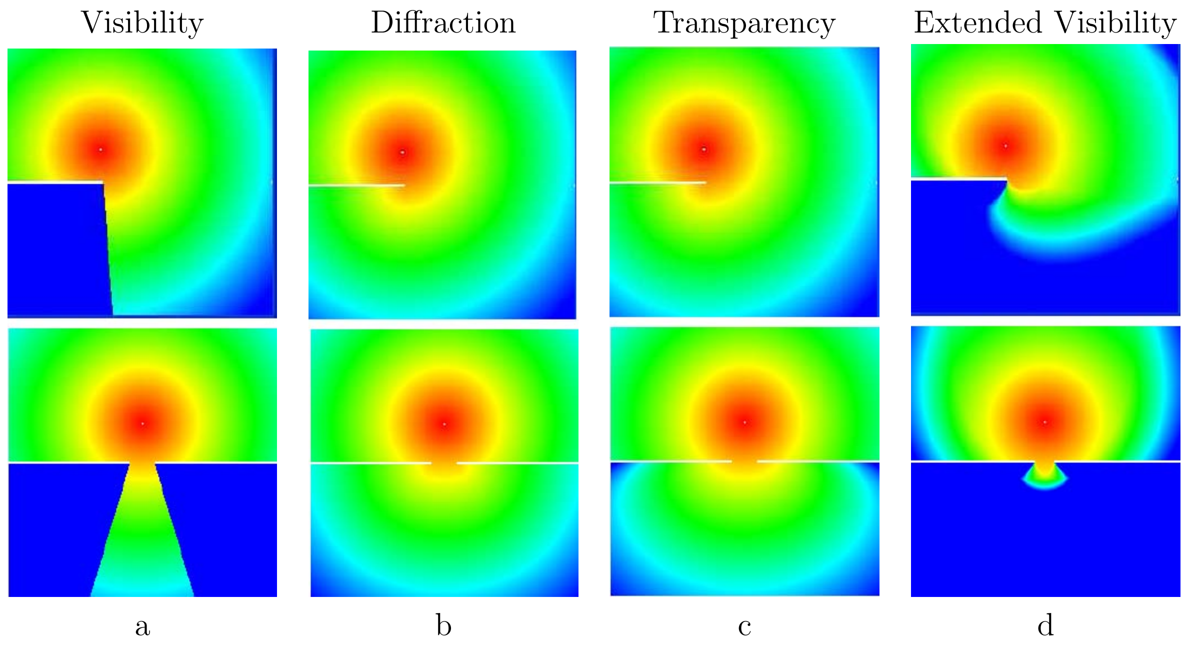


Figure 2.9: The images show how the weight function is modified by the introduction of a cut surface (white line) by using: the visibility criterion, the diffraction methods, the transparency method and the Extended Visibility criterion.

Cuts or fractures are often characterized by sharp features, in order to render that features correctly, surfels overlapping a crack are clipped against a plane lying on the other side of the crack [PKKG03].
In their model a crack is codified by a sequence of phyxels (called *crack nodes*) which identifies the crack propagation front. For cracks starting from the surface (e.g. when a cut is being made), the first and last node of the sequence lie on the surface, while for cracks generating inside the volume the front is circular. Every time a new crack node is added to the sequence, i.e. every time the front propagates, new surfels are added to represent the two new pieces of surface.
This technique avoids the classical problems of the mesh-based methods, i.e. fragmentation and degeneracies due to re-meshing. On the other hand the crack fronts can split and merge and these events need to be handled explicitly to maintain the topology of the crack front(s) consistent.
An alternative approach has been proposed by Steinemann et al. [SOG06], where the surface is represented by a triangle mesh, which vertex moving according to equation 2.33 basing on a set of neighboring phyxels. When a cutting tool penetrates the object, the cut surface is triangulated and used to update the current object's surface.
As for the previous solution, the branching and merging of crack fronts has to be handled

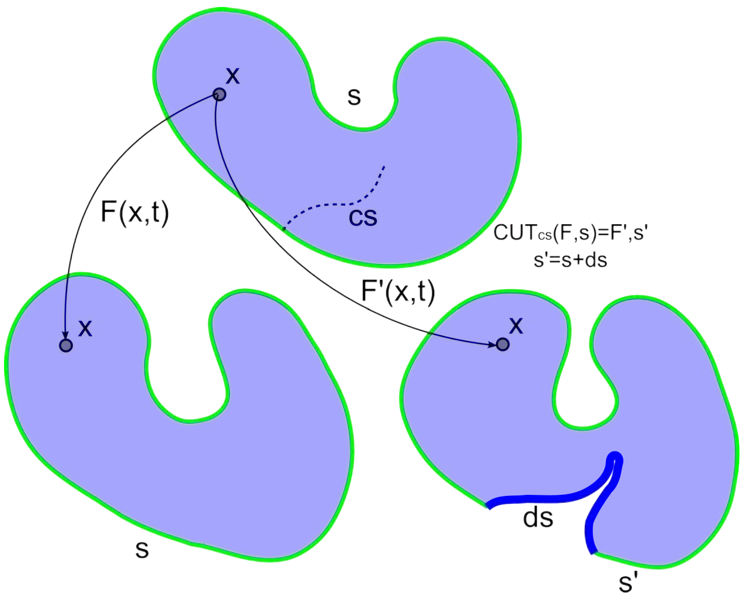


Figure 2.10: An abstract representation of a discontinuity

explicitly. The set of neighbors of surfels is updated by extending the connectivity graph to the surface's vertices.
Compared to the point sampled method described above, the use of a triangle mesh give advantages in terms of rendering speed. Unfortunately, if multiple cuts are executed in the same region triangle fragmentation may occur, causing degradation in terms in performance.

2.3.3 An abstract representation of a Discontinuity

Many solutions have been proposed to the problem of virtual fracturing/cutting of deformable objects, using different methods for physical simulation and rendering.

For the sake of generality, we introduce the problem in abstract terms. We characterize a deformable object as a time dependent function $F : \Omega \times Time \rightarrow R^3$ and a *description* of its surface s, as shown in Figure 2.10. The function F gives, for each point of the object domain at rest shape Ω, its position at a given time t. F is usually at least C_0 continuous except on the boundary of Ω. A *cut* is a function that modifies F and s on the base of a *cut surface* inside the volume Ω: $Cut_{cs}(F, s) = (F', s')$.
In this terms, the problem of virtual cutting can be expressed as the problem of defining the function Cut.

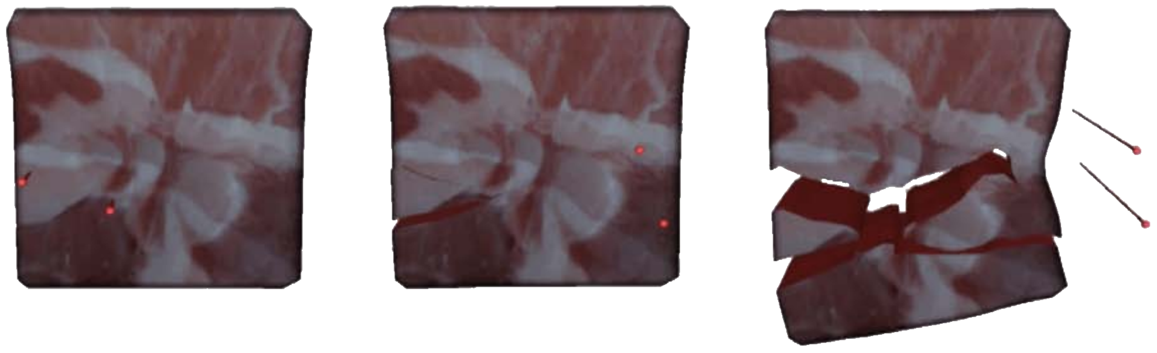


Figure 2.11: Example of cut with multiple advancing fronts using the Splitting Cubes, frames captured during an interactive simulation. The cut path is defined by the counter clock-wise motion of the two blades.

2.4 The Splitting Cubes Algorithm

The *Splitting Cubes* algorithm is a technique for providing a tessellation of an evolving surface embedded in a deforming space.
In this specific case the surface evolves when a cut exposes new parts of the object boundary, and when the space deforms.
The key idea of the Splitting Cubes algorithm is to embed the object in a regular 3D grid whose nodes are moved according to the deformation function F. The deformation function F is continuously defined over the entire domain by interpolation of nodal values. Since cells are cubic, the deformation function is interpolated inside each cube from its 8 nodes. This scheme allows us to implement discontinuities of the deformation inside a cell, by varying the interpolation values of the nodes, depending on which edges are cut.

We introduce the details with a practical 2D example.
Figure 2.12.a shows a tessellated surface crossing a few cells of the regular grid (in 2D). The cyan arrows leaving from the vertices of the tessellation show the dependencies of that vertex on the cell nodes, i.e. from which nodes we compute its position. The red curve shows the intersection of a cutting tool path with one edge of the embedding grid.
The tessellated surface is defined cell by cell on the base of the configuration of cut edges. Positions of vertices is derived by position and normal of the intersection points.
In this sense, the main philosophy of our method is similar to what is done in the vari-

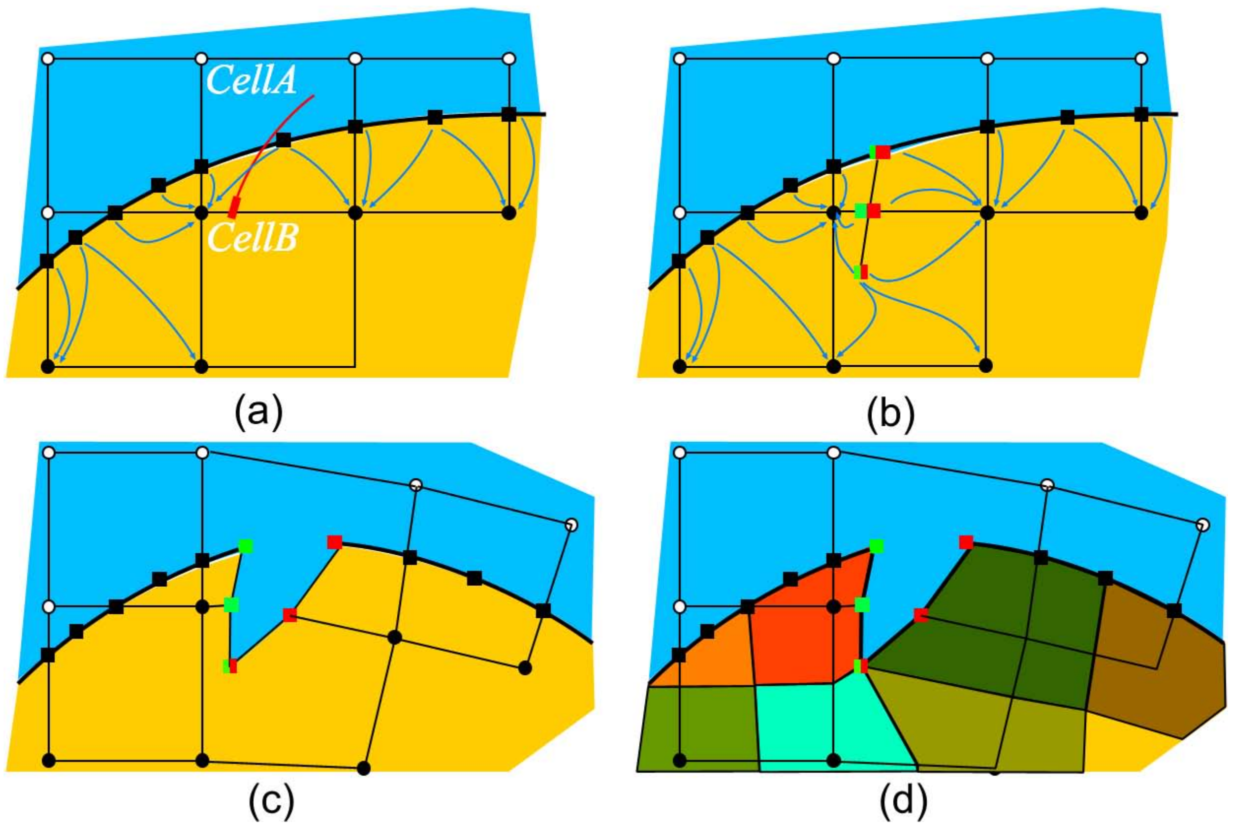


Figure 2.12: The Splitting Cubes idea. (a) The object is embedded in a regular grid, the vertices of the tessellated surface depends on the grid nodes. (b) A cut surface crosses an edge and changes the configuration of cells CellA and CellB. (c) The new surface in the deformed space. (d) Dual interpretation: each node of the grid models a portion of material.

ous extensions of the Marching Cubes approach [LC87, WMW86] that exploit hermitian data [KBSS01, JLSW02]. However, the cut shown in Figure 2.12.b would generate an invalid configuration for the standard Look Up Table (LUT) of the Marching Cubes both for cell $CellA$ and cell $CellB$ (one edge of the cell is intersected). On the contrary, the Splitting Cubes algorithm includes these configurations. The reason relies in the nature of the cut surface.

In the literature the cut surface is regarded as the surface swept by the cutting tool, which is usually identified with a segment (or a set of segments in order to approximate the a curved blade). We use a more topological definition to explain our technique: the cut surface is the boundary of a protrusion of the *empty space* surrounding the object. We define the empty space as the absolute complement of the object's volume domain M.

When a cutting tool penetrates into the object, it actually extends the empty space into the object. Indeed, the cut surface represents a portion of the boundary of the empty space surrounding the object.
When the object is in its rest position, the empty space always degenerates locally to a surface where a cut occurs (Figure 2.12.(b)). However, that is not always verified when the object is deformed (Figure 2.12.(c)).
Although at rest shape the volume of this protrusion is zero, its boundary (the blue curve in Figure 2.10) is topologically well defined and can be tessellated, which is exactly what the Splitting Cube does by sampling the cut surface on the cells edges and using these points to define a tessellation.

The tessellation is defined individually cell by cell. Figure 2.12.b shows the tessellation for the configurations of cells $CellA$ and $CellB$. We can see that the cut generates two vertices on the edge and one inside the cell $CellB$. Furthermore, the dependencies of the vertices inside the two cells have been changed to reflect the cut.
The Splitting Cubes LUT(Look-Up-Table) contains all the 2^{12} possible configurations determined by cuts on cell edges, specifying for each configuration the relative triangulation along with the dependencies of the vertices from the cell nodes. Figure 2.12.(d) shows a dual interpretation of the Splitting Cubes.
Every node represents an amount of material. The material between two adjacent nodes is continuously defined if and only if the corresponding edge is not cut. Similarly , when a cell has no cut edge, then that cell is considered as an interval of continuously connected material. Since cell shares nodes, then the whole cubic meshes defines a continuous piecewise interpolation function. Following that definition, the splitting cubes can be seen as a cuboid version of the Virtual Node algorithm [MBF04].

It is important to remark the splitting cubes grid is an instrument to represent and interpolate a generic continuously defined function. Furthermore, the splitting cubes grid give us the possibility to modify that function to model discontinuities on a resolution that is limited by the edge length. This limitations is compensated by the robustness and the efficiency achieved by the method.

2.4.1 Splitting Cubes: a 2D illustrative example

Figure 2.13 shows the six configurations for a cell in the 2D case. For each configuration a tessellation of surface inside the cell is given. The cyan arrows leaving the vertices and pointing to the cell nodes show the dependencies.
We start considering the 2D case, where the cells are quadrilaterals (besides, these are also the configurations for the faces of a 3D cell). The first column shows the configuration

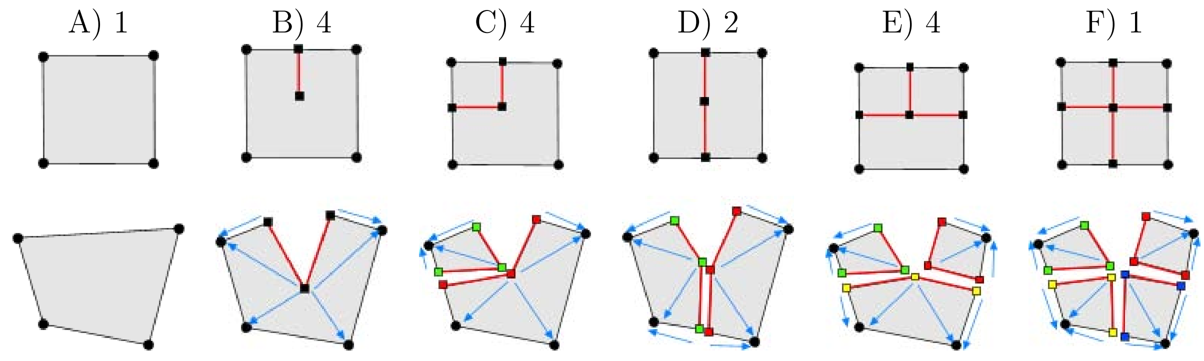


Figure 2.13: The six configurations for a face of the splitting cube (i.e. the 2D case). We show: the configurations at rest shape (first row); the configurations at a given deformed shape (second row); the need for the internal face vertex to avoid volume loss (right most column).

in the parametric domain while the second shows a possible deformation of the cell with the vertex-node dependencies. The number next to the case letter indicates how many equivalent configurations are obtained by rotation or mirroring.
The configurations **B**-**F** are tessellations of the cut surface as derived by the cut edges. Note that each cut edge will always create two vertices, called *edge-vertices* which always depend on one of the two extremes of the edge. This choice reflects the discontinuity of the deformation function F along the edge. The two edge-vertices will be free to move apart in the deformed space.
Similarly, the vertex in the middle of the face, called *face-vertex*, is replicated for each connected component and depends on the nodes constituting its connected component. Face-vertices become necessary if we want to preserve the amount of material, otherwise a cut in the parametric space will reduce the total volume.
We define as *connected component* a portion of the cell where every pair of points can be connected by a curve without intersecting the cut surface. In the configurations **A** and **B** of Figure 2.13, the cell includes only one connected component, while in **C** and **D** there are 2 connected components, finally configurations **E** and **F** shown respectively 3 and 4 connected components. Obviously, extending that example to a 3D cubic cell, there should be up to 8 different connected component.

2.4.2 Splitting Cubes: from 2D to 3D

The configuration for the 3D cell are derived by extending the 2D case (as described in Section 2.4.1). Let us consider a cell with only one edge cut. In this case, extending the corresponding 2D configuration (which is the case **B** of Figure 2.13), the resulting surface

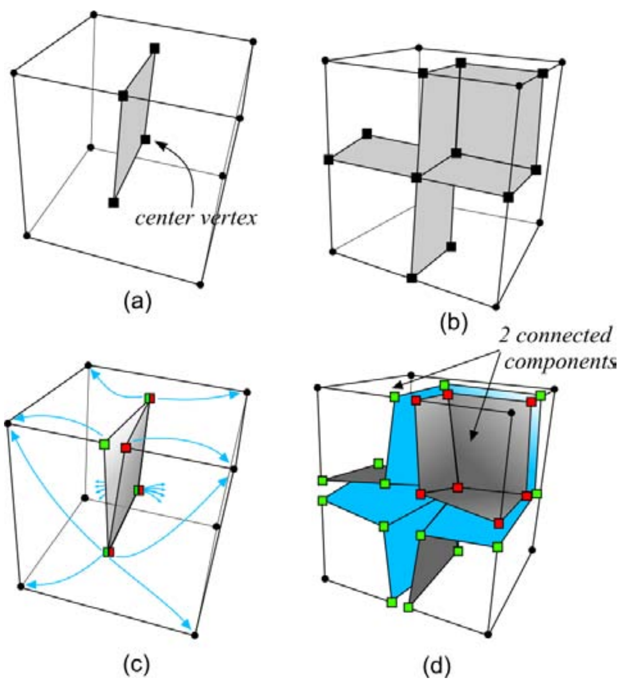


Figure 2.14: Two examples of cuts. (a-b): The quads created by cutting 1 and 5 edges. (c-d): the resulting tessellation.

will be represented as shown in Figure 2.14.(a). More in particular, we build a quad with two edge-vertex, two face vertices and a *center vertex* placed inside the cube.
That configuration is replicated for each possible edge become cut, duplicating appropriately the vertices and assigning the dependencies accordingly to the different connected components. Figure 2.14.(b) and 2.14.(d) shows a case with 5 edges cut generating two connected components, with the resulting tessellation, while Figure 2.14.(d) illustrates vertices dependencies.
The tessellation and the dependencies are computed once for all and stored in a LUT composed of 2^{12} entries. We will show details about the LUT generation in Section 2.5.

2.4.3 Position of the vertices

While the connectivity of the vertices added by a cut is stored in the LUT, their position has to be found on-the-fly. For the edge vertex the choice is trivially the cutting point along the edge.
The position of face points is less obvious to find. Our goal is to provide a tessellation that mimics the cut surface inside the cell, so we cannot use simplistic solutions as the center of the face or the average of the edge vertices.

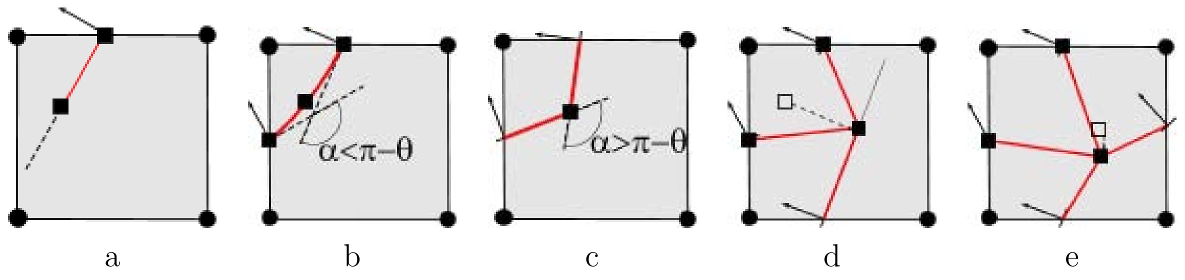


Figure 2.15: The figure shows how the vertices of the tessellation are derived from the intersection o the cut surface with the edges of the cell and the normal to the cut surface at the crossing point. It is important to notice that the normal represented for each vertices on examples (d) and (e) were defined during the cut, the system uses such information to place the central vertex in an unique position at undeformed space. It turns out that those normals may be slightly different from the ones defined by the geometry.

Instead we take into account the normal of the cut surface at the edge-vertices, that we obtain from the movement of the tool. This normal and the relative edge-vertex define a plane the we call *Cut Plane*. In other terms, a cut plane is the approximation of the cut surface around a edge vertex.
To explain the different cut combination together with the position of surface's vertices we refer to the schema shown by Figure2.15.

For cases b and c, we use the same approach described in [KBSS01]: if the angle α formed by the two cut planes is close to π, i.e. $|\pi - \alpha| < \theta, 0 < \theta < \pi$, we assume the two edges have been cut with a smooth movement. In that case we place the face vertex in the middle of the *Bezier curve* defined by the edge vertices and the intersection of their associated cut planes (see Figure 2.15.(b)). Otherwise we place the vertex exactly at such intersection point, showing a sharp feature (see Figure 2.15.(c)).
The cases d and e would clearly require a tessellation with more detail for representing the cut surface exactly.
We use a strategy that leads to little or no visible artifact with the assumption that in a simulation step, no more than two edges are cut on the same cell. In the case where this conditions is not satisfied the set of cuts are re-distributed over different time-steps. Using this strategy we smoothly proceed from configuration a to d.
Any further cut will find the face vertex already placed, so we redefine (move) the face vertex by projecting it into the new cut plane, as shown in Figure 2.15.(d) and Figure 2.15.(e).
Similar considerations hold for the position of the central vertex. The first time the cell is cut, if the cut does not split the cell in two parts, the central vertex is positioned in the average position among the face vertices, otherwise it is placed so that it minimizes the sum of the squared distances from all the planes by using a quadric metric as in [GH97].

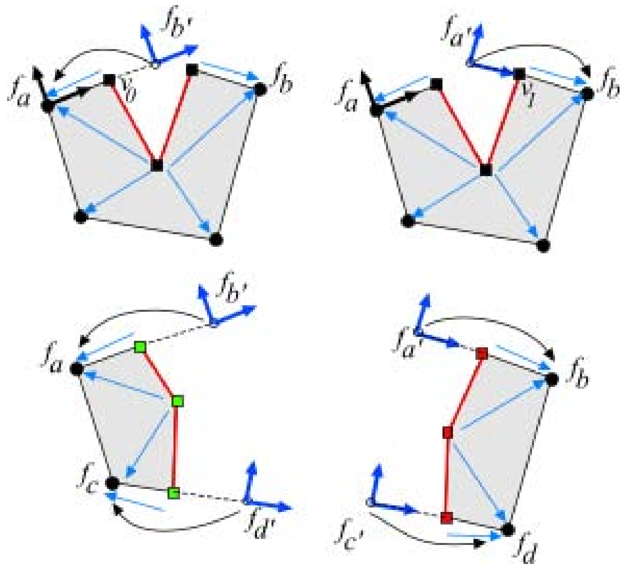


Figure 2.16: Two types of cut (case B and case D of figure 2.13). The frames rendered in blue are virtual and depend on the frame pointed by the arrow.

Again, for any further cut we project the position onto the new cut plane.

2.4.4 Interpolation inside a cell

As previously stated the space inside the cell is deformed accordingly to the cell nodes. For each node of the grid, we attach a reference frame, that is updated for each time step of the simulation. The reference frame of the node i at time step t is defined by a pair $f_i(t) = (A_i(t), O_i(t))$ where $A_i(t) \in I\!R^{3x3}$ defines the three axis and $O_i(t)$ the origin. The reference frames are initialized by the three orthogonal directions of world reference frame and the undeformed position of the node. We indicate as $f_i^0 = (A_i^0, O_i^0)$ the reference frame value.
Given the deformation function $F(t)$ at time t and its Jacobian $J_i(t) \in I\!R^{3x3}$ evaluated at node i, then the reference frame $f_i(t)$ is updated as follows:

$$f_i(t) = ((J_i(t)^{-1})^T(A_i^0), F(O_i^0)) \tag{2.36}$$

Given that we updated the reference frame $f_i(t)$ of all nodes, then the position in the deformed space of a generic point p is found as:

$$\sum_{j \in cell} a_j p_j f_{j_t} \tag{2.37}$$

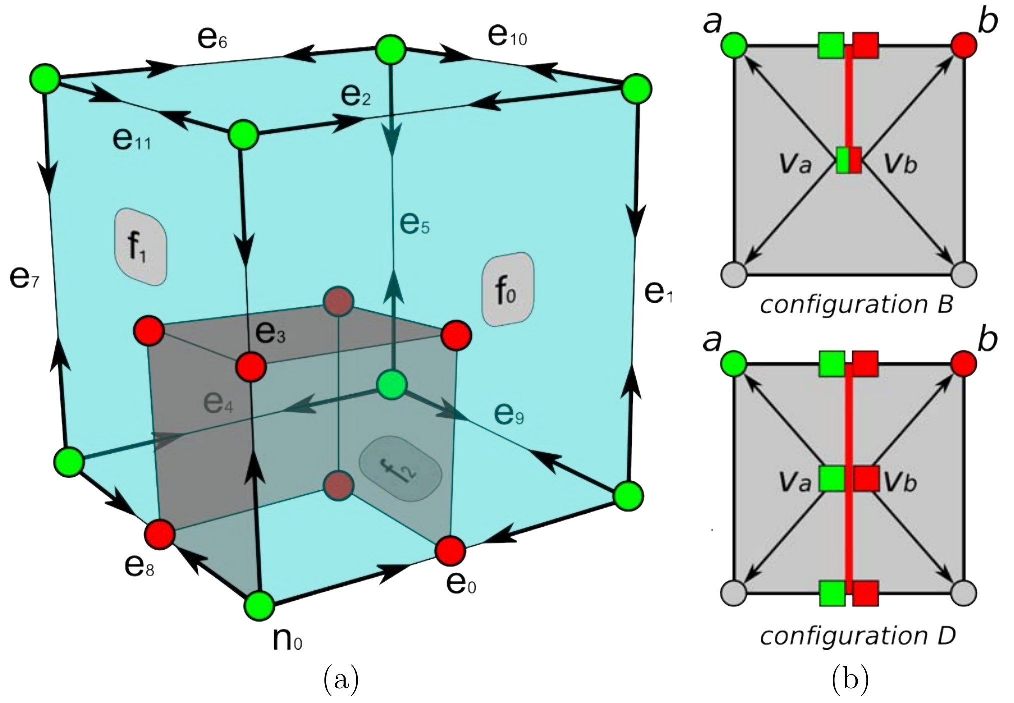


Figure 2.17: (a) tessellation for the cell node 0. (b) two configurations that show why dependency cannot be computed locally to the nodes.

where p_j is the projection of p on the frame f_i and a_j are the scalar coefficients of the trilinear interpolation. This interpolation scheme uses all of the 8 nodes of a cell, but tessellation vertices must depend only on a subset of them.
In order to illustrate this problem we use the example shown in Figure 2.16. In that case the position of the edge vertex v_0 is interpolated from f_a and f_b (since the coefficient of the trilinear interpolation are 0 for the other 6 frames). However, since we want to represent the discontinuity of F along the edge, intuitively v_0 should only depend on f_a.
To solve this problem we proceed similarly to [MBF04]. We interpolate v_0 using a copy of $f_b(t)$, indicated as $f_{b'}(t)$, which is called *virtual reference frame*. The center of the virtual reference frame b' is computed by the function $b' = F'(b) = F(a) + (b - a)\ J_{F_t}(a)$, i.e. the deformation function in b as approximated by its value in a by Taylor series, the same transformation is applied to obtain the directions composing the virtual reference frame.

2.5 Construction of the Look-Up-Table.

As explained in Section 2.4, each one of the 2^{12} possible configurations of the cut edges of a cell corresponds to a row of the LUT, which encodes two data: the tessellation representing the piece of surface internal to the cell and, for each vertex of such tessellation, its dependency from the cell nodes.

Fortunately we can take advantage from noticing that the tessellation of a cell can be expressed as the union of 8 tessellations, one for each cell node, and that each tessellation only depends on the cell edges entering the node. This can be better understood recalling the dual interpretation of the Splitting Cubes shown in Figure 2.12.(d) where the material (and therefore its boundary and the tessellation of its boundary) is associated to the grid nodes.
We define the tessellation for a single node and then obtain the tessellation for the cell by translating the vertices appropriately.
We show a practical example on Figure 2.17.a. Let us consider the tessellation for the node n_0. For each cut edge leaving this node ($e0$, $e3$ and $e8$ respectively), we build a quad which has one point on the edge, one point on each of the two faces sharing the edge and the point at the cube center. We obtain the tessellations for the other 7 nodes simply rototranslating the frames and reapplying this scheme.
You may note that we will create duplicate vertices. As explained below, some vertices actually need to be duplicated and some other do not; to know it which ones have to be duplicated we must compute their dependency.

Unlike the tessellation, the dependencies of the vertices cannot be treated locally to a cell node. Figure 2.17.b shows two configurations for a face. The vertices v_a and v_b are created both when tessellating node a and node b. In the configuration **B** the face is partially cut and v_a and v_b depend on the same set of grid nodes and therefore will always occupy the same spatial location. Conversely, in the configuration **D** v_a and v_b depends on two different set of nodes and therefore they will move apart.
As explained in Section 2.4, each vertex depends on the cell nodes of the same connected component. Therefore we applied a simple principle: a vertex depends on all the nodes of the same cell that are *reachable* through the material. We consider each grid node reachable by the vertices of its associated tessellation and two nodes of a cell as mutually reachable iff they are connected by a path of uncut edges.
Furthermore, since we need continuity of the deformation function across the cell boundary, the interpolation weights of the cell nodes for a edge-vertex will be 0 except for the 2 nodes of the edge and the interpolation weights of the cell nodes for a face-vertex will be 0 except for the 4 nodes of the face. Therefore we only need to find reachable nodes of the edge for the case of the edge-vertex (which is only one node) and reachable nodes of the face for the case face-vertex, since the weight for the other nodes would be zero even if they were reachable.
Considering the tessellation, when two vertices have the same dependencies we do not create two vertex instances, but we statically unify them in the tessellation contained in the LUT so no useless duplicate vertex are created.

The LUT and the necessary code to read it can be found in the IDOLib library [Vis05a].

2.6 Physical response to cutting

In the previous section we introduced the Splitting Cubes algorithm, which enables to introduce cuts and to setup the corresponding discontinuity in the deformation function F *inside* each single cell, **without making any assumption on the physical model**. However, the deformation function must be changed to reflect the discontinuity introduced. This sections shows how to modify the deformation function to show discontinuities using point-based animation.

2.6.1 Nodes-Phyxels bounds

The deformation function at a grid node depends on the set of phyxels within a certain support radius r that are visible, i.e. the segment connecting the node to the phyxel does not intersect the boundary surface. This set is usually referred as *kernel* of the node.
The kernel of a node is kept up to date when new pieces of surface are added. Updating a kernel means to check which of the phyxels in its kernel are still visible. To perform this check it is sufficient to test only for the nodes within a radius r from the cell where the new piece of surface was created. Since the grid is regular it is possible to get this set of node in a constant time.

2.6.2 Phyxels-Phyxels bounds: the Extended Visibility Method

In Section 2.3.1 we reviewed the methods to update inter-phyxels relations after a cut. Here we propose to extend the visibility criterion by substituting the segment of sight between two phyxels with a pair of cones as shown in Figure 2.18.
Consider the common base of the two cones, that we call *visibility disk*. From each point on this disk we trace two rays, directed to each phyxel and say that this single point is *occluded* if at least one of these two rays intersects the cut surface.
Then we define the weight function as:

$$w'(p_i, p_j) = w(p_i, p_j)\left(1 - \frac{1}{DiskArea}\int_{p\in Disk} IsOccluded(p)dp\right) \tag{2.38}$$

In this manner we replaced the binary value of the previous visibility criterion with a criterion which returns a scalar value in the range $[0, 1]$ that we use to weight the inter phyxel

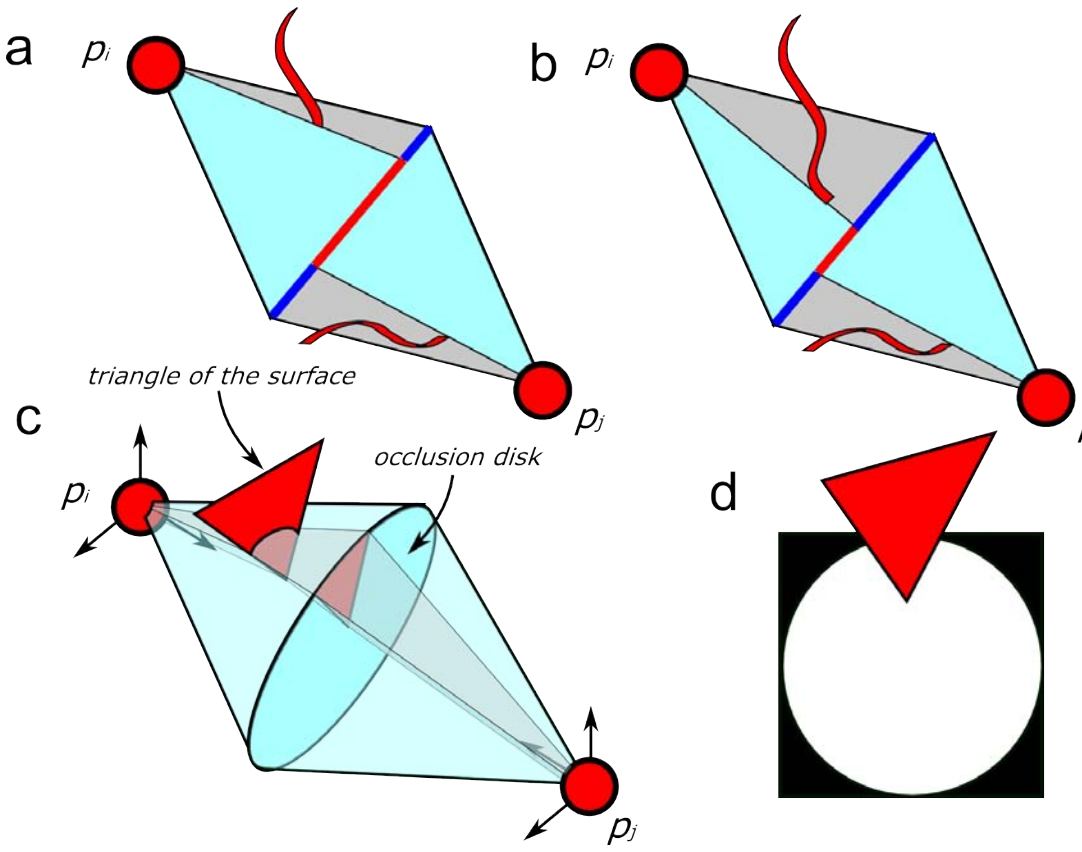


Figure 2.18: (a-b) The cut (represented with two red curved stripes in 2D) surface partially occludes the visibility disk. The percentage of weight remaining is represented with a red line. (c) Hardware implementation of the Extended Visibility criterion. (d) A triangle of the cut surface as seen by phyxel p_i.

bound.
Figure 2.9 shows the value of the weight (in a color ramp from red= 1 to blue= 0) around a phyxel in the proximity of a cut surface using the visibility criterion, the diffraction method, the transparency method and finally our *Extended Visibility criterion*.

The first row shows the case were the cut surface is made of a single connected component. In this case, all the methods correctly implements the discontinuity with respect to the cut surface. Nevertheless the visibility criterion introduces an undesired discontinuity in the weight function on the horizon line, while in the diffraction and the transparency methods the weight function decays smoother.
The Extended Visibility criterion produces a reasonable compromise: the weight function does not exhibit unwanted discontinuities as the visibility criterion does, but decays around the tip point of the cut surface faster than diffraction and transparency methods.

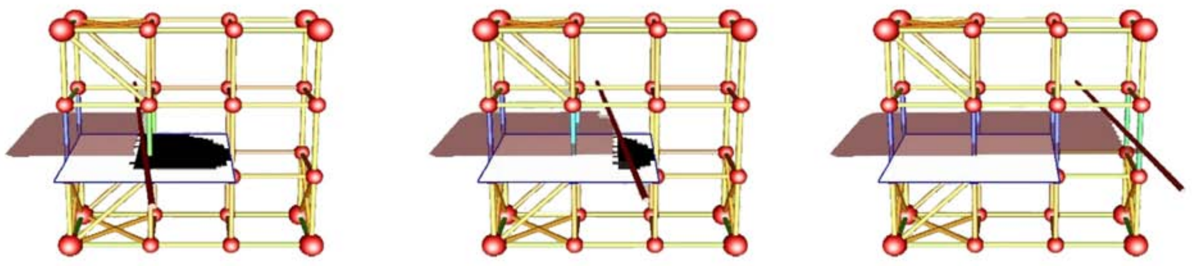


Figure 2.19: A sequence of steps during a cut: the kernel value smoothly decreases as the visibility disc is 'obscured'.

The second row shows the case where the cut surface is defined by two connected components, e.g. in time immediately before two crack fronts merge.
Using either diffraction or transparency methods, the weight functions change discontinuously with respect to the growth of the cut surface at the point when the two cut surface merge, because both methods depend on the tip point. On the contrary, using the Extended Visibility criterion the weight function smoothly decays to 0 in the region under the cut surface. Note that the merging of crack fronts is a common event if we propagate cracks or if we perform a cut with scissors.

2.6.3 Implementing the Extended Visibility Method.

The choice of using cones may seems quite arbitrary, in reality it lead to an easy GPU implementation of the Extended Visibility criterion.
Let p_i and p_j be two *connected* phyxels, i.e. for which $w(p_i, p_j) > 0$, and cs a cut surface potentially occluding the disk between p_i and p_j (see sequence shown by Figure 2.19).
Consider the smallest square enclosing the occlusion disk. We associate a small single-channel texture to this square to store which samples of the disk have been occluded. Initially, as shown in Figure 2.18, it shows a circle representing the intact visibility disk. This texture is permanently associated with the couple of phyxels and updated every time a new piece of surface (e.g. produced by the Splitting Cubes algorithm) could potentially occlude their visibility disk. Therefore we use $\#phyxels \cdot k$ small textures, where k is the average number of neighbors of a phyxels.
To update the visibility disk we render cs twice: once from p_i towards p_j and once from p_j towards p_i, always setting the far plane of the projection on the mid point of the segment p_i and p_j. To perform this render we set the size of the *viewports* equal to the size of the texture's square, so that each sample in the disk project on the same pixel for the two renderings.

Using a fragment shader, we discard those fragments projecting onto a pixel already written, so that all the fragments that are written into the frame buffer correspond to newly rasterized fragments, i.e. newly occluded samples. By using the hardware occlusion query, we count these fragments and finally update the weight as follows:

$$w'(p_i, p_j) = w(p_i, p_j)\frac{n - \#occluded}{n} \tag{2.39}$$

which is the discrete version of equation 2.38 where n is the number of texels composing the circle of the visibility disk.

2.7 Initial Setup

Initially we load a watertight mesh creating the cubes that are intersected by such mesh. The initial surface is obtained by considering the original mesh as a big cut surface.
Once the cut is done, we obtain two surfaces: one bounding the object and one bounding the empty space around the object, that we simply throw away.
Phyxels are uniformly sampled in the volume in the setup of our experiments, however our method doesn't impose any restriction about phyxels distribution. A good choice could be to sample phyxels following the method proposed by Pauly et al. in [PKA+05].
Finally, two parameters must be tuned in Extended Visibility method:

Texture Size A large texture requires a longer rendering time but provides a smoother change of the weight and *vice versa.*

Radius of Visibility Disks A bigger radius provides a smooth weight change. Nevertheless since the cells generating potential occlusive surface are increased, then the total number of rendering operations increases, effecting the performance of the simulation.

We found experimentally that a good tradeoff between smoothness and performance is achieved by assigning a radius of the visibility disc equal to half of the distance between the two phyxels and to use a 32×32 buffer for rendering (which means 16 pixel radii for the visibility disks).

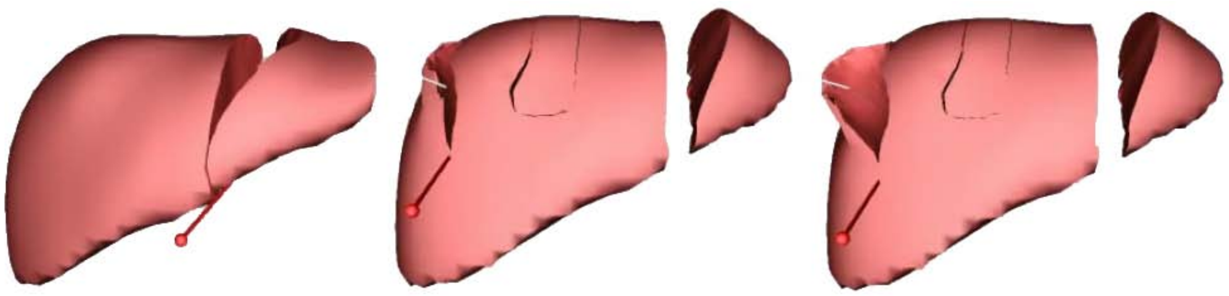


Figure 2.20: Cutting of a deformable model of the liver

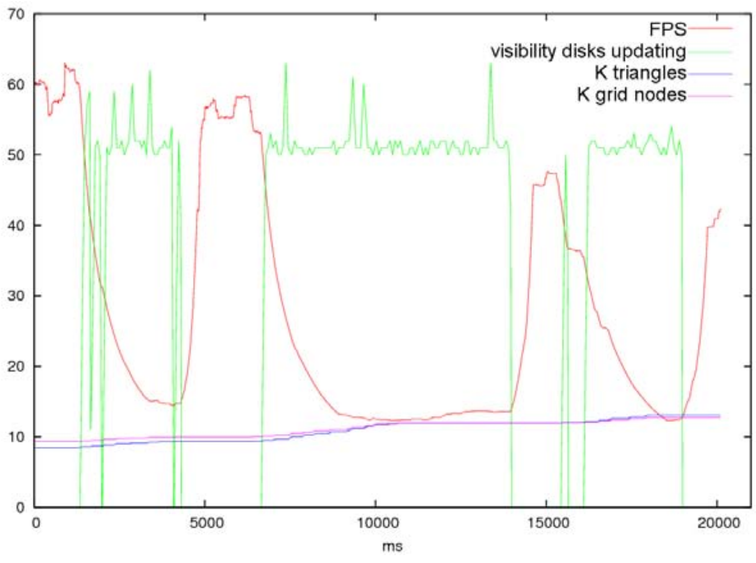


Figure 2.21: Performance of the method recorder during sequence shown in Figure 2.20

2.8 Results and discussion

The approach we presented was implemented on a Windows XP platform using C++ and OpenGL running on a dual core P4 @ 3GHx PC equipped with 2GB Ram, and a NVIDIA GeForce 8800GTX with 768MB video memory; the GPU code was written in GLSL. The tests shown in the next sections were all recorded in real time.
The graph in Figure 2.21 shows the performance of our approach recorded during the interactive sequence shown by Figure 2.20. The object was filled with 566 phyxels and the initial surface created was made of 8452 triangles in 1177 grid cells. The radii of the visibility disks were set to 16 pixels wide.

Although the number of disks to update obviously depends on the speed at which the

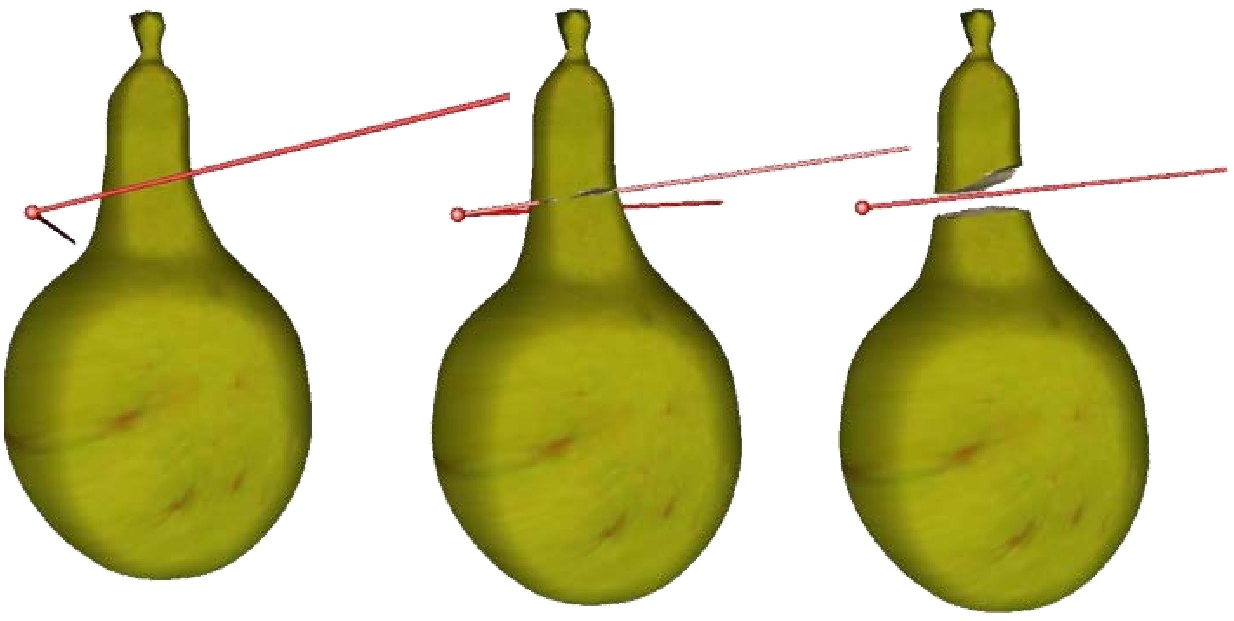


Figure 2.22: Scissors cutting a deformable pear.

cut surface grows, the time spent in updating the visibility disks is almost constant. This is due to a time-critical implementation that assigns a fixed time slot per frame for disks updating, giving the possibility to distribute the load over few consecutive frames. Note that the time for updating a disk is short and predictable (two renderings) and therefore the updating of all the disks is easily made interruptible by updating a few disks and then returning. It is clear that this may possibly cause a delay of the time step the cuts appears open but processing the disks in a FIFO order and distributing the load uniformly over the frames substantially avoids noticeable discontinuities during the opening of the cuts.

The time spent to interpolate the tessellation vertices is about 2ms for 4103 vertices. If we tessellate the surface by building a triangle mesh directly on the grid nodes, such a triangulation will have average triangle size of the order of a grid cell. In this sense, the time interpolation can be considered as an overhead introduced by the Splitting Cubes. The time for generating the tessellation includes, for each grid cell: computing the cell configuration, accessing the LUT, instantiating new triangles and setting the dependencies of the vertices. The generation of the initial surface of the liver required to process 1177 cells and took 282 ms. In general, the processing of a grid cell takes less than 5 ms.

The memory consumption is also linearly related to the area of the cut surface (please note that only the grid nodes of the cells containing the surface are actually stored in main memory).

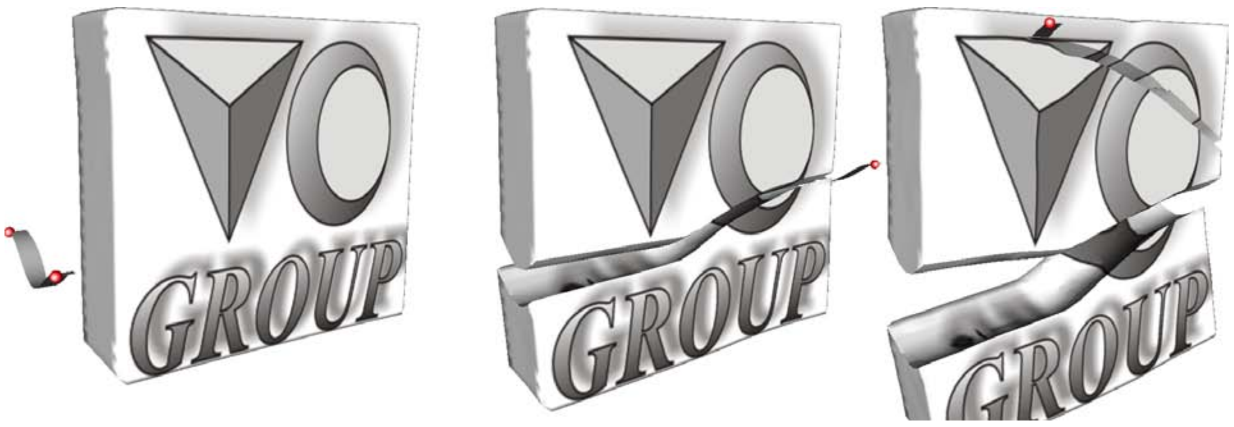


Figure 2.23: A cut with a curved tool.

As expected, the number of triangles grows linearly with the cut surface.

The Splitting Cubes algorithm does not pose restrictions on the tool path. Thanks to the implicit definition of the tessellation, the surface is always intrinsically consistent and the self intersection of the cut surface does not need any special treatment.
To support this claim figure 2.22 shows two cutting tools that act as scissors for cutting a pear while it deforms. Another sequence shows a pair of parallel blades rototranslating into a cube-like object, see figure 2.11. Finally Figure 2.23 and Figure 2.24 shows two interactive cuts using curved tools.

With respect to the previous approaches, we can summarize the main advantages given by the Splitting Cubes Algorithm as follows:

It is independent of the physical model The major benefit of the Splitting Cubes algorithm is that it decouples the model for physical simulation from the problem of virtual cutting. The dynamic re-tessellation, the intersection with other objects and the representation of discontinuities are threaded independently with respect to the underlying deformation function. Therefore, it can be seamlessly integrated with other methods for physical simulation.

It is robust Since every possible configuration of cut edges corresponds to a predefined tessellation, there are no possible states of the system that can lead the algorithm to end inconsistently.

It is well conditioned for collision detection Since we do not have to care about bad shaped triangles but only with the edges of a deformed regular grid, the task of detecting intersections between the tool and the object is better conditioned than in

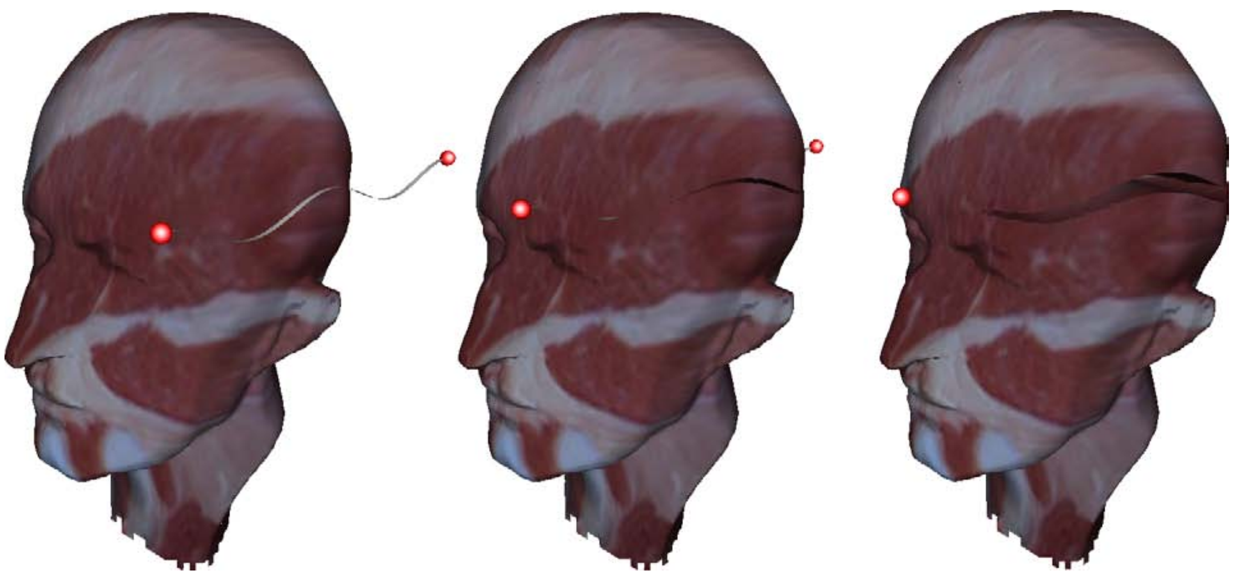


Figure 2.24: Another cut with a curved tool.

mesh-based approaches.

It handles implicitly the topology changes Topology changes come for free while edges are cut, without having to analyze the crack front as in [PKA+05, SOG06].

It is simple The Splitting Cubes algorithm is conceptually simple and easy to implement.

It is efficient The surface update requires a direct access to the LUT for each cell affected by the cut. Since the table can efficiently indexed, then its access time can be considered negligible with respect to the rest of the simulation.

Furthermore, we proposed a new method to create discontinuities on Meshless Methods without the need for exploring the graph of adjacency as in [SOG06], or the cut surface as in [PKA+05]. The Extended Visibility Method is suitable a for GPU implementation since it requires two renderings of few triangles (See Section 2.6.3 and 2.6.2 for details).

There are several directions of improvement/exploitation of the Splitting Cubes approach:

Original Boundary With the current solution the detail of the representation and the granularity of the cut are coupled. We could preserve the original surface triangulation by handling explicitly its intersection with the tessellation given by the LUT. This approach prevents visual artifacts related to the snapping of the vertices when the configuration of a cell changes.

Granularity of the cut Similarly to [MBF04], the efficiency of the Splitting Cubes algorithm comes at the price of limiting the granularity of cuts to the size of the cell. This limit can be overcome by considering that the Splitting Cubes Algorithm can be also implemented in an adaptive fashion, therefore replacing the grid with a hierarchy. However it is not immediately clear about how to switch to a hierarchical approach and, at the same time, maintain robustness and efficiency. In general, if we allow recursive subdivision of cells (typically 1 : 8) we implicitly introduce fragmentation, slowing the whole system.

For example, suppose we would like to detect intersections with the grid edges. Since we lose the implicit spatial indexing given by the regular grid, then this procedure will be more complex.

In the main application that motivates this research, i.e. a virtual cutting for surgical training, the interaction happens within a scale frame. More precisely, for such applications, robustness and interactivity is, in general, more important than the granularity of the cut.

Collision Detection The Splitting Cubes does not add any particular issue regarding the collision detection problem and existing techniques can seamlessly be integrated in the framework.

Furthermore the parametrization of the volume given by the grid could offer some advantage for self-collision detection.

Chapter 3

Modeling the interior of an object

Textures provide a simple and efficient way of modeling 3D objects by separating appearance properties from its geometric description.
Textures have been extensively used in computer graphics for modeling the external structure of objects, either through photographs [SKvW$^+$92] or through procedural models [Per85, EMP$^+$94].
When objects undergo topological changes (for example cuts or fractures) their internal structures are revealed, and this phenomenon has motivated researchers to design approaches for modeling the texture of *internal surfaces* of objects. However, the simple procedure of taking photographs of objects and "pasting" them on a 3D model has been successfully applied to external but not to internal surfaces.
In interactive cutting and fracturing, which is our specific application domain, the main challenge is dynamically synthesize the appearance (textures) of internal surfaces, as they appear as the consequence of cuts or fractures, minimizing the introduced overhead.
We consider the *expressing power* of a texture synthesis algorithm as its capacity of capturing and reproducing features which are present at different scale in the input image. Expressing power is one important characteristic of a synthesis algorithm, since it measures its versatility in modeling the different appearances which are present in real world. As explained in Section 3.1, we refer to texture appearance classification based on regularity as proposed by [LLH04].
We group the existing algorithms for modeling internal appearance of objects in two classes: Solid Textures [HB95, GD96, DGF98, JDR04, QY07, Wei03, DLTD08, KFCO$^+$07] §3.3.1 and Solid meshes [CDM$^+$02, ONOI04, TOII08] §3.3.2.

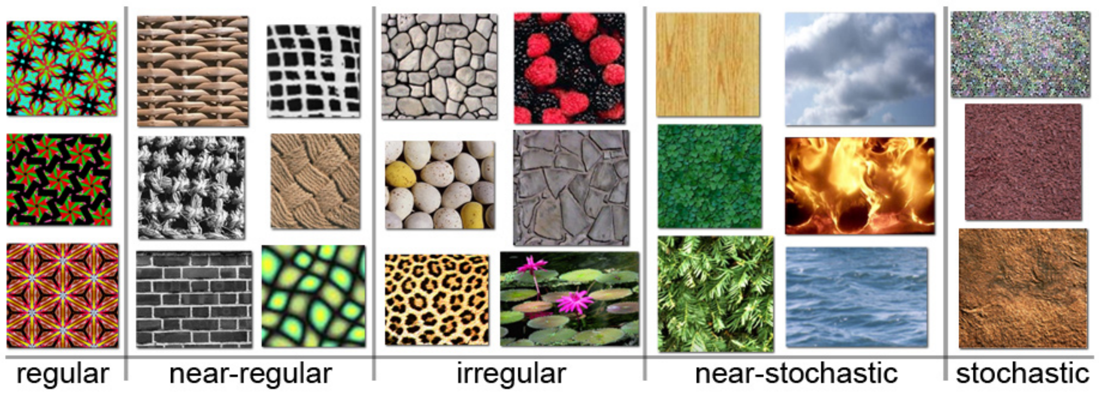


Figure 3.1: Classification of textures as proposed by [LLH04].

3.1 texture classification

By considering its appearance a texture can be classified into an interval which smoothly varies from *Stochastic* to *Structured* [LLH04] :

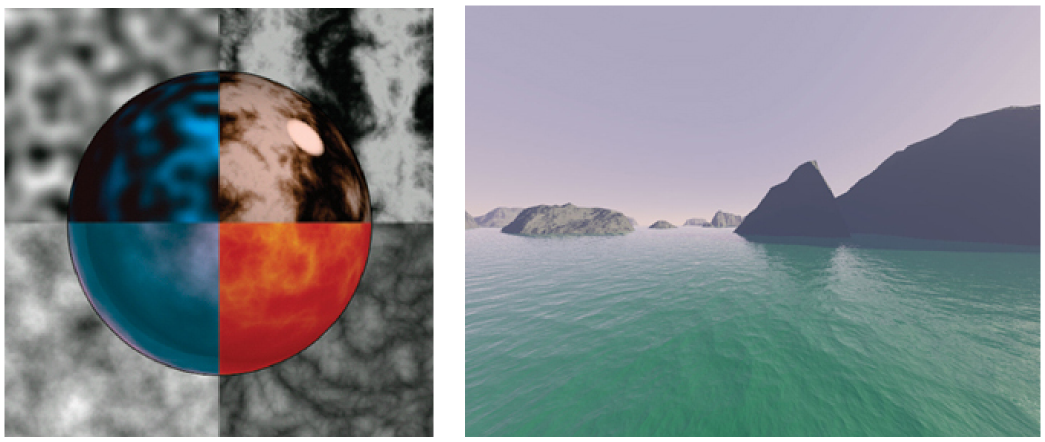
Stochastic textures This class of textures looks like noise showing a high degree of randomness. An example of a stochastic texture is roughcast.

Structured or regular textures These textures present regular patterns. An example of a structured texture is a stonewall or a floor tiled with paving stones.

The 2D textures classification proposed by [LLH04] and illustrated by figure 3.1, is extendible to 3D textures, by considering the presence of regular patterns along the 3 directions, instead of 2, of a solid texture.
The algorithms for modeling object's internal color are often defined by extending basic concepts of 2D texture synthesis. For this reason, we first give a brief introduction about 2D texture synthesis algorithms, then in Section 3.3 we provide a detailed description of existing methods for modeling and representation object's internal appearance.

3.2 A brief introduction to 2D Texture Synthesis

The problem of texture synthesis is typically posed as producing a large (non-periodic) texture from a small example.
In the next sections we provide a brief overview of the existing 2D texture synthesis techniques, classifying existing methods as: Procedural, Statistical features-matching, Pixel



(a) (b)

Figure 3.2: Application of Perlin noise: (a) a set of synthesized materials [Fer04], (b) a landscape is synthesized by Terragen [pla] using Perlin noise.

based, Patch based and finally Optimization based. We refer to [KLTW07] for a more detailed description of existing methods for example-based 2D texture synthesis.

3.2.1 Procedural Methods

Procedural methods synthesize textures as a function of pixel coordinates and a set of tuning parameters.
Among all procedural methods, the most used in Computer Graphics is *Perlin Noise*[Per85]. The key idea of Perlin noise consists of trying to replicate the patterns presents in nature. Indeed, we can observe that nature creates patterns that are fractal, i.e. different level of details show different variations, or, in other words, each level of detail is defined by a pseudo-random pattern. For example, a terrain is formed by large variations (mountains) upon that rises medium variations (hills) and small variations (rocks). The main goal of [Per85] consists in replicating this phenomenon by applying noise functions on each level of detail. Noise functions are combined together to create the final result.
More precisely, Perlin Noise is a gradient noise function that perturbs mathematical functions in order to create pseudo-random patterns. Perlin noise has been widely used in various application domains, to cite a few: rendering of water waves, rendering of fire, realistic synthesis of the appearance of marble or crystal.
Among the procedural methods, [Lew87] derives noise functions by generalizing stochastic subdivision, while [Gar84, Gar85] synthesizes terrains, water fields and clouds by summing

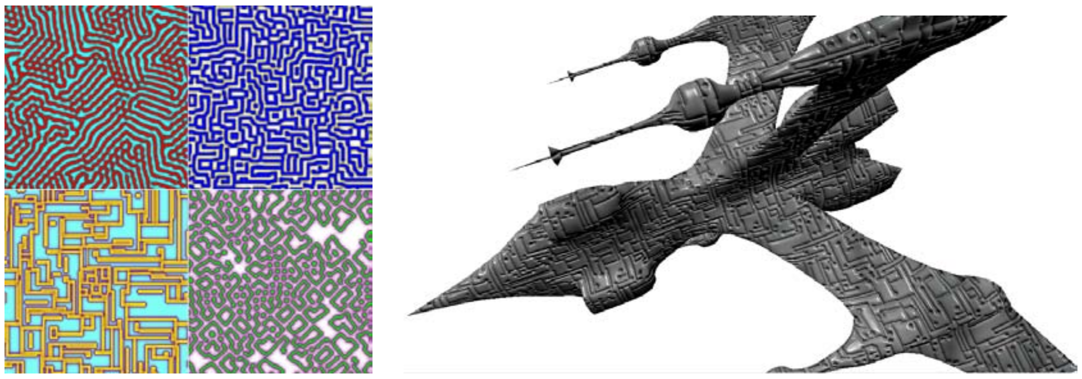


Figure 3.3: Examples of textures generated by the generalized reaction-diffusion model [McG08].

and multiplying sine waves.
Textures can be synthesized also by simulating natural processes. In [WK91] and [Tur91] authors extends the reaction-diffusion model (which is originally proposed in chemistry to simulate the formation of biological patterns), to generate organic texture patterns. This method has been recently generalized to allow the synthesis of new class of textures [McG08]. Other examples of natural process simulation are [FLCB95, Wor96], which generate noise by simulating biological cells, which are modeled as small interacting geometric elements distributed on the domain.

2D Procedural methods are, in general, efficient and easily extendable for solid-texture synthesis. Moreover procedural methods constitute a useful and simple tool for the synthesis of a set of textures, Figures 3.2 and 3.3 show possible application scenarios.
However, the main problem of procedural texture synthesis is its lack of expressive power. Since a procedural method is limited to encode a specific texture appearance, if we want to encode a new texture we must define a new function. In general, it is tricky to define analytically texture patterns which are present in real world. In this sense, maybe it is more useful a synthesis method that captures informations from a template image provided by the user.

3.2.2 Statistical features-matching methods

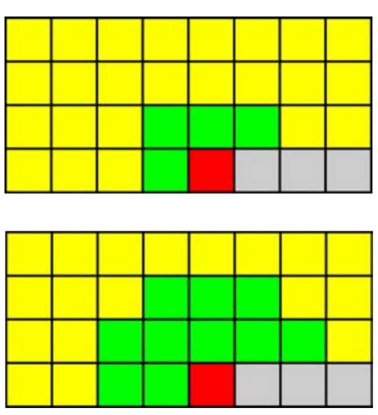
The main strategy of this class of methods consists of capturing a set of features from the template image and transfers it to the synthesized image. Generally, the synthesized image is initialized using random noise, then the algorithm transfers characteristics captured from the template image, such that the resulting image looks like the template image.

[HB95] is the first relevant work in this class of methods. In this approach, *Histogram matching* is used to iteratively modify the synthesized image in order to converge to the appearance specified by template image. More precisely, the color range of an image is quantized into a set of uniform intervals bins, each of which represent an subinterval and the related probability of a pixel to fall into such interval (evaluated using color distribution).
Histogram matching is used to make the color distribution of a source image to match with the color distribution of a target image. Histogram matching is based on the cumulative distribution function $CDF_H : [bins] \rightarrow [0,1]$ and its inverse $CDF_H^{-1} : [0,1] \rightarrow [bins]$, were H is an image histogram. Given a source I and a target images I', and considering their histograms H and H', histogram matching consists of substituting each color of source image $v \in I'$ with the one having the same CDF value in target image I':

$$v' = CDF_{H'}^{-1}(CDF_H(v)) \tag{3.1}$$

Image's sub-bands are captured through *steerable pyramid* [Per91, SFAH92, SF95]. Steerable pyramids captures at different frequencies the gradient with respect to a set of possible directions. Given that the steerable pyramids are invariant with respect to translation or rotation, they are particularly suitable for detail transfer.
The overall algorithm can be schematized as follows: after a first histogram matching between noise and template images, the algorithm continue by iteratively applying histogram matching across each pairs of steerable pyramids levels. Finally, the image is fully reconstructed from the processed pyramids levels. This method, together with its extension [PS00], works well only on stochastic textures, while it fails in general if the template image is structured.

In [dB97] images are represented using *Laplacian pyramids*. Laplacian pyramids [Bur83] encodes the differences between two adjacent levels of a gaussian pyramid. For each value at a certain level of resolution is associated a *parent structure* which encodes a set of local texture measures, called *features* (typically related to human perception) evaluated for each ancestor. Two pixels at a certain level of the Laplacian Pyramid are considered to be *similar* if and only if their parent structure is similar, i.e. the sum of squared differences between their features is less than a certain threshold. This method, starting form a noise image, runs across laplacian pyramids levels, substituting each value level with a similar pixel coming from template pyramid. This method improves the results obtained by [HB95] especially for the near-stochastic cases.


(a)

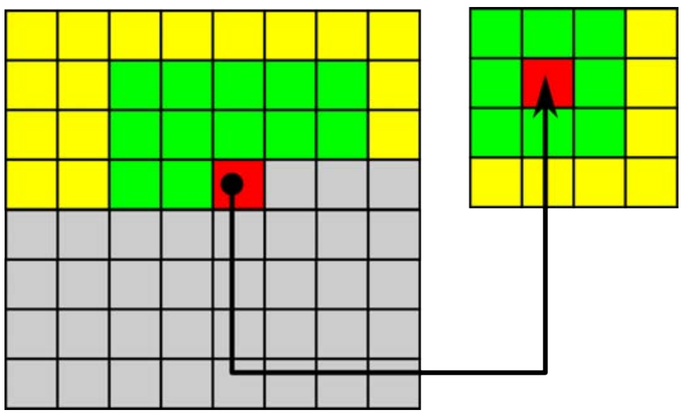

(b)

Figure 3.4: (a)Two possible causal conditioning neighborhood kernels [PP93]. The Pixel in red is the one that has to be synthesized, while its conditioning neighborhood kernel is indicated in green. The conditioning neighborhood kernel is a subset of the pixels that were synthesized in a previous step, indicated in yellow. (b) The L-shaped causal conditioning kernel defined across the same level and respective squared-shape kernels coming from next level [WL00].

3.2.3 Pixel Based

The main idea of Pixel-Based methods consists of relating pixels color and spatial neighborhood.

This intuition was developed in [PP93]. In this approach the color of a pixel is fully determined by a probabilistic *causal conditioning neighborhood* kernel. After an initial phase of training, where each pixel of the template images is correlated to its neighborhood kernel, the image is synthesized pixel by pixel in a scan-line order. Figure 3.4, shows two possible neighborhood kernels for a row-major scan-line order. The neighborhood kernel is designed such that the pixel that has to be actually synthesized (indicated in red) depends from a subset (indicated in green) of previously synthesized ones (indicated in yellow).

Efros and Leung in [EL99] extend this idea by reinterpreting the algorithm as a nearest neighbor-search problem. More precisely, instead of constructing an explicit probabilistic model relating a pixel to its neighbors, they search for the sample that minimizes a distance function d in the template image. The distance function d relates two pixel's neighborhood and is usually defined as the weighted sum of squared differences. Weights are calculated by using a two-dimensional Gaussian kernel.

Given that an initial portion of the image is synthesized, the algorithm continues synthesizing pixels in a spiral order as follows: for each pixel p, find the set of candidates $\Omega(p)$ that minimizes d (candidates comes from the template image), then pick one randomly and assign its color to p.

Wey and Levoy in [WL00], starting from [PP93], define an algorithm that, thanks to its simplicity and its performance has become very popular in the Computer Graphics community. They proposes two versions of the synthesis algorithm.
The first approach works on a single resolution level: starting from an image made of random noise, pixels are synthesized in a scan-line order using a causal conditioning neighborhood kernel. Similarly to [EL99], the color of current pixel is substituted by the one that minimize a distance function, choosing from the template image.
This algorithm is extended to work in a multi-resolution fashion. The multi-resolution approach is based on the *Gaussian pyramids* of template and synthesized images. The main difference with respect to single-resolution consists in how distances are measured. Indeed, in this case distances are calculated by combining different levels of the Gaussian Pyramid. More precisely, distances are defined by using simultaneously:

- L-shaped causal conditioning kernel defined across the same level.
- The whole sequence of square kernels coming from higher levels (considering that all those pixels were already synthesized, then is possible to use a square kernel).

Figure 3.4.b shows an example of causal conditioning kernel, together with the associated inter-level kernel. The multi-res algorithm starts from the highest level (which has to be initialized) synthesizing iteratively each level of the gaussian pyramids, until the final image is completed.
Thanks to the multi-resolution approach it is possible to capture multi-scale features and inter-scale dependencies present in the texture, producing a significant improvement on final results (see figure 3.5). Moreover, by considering that neighborhood has a fixed size, the search process can runs faster by using a tree-structured vector quantization (TSVQ) [GG92]. The overall synthesis time is reduced then from 503 to 12 seconds to synthesize an 192^2 image, while [EL99] needs 1941 seconds (timings were measured using a 195 MHz R10000 processor [WL00]).
Some significant extensions of [WL00] were proposed in literature. For example Michael Ashikhmin in [Ash01] introduce the concept of pixels *coherence* to improve results using natural textures. Coherence tend to group the pixels that are together in the input image. While *image analogies* [HJO$^+$01] uses pixel's neighborhood similarity to learn and reproduce image filters from exemplars.

The Pixel-based methods presented are considered *order-dependent*, i.e. the resulting image depends on the order the pixels are synthesized. In [WL01] Wei and Levoy modify

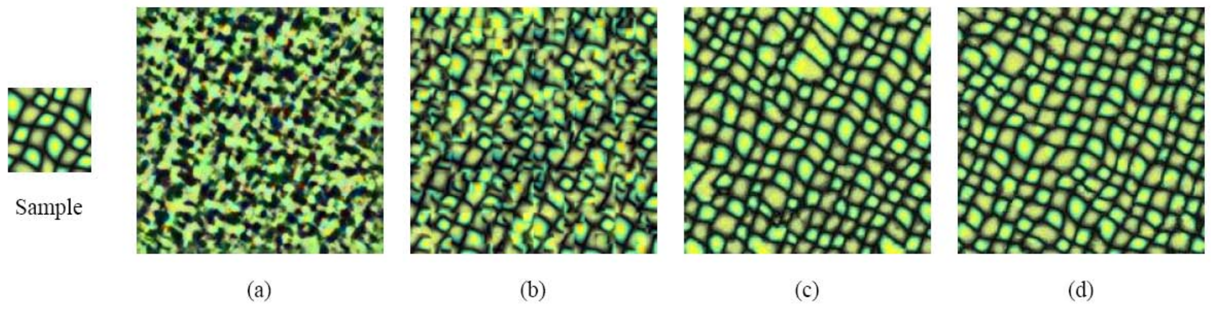


Figure 3.5: A comparison of texture synthesis results using different algorithms (a) Heeger and Bergen's method [HB95] (b) De Bonet's method [dB97] (c) Efros and Leung's method [EL99] (d) Wei and Levoy method [WL00].

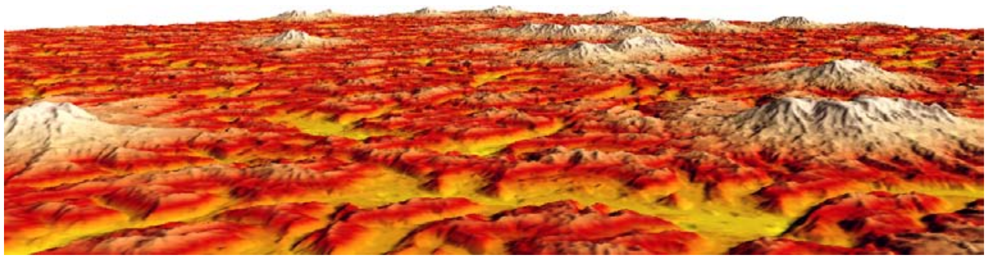
their original pixel-based algorithm to make it *order-independent*. The main idea can be summarized as follows: the value of a synthesized pixel is stored in a new image (instead of overwriting), while the kernel used for neighborhood search is made by pixels that were synthesized in the previous step. Since the algorithm is order-independent, the kernel can be designed as a square of pixels. The algorithm proceeds by repeating the synthesis step on pyramid levels until convergence.

In 2005 Lefebvre and Hoppe [LH05] parallelized the order-independent synthesis. By using the GPU, synthesis can be performed in real-time (some results in 3.6.a), opening new application scenarios. One year later, in [LH06], the same authors proposed to enrich the neighborhood search space. That space, called *appearance-space*, extends the RGB color space by including neighborhood information (like gradients), features information (like distance to image's edges). The search space is reduced in dimensionality by using principal component analysis (PCA). Thanks to the high concentration of per-pixel information the kernel size is reduced, with a resulting improving of speed performances (from 3 to 4 times faster respect to [LH05]), see figure 3.6.b.

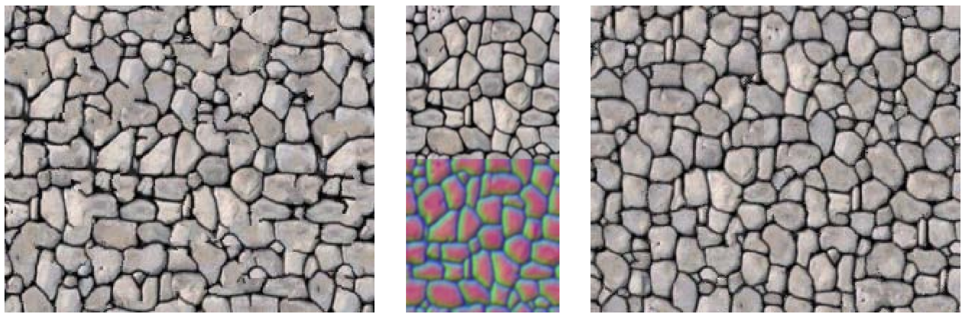
3.2.4 Patch Based Methods

This class of methods relies on a different philosophy: the template image is divided in a set of patches, which are re-arranged in the output image. Patch-based methods can be seen as an extension of pixel-based methods, but instead of copying one pixel at time an entire patch is copied to the output image. Managing patches instead of pixels, can be seen as a way to keep as more as possible the original structure of the template texture. The main challenges raised by this methods are:

- How to arrange patches to produce randomness in the synthesized image?



(a)



(b)

Figure 3.6: (a)Examples of results generated by using the parallel controllable texture synthesis algorithm [LH05]. (b)A comparison between [LH05] (Left) and [LH06] (Right), thanks to the high concentration of per-pixel information expressed in the feature image (Center) the resulting texture preserves the original pattern.

- How to avoid artifacts between adjacent patches?

The first patch-based method is the *chaos mosaic* algorithm [GSX00]. In this approach patches were arranged by using an iterative discrete chaos mapping [Sch88], which is called *Cat-Map*, that creates visually stochastic patterns. The mismatch across adjacent patches were resolved by using a constrained texture synthesis [EL99]. However, the quality of results produced by chaos mosaic is often unsatisfactory, due to visible artifacts between patches.

In 2001, Efros and Freeman proposes a new patch-based algorithm [EF01]. Between adjacent patches an overlap region (usually 1/6 of patch size, see Figure 3.7.b) is used to *quilt* appropriately adjacent patches making sure they all fit together. Patches are placed in the output image in a scan-line order. Patches are chosen step-by-step randomly from a set of candidates, which is constituted by the ones that minimize an error. Such error is defined as the squared differences of color between overlapping pixels. As shown in Figure

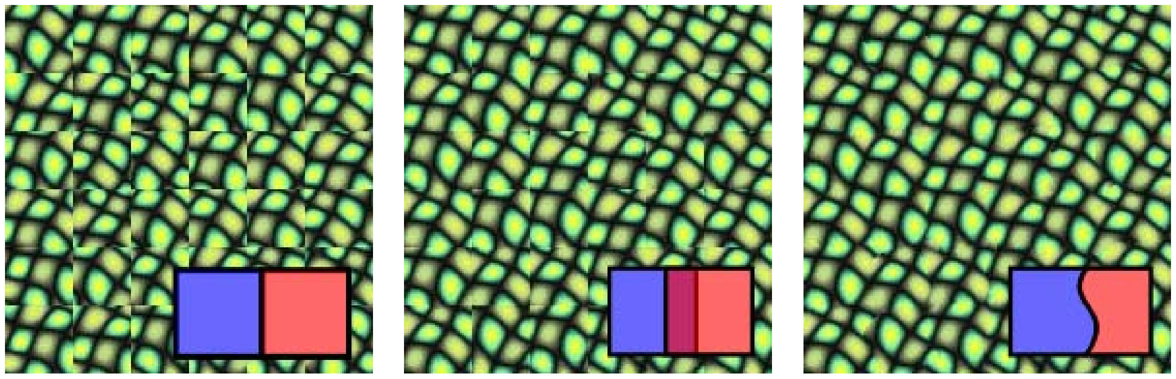

(a) (b) (c)

Figure 3.7: Texture quilting algorithm [EF01]: (a) Patches are chosen randomly. (b) Patches are placed in order to minimize error on overlap regions. (c) The overlap region is cut to reduce artifacts between adjacent patches

3.7.c, once patches are placed, overlap region is quilted appropriately to minimize the error. In 2003 Kwatra et al. [KSE+03] improve this approach by minimizing the error using a different strategy: a graph-cut algorithm.

In [CSHD03] Cohen et al ensure the tileability of adjacent patterns by using a different strategy: a set of patches, called *Wang tiles*, were extracted from the template texture such that the edges are pairwise tileable, i.e. the colors across the edges are coincident. The final images is constructed by simply assembling Wang tiles in a way that adjacent edges share compatible colors. That algorithm was extended to work on the GPU in 2004 [Wei04].

3.2.5 Texture optimization method

This technique, introduced in [KEBK05] relies on a global optimization framework to synthesize a new texture.
It essentially consists of minimizing an energy function E_T which considers all the pixels together. This energy function measures the similarity with respect to template texture and it is locally defined for each pixel. Similarly to pixel-based methods, E_T is defined using a local neighborhood:

$$E_T(x; \{z_p\}) = \sum_{p \in X^\dagger} \|x_p - z_p\|^2 \tag{3.2}$$

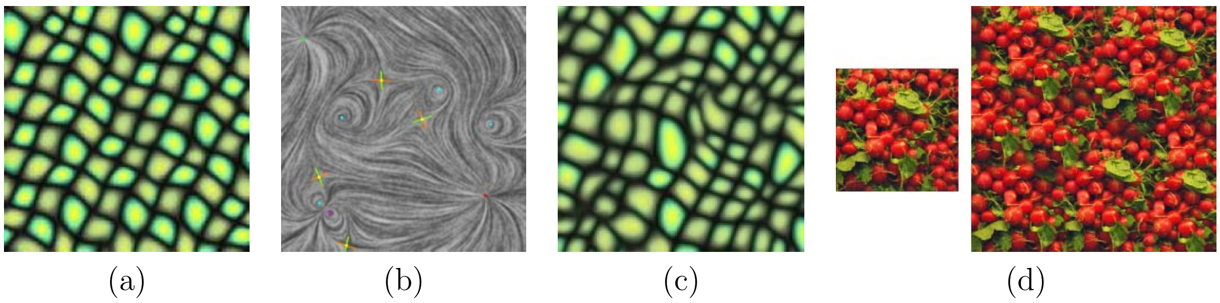


Figure 3.8: Examples of textures synthesized by using [KEBK05]: (a) Simple texture synthesis (b) The flow field used to lead the synthesis process in (c). (d) An example of synthesis with a structured texture.

where x_p is the local neighborhood of pixel x in the synthesized image, and z_p its most similar neighborhood in the template image. In order to make the minimization process more robust, terms of equation 3.2 are weighed by considering distance to the center. Those local energy contributes are merged together in a global metric which is minimized.
While the local energy term of equation 3.2 is related to pixel-based methods, the global minimization process is related to patch-based methods, since pixels are considered together at each minimization step.
Minimization is performed by using an EM (Expectation-Maximization) algorithm [MK97], which consists in two main steps:

E step keeping fixed $\{z_p\}$, minimize E_T by modifying x (and consequently x_p). In other words, colors of synthesized image are modified to resemble locally, as more as possible, to the precomputed neighborhood sets $\{z_p\}$.

M step keeping fixed x minimize E_T by updating $\{z_p\}$. For each synthesized pixel x, the corresponding z_p is updated by finding the best matching neighborhood in the template image.

This steps are repeated in a multi-resolution and multi-scale fashion.
Despite its slow convergence, this method produces very good results (see Figure 3.8.a and 3.8.d), furthermore energy formulation can be easily extended to create flow-guided synthesis (see Figure 3.8.b and 3.8.c).

3.3 Designing internal properties of meshes

The internal appearance of an object $\mathcal{M}$ is defined by a function $\mathcal{F}$ which maps each point p belonging to $\mathcal{M}$ to the respective color attribute $color(p) = \mathcal{F}(p), p \in \mathcal{M}$. This mapping is usually extrapolated from a reduced input provided by the user.
We can divide the methods, and relative data structures, that models the internal appearance of an objects in two main categories: *solid textures* and *solid meshes*. Intuitively the main difference is that solid textures are independent respect to the object on which they're mapped, while solid meshes are built upon the object.
Next chapter makes this classification more clear, providing an exhaustive description of existing approaches. We finally provide a comparison between this two main class.

3.3.1 Solid Textures

Solid textures were introduced in Computer Graphics by [Pea85], *"This paper introduces the notion of "solid texturing". Solid texturing uses texture functions defined throughout a region of three-dimensional space"*. While traditional textures relies to 2D, solid textures are defined across a 3D space.
If we embed an object into the domain of a solid texture we implicitly define its surface's color without introducing distortion. Moreover, since solid texture defines the color for each point belonging to object's volume it can be utilized to texturize surfaces revealed by cuts or fractures.
Similarly 2D texture synthesis, methods proposed in literature for the synthesis of solid textures can be classified in:

Procedural methods The color is a function of the 3D positions and a set of parameters provided by the user.

Statistical features-matching methods Extracts statistics from 2D textures and replicates on the solid texture.

Pixel based methods The color of each pixel belonging to solid texture depends on its neighbors.

Optimization based methods The solid texture is the result of a global minimization.

In the following sections we presents most significative approaches according to this classification.

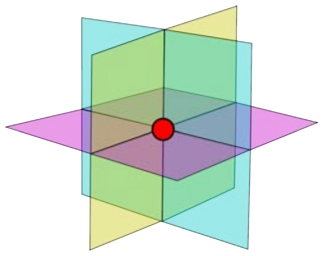


Figure 3.9: A 3D neighborhood composed by 3 orthogonal 3D slices.

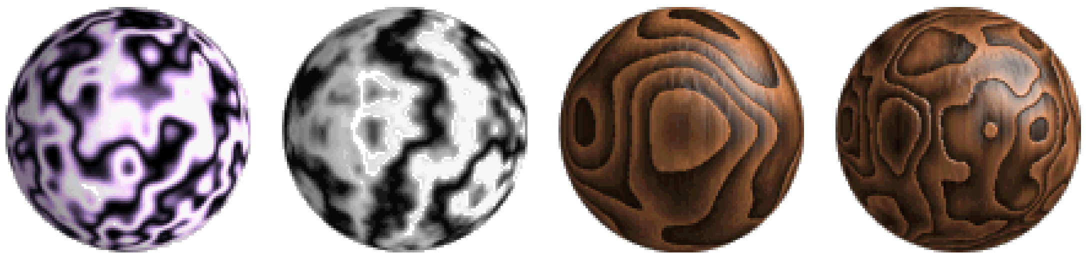


Figure 3.10: Examples of solid textures produced by using Perlin noise

3.3.1.1 Notation

We introduce a simple notation used in the following sections, the reader may refers to Figure 3.9 for better understanding.
We call *voxel* the cells belonging to solid texture to distinguish from *texel* which belongs to a 2D texture. The *3D neighborhood* of a voxel v is formed by assembling 2D neighborhood centered in v slicing the solid texture along each axis. A *3D slice* refers to each orthogonal 2D neighborhood defining a *3D neighborhood*.

3.3.1.2 Procedural Methods

Procedural methods for the synthesis of solid textures are, in general, derived directly from the 2D methods. Indeed, thanks to their “dimension-independent” formulation, procedural methods are, in general, easily extendible to the 3D case.

For example the noise functions defined by Perlin in [Per85] can be used to synthesize solid textures. Solid noise is a 3D function used to perturb a basis 3D function in order to create realistic solid patterns. For example solid marble can be obtained by perturbing a sinus function:

$$Marble(x, y, z) = sin^n(x + Noise(x, y, z)) \tag{3.3}$$

Perlin noise has been largely used in Computer Graphics to produce solid textures of marble, rocks or wood (see Figure 3.10 for application examples).

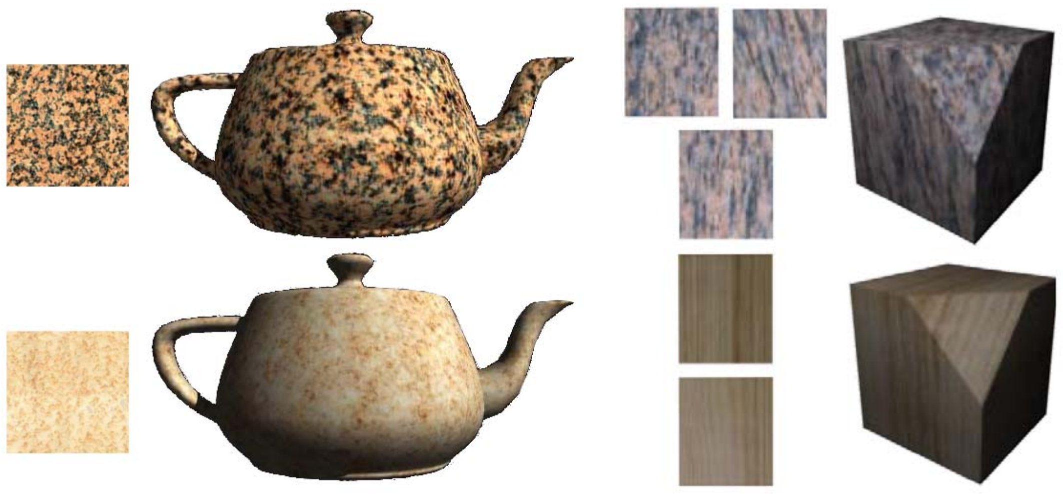
A procedural methods for solid texture synthesis is, in general, easy to implement and computationally light. Since the color of a voxel is a function of its coordinates, procedural methods can synthesize each voxel independently, while the majority of methods requires the synthesis of the entire solid block.
As in the 2D case procedural methods can be defined to simulate natural process. For example, Buchanan in [Buc98] proposes to synthesize wood solid texture by simulating the growth process of a tree, Dorsey at al proposes to simulates the natural process of weathering stones [DEL$^+$99], while [HTK98] uses a mass-spring model to simulate the propagation of crack patterns.
As in the 2D case, main problem of these methods is their lacks of generality. In general a procedural method corresponds to a specific type of pattern.

3.3.1.3 Statistical features-matching methods

Similarly to 2D, the main purpose of this method is to extract a set of statistical properties from the template image in order to replicate them in the synthesized texture. However, solid texture synthesis is a more complex scenario: properties are defined in 2D, while the syntheses is performed in 3D. Since no 3D information is provided, these methods transfers statistical properties which are defined over a 2D space, to an higher order space, i.e. the 3D space embedding the solid texture.

For example, [HB95], which has been introduced in section 3.2.2, can be easily extended to produce solid textures. Since the CDF (Cumulative Distribution Function) expressed by image histogram is independent of the dimensionality of input data, then it is possible to apply the same histogram matching on a solid texture, rather then an image. In this specific case, target histograms relies on 2D steerable pyramids of the template image, while source histograms relies on 3D steerable pyramids of the solid texture. Initially the solid texture is initialized with random noise, then the algorithm continues by repeatedly applying multi-scale histogram matching operations (as the 2D case, see Section 3.2.2 for details). Finally the solid texture is reconstructed by collapsing the tri-dimensional steerable pyramid.

Ghazanfarpour and Dischler [GD95] propose to use *spectral analysis* for solid texture synthesis. Spectral information is extracted from the template images using the *Fast Fourier Transformation* (FFT), and used to obtain a basis and a noise function. Finally, the solid texture is obtained procedurally as in [Per85].
This method is extended by [GD96] to use multiple images. Each image defines the ap-



(a) (b)

Figure 3.11: [Examples of solid textures synthesized by using statistical features-matching methods](a) Examples of solid textures produced by [HB95]. The textured model is carved from the synthesized texture block (b) Anisotropic solid textures generated by [DGF98] using multiple template images.

pearance of the solid texture along an imaginary axis-aligned slice. The algorithm is built upon the assumption that the appearance of axis-aligned cross-sections are invariant with respect to translation, while a the non-orthogonal ones blend the appearance of the three template images according to their orientation.
Modifications are realized by using *spectral* and *phase* processing of image FFT. The synthesis process take as input a solid block initialized with noise, and modifies axis-aligned slices, extracted from the solid texture, according to the corresponding template image. Since each voxel belongs to three different slice, it defines three possible colors, which are simply averaged. By repeating this step, the noise block slowly converges to the appearance of template images.
In [DGF98] this approach was modified to avoid phase processing. The solid texture is generated by repeatedly applying spectral and histogram matching. The reader my refers to [DG01] for a survey on spectral analysis methods.
While methods based on spectral analysis([GD95],[GD96] and [DGF98]) produces pleasant results using stochastic textures, it usually fails with structured textures.

A significantly different approach in generating structured textures is proposed by Jagnow et al. [JDR04]. This method is based on classical stereology [Und70, Hag90]. Stereology is an interdisciplinary field that provides techniques to extract three-dimensional information

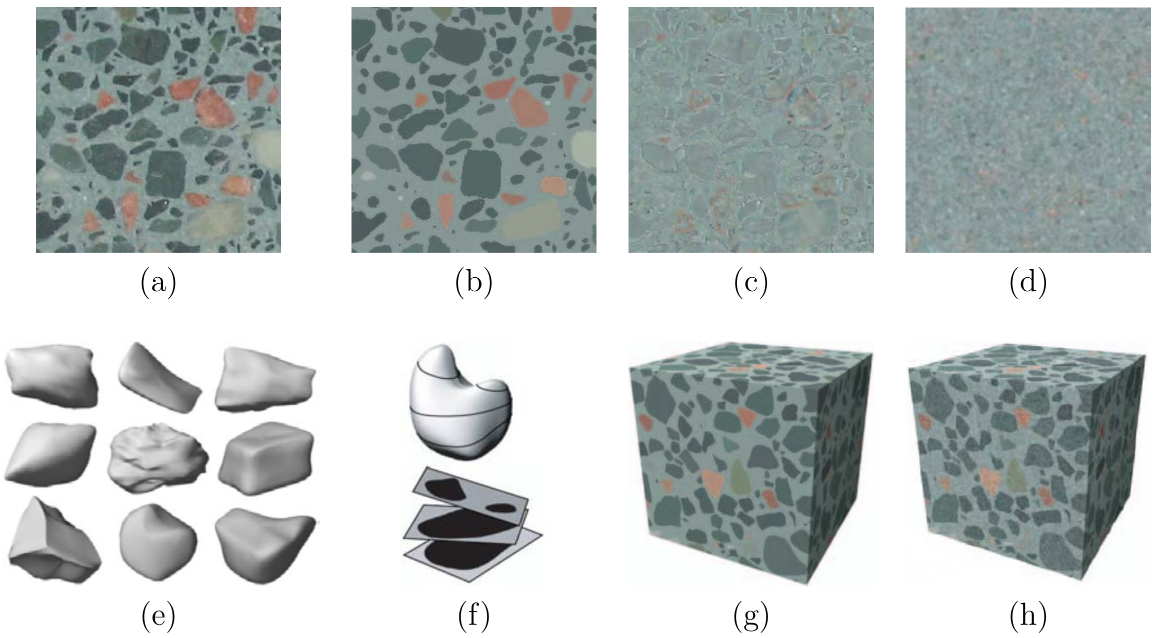


Figure 3.12: The synthesis pipeline of [JDR04]: (a) The initial image, (b) the profile image, (c) the residual image, (d) the synthesized residual solid texture, (e) 3D meshes of the different particles, (f) slices of a particle, (g) solid texture obtained by distributing particles (h) final result.

from measurements made on two-dimensional planar sections.
Figure 3.12 give an overview of the method. The initial image *(a)* is filtered to extract two components: a profile image *(b)* and a residual image *(c)*. The profile image, together with particle's shape *(f)* is used to infer, trough stereological techniques, the 3D distribution of particles *(e)* (encoded as triangle meshes), while the residual image is used to synthesize a residual solid texture *(d)* . The final solid texture *(h)* is obtained by adding residual solid texture, which encode the fine details, to the solid texture obtained by the distributing particles *(g)*, which encode the rough structure.
In this method, stereology relates the particle's area distribution in the profile image, with particle's area distribution revealed by an arbitrary cross-section of the solid texture. Profile image and particles's shape concurs to define the 3D particle's distribution, since:

- The profile image captures the distribution of particle's area. This distribution must be replicated in the solid texture, so that is preserved along every cross-section.
- On other hands, a cross-sections of the solid texture cuts some particles defining an area distribution which is obviously related to particle's shape. Authors propose to capture the area distribution generated by a particle by cutting randomly its meshed

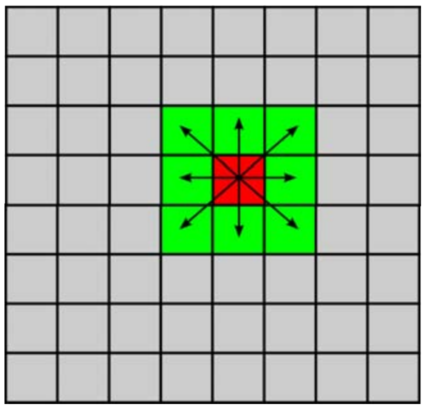


Figure 3.13: Displacement configurations of [QY07] generated by a 3^2 kernel.

model (Figure 3.12.f).

These probability distributions concur to extract a *particle density function* which defines implicitly how particles has to be distributed.
In [JDR08], Jagnow et al. presents an interesting analysis about how different methods for approximating particle's shape influence the perception of the generated solid texture.
This stereology-based synthesis technique, despite its impressive results, can be applied only to a limited set of textures.

Aura 3D textures [QY07] overcome this lack of generality. Aura 3D solid texture synthesis is based on Basic Gray Level Aura Matrices (BGLAM)[EP94, QY05]. The information stored in BGLAMs characterizes the co-occurrence probability of each grey level at all possible neighbor positions, which are called also *displacement configurations* (see Figure 3.13). The synthesis algorithm is based on the consideration that two textures look similar if their *Aura matrix distance* is within a certain threshold. Aura matrix distance between two images is defined considering their BGLAMs. This approach, similarly to [GD96] and [DGF98], produces a solid texture given a set of oriented template images. Usually two or three axis-aligned template images are enough to define the anisotropic nature of a solid textures, nevertheless this method supports an arbitrary number of input textures. As previously introduced, the structure of a texture is captured by the BGLAM. More precisely, given a grey level image I quantized into G grey levels, and considering the $n \times n$ squared neighborhood of a texel t, there are $(n^2 - 1) = m$ possible BGLAMs, one for each possible displacement configuration with respect to t (see Figure 3.13). The *BGLAM distance* $A_i \in R^{GXG}$ for a given displacement configuration $i : 0 \leq i < m$ is computed as follows:

- Initialize A_i with zero.
- For each texel s belonging to I, consider its neighbor k defined by the current displacement configuration i.
- Increment $A_i[g_s][g_k]$ by 1. Where g_s and g_k are respectively the grey levels of s and k.
- Normalize A_i, such that $\sum_{i,j=0}^{G-1} A[i][j] = 1$.

Then, the distance $D(A, B)$ between two BGLAMs is defined as follows:

$$D(A, B) = \frac{1}{m} \sum_{i=0}^{i<m} \|A_i - B_i\|, \tag{3.4}$$

where $\|A\| = \sum_{i,j=0}^{G-1} A[i][j]$.

This formula relates only two 2D textures. In the case of solid texture synthesis, it has to be extended in order to consider the distance of a voxel (with its volumetric neighborhood) from a set of oriented slices. Such extension is the Aura matrix distance. Aura matrix distance is defined by blending appropriately the BGLAM distances between 3D slices and template images. This method can be generalized to support an arbitrary number of template textures. As usual, the solid texture is initialized with random noise, then the synthesis process consists of minimizing the Aura matrix distance of each voxel with respect to template textures. In detail, the algorithm repeats the following steps:

- Choose randomly a voxel v.
- Among all possible grey levels $0 \ldots G-1$, select the subset of candidates C_G that reduces the current Aura matrix distance from template images.
- Substitute the grey value of v, by choosing randomly from C_G.

Since BGLAM works only with grey levels, the RGB channels of the template images must be decorrelated in a way such that the algorithm can work independently on each channel. The algorithm produces good results, especially for structured textures (see Figure 3.14.a). Unfortunately it has some drawbacks (see Figure 3.14.b,3.14.c,3.14.d):

- It converges slowly (about one hour of computation needed to produce a 128^3 solid texture).

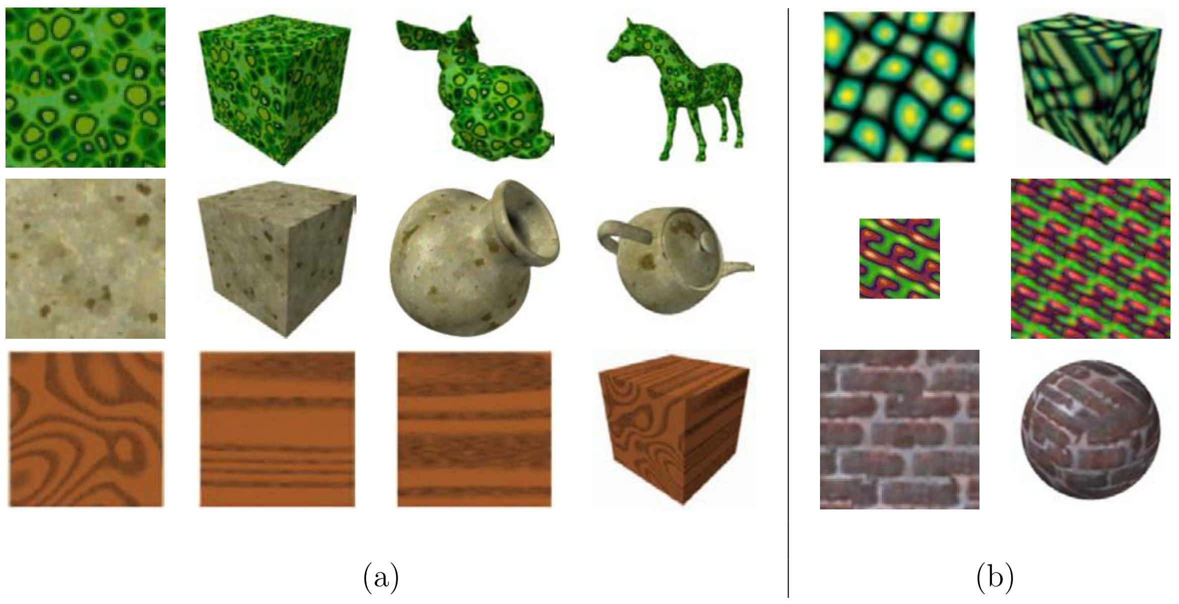


Figure 3.14: (a) Successfully examples of textures synthesized by Aura 3D synthesis [QY07]. (b) Examples of failures of [QY07]: The top row shows the effect of a convergence on a local minima, Middle row shows that independent synthesis of decorrelated channels leads to visual artifacts (courtesy of [KFCO$^+$07]), while the bottom row show an inconsistency generated by an oriented structural texture.

- It can converge to local minima, producing inconsistent results.
- Independent synthesis of channels may lead to visual artifacts.
- Oriented structural textures can cause inconsistencies in the solid textures.

3.3.1.4 Pixel based methods

Pixel-based methods for 2D texture synthesis (previously described in 3.2.3) have been extended to synthesize solid textures.
Similarly to the 2D pixel-based synthesis, the main intuition is to characterize a pixel by using its neighbors only. Again, the solid texture is produced by modifying a single pixel at time, searching in the template texture for the candidate which has a similar neighborhood. Despite the underlying principles are the same, volumetric synthesis entails novel problems:

- How to compare 3D neighborhood of a voxel with 2D texel's neighborhoods coming from template textures?

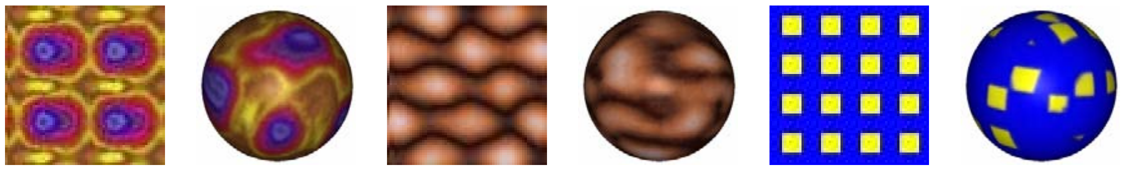


Figure 3.15: Examples of solid textures produced by [Wei02]

- How to handle multiple oriented template textures that concurs to define a single voxel color?

In [Wei03] and [Wei02] Wei extends [WL00] to synthesize textures from multiple sources. This method, originally proposed to synthesize 2D textures by mixing multiple sources, is modified to create solid textures from a set of oriented slices. As in [GD96, DGF98, QY07], the user defines the appearance of the solid texture along its principal directions by providing a set of axis-aligned slices T_x, T_y, T_z.
For each voxel v, 3D slices are used to select best-matching texel of template images. As in [WL00], three candidates colors p_x, p_y, p_z were selected by minimizing the energy function E, defined as the squared differences between 3D slices and 2D neighborhood:

$$E_x(v, p_x) = \|v - p_x\|^2 + \|I_x - N(p_x)\|^2; \tag{3.5}$$

$$E_y(v, p_y) = \|v - p_y\|^2 + \|I_y - N(p_y)\|^2; \tag{3.6}$$

$$E_z(v, p_z) = \|v - p_z\|^2 + \|I_z - N(p_z)\|^2; \tag{3.7}$$

Where p_x, p_y, p_z are texels chosen form the respective templates textures T_x, T_y, Tz, while $N(p_i)$ represents 2D neighborhood of a texel p_i. Voxel color is finally assigned by averaging the candidate colors p_x, p_y, p_z.
The synthesis process starts with a block of noise and runs over voxels changing the colors. As in [WL00], by using gaussian pyramids, the entire process is performed in a multi-resolution fashion.
This method is simple to implement but, as shown in Figure 3.15, resulting textures exhibits blurring and usually don't preserve patterns which are presents in template textures.

In 2008 Dong et al.[DLTD08] propose a new method to synthesize solid textures called Lazy Solid Texture Synthesis.
The main advantage provided by this method is the possibility to synthesize texture in real-time, which make it particularly suitable for interactive simulations such as real time fracturing or cutting objects. More precisely two main characteristics make this method suitable for real time application:

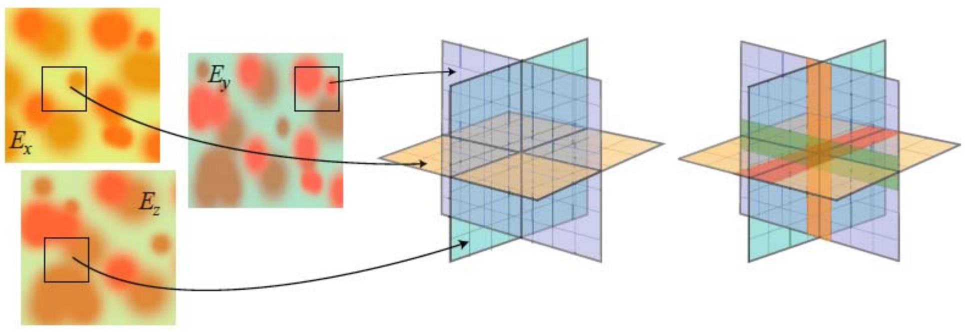


Figure 3.16: Left:Three exemplar composing a candidate. Right: The overlap region defined by a candidate. Courtesy of [DLTD08].

Parallelism The algorithm can be parallelized. The authors proposes a GPU parallel implementation that provides real-time synthesis.

Granularity of the synthesis Thanks to its locality, this algorithm can synthesize a small subset of voxels near to a visible surface instead of synthesizing the whole volume.

The main idea consists in performing synthesis on pre-computed sets of candidates. A candidate is essentially a 3D neighborhood created by selecting slices from the template images. Each candidate defines an overlap region (see Figure 3.16). The cardinality of possible candidates is huge if we consider that we can create candidates by combining triple of 2D neighborhood selected from template textures.
This space can be reduced by pruning candidates that produces color incoherences. More precisely, a candidate can be classified according to two metrics:

Color Consistency Measured as the coherence of a candidate along its overlap region. Based on similarity of colors, it is evaluated by summing squared color differences in the overlap region.

Color Coherence Which is the ability of the candidate to form coherent patches from template textures [Ash01]. This is evaluated by considering the amount of neighboring texels which forms contiguous patches.

The synthesis is performed in multi-resolution, from coarse to fine level, by using gaussian pyramids. Starting from an initial block, which is formed by tiling the best overlap regions (valuated using color consistency), the synthesis pipeline, as in [LH05], is divided into three main steps:

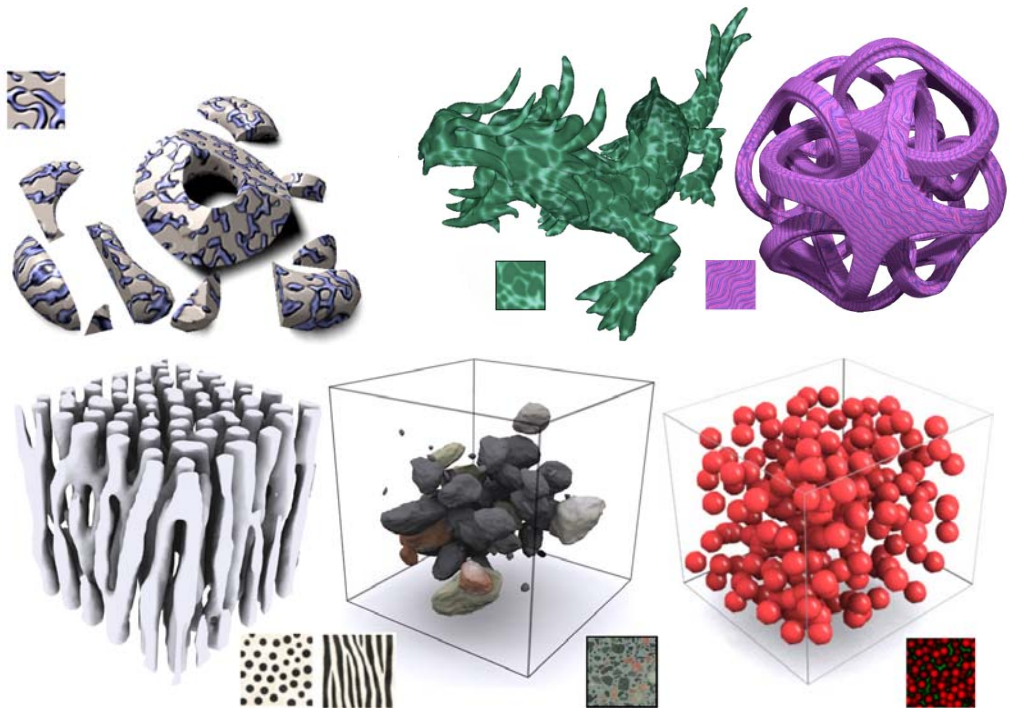


Figure 3.17: Some example of solid textures synthesized by Lazy Solid Texture Synthesis [DLTD08] using single or multiple exemplars.

Upsampling This step is used when the algorithm switch to a finer resolution level. Upsampling is simply realized by colors inheritance.

Jittering Introduce variance in the output data. It is realized by deforming colors in the solid texture.

Correction It makes the jittered data to look like template textures. It consists into searching for each voxel the candidate which is more similar to its 3D neighborhood, the searching phase is speeded by using PCA projection(Principal Component Analysis)[LH05].

As previously stated, thanks to the locality of the data involved by the process, it is possible to synthesize on demand a block of voxels instead of synthesizing the entire block. The granularity of the synthesis is limited by neighborhood size. It follows that, in the case we have to texturize a triangle mesh, we can to limit the synthesis to a solid shell following the surface.
As shown in Figure 3.17 this method produces nice results for a wide variety of input textures.

3.3.1.5 Optimization based methods

The 2D optimization based texture synthesis method [KEBK05](see Section 3.2.5 for details) has been extended by Kopf et al [KFCO$^+$07] to synthesize solid textures.
As in [KEBK05], the main goal of this method is to make the solid texture look like the 2D template texture by minimizing globally an energy function. Since EM energy minimization process proposed by [KEBK05] can stop in a local minima, provoking blurring or creating artifacts in the output, [KFCO$^+$07] proposes to improve convergency by interleaving minimization with histogram matching operations.
For the case of solid texture synthesis the global energy equation, expressed by 3.2, is reformulated in order to consider a 3D neighborhood:

$$E_T(v; \{e\}) = \sum_v \sum_{i \in \{x,y,z\}} \|S_i^v - E_i^e\|^r \tag{3.8}$$

where the voxel v iterate across the whole solid texture, S_i^v are 3D slice at voxel v, while E_i^e is the 2D neighborhood centered on texel e coming from template textures i. The exponent $r = 0.8$ makes the optimization more robust [KEBK05].
The terms of above equation can be rewritten as follows:

$$\|S_i^v - E_i^e\|^r = w_i^v \|S_i^v - E_i^e\|^2 \tag{3.9}$$

Where $w_i^v = \|S_i^v - E_i^e\|^{(r-2)}$. Equation 3.8 is then rewritten with respect to each single voxel belonging to S_i^v:

$$E_T(v; \{e\}) = \sum_v \sum_{i \in \{x,y,z\}} \sum_{u \in N_i^v} w_i^{v,u} (S_i^{v,u} - E_i^{e,u})^2 \tag{3.10}$$

By setting the derivative of 3.10 to zero, and assuming the weights $w_i^{v,u}$ are constant, it turns out that the optimal value for a voxel is simply the average of $E_i^{e,u}$, but this formulation can cause blurring in the synthesized texture.
To overcome this problem weights are recomputed by using histograms. More precisely, [KFCO$^+$07] proposes to reduce weights that increases the difference between the current histogram and the histogram of template textures. Minimization is realized by using again the same Expectation-Maximization process of the 2D case. Starting from a solid block initialized by choosing colors randomly from the template textures the synthesis is performed in multi-resolution. To enforce preservation of strong features it's possible to include a feature map [LH06] in the synthesis process.
The ability of this method to preserve sharp features is superior if compared with previous works (see Figure 3.18.(c)). Moreover, using an user-defined constraint map, it is possible to tune the minimization to create predefined patterns (see Figure 3.18.(b)).
Since the optimization is performed globally, this method requires that entire block is synthesized. Furthermore, the time needed for minimization process is high (from 10 to 90 minutes to synthesize a 128^3 block).

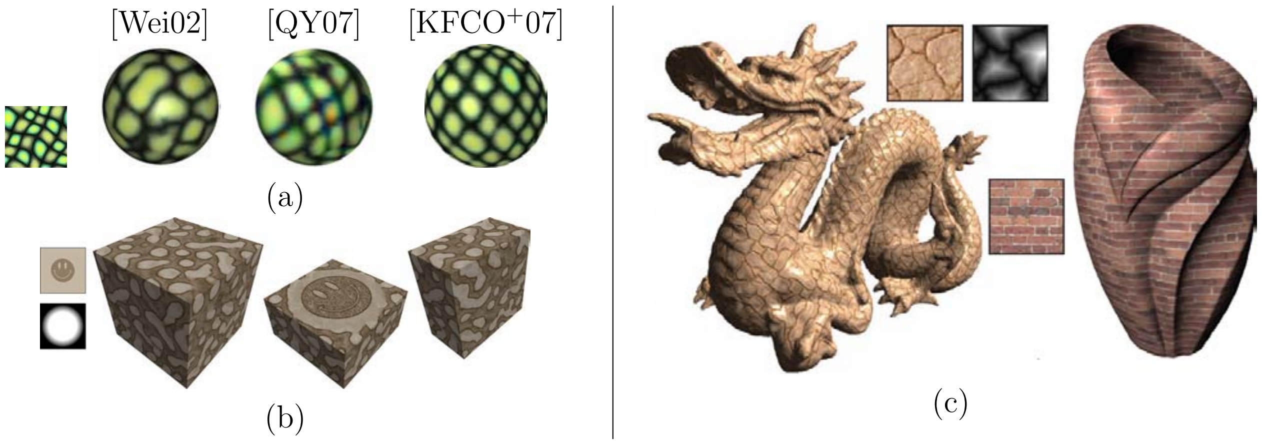


Figure 3.18: (a) Comparison of different methods in solid texture synthesis from 2D exemplar, [KFCO+07] preserves sharp features, while [QY07] and [KFCO+07] introduces blurring. (b) An example of constrained synthesis. (c) Examples of surfaces carved from a texture block synthesized using [KFCO+07].

3.3.2 Solid Meshes

Solid textures are not the unique way to model the interior of an object, alternative solutions were proposed in literature. Solid textures do not care about the object on which colors are projected. Alternatively object's volume can be used as domain on which the synthesis is performed, we call this class of approaches as *solid meshes.*
Main challenge of solid meshes consists in creating an appropriate representation of object's volume and use it as the synthesis domain.
Since in solid meshes the synthesis process is constrained by the mesh, it's possible to obtain interesting effects, like, for example, to orient the textures to follows mesh's shape, or to define a layered texture.
Those methods are, in general, semi-automatic: the user specifies some appearance property of the object and the system infers how to synthesize its interiors. Furthermore, those methods are fast and, contrarily to solid textures, they don't store explicitly the color of each voxel, but it is implicitly defined by the domain model.

We consider [CDM+02] to be the first example of solid mesh synthesis. In this method the interior of on object is defined by using a simple scripting language which allows to define nested textures. This effect is realized by using a signed distance field.
Despite the interesting results shown in the paper, textures are generated procedurally so the set of possible appearances is limited. Furthermore, defining internal textures by using a scripting language is not user-friendly. Other methods that we classified as solid meshes synthesis are [ONOI04] and [TOII08], which will be described in the next subsections.

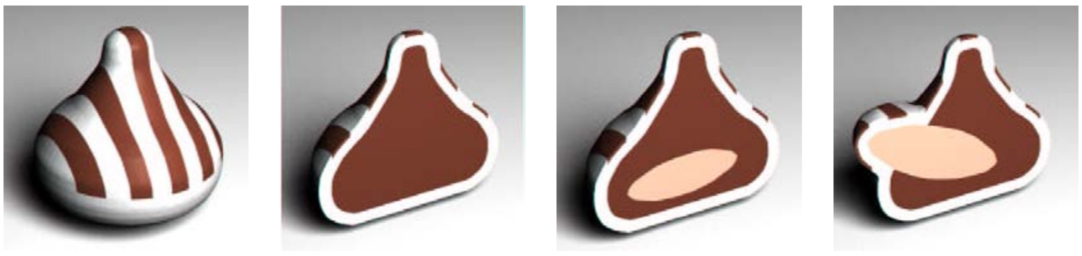


Figure 3.19: Examples of layered textures created by [CDM+02].

3.3.2.1 Volumetric Illustrations

In 2004 Owada et al. proposes a novel method to create solid meshes [ONOI04].
The user specifies the interior of an object by using a *browsing interface* and a *modeling interface*.
The browsing interface is a model viewer which allows the user to visualize the internal structure of an object (See Figure 3.20.a). The interface is simple and intuitive: the user freely sketches 2D path lines on the screen [IMT99] to specifies the direction along the object has to split. This paths were projected on the 3D mesh to define the cross-section. Once the surfaces is split in two parts, its internal surface is re-triangulated, according to the split section, such that internal appearance can be finally rendered [ONNI03].
The modeling interface provides an intuitive way to specify the internal structure of the model. When an object reveals its internal surface it is possible to specify a texture for each closed volumetric region. It allows, for example, to define multiple appearances in the case that the domain contains multiple closed regions (see Figure 3.20.e). That information is used, together with the triangle mesh, to perform synthesis on cross-sections. More in details, once each region of the mesh is linked with the respective template texture, then cross sections can be synthesized on-the-fly using a 2D synthesis algorithm. That operation requires the parametrization of cross-section, since no volumetric textures were created.
The system allows to use three different kinds of textures:

Isotropic textures This kind of textures does not care about the mesh, they can simply synthesized in the parametric space of a cross-section using a standard 2D synthesis algorithm [WL00] (see Figure 3.20.b).

Layered Textures The appearance of this texture changes according to the depth. A smooth 2D depth field is calculated in the cross section [TO99], then the synthesis is performed again using 2D texture synthesis algorithm [Tur01, ZZV+03] which allows to distort the texture according to the depth field (see Figure 3.20.c).

Oriented textures This texture has distinct appearances in cross-sections that are per-

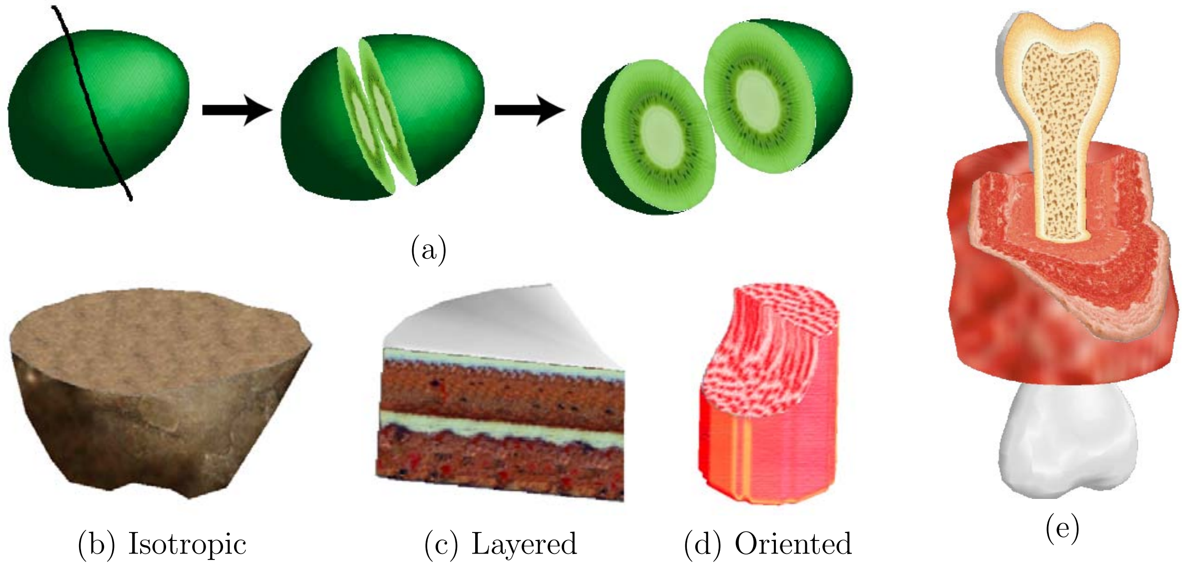


Figure 3.20: (a) The browsing interface of [ONOI04]. (b) Isotropic texture. (c) Layered texture (d)Oriented texture. (e)Example of subdivided domain (bone and meat) modeled by [ONOI04].

pendicular and parallel to a flow orientation (One example of oriented texture is shown by Figure 3.20.d). The user defines by sketching a main flow direction, the system use this vector to orient a 3D flow field defined into the volume, which is calculated by using [TO99].
A reference volume is synthesized simply by sweeping the texture image along the y direction. This reference volume is used, together with the 3D flow field, to "texturize" properly the cross-section. A 2D pixel-based texture synthesis technique [WL00] is used to synthesize colors of the parameterized cross-section. To make the synthesis process dependent on surface's orientation, the neighborhood search step is modified as follows: Given a pixel p, with normal n, the set of candidates used in coherent search, is formed by slicing patches from the reference volume along planes orthogonal to n.

As shown by Figure 3.20, the user easily produces nice results with a few mouse clicks. Thanks to the user-friendly interface, this method is an interesting solution to produce nice scientific illustrations (useful in medicine, biology or geology).
However the expressing power of this method is limited.

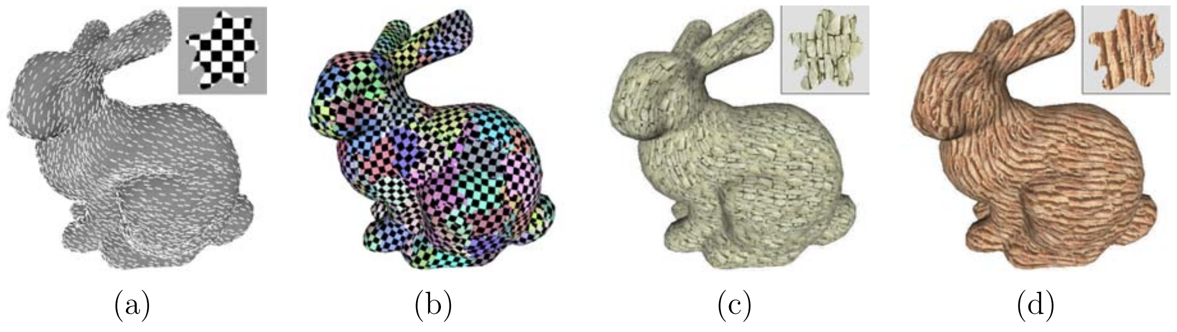


Figure 3.21: 2D Lapped textures [PFH00]: (a) The continuous tangent field and the 2D template patch. (b) The local surface parametrization. (c)&(d) Some results.

3.3.2.2 Lapped Solid textures

Lapped textures[PFH00] is a technique to synthesize textures on surfaces. It consists mainly into overlapping properly an irregular patch to cover the entire surface. Figure 3.21 illustrate how the lapped textures works: by using a continuous tangent field and a 2D template patch (Figure 3.21.a), the surface is locally parameterized (Figure 3.21.b), so that it's possible to texturing it by repeatedly pasting patches (Figure 3.21.c and Figure 3.21.d). The method does not require to store explicitly the color, since it is implicitly defined by texture coordinates.

Takayama et al.[TOII08] propose to extended Lapped Textures to fill volumes instead of surfaces. The basic concepts behind the 2D and the 3D version are similar. As we previously introduced, 2D Lapped Textures paste irregular patches over triangles, similarly 3D Lapped Textures paste solid patches over tetrahedrons. Moreover, 2D Lapped Textures uses a tangent field on the surface to orient textures, similarly 3D Lapped Textures uses a smooth tensor field (three orthogonal vector fields) along the volume to arrange solid patches. Furthermore, like in 2D Lapped Textures, 3D Lapped Textures requires to store only the 3D texture coordinates of each vertex belonging to the tetrahedral mesh. Finally in order to avoid artifacts in the final texturing, [PFH00] proposes to use a ”splotch” alpha mask (As shown by Figure 3.21), while in 3D lapped texture a volumetric alpha mask is used to make solid texture have a ”splotch” shape.
[TOII08] classifies solid textures by considering both their anisotropy and variation (See Figure 3.22). *Anisotropy level* describes the appearance of a cross section varying with respect to the orientation of the slice, while *Variation level* express the number of directions along which the texture changes. The tileability of the texture depends on the variation level, i.e. a solid block is tileable along the directions that preserve the appearance.
The user first select the appearance class (according to table) he wants to model. Then if

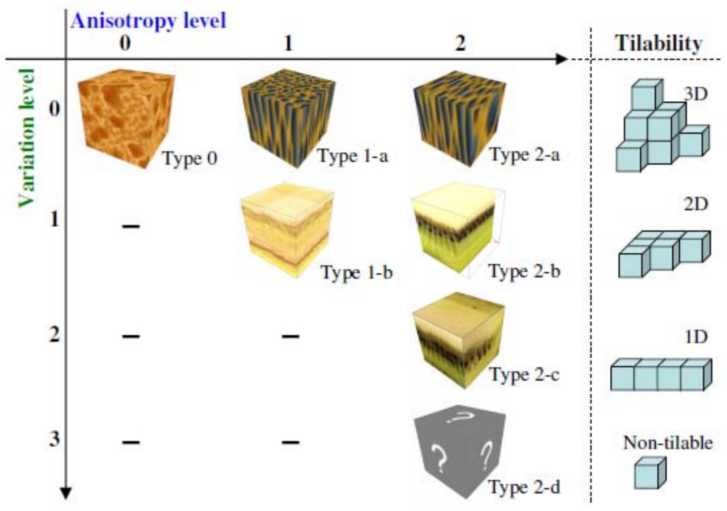


Figure 3.22: Classification of solid texture appearance according to [TOII08].

required by texture class, specifies directions by sketching strokes (The interface is similar to [ONOI04]).
The system creates a consistent global tensor field to make texture follow the orientations in the case the solid texture is anisotropic. The tensor field is calculated by Laplacian smoothing of user-defined direction along the tetrahedral mesh (see [TOII08] for details). Then, the algorithm can be summarized as follows: Initially a patch is pasted by the user in the object's volume, then tetrahedrons that are inside the alpha mask were marked as "covered", then the "covered" region is expanded until it include the entire tetrahedral mesh, by repeatedly pasting patches.
Each pasting operation implies that covered tetrahedrons must be transformed in texture space according to the tensor field, such that it is possible to assign per-vertex 3D texture coordinates.

This method has a nice formulation, it can model a wide variety of textures (as shown by Figure 3.23), and requires low memory consumption. However, it has one strict requirement: the initial set of solid textures has to be provided a priori, together with relative alpha masks.

3.3.3 Solid Textures vs Solid Meshes

Solid texture synthesis has both advantages and disadvantages with respect to solid meshes, we list advantages and drawback of both approach in table 3.1.

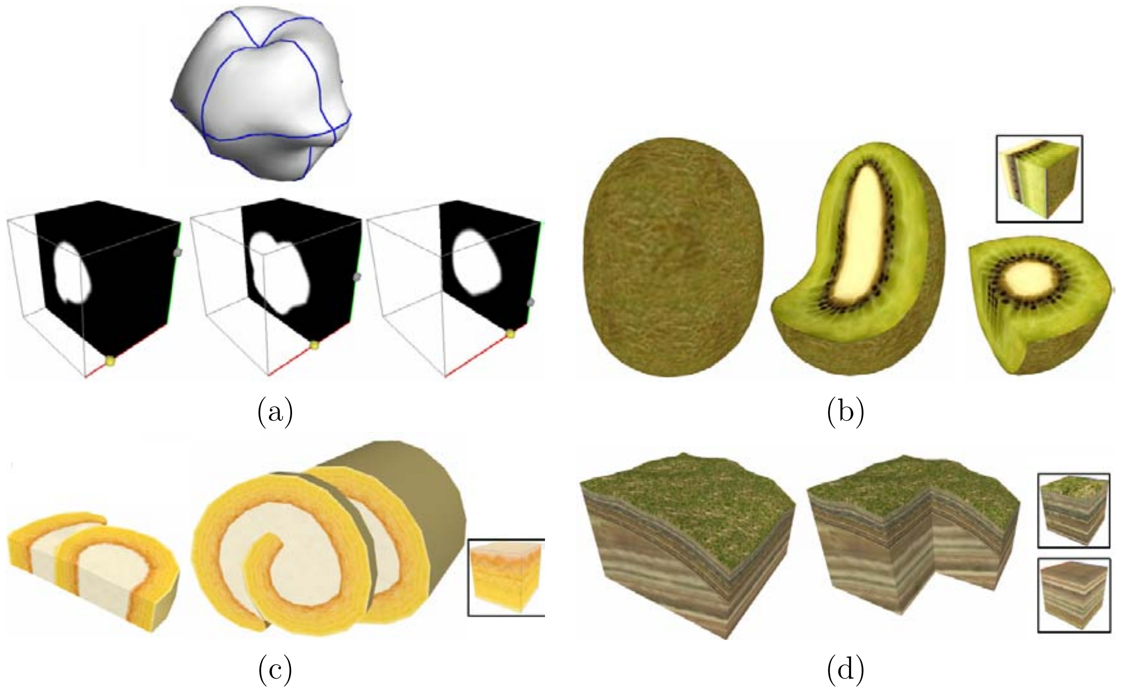

(a) (b) (c) (d)

Figure 3.23: Lapped Solid Textures [TOII08]: (a) The alpha mask used to modify the shape of volumetric sample used for synthesis process. (b) & (c) & (d) Examples of results.

Solid textures introduces no distortion, while solid meshes can produces distortion depending on the surface. Furthermore solid textures can be reused to color every surface, while solid meshes are adaptable only to a single surface. Solid texture creation is completely automatic, while solid meshes requires a minimal user-interaction.
On the other hand, solid texture are no customizable. At the moment none of the techniques allows to orient textures along object's shape, or to produce layered textures. While this effects are producible by using solid meshes. Moreover, solid textures requires a huge memory to store data, which grows cubically with respect to the amount of edge size. Solid meshes usually just store the minimal amount of information used to retrieve colors

	Solid Textures	Solid Meshes
No Distortion	+	-
Versatile	+	-
Completely Automatic	+	-
Customizable	-	+
Memory consumption	-	+
Preprocessing time	-	+

Table 3.1: Comparison between solid textures and solid meshes.

on demand. Finally to produce a solid texture requires, in general, a long preprocessing time ([DLTD08] which uses GPU is an exception), while solid meshes usually just need some seconds.

3.4 Texturing Internal surfaces from a few cross sections

We introduce a new texture modeling paradigm. As shown in Figure 3.24, the input data to our modeling paradigm consists of the boundary representation of a 3D model, plus a few photographs of cross-sections of a real object, which we refer to as exemplars. This data is sufficient for defining plausible appearance properties of internal structure at any point inside the 3D model and, as a result, we can generate instances of the 3D model that reveal internal surfaces with highly realistic texture.

Our modeling paradigm can be applied for carving other 3D models out of organic objects, interactively texturing cut surfaces in virtual cutting or fracture simulations, transferring internal textures of real objects to arbitrary 3D models without topology restrictions, or producing artistic combinations of internal textures from different objects.
We also relax the requirement of full cross-section photographs, allowing the user to input partial samples or even synthetic exemplars. With our approach, it would be possible to compute a full 3D volumetric texture and then synthesize an internal surface as a cut on the 3D texture.
However, the computation of the 3D texture is not a requirement. We have designed an efficient algorithm for on-demand computation on a per-point basis inside the 3D model, which, in our implementation, yields a throughput of 20000 points/sec. Our on-demand solution allows for a texture resolution that is not limited by storage, at most by the resolution of the input exemplars.

Our texture morphing algorithm rests on two main technical contributions.
The first contribution is the definition of a domain and a procedure for interpolating appearance properties from planar oriented exemplars in 3D. We observe that splitting a physical object with planar cuts produces a *binary space partitioning* (BSP) [FKN80] of the object, thus we naturally employ a BSP tree for decomposing our interpolation domain. Then, we employ curved projections and scattered data interpolation for defining source points for interpolation and their weights.
The second contribution is an efficient single-point multi-exemplar morphing algorithm, inspired on the elegant warping-based morphing algorithm of Matusik et al. [MZD05], with notable differences. We adapt the computation of texture warpings to our BSP-based interpolation domain, and we present an efficient method for computing inverse warpings on a per-point basis through first-order approximation, plus a local orientation-aware histogram matching procedure for feature enhancement.

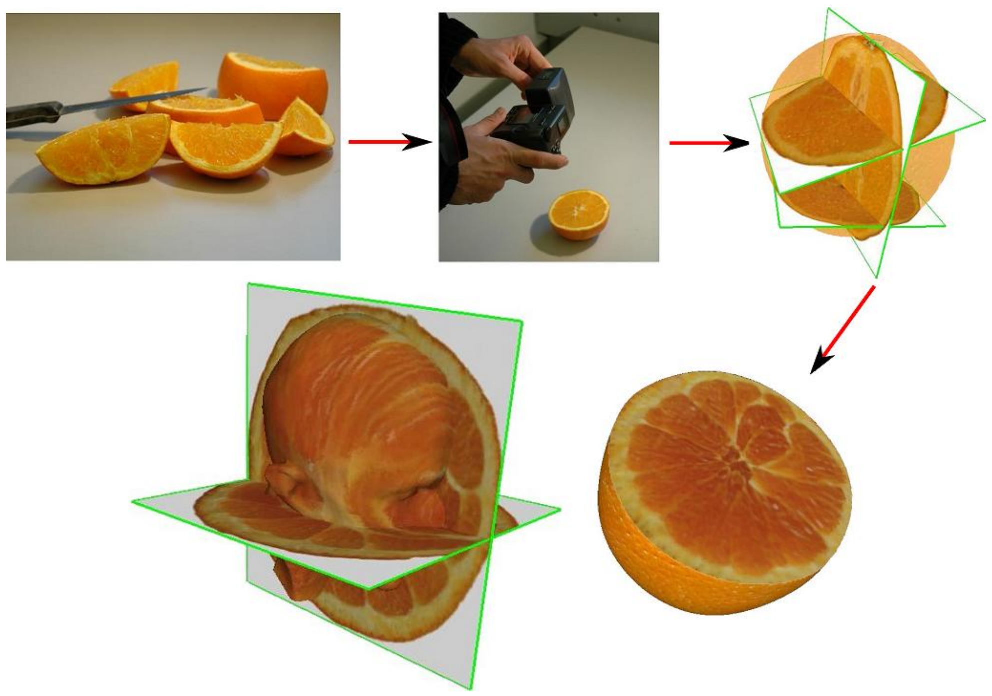


Figure 3.24: Our Paradigm for Digital Content Creation: We capture images of internal surfaces of real objects and, in an interactive editor, we place them in the local reference frame of a 3D model. Relying on our two main technical contributions (a natural interpolation domain for object cross-sections and a novel texture morphing algorithm), we can, among other results, produce models that reveal internal surfaces with highly realistic texture, or carve 3D models out of organic objects.

We implement our modeling paradigm through an interactive and very intuitive editor where the user can place the exemplars inside the 3D model, trigger the preprocessing of the data, and then use the precomputed data for the actual on-demand computation of internal textures.

3.4.1 Texture Synthesis Pipeline

The synthesis of 3D textures from cross-section images can be decomposed into two procedures:

- At *runtime*, texture colors are synthesized on arbitrary 3D positions from predefined model cross-sections through a morphing operation.
- As a *preprocessing* step, the user must collect input images, set them up together

with 3D geometry in an interactive 3D framework, and compute data necessary for the runtime morphing.

In this section, we outline both the runtime and preprocessing tasks. Section 3.4.2 describes the interpolation domain in which we perform texture synthesis through morphing, while section 3.4.3 describes the morphing algorithm itself.

In cases where the texture of internal surfaces must be stored for repeated use, our synthesis method must be accompanied of parametrization and packing of surface triangles in a texture atlas. We have not aimed at optimizing these steps, we have simply packed triangles individually in an area-preserving manner [CH02].

3.4.1.1 Runtime Texture Synthesis

Our texture synthesis algorithm produces the color of one target point at a time. Given the 3D position $\mathbf{p}$ and orientation $\mathbf{n}$ of the target point, together with a set of input exemplars $\{\Omega_1, ..., \Omega_n\}$, the synthesis function $\mathbf{c} = f(\mathbf{p}, \mathbf{n}, \Omega_1, ..., \Omega_n)$, outputs the color $\mathbf{c}$ of the point.

The texture synthesis is executed on an interpolation domain $\mathbb{D}$, which typically constitutes the interior of a 3D model $\mathcal{M}$. Given a target point $\mathbf{p} \in \mathbb{D}$, we classify its location in a binary space partitioning (BSP) of $\mathbb{D}$, constructed as described in §3.4.2.1. Each region of the BSP is defined by a subset of exemplars, which are employed for synthesizing the output color $\mathbf{c}$.
We identify a source point $\mathbf{p_i}$ and an interpolation weight w_i for each exemplar, through projection and scattered data interpolation, as described in detail in §3.4.2.2. Once we know the source point and the weight for each contributing exemplar, we apply texture morphing as described in §3.4.3.2.
Our morphing algorithm is an extension of the work of Matusik et al. [MZD05], with notable differences. In our setting, the source points are different on each exemplar image, and each target point must be synthesized independently, as the contributions of the exemplars vary according to its position. Hence, we have designed an optimized algorithm for synthesizing the color of each target point on demand, allowing for real-time synthesis of thousands of pixels. Nevertheless, we account for the effect of orientation on small-scale features (See §3.4.3.3), and we ensure texture continuity across spatially-adjacent target points.

3.4.1.2 Preprocessing of Input Exemplars

The preprocessing is composed of four main tasks:

- Generating exemplars and placing them in the interpolation domain $\mathbb{D}$ (See §3.4.2.3).
- Constructing the BSP of the interpolation domain $\mathbb{D}$ (See §3.4.2.1).
- Performing precomputations (e.g., feature detection, histograms, etc.) on each of the exemplars independently (See §3.4.3.3 and §3.4.3.1).
- Defining pairwise warping functions between exemplars that bound common BSP regions (See §3.4.3.1).

One major design desiderata for our algorithm was to rely on a small amount of data (in contrast to the richness of possibilities that our synthesis algorithm can produce).
Therefore, we allow for some user interaction in order to optimize the quality of the input data at the preprocessing stage. The user has the possibility to interactively place exemplars in the interpolation domain (§3.4.2.3), analyze and judge the need for further input exemplars (§3.4.2.3), or introduce a priori knowledge about the textures for guiding the interpolation (§3.4.2.2).

Thanks to the versatility of our algorithm, exemplars may be photographs of cross sections of a real object, or synthetic images. Similarly, photographs of a certain object can be used for synthesizing textures on yet a different object, enabling efficient 3D texture transfer.

3.4.2 Texture Interpolation Domain

In this section, we describe the construction of the BSP of the interpolation domain $\mathbb{D}$ using exemplars, among with how interpolation is performed inside each region of the BSP. Then, we propose approaches for defining input exemplars and placing them in the interpolation domain.

3.4.2.1 Binary Space Partitioning

As mentioned earlier, we perform texture synthesis on a domain $\mathbb{D} \subset \mathbb{R}^3$ corresponding to the interior of a model $\mathcal{M}$. Using a sorted sequence of input exemplars $\{\Omega_1, ..., \Omega_n\}$, we construct a BSP of $\mathbb{D}$ in the following way.
We assume that each exemplar Ω_i constitutes a planar region in $\mathbb{D}$. Then, given a subsequence of exemplars $\{\Omega_1, ..., \Omega_i\}$, defining BSP regions $\{\mathbb{D}_1, ..., \mathbb{D}_m\}$,the addition of the next exemplar Ω_{i+1} subdivides one of the regions, $\mathbb{D}_j$, into two new ones.
Figure 3.25 shows the BSP of a bunny model produced by a set of exemplars. It is important to highlight that a BSP is actually a natural choice as data structure in our application. Cutting a solid object by successive bisection of one of the existing pieces with a planar

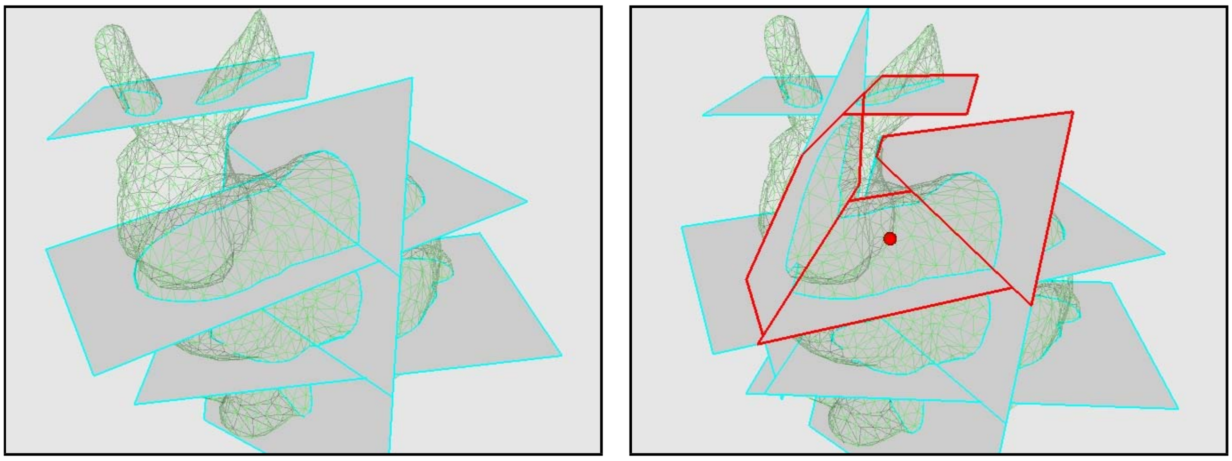


Figure 3.25: *BSP-Tree* Produced by Exemplars: (a)The intersection of the top exemplar plane with the ears of the bunny yields two connected components. (b)This situation is fixed by adding another exemplar. The exemplars bounding the target point (in red) are highlighted.

cut produces indeed a BSP of the original object.
Each region $\mathbb{D}_j$ of the BSP of $\mathbb{D}$ is bounded by curved surfaces $\{\mathcal{S}_i\} \in \partial\mathbb{D}$, and planar exemplar subdomains $\{\Omega_{i,j} = \Omega_i \cap \mathbb{D}_j\}$. We enforce the user to produce exemplar subdomains $\{\Omega_{i,j}\}$ topologically equivalent to a disk. If this condition does not hold, the user must introduce additional exemplars, as shown in Figure 3.25. In §3.4.3.1, we discuss the computation of pairwise warpings between exemplar subdomains.

3.4.2.2 Source Image Points and Interpolation Weights

In order to synthesize the color at a point $\mathbf{p}$ in the BSP region $\mathbb{D}_j$, the texture morphing algorithm takes as input source points on the exemplars that bound $\mathbb{D}_j$, along with associated interpolation weights. Here, we describe an algorithm based on (possibly curved) projections for finding the source points and computing their weights.

Conceptually, we define the source point $\mathbf{p}_i$ of an exemplar subdomain $\Omega_{i,j}$ by projection of $\mathbf{p}$ onto $\Omega_{i,j}$ along a path line $\gamma(\mathbf{p}_i)$ emanating from $\Omega_{i,j}$ and flowing through $\mathbb{D}_j$.
If no assumption is made on texture isotropy, there is no ideal projection scheme a priori. Any method producing smooth, evenly distributed path lines emanating from $\{\Omega_{i,j}\}$ and covering the entire subdomain $\mathbb{D}_j$ would be valid.
In our implementation, we have adopted the following approach. We compute the barycenter of the exemplar $\Omega_{i,j}$, we define a curve γ emanating from the barycenter, and we sweep and rotate the exemplar plane along γ, until it contains the target point $\mathbf{p}$. In this configu-

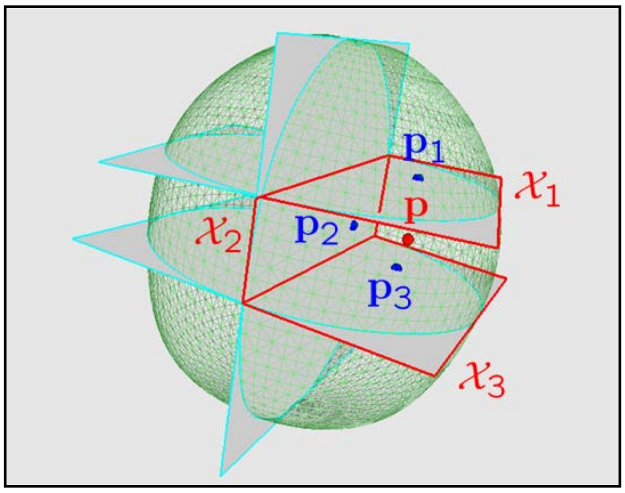


Figure 3.26: Interpolation Inside BSP Regions: Given a target point $\mathbf{p}$, we define source points $\mathbf{p}_i$ on the exemplars Ω_i that bound the BSP region where $\mathbf{p}$ is located.

ration, we compute the ray from the swept barycenter to $\mathbf{p}$, and we map this ray onto $\Omega_{i,j}$. We define the location of the source point $\mathbf{p}_i$ along the mapped ray by preserving the ratio of distances w.r.t. the barycenter and the boundary of the subdomain in the original and swept configurations. If we can assume no knowledge about texture isotropy, we define γ as the line normal to the exemplar. In cases such as the oranges in Figure 3.24, we exploit a priori knowledge by defining γ as a great arc on the sphere.

Given the set of source points $\{\mathbf{p}_i\}$, we define interpolation weights for morphing based on scattered data interpolation. Specifically, we compute the (pre-normalized) weight w_i of each source point $\mathbf{p}_i$ based on Shepard interpolation: $w_i = \frac{1}{\|\mathbf{p}-\mathbf{p}_i\|}$. In BSP regions that do not lie on the boundary of $\mathbb{D}$, the source points $\{\mathbf{p}_i\}$ define a convex polyhedron that bounds $\mathbf{p}$, and then it is also possible to use convex weights given by e.g., mean-value coordinates [Flo03].

Our interpolation procedure is guaranteed to be continuous inside regions $\mathbb{D}_j$ of the BSP, as well as across regions. The projection operation and the interpolation weights are all continuous inside a given region. When a BSP boundary is crossed, the interpolation is dominated by a single source point, thus ensuring continuity as well. Since the morphing algorithm described in §3.4.3 is also continuous, the complete texture synthesis procedure is C^0 continuous. Of course, the features of the input exemplars may not be continuous, therefore we do not enforce continuity on the output textures.

3.4.2.3 Defining Input Exemplars

The generation of input data for the synthesis algorithm encompasses:

- The definition of texture attributes for the exemplars.
- The definition of a polygonal representation of the model $\mathcal{M}$.
- The placement of exemplar planes in the domain $\mathbb{D}$.

We typically collect exemplars by taking photographs of cross sections of real objects, but it is also possible to let an artist define exemplars and their attributes. For example, Figure 3.30 shows a case where exemplars were generated using 2D texture synthesis techniques [LH05].
When trying to simulate the textures of a real object, the model $\mathcal{M}$ should approximate the boundary of the real-world object that is cut for generating the exemplars. High quality results could be generated by scanning the real-world object, but a coarse approximation (such as the orange model used in Figure 3.24) proved to be enough in our examples. Once the representation of $\mathcal{M}$ is available, we provide the user with an interactive tool for placing exemplar planes in $\mathbb{D}$. The user may typically judge how many exemplars to add based on visual examination of the textures, but we have also incorporated a warping quality metric [MZD05] to aid with this judgement.

Since the model $\mathcal{M}$ may not exactly correspond to the real cut object and it is very hard to accurately place the exemplars, we allow the user to place the exemplars approximately, and then we warp them so that the texture images exactly fit the boundary of $\mathcal{M}$. In our implementation, we constrain the image boundaries to the boundary of $\mathcal{M}$ and we perform a relaxation process in the interior.

3.4.3 Texture Morphing

In this section, we describe our algorithm for synthesizing the color of a 3D point as a *morphing operation.*
First, we summarize the texture morphing algorithm of Matusik et al. [MZD05], which comprises a preprocessing part for computing image warpings, and the actual runtime morphing algorithm.
Our algorithm presents differences in the preprocessing of warpings, such as finding multilevel feature correspondences, and allowing for user interaction. But, most importantly, our runtime morphing algorithm is designed for efficiently synthesizing the texture of individual 3D points *on-demand.* This aspect is essential when synthesizing texture on the

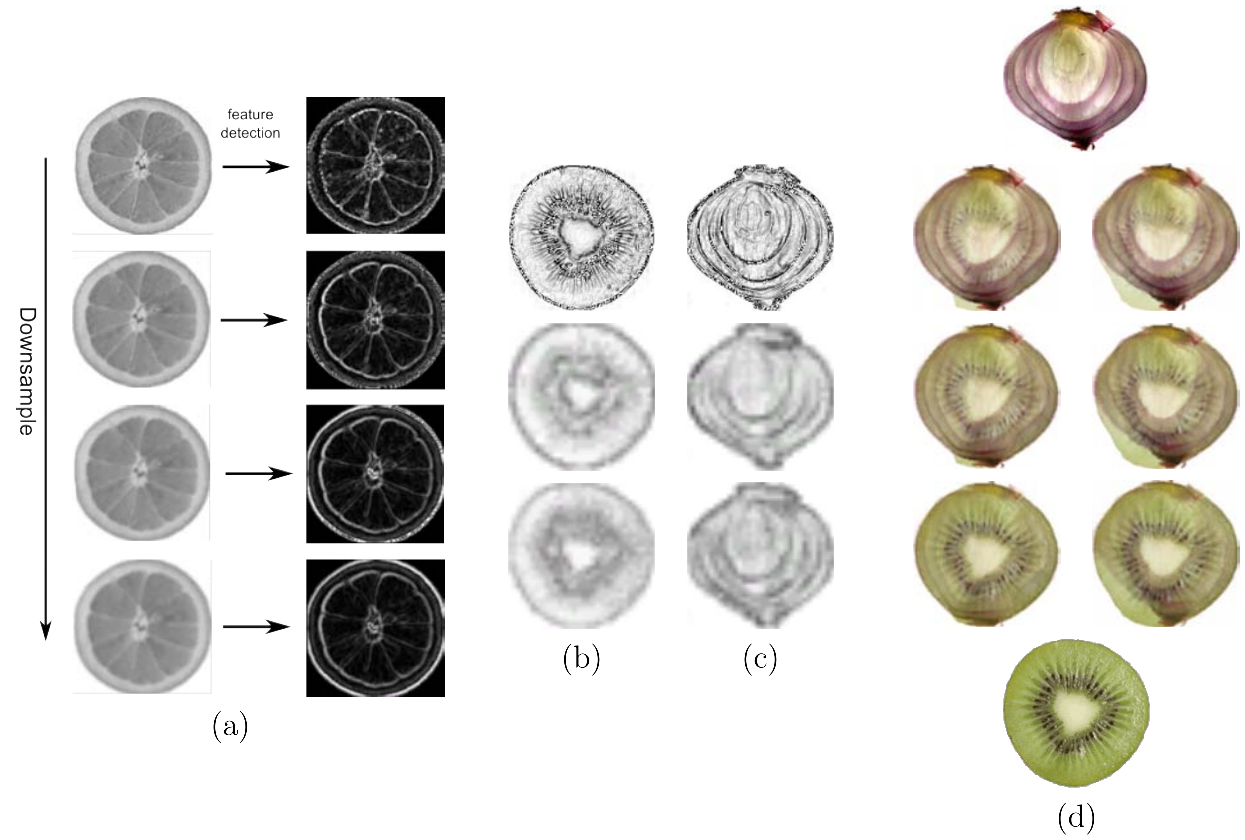


Figure 3.27: Morphing Using Gaussian Stacks: (a) Computation of gaussian stack. (b)&(c) Feature maps for a kiwi and an onion. On the top, feature map at full resolution; on the center, feature map at a level of our Gaussian stack; and on the bottom, feature map at the same level without Gaussian stack. Notice the blurry region in the center of the kiwi's feature map. (d)Morphing with our Gaussian stack (left) vs Morphing without Gaussian stack (right), which noticeably increases blur.

internal surface of an object, as no pair of surface points shares interpolation weights for morphing.
In the second part of this section, we introduce the definition of warping in our interpolation domain, and we describe an approximation to the inverse warping that allows for fast morphing at a single point at a time. Moreover, we present an approach for feature enhancement based on local high-frequency histogram computation.

3.4.3.1 Morphing Images

Given a set of input images $\{\Omega_i\}$, Matusik et al. [MZD05] defined morphing as a convex combination of warped versions of the images. Hence, their algorithm relies heavily on the computation of a warping for each pair of images (Ω_i, Ω_j). They defined a bijective mapping $f_{ij} : \Omega_i \subset I\!R^2 \rightarrow \Omega_j \subset I\!R^2$, such that a point $\mathbf{p}_i \in \Omega_i$ maps to a point $f_{ij}(\mathbf{p}_i) \in \Omega_j$, where $f_{ij}(\mathbf{p}_i) = \mathbf{p}_i + \mathcal{W}_{ij}(\mathbf{p}_i)$. The 2D vector $\mathcal{W}_{ij}$ is referred to as the *warping*, and can be regarded as a translation.

3.4.3.1.1 Computation of Warpings Matusik et al. followed these steps to compute each mapping f_{ij}:

- Apply an edge detector [RT01] to Ω_i and Ω_j to compute feature images.
- Compute a stack of feature images by downsampling.
- Perform multilevel feature matching by minimization of the Euclidean norm of feature image differences, together with a regularization term that measures image deformation.

In our setting, we first need to align pairs of exemplars if they share a boundary, and we scale them such that their axis-aligned bounding boxes match.
As opposed to Matusik et al., we apply multilevel feature detection, by computing Gaussian stacks [LH05] of the input exemplars, and applying the edge detector and further downsampling to each image of the Gaussian stack independently. Figure 3.27 shows the improvement obtained in the morphing by incorporating the Gaussian stack.
We perform feature matching through an iterative optimization of Matusik's matching energy, but we also allow for user interaction to add a priori knowledge into the process. The regularization term in the matching energy measures the norm of the Jacobian for each warped triangle. If the user considers that the warping is mostly dominated by a global scaling or rotation, he/she may estimate this global transformation and remove it from each triangle's Jacobian using polar decomposition [HS88]. We also let the user grab specific vertices and constrain them, while the regularization term guarantees a smooth warping.

3.4.3.1.2 Computation of Morphed Images Based on all pairwise mappings, Matusik et al. computed the morphed image from convex color interpolation weights α_i and convex warping interpolation weights β_i. In order to synthesize the color $\mathbf{c}$ at pixel $\mathbf{p}$, their algorithm performs a convex combination of the images evaluated at pixels $\mathbf{q}_i$, $\mathbf{c}(\mathbf{p}) = \sum_i \alpha_i \mathbf{c}_i(\mathbf{q}_i)$, where each $\mathbf{q}_i - \mathbf{p}$ represents the inverse warping of $\mathbf{p}$ based on the convex combination of warpings. In other words, $\mathbf{q}_i$ is the pixel in image Ω_i that maps to $\mathbf{p}$: $\mathbf{q}_i + \sum_j \beta_j \mathcal{W}_{ij}(\mathbf{q}_i) = \mathbf{p}$. The complete morphing function can then be expressed using the concept of inverse warping as:

$$\mathbf{c}(\mathbf{p}) = \sum_i \alpha_i \mathbf{c}_i(\mathbf{p} + \left(\sum_{j \neq i} \beta_j \mathcal{W}_{ij} \right)^{-1} (\mathbf{p})). \tag{3.11}$$

The evaluation of the inverse warping requires that each warping $\mathcal{W}_{ij}$ must be scaled by its associated weight, and the weighted sum is then computed over the complete image, searching for the pixel that maps to $\mathbf{p}$. This search can be implemented, e.g., by rasterization of the warped mesh with original locations as attributes. When the morphing algorithm is applied to a complete texture image, the cost of computing the inverse of the scaled warping is amortized over all target pixels. However, this approach is far from optimal when the interpolation weights vary for each target point, as is our case. In §3.4.3.2, we describe our optimized solution for single-point synthesis.
Matusik et al. complete the morphing process by applying a histogram matching for feature enhancement [HB95]. Unlike in our method, Histogram computation also presents a cost dependent on the size of the exemplars, which is amortized for complete texture synthesis. In §3.4.3.3, we again present an optimized solution for single-point synthesis.

3.4.3.2 Single-Point Multi-Exemplar Morphing

We now present the definition of warping in our BSP-based interpolation domain, together with an efficient approximation of the inverse warping that can be explicitly evaluated.

3.4.3.2.1 Warping in the Interpolation Domain In the interpolation domain $\mathbb{D}$, morphing takes place among different source points $\mathbf{p}_i$ for each exemplar. For two exemplars (Ω_i, Ω_j), the warping vector $\mathcal{W}_{ij}$ cannot be defined as the difference vector between a point $\mathbf{q} \in \Omega_i$ and its corresponding point $f_{ij}(\mathbf{q}) \in \Omega_j$. Instead, we account for the translation between the reference systems of the two exemplars, and we define the warping as:

$$\mathcal{W}_{ij}(\mathbf{q}) = (f_{ij}(\mathbf{q}) - \mathbf{p}_j)) + (\mathbf{p}_i - \mathbf{q})\,. \tag{3.12}$$

As noted in §3.4.3.1, it is highly inefficient to compute the exact inverse warping in the context of single-point morphing, but we have devised an efficient approximation, presented next.

3.4.3.2.2 Efficient Warping Approximation For texture morphing in the interpolation domain $\mathbb{D}$, we slightly modify (3.11). The source points $\{\mathbf{p}_i\}$ vary across exemplars,

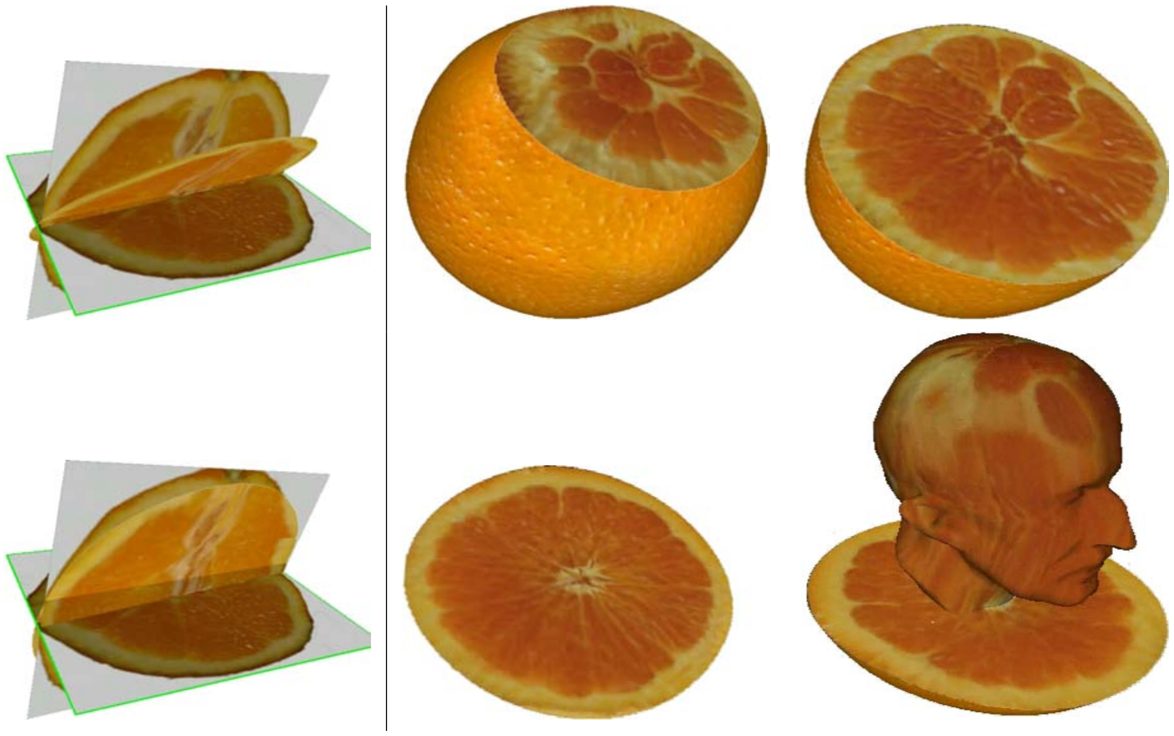


Figure 3.28: Showing the internal appearance of an orange.

and we use the same weights $\{w_i\}$ for color and warping interpolation. Then, we obtain the following morphing equation:

$$\mathbf{c}(\mathbf{p}) = \sum_i w_i \mathbf{c}_i(\mathbf{p}_i + \left(\sum_{j \neq i} w_j \mathcal{W}_{ij}\right)^{-1} (\mathbf{p}_i)), \tag{3.13}$$

where $\mathbf{q}_i - \mathbf{p}_i = \left(\sum_{j \neq i} w_j \mathcal{W}_{ij}\right)^{-1} (\mathbf{p}_i)$ is the inverse warping of $\mathbf{p}_i$.
In order to find the inverse warping on an exemplar Ω_i, we approximate the warping from each other exemplar $\Omega_j, j \neq i$ based on the value at the corresponding point of its source point, $\mathbf{p}_j$.
In other words, $\mathcal{W}_{ij}(\mathbf{q}) \approx \mathcal{W}_{ij}(f_{ji}(\mathbf{p}_j)) = \mathbf{p}_i - f_{ji}(\mathbf{p}_j)$. From this approximation, we reach the estimate for the inverse warping in (3.13), which yields:

$$\mathbf{q}_i \approx w_i \mathbf{p}_i + \sum_{j \neq i} w_j f_{ji}(\mathbf{p}_j). \tag{3.14}$$

The approximation results in the evaluation of the convex combination of the source texel $\mathbf{p}_i \in \Omega_i$ and the corresponding points of all other source points $\mathbf{p}_j \in \Omega_j, j \neq i$. Remarkably, this approximation produces the accurate result if one of the weights $w_k = 1$, and this contributes to the continuity of the morphing as the source point crosses boundaries of BSP regions. Our approximation obviously does not yield the same morphs as using the exact warping, as the warping is a highly nonlinear function, but our approach produces plausible, sharp results.

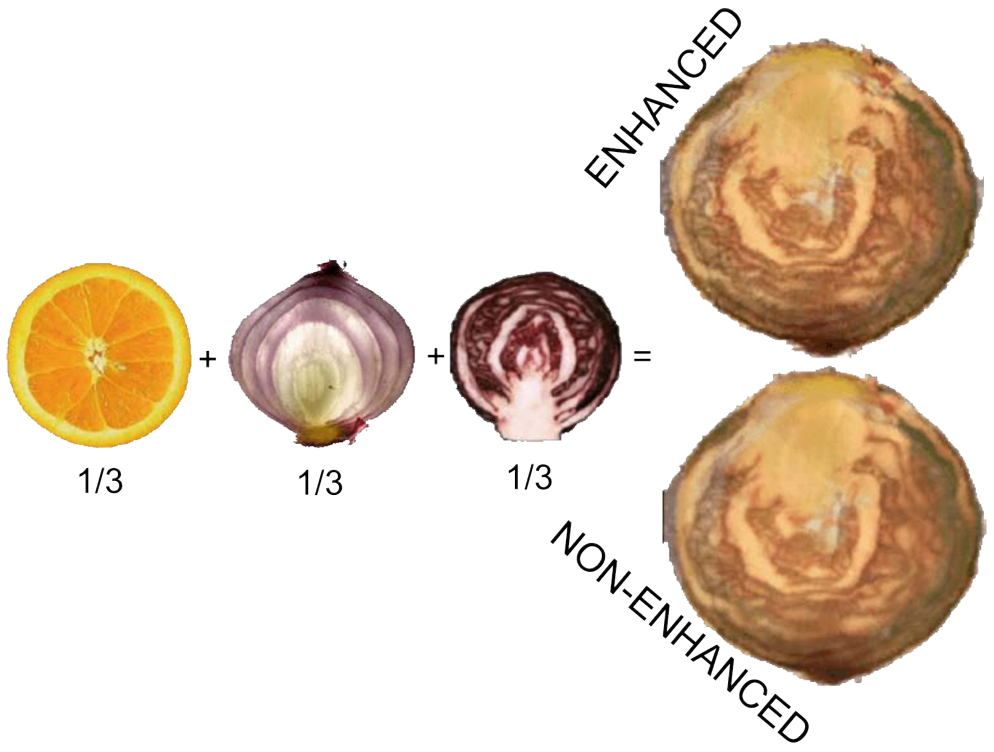
3.4.3.3 Feature Enhancement

The morphing algorithm we just described inevitably produces a certain blending of the source exemplars. It exploits the warping for morphing large and medium scale features effectively, but small scale features are blended.
As mentioned before, Matusik et al. [MZD05] proposed a histogram matching technique for reinserting small scale features into the final synthesized texture. The basic idea is to compute the histogram of high-frequency spectra in the target texture, and replace colors based on the probability distribution functions of the source exemplars. Matusik et al. employed steerable pyramids [HB95] for matching histograms at high-frequencies and then recovering the full texture colors.

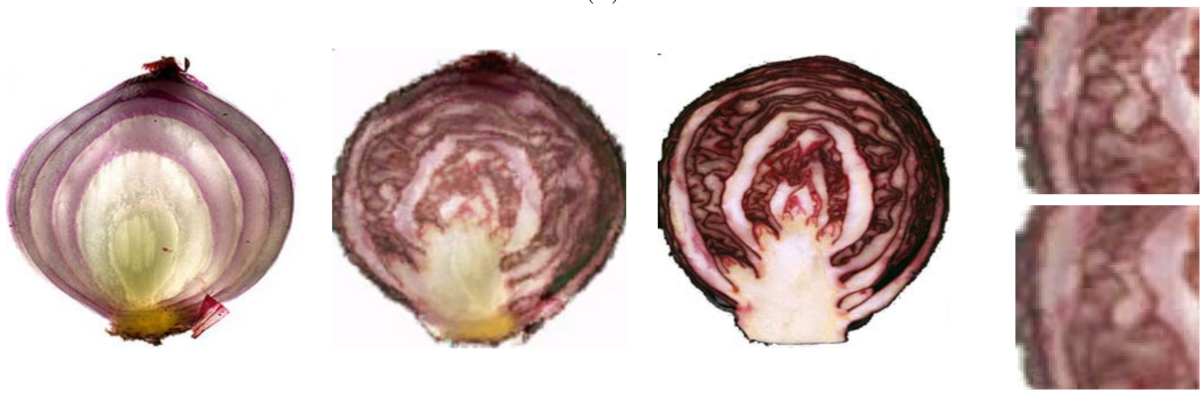
Although effective, this feature enhancement approach requires the computation of the histogram on the full synthesized texture image, which is inefficient for our per-point on-demand synthesis algorithm. However, one can observe that the histogram in the high-frequency spectrum can be well approximated by windowing the computation. In other words, it suffices to compute the histogram in a local kernel around the target point. We exploit this observation by precomputing local histograms (with a 7×7 kernel) for every pixel of the input exemplars, and similarly computing at runtime only a local histogram around the target point. In fact, since textures are computed on a surface, we anyway must synthesize the texture on a local kernel around each target point. Reusing the texture values from neighboring points also produces an orientation-aware histogram matching, as the small-scale features depend on the local orientation of the surface. Figure 3.29 shows the successful feature enhancement achieved with our local histogram computation.

3.5 Results

We have applied our appearance modeling framework in a variety of examples that show the diversity of problems where it can be used, as well as its versatility in terms of input data. Our examples have been generated on a laptop, with 1.7 GHz Intel Centrino processor and 1 GB of RAM. With such commodity hardware, our on-demand texture synthesis algorithm is capable of producing a throughput of approximately 20 000 pixels/sec. Histogram matching, with a 7^2 kernel, takes 50% of the computations.



(a)



(b)

Figure 3.29: Feature enhancement using local histogram matching: (a) Feature enhancement from multiple sources. (b) Morphing from an onion to a cabbage. The rightmost column compares a portion of the morphed texture, with feature enhancement through our local histogram computation (top), and without feature enhancement (bottom).

One of the applications where our modeling paradigm shows great benefit is the simulation of cutting and fracture.
Figure 3.32 shows two examples of interactive cutting simulation.Note that the simulations

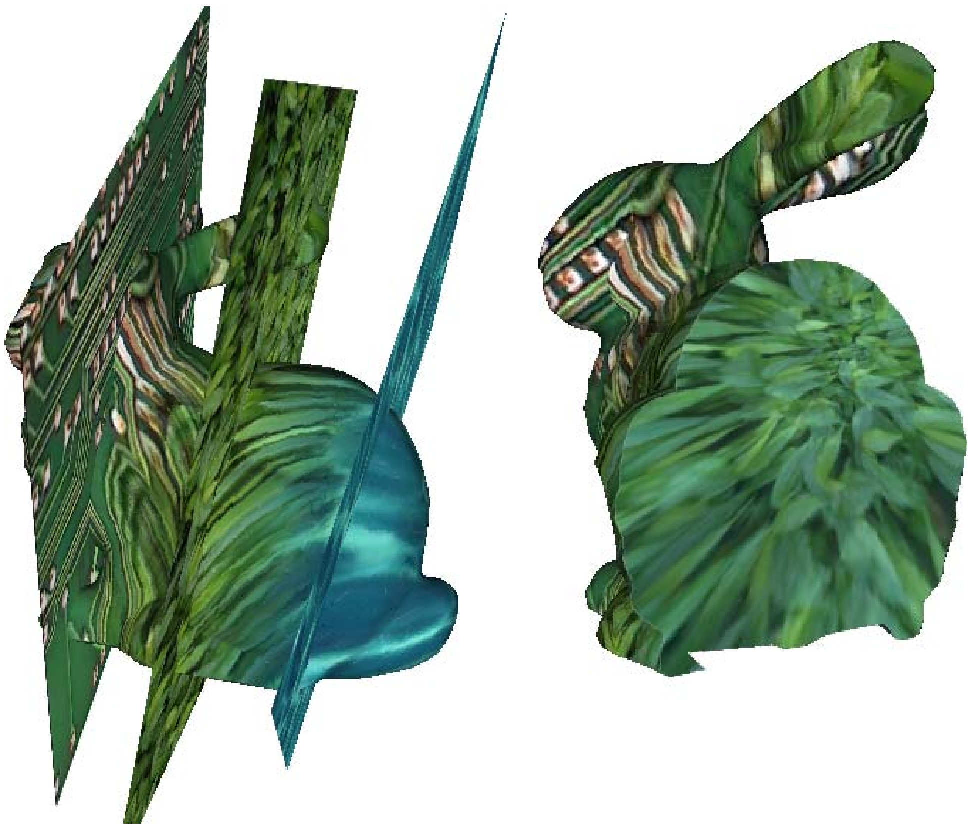

(a) (b)

Figure 3.30: Bunny with Patterned Textures: (a) Exemplars (a circuit board, leaves, and water reflections) and bunny's surface. (b) Cross-section of the textured bunny.

were created interactively, although they were later ray-traced offline. During interactive cutting or fracture, the on-demand synthesis of texture on internal surfaces plays a crucial role on the richness and realism of the results. The top row of Figure 3.32 depicts the synthesis of internal surfaces of the apple with the texture of an orange. The slices appear crisp and clear, and one can easily distinguish the border and the different features of the orange, even though we only performed three cross-sections on the real orange. Notice that the cuts in the simulation are not planar, yet our technique successfully captures the changes in orientation.
The bottom row of Figure 3.32 depicts a similar animation, where texture was synthesized from cross-sections of a cabbage and an onion (See Figure 3.29). The morphing between onion texture (bottom of the apple) and cabbage texture (top of the apple) is clearly visi-

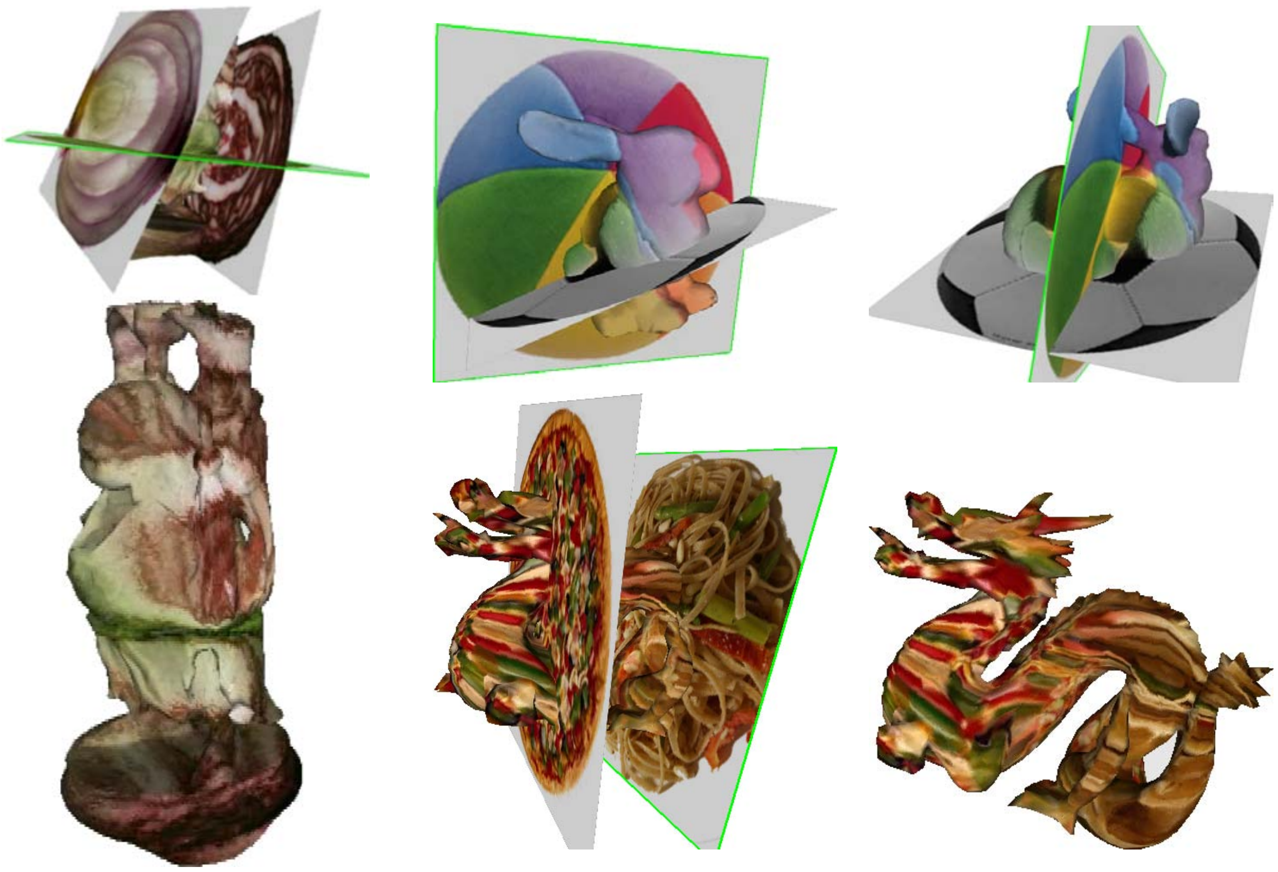

(a) (b)

Figure 3.31: Versatile Texturing: (a) Carving happy-Buddha from multiple exemplars. (b)Highly diverse textures are morphed onto a bunny and a dragon.

ble, while features are sharply captured.

Our versatile texturing approach allows the combination of highly diverse textures, as shown in Figures 3.30 and 3.31. In these examples, we use a simple sphere as the containing object M. Notice also that the colors of exemplars do not match at their intersections, but our morphing was able to handle this situation without artifacts.

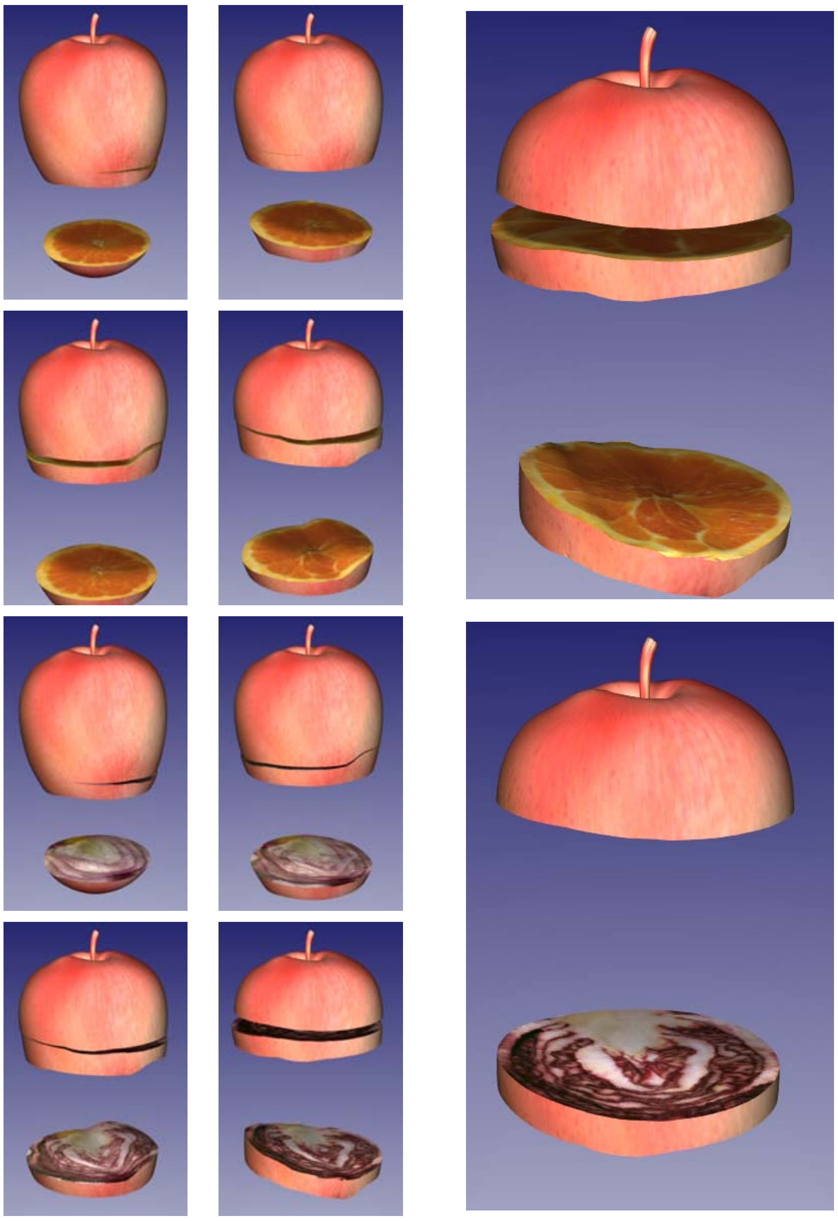


Figure 3.32: New Generation of Transgenic Fruits. Virtual slicing of an apple, showing internal textures generated with our algorithm. Top: orange texture morphed from three cross-sections. Bottom: morph between onion and cabbage cross-sections.

3.6 Conclusions and Future Work

The texturing technique presented in this paper provides a very simple yet powerful paradigm for creating appearance models for 3D objects. We have shown its application for carving objects or texturing internal surfaces in cutting simulation. The simplicity of the method, where a user takes cross-section photographs of real objects and places them in an interactive 3D editor, makes it highly amenable. As shown in the examples, our method produces highly realistic textures for internal surfaces of models that resemble real objects, but it also produces plausible textures for non-realistic examples. Our algorithm captures successfully the morphing of global and medium-scale features through multi-level warping, and reintroduces small features efficiently through local histogram matching. Moreover, an efficient approximation of warping enables the implementation of the algorithm as an on-demand routine for texturing internal surfaces during cutting simulation.

In many of the examples we have produced (e.g., morphing between onion and cabbage, the bunnies, or the dragon), it is practically impossible to find a warping between the exemplars that completely avoids blur. In essence, the feature images are not pure deformations of each other, hence a warping is not sufficient for capturing the transition. Although our method succeeds at producing plausible results with little blur, the automatic feature matching may lead to warpings that produce high feature distortion when morphing between images. A purely morphing-based technique may not be the best solution for such examples, and it would be interesting to study combinations of morphing-based texturing with techniques from stereology [JDR04] or global optimization optimization [KFCO$^+$07].

Our current implementation is limited to synthesis of color, but it would be interesting to investigate other appearance attributes. It is not obvious, however, that our morphing-based approach will be applicable to techniques such as bump mapping, as its major strength is capturing global features. We also consider the possibility of designing a parallel implementation on graphics hardware, as this could accelerate the morphing stage of the algorithm. Moreover, there could be cases where the texture does not need to be stored, as the user simply looks at an internal surface once, while sweeping through the object. The BSP-tree poses probably the biggest difficulties for a parallel implementation, and one option could be to limit the cross-sections to be axis-aligned, and implement the BSP-tree as a K-d tree.

Chapter 4

Additional Results

In this chapter we present some additional results we achieved. In the first section §4.1 we present a novel algorithm for interactive rope simulation which allows the user to create complex knots. In the second part §4.2 we present a system to produce a 3D head model from a set of input photos, that model is used to simulate sound scattering for realistic rendering of 3D sound.

4.1 A Robust method for Real-Time thread simulation

The main use of a *thread simulator* is certainly in endoscopic surgical simulation. Handling the surgical thread to make knots (which is required in many surgical procedures) is one of the most difficult tasks for a surgeon, which requires ambidexterity with the endoscopic forceps.
Simulating a thread is a intriguing problem because, although it seems simpler than for more complex 3D structures, the interaction with a thread involves some worst case situations for *self-collision detection* and *contact handling*, which are both fundamental to make knots. Furthermore, the thread is almost inextensible, and from the point of view of the simulation, this means that the methods modeling elasticity and using explicit time-integration schemes are poorly conditioned.

We implemented a method for simulating surgical thread based on Position Based Dynamics [MHHR07], modeling stiffness, bending, torsion and providing feedback for the haptic device.
The collision detection is carried out by *spatial hashing*[THM+03]. We speed up collision

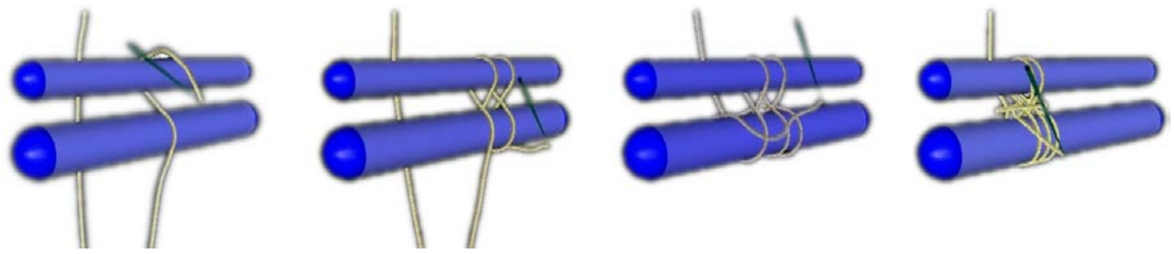


Figure 4.1: An example of knot tying performed by our algorithm.

detection introducing a hierarchical test on the curvature of the thread to early discard those portion of the thread that cannot self intersect.

In Section 4.1.1 we briefly review the state of the art in thread simulation, while in Section 4.1.2 we describe the detail of our implementation, finally results and conclusion are reported in Section 4.1.5.

4.1.1 Previous Work

Most of the approaches to thread simulation are energy-based (i.e., define a system energy and derive it for computing the forces), and model the thread as a one dimensional chain of mass points, where adjacent mass points are connected by springs. If the spring are pure elastic elements following Hooke law, some care needs to be taken to avoid oscillation and instability of the simulation.
In [PLK02] stability is enforced by making adaptive the number of points and using a modified integration scheme.
In [WBD$^+$05] the relation between points also includes bending and torsion. A dissipation factor is also considered in the form of friction forces consequent to contact.
A continuous model is proposed in [LMGC02], where the thread is modeled as a spline and energy of the system is defined with an integral over the spline. Although the mass is distributed along the thread and not lumped at the points, the energy term is ultimately computed by discretizing the computation of elongation and bending of the curve.
A non energy based model is presented in [BLM04] under the name of Follow The Leader (FTL) algorithm. It consists of first moving the nodes constrained by external action (e.g., grasped by a tool), and then heuristically moving the others trying to preserve the original inter-points distance. Although the absence of physical simulation is apparent, this method allows making complex knots at interactive frame rate.
A critical aspect of all the methods is how self-collision is detected and how contact is handled. A common approach is to use a hierarchy of bounding spheres. For the simplicity of the thread structure and for the low number of points generally used (few hundreds),

it turns out to be an effective solution, although the sphere is the worst fitting geometric primitive for a segment. As for more general simulation models, the collision generates impulse forces trying to restore non contact situation and the friction due to continuous contact is modeled as a dissipative force. Recently Spillmann and Teschner in [ST07] and [ST08] present two novel solutions for the simulation of ropes allowing knot-tying and adaptive refinement.

4.1.2 Our Approach

We model the physics of the thread using Position Based Dynamics [MHHR07].
This approach consists of modeling the physics as a set of constraints and then running the simulation iterating three steps:

1. Moving the mass points according to their velocity and external action (e.g. grasping).
2. Moving the mass points to satisfy the constraints.
3. Performing time integration.

If the direction of movement is along the gradient of the constraint function then the linear and angular momentum are preserved and no ghost forces are introduced. This characteristic coupled with a Verlet scheme (which does not store velocity) guarantees a unconditionally stable method (please refer to [MHHR07] for further details).
The main reason we choose Position Based Dynamics relies in its stability in collision and contact handling, where energy based method generally fails.

4.1.2.1 Stiffness and Bending

We use a *Distance Constraint* $C(p_0, p_1) = \|p_1 - p_0\| - L_{rest}$, both to keep the inter-points distance constant (see Figure 4.2.a) and to model resistance to bending by constraining the distance between every second mass point as proposed in [Pro95] (see Figure 4.2.b).

4.1.2.2 Contact constraint

Self contact constraints are added whenever the cable self-intersects, i.e. when the distance between two non adjacent edges is less than the radius of the thread r. The constraint is of the form $C(p) = [p - (p_{n0} + p_v)]$ where $p_v = (\|p_{n0} - p_{n1}\| - r) \cdot \frac{p_{n0} - p_{n1}}{\|p_{n0} - p_{n1}\|}$ is the penetration vector and p_{n0} is the current position of point p.

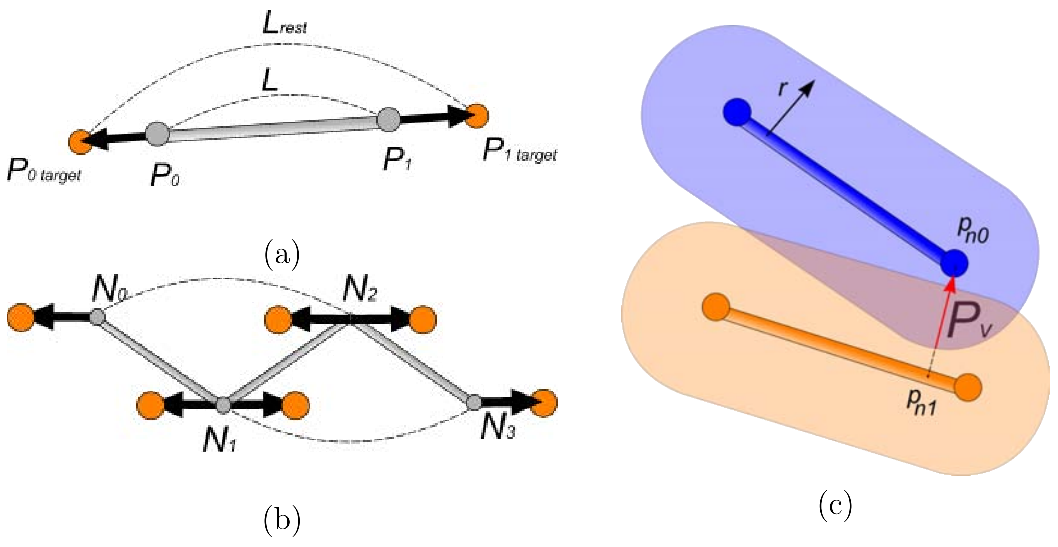


Figure 4.2: Stiffness,Bending & Collision constraints formulation: (a) *Stiffness constraint*: The two particles p_0 and p_1 are displaced in opposite directions along the edge in order to restore the original length of the thread L_{rest}. (b) *Bending constraint*: A stiffness constraint is added between node N_i and node N_{i+2} in order to oppose to bending operations. (c) *Contact constraint*: Given the section of the thread r the cable self-intersect if the distance between two non adjacent edges is less than r. The penetration vector P_v defines how to move the colliding segments in order to resolve the collision

4.1.2.3 Friction constraint

If a segment composing the thread is in contact, then its movement should be limited by adding friction.
In the original approach [MHHR07], friction is modeled by manipulating the velocities. Instead, we chose to integrate the friction in the constraints projection. The main reason is that the projection of the other constraints could bring the particles in a non-contact state *before* the velocities are updated and the risk of oscillating between contact state and non-contact state is greater.
Let $\Delta_i p$ be the displacement computed by the projection of a set of constraints, the point p be in contact with a surface locally approximate by a plane P_{fric}. Then the displacement is modified as: $\Delta'_i p = \Delta_i p \cdot \mu P v_{frict}$, where μ is the friction constant and Pv_{frict} is the penetration vector. In other words we apply the Coulomb friction in the constraint formulation, i.e. replacing forces with displacement. If the contact involves the thread and a surface, then the friction plane P_{fric} is defined by the normal over the nearest point of surface, while if the contact involve two segments then the friction plane is defined as the plane passing through the two segments (figure 4.3). The choice of integrating the friction in the constraint resolution is an heuristic that may make it more difficult to tune the

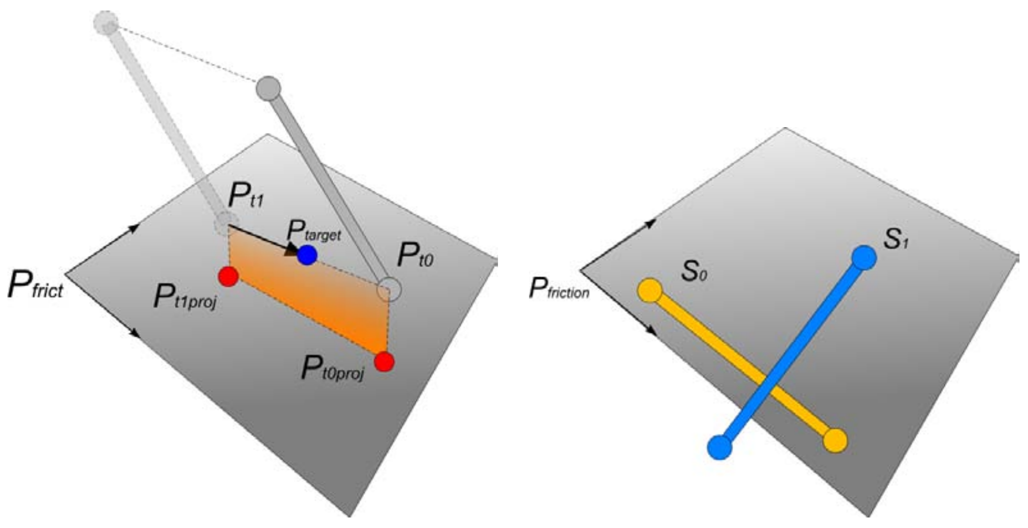


Figure 4.3: Friction & Contact constraints formulation: (a) An example of friction. (b) Friction plane definition in the case of 2 segments in contact: the friction plane is defined as the best plane fitting the 2 segments.

parameter, but in our experiments it leads to a more stable simulation.

4.1.2.4 Torsion constraint

To add a torsion constraint we first have to evaluate the torsion angle. While stiffness, bending and contact constraint can be easily defined over a one dimensional thread, the torsion cannot be derived solely on the position of the mass points. For this reason we add to the system a *material* vector for each segment. This vector is orthogonal to the segment and it is meant to have fixed orientation in material coordinates.
Without loss of generality, let us assume that at the beginning of the simulation the thread is straight and the torsion is 0 everywhere, so that all the material vectors have the same orientation.
For each segment we define the torsion angle as the angle between the actual material vector and a material vector computed assuming no torsion in the thread.

Figure 4.4.a shows two adjacent segments where the node p is moved from position p_a to the position $p_{a'}$ keeping the segment b fixed. The movement is done so that a torsion is created therefore the vector $vt_{a'}$ is not in the same plane as vt_b. Figure 4.4.b shows the same final configuration but this time the movement is a single rotation and no torsion is created, therefore $v_{a'}$ is in the same plane as vt_b. In other words the vector $v_{a'}$ is the value the material vector would have if the configuration was reached without applying a torque

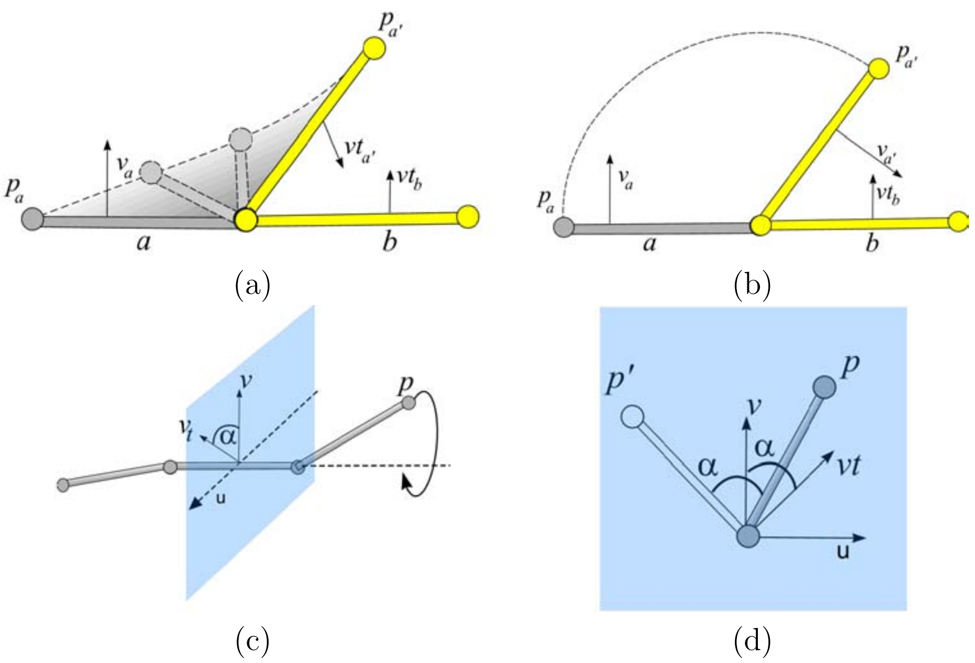
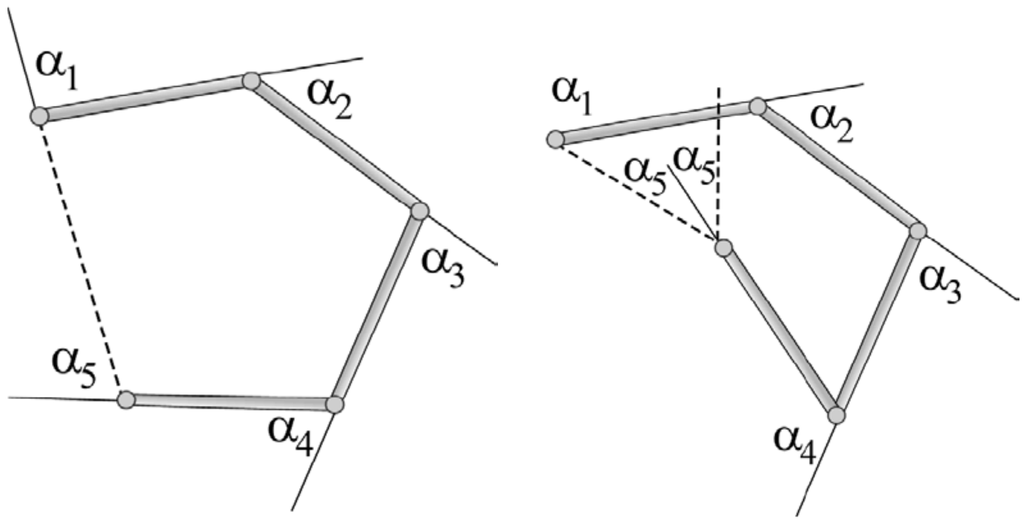


Figure 4.4: Torsion constraints formulation: (a) Computation of the material vector. (b) Finding the material vector under the assumption that there is no torsion (c-d) Definition of the torsion constraint.

to the thread.
Figure 4.4.c shows a configuration of the thread where the two material vectors have been computed. Consider the projection of the node p onto the plane orthogonal to the central segment. If we rotate p by the torsion angle α the torsion would be 0, therefore we express the constraint as $C(p) = |\alpha|$.

4.1.2.4.1 Material vector computation The material vector of each segment known at the beginning and it is updated step by step. Let us consider the segment s_i with material vector v_i at time t and $t+1$. We find the rototranslation RT_i as $\min\{RT|RTs_{it} - s_{it+1}||\}$ and update the material vector as $v_{it+1} = RT_i v_{it}$. This approximation relies on frame to frame coherency and on the fact that the thread has a very stiff behavior, so that the segments have almost constant length.



(a) (b)

Figure 4.5: Simple pruning test for self collision detection based on angles: if $\sum_{i=0}^{i<n} \|\alpha_i\| <= 2\pi$, then self-intersection can not occur.

4.1.3 Collision Detection

Because the segments of the thread are equally sized we used a regular partition of the space and the Spatial Hashing technique with temporal marks introduced in [THM+03]. We also exploit the one dimensional nature of the model to define an quick rejection test based on the discrete curvature of the thread. The idea is to consider the angles between consecutive segments of the piecewise rectilinear curve and to check if such angles would allow a self-intersection of the curve.

Let us consider Figure 4.5.(a) representing a thread made of four segments where two ends have been joined to form a non selfintersecting (i.e. simple) polygon. The angles indicated with α_i are the *turns* taken to walk the polygon clockwise.
In general it holds $\|\sum_{i=0}^{i<n} \alpha_i\| = 2\pi$ with α_i signed (positive clockwise, negative counterclockwise). If the polygon is convex, than $\|\sum_{i=0}^{i<n} \alpha_i\| = \sum_{i=0}^{i<n} \|\alpha_i\|$ while if it is non convex $\|\sum_{i=0}^{i<n} \alpha_i\| < \sum_{i=0}^{i<n} \|\alpha_i\|$.

We use the fact that if a polygon is not simple then it cannot be convex to derive a rejection test for self intersection, i.e. if we can prove the polygon is convex that it cannot be non simple.

Since the thread is embedded in *3D*, we would need to find a plane on which the projection of the thread (plus the closing segment) is a convex polygon, which would become

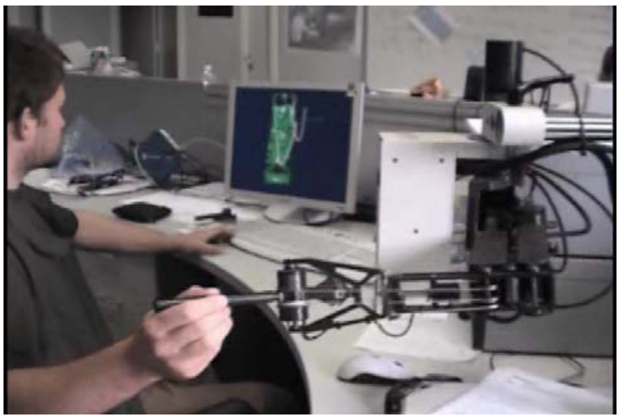


Figure 4.6: Our simulation framework include a 6 degree haptic device to allow real-time knot tying

too much demanding for a quick rejection test. Instead we simply compute the *3D* angles between consecutive segments $\overline{\alpha_i}$ and test if $\sum_{i=0}^{i<n} \|\overline{\alpha_i}\| <= 2\pi$. If this inequality holds, the thread is guaranteed not to self-intersect. The intuitive meaning of this test is that if we flatten the thread on to a plane preserving all the angles and so that they have the same sign, the thread will not self intersect because it simply is not *bent enough* to self intersect. We use this test in a hierarchical fashion, starting from the whole thread and recurring on failure to reject the self-intersection.

4.1.4 Visual and Haptic feedback

In order to have a smooth shape of the rope, we refine it at rendering time using the *Chaikin's Algorithm* [Cha74], starting from the set of particles used during the simulation. We render each segment of the rope using textured *impostors*, in order to minimize the number geometric primitives. We draw a set of connected polygons that were transformed respect to the eye point of the scene using a cylindrical symmetry. The overload due to this rendering technique is negligible and it has been successfully used to produce the final visual examples shown on videos. The framework for knot tying simulation include a 6 degree haptic device to allow real-time knot tying, the FREEDOM6S interface [J.D98](See Figure 4.6). The user interact by grasping and freely moving a grasped node using the haptic device. The resulting *feedback force* is computed by considering the two segments adjacent to the grasped node as linear springs. While the frame rate of haptic device is guaranteed by performing a linear extrapolation in a dedicated *thread*.

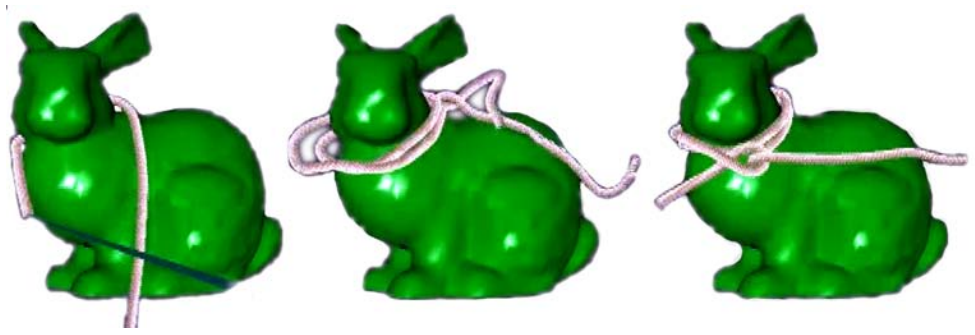


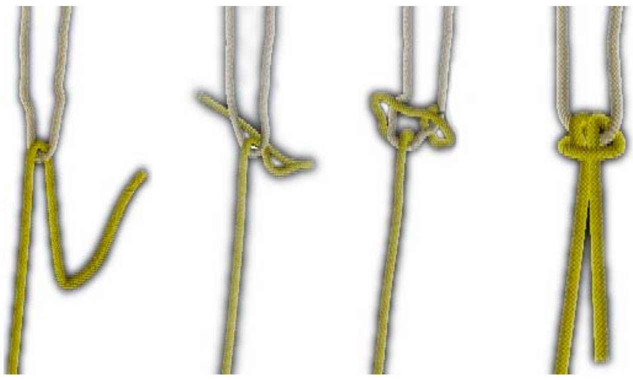
Figure 4.7: Strangling the bunny

4.1.5 Conclusions and future work

We applied our model to simulate real time knot tying. Figures 4.7, 4.8 and 4.1 show some examples of interactive knot tying. The simulation runs at 70fps on a Intel Pentium 4 - 3,0 GHz using 90 particles and 320 sampled points for ropes rendering.
We implemented the technique using the VCG library for the geometry concepts [Vis05b], IDOLib for simulation primitives (time integration and contact)[Vis05a] and the Haptik Library [dPP07] for interface with generic haptic hardware.

With respect to the *Follow-The-Leader* algorithm proposed in [BLM04] that only takes into account stiffness and assumes the thread has no mass, our solution also considers gravity and torsion. Thanks to the stability and controllability of position based dynamics, our algorithm can model a very stiff thread, that is more similar to a real surgical thread with respect to the explicit mass spring formulation described in [WBD+05].

For future research, we plan to formulate new constraints on the thread in order to simulate the suturing process. In this case, it may be rather simple to constrain the cable to slide trough holes. Another direction of research is to investigate the possibility of adapting sampling of the thread and adaptive time step of integration. This was not necessary in our current setting but it will be when using longer threads without losing the detail of the representation.


(a)

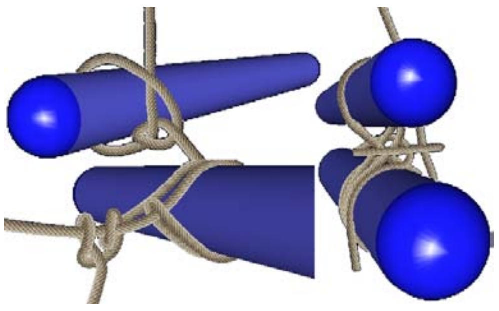

(b)

Figure 4.8: (a) Example of a knot with 2 ropes. (b) Detail views on other knots.

4.2 Reconstructing head models from photographs for individualized 3D-audio processing

Audio has not been considered much as a mean to provide realism, but techniques like binaural rendering can greatly enhance the perception of realism. Binaural rendering can be fully exploited only by using individualized HRTF filters, otherwise it can produce localization artifacts (please refer to [Beg94] for a comprehensive overview on 3D sound for Virtual Reality and Multimedia).
HRTFs are complicated functions of frequency and spatial variables, their shape can noticeably vary between subjects and it's closely related to the features of the head (see [Bla97] for a comprehensive overview about psychophysics of spatial hearing). In particular, the primary role in determining sound perception seems to be associated with the peculiar features of ears, while a secondary contribution is assigned to the size of the main features of the head (nose, chin, head width and height...). This makes the modeling of accurate *individual* HRTFs a central issue in the context of audio rendering techniques.
Among the several proposed solutions, a very promising direction is the simulation of the measurements made on a 3D head model. Unfortunately, the quality of the final result is strongly related to the accuracy of the geometric model.

The system we propose is an automatic way to build a 3D model of a human head, starting from a few photos of the subject and some key-points indicated on them. The needed input can be produced in a few minutes, and the rest of the procedure is fast and completely automatic. It involves mechanisms to extract information from the photos , and to accordingly deform a 3D dummy head model in order to reproduce the ears and face features of the subject.
The final structure of the system proves to be:

- Usable for large scale application, thanks to the very low amount of input needed.
- Completely automatic. Once the input data are collected, the 3D model reconstruction and the HRTF calculation are performed with no further intervention.
- Fast and reliable: the system is structured to produce realistic results even in the case of low quality or incongruent input data.
- Designed to accurately reproduce the ears shape, but also able to recreate the main face features.
- Geometrically accurate: validation shows that the accuracy in reconstruction is adequate to calculate a satisfactory HRTF.

4.2.1 individualized HRTF modeling

4.2.1.1 Related Work

Listening to 3D audio over headphones using non-individualized HRTF filters can lead to localization artifacts, like notably increased "inside-the-head" perception and front-back confusions [Bla97, WAKW93]. To address this issue, several approaches have been proposed to model individualized HRTFs for 3D audio processing [Gar05]. They can be classified into four major groups:

1. Directly measuring HRTFs by placing microphones inside the ears of the subject [KB07]. This process can take up to several hours and is extremely uncomfortable for the user.
2. Perform a set of Perceptually-oriented tests to choose a preprocessed HRTF [MMO00]. Unfortunately, efficient protocols to quickly select the most appropriate matches are still not well defined.
3. Choose the right preprocessed HRTF by considering direct measurements of morphological parameters on the subjects [Lar01, DAA99, ADMT01]. This approach requires a large database which might not be fully reliable.
4. Simulate virtual measurements of HRTFs filters by running finite element simulations [Kat01, KN06] or ray-casting [ARM06] on a 3D head model. Using an approximate finite element simulations [TDLD07] provides results in a reasonable time, however, this method require a 3D input mesh of the subject.

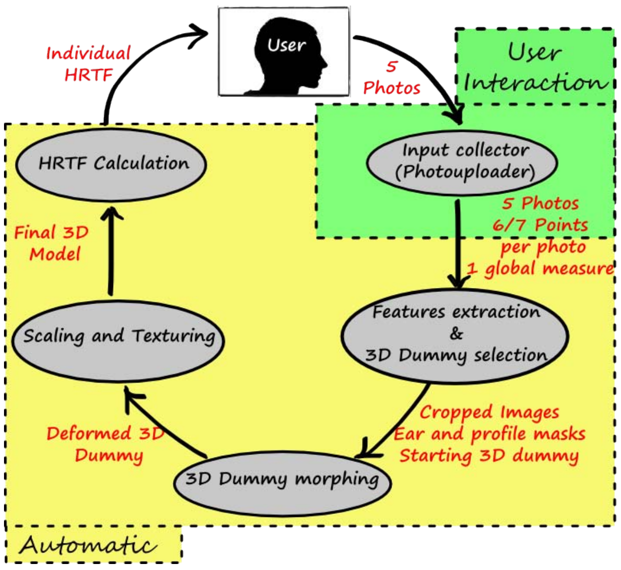


Figure 4.9: A scheme of the whole HRTF calculation system

4.2.1.2 Overview of our system

In order to be able to reach the goals listed in the introduction, the whole system was designed as a custom solution which employs different techniques at their best for this peculiar application.
A scheme of the whole system is shown in Figure 4.9. We can individualize four main components involved in the generation of the 3D model. The whole process can be roughly divided into two parts:

- The first one deals with the treatment of the input data, in order to extract relevant features and select the best 3D dummy to be morphed.
- In the second part, work is performed on the selected 3D dummy mesh, in order to morph and scale it to resemble the geometric features of the head of the user.

Though, for sake of clarity, the system has been subdivided in several parts, only the first component (Input Collector) needs an intervention by the user. The rest of the system has been designed to be completely automatic.

4.2.1.3 Photouploader

In order to collect data in a user-friendly way, we propose a simple tool called *Photouploader*. By using Photouploader the user specifies the input photos and indicate some key-points (6 or 7) on each image. Moreover, the system requires a global scale measure in order to scale the final model (we chose nose length).

4.2.2 Reconstruction of head models

4.2.2.1 Related Work on ear reconstruction

The shape of the ears plays a key role in our reconstruction system. Ear biometrics try to individuate a model for external ear features:[Ian89] is often cited as a very convincing one. Unfortunately, these biometry measures are not well-defined, so it's very hard to use them in automatic methods. Ears are mainly considered in the field of recognition and security by analyzing both 2D [JM06] and 3D [CB05, CB07] ear data. Unfortunately the information extracted and used for recognition is usually not directly linked to geometric features (like curves or size of the ear).

4.2.2.2 Related Work on head reconstruction

Several works in the field of 3D reconstruction focus on the reconstruction of 3D faces from images.
Very realistic results can be achieved [HA04, D'A01], but the produced geometry is usually not accurate enough. An approach which is more related to our goal is the morphing of face model to fit images [BV99, Bla06, JHY$^+$05] and 3D scans [BSS07]. These methods are very accurate, but they don't take into account the whole head, and especially the ears.

Regarding 3D head reconstruction from 2D data, some methods [LMT98, LLY04] obtain low resolution complete and textured 3D head models. Even in these cases, geometry is not accurate enough for our requirement. [FOT04] can generate accurate models, but a very complex acquisition apparatus (28 digital cameras and two projectors) is needed. Moreover, most of the works don't take into account the scale of the model, that is a key issue for scattering calculation.
In conclusion, a numerical comparison between the cited methods and our system is difficult due to the differences in goals (geometry accuracy vs. visual resemblance, 3D faces vs. 3D heads). Hence, we will consider laser scanning as a reference, since it is the most reliable technique for geometry acquisition.

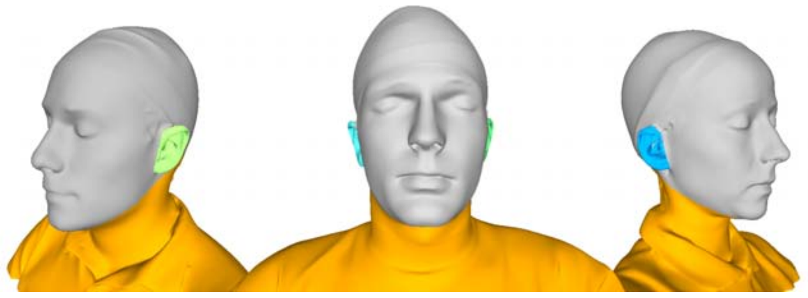


Figure 4.10: Three elements of the 3D dummy library

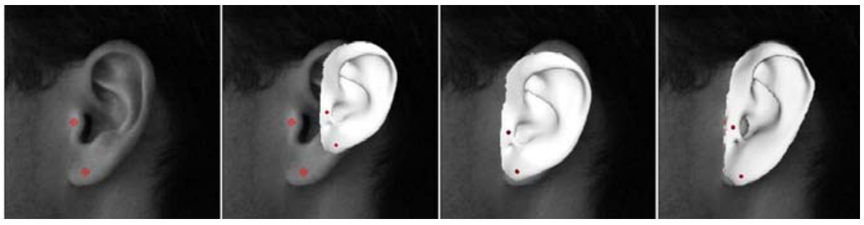
4.2.2.3 Features extraction from images and starting 3D dummy selection

Once the input data are collected, the automatic model production process starts. The goal of the first element of the system is the selection of the best starting dummy from a library of 3D heads. The library of 3D heads is composed of ten models obtained via 3D scanning. As shown in Figure 4.10, each 3D model has differently colored parts, which undergo different morphing operations.
As already stated, although face features are important and can't be ignored, the shape of the ears plays a key role in the final HRTF profile. So the dummy which best matches the ear features (extracted from images) is selected for morphing.
The sub-image representing the ear is automatically cropped (by using key-points) from original photos, then the ear external border is extracted by following the ear edge starting form an initial seed which is automatically placed. The dummy selection is performed by analyzing each of the ears of the library models. A model of perspective camera is used to rigidly align segmented ears from photos with respect to 3D ears coming form the dummy model.
A low accuracy morphing (see Figure 4.11.a) is applied on each 3D ear model, so that it is slightly deformed to fit both the external mask and the internal features. After this operation, a similarity measure between a rendering of the 3D ear and the extracted image (based on the position of feature edges) is calculated, and the most similar dummy is selected.

Similarly, profile and frontal photos are used to provide head contours together with each aligned cameras to compute head deformation (Figure 4.11.b).

4.2.2.4 Dummy morphing

The 3D Morpher is the core of all the system: it applies a peculiar deformation to the dummy 3D mesh. Using the set of cameras which defines the alignment of the dummy model with respect to each image, a set of viewport-dependent 2D-to-3D model deformation


(a)

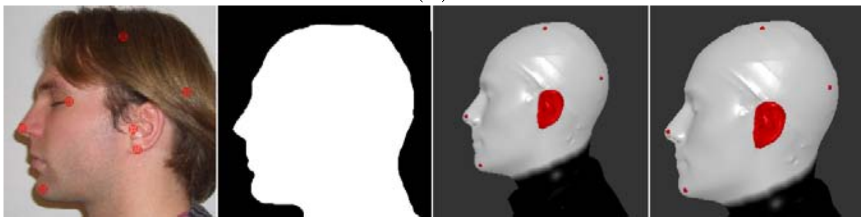

(b)

Figure 4.11: Ear and Head selection and alignment: (a) An example of ear selection: starting image, dummy ear camera position before and after alignment, ear shape after low accuracy morphing. (b) An example of head alignment: starting image, extracted mask, dummy head camera position before and after alignment.

is calculated. The set of deformations is then combined to morph the dummy model to its final shape.
The entire morphing process can be subdivided into the following steps:

Single View Head Deformation Three energy-driven deformations, one for each photo (right and left profile, frontal) are calculated. Each one tries to match the geometry with respect to a single point of view (*view dependent deformation*).

Global Head Deformation View dependent deformations are merged (according to camera positions and orientations) to a global smooth deformation.

Ear deformation Ears, which have been preserved in previous steps, undergo an accurate deformation using close-up ear cameras and images.

Morphing the geometry to match an input image requires the computation of a mapping between the photo and a rendered image of the dummy head (taken from the associated rigid-aligned camera position). This mapping operation is usually referred as *warping* in literature.

4.2.2.4.1 Warping Computation Our warping function is an extension of the technique described in section 3.4 for textures warping computation.

The original energy function (see [MZD05] for details) is modified by adding a term $Kp = \sum_{P_i \in keypoints} |(P_i(photo) - P_i(model)|$ that measures the sum of distances between the user-defined key-points on the input photo and the relative key-points on the dummy head (transformed to screen-space coordinates).
Energy function can be schematized as follows:

$$E = L2 + \alpha * J + \beta * Kp \tag{4.1}$$

where $L2$ is the *per-pixel error* scalar feature strength and J is the *Jacobian term* which controls the smoothness of the warp field.
Once we calculate the warping function, the displacement for each 3D dummy vertex is calculated by projecting the vertex on the associated camera plane, evaluating its warped position, and un-projecting it back to world space (without changing z-value).
The size of images used for morphing is 256^2. This size represents a good compromise between detail preservation and processing time: higher resolution images could be used, but the gain in detail would not justify the longer time necessary for computation.

4.2.2.4.2 Single View Head Deformation In this phase we apply the warping between a rendered image of the dummy head and the input photograph, using associated camera parameters.
In the original method, both external and internal features would be taken into account for deformation. But in a real scenario head photographs could reveal strong sharp features that are difficult to be represented geometrically (such as beard, eyebrows..), furthermore peculiar lighting environments could lead to incorrect edges warping. So the deformation is applied using only the binary masks which define the external profile of the head.
The internal features of the face are then deformed by fitting a group of key-points associated to those indicated by user. Figure 4.12.a shows the key-points involved in head warping: for frontal deformation we use 5 key-points: two for the eyes, one for the nose and two for the mouth; while for lateral deformation we chose to use one side eye constraint plus a set of four points around the ear. These points define the bounding box of the ear, so that it is preserved for a latter deformation. Sequence 4.12.b shows the deformation process involving the dummy mesh, using one lateral image.
Moreover, the frontal warping is controlled via *symmetrization*. Because of possible non-symmetric head contours extracted from frontal photo or non perfect input image (i.e. tilted or slightly rotated head), simple warping can produce asymmetric head shapes (see Figure 4.13.a). To overcome this problem we symmetrize the warping as follows:

- We establish a symmetrization line on the rendered image, so that the rendered image is divided into two subspaces (Figure 4.13.c). The symmetrization line is defined as the line passing through the nose key-point and the point in the middle of the eyes key-points.

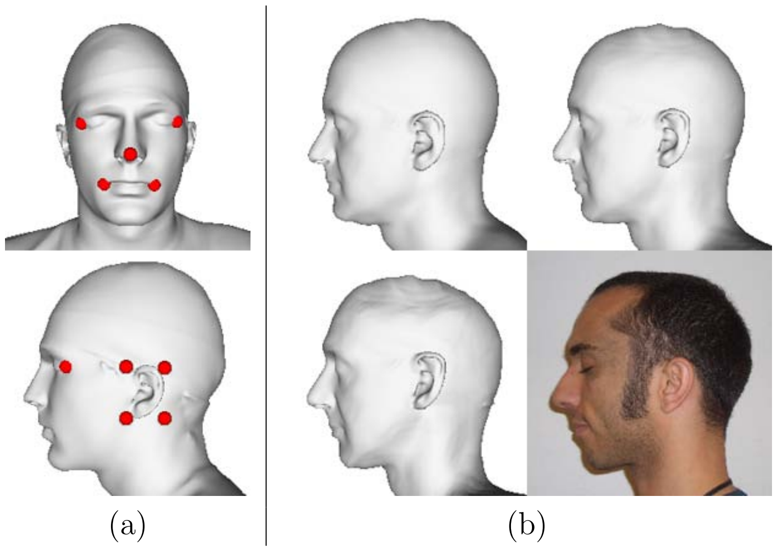


Figure 4.12: Lateral head deformation:(a)Keypoints used for frontal and lateral head deformation. (b)Lateral head deformation sequence.

- We obtain a mapping $Mirr(x, y)$ between the two regions of the rendered dummy, by mirroring over the symmetrization line.
- We finally average the warping of mirrored points:

$$Warp(x, y) = \frac{Warp(x, y) + Mirr^{-1}(Warp(Mirr(x, y)))}{2} \tag{4.2}$$

Figure 4.13.b shows the effect of warping symmetrization.

4.2.2.4.3 Global Head Deformation At this morphing stage, each vertex can be translated in three different ways (one for each viewpoint). These three camera plane warpings are unified to a single smooth deformation as follows:

- Lateral deformed positions are unified through a weighted sum (weights decrease proportionally respect to ear distances)
- Unified lateral and frontal deformations are summed by assuming they are perpendicular, so that displacements in x- and z-axis are independent: the final displacement in the common direction (y-axis) is a weighted sum of the two contributions.

4.2.2.4.4 Ear deformation Accurate ear deformation is key to the final quality of the results: in this case both internal and external features extracted from images can be used

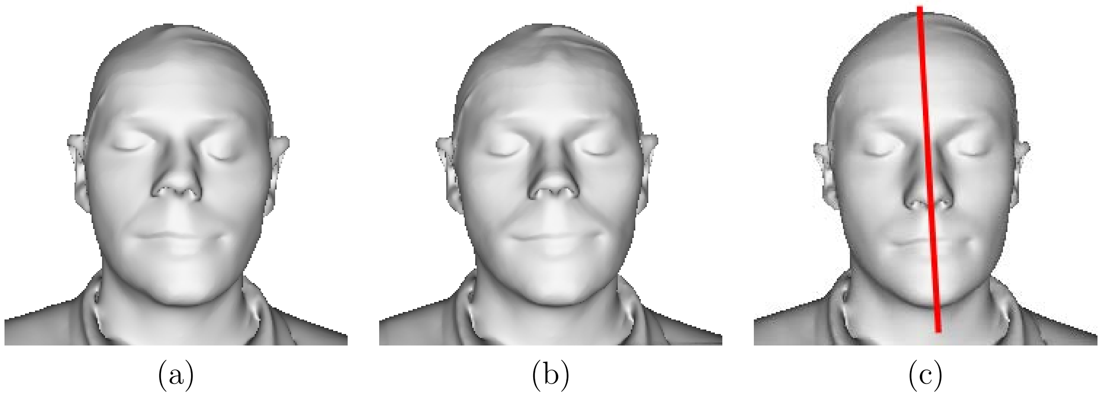


Figure 4.13: Symmetrization: (a) & (b) Non-symmetrized versus symmetrized. (c) Symmetrization line of undeformed model.

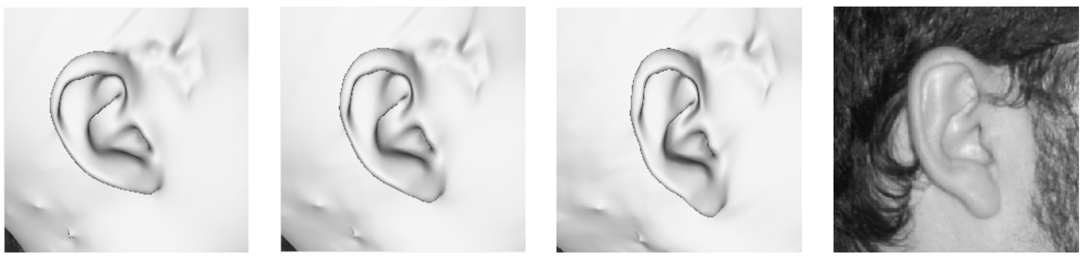


Figure 4.14: Example of ear morphing sequence.

to compute the deformation (Figure 4.14). The morphing sequence is organized as follows: 3D ear rendering is morphed to fit the external ear silhouette, then an additional warping is used to match internal features.
The colors of input ear images are previously modified in order to match the histogram of frequency spectra of the rendering of the dummy model.

4.2.2.5 Global scaling and texturing

Scaling is one of the key issues about the accuracy of reconstruction. If the size of the model is incorrect, the computed HRTF will be wrong. The scaling operation is performed using the measure provided by user with the Photouploader (see Section 4.2.1.3), which is the nose length.
Another feature of the system is the possibility to texture the obtained head model: the input photos are deformed to match more precisely the geometry (essentially by applying the inverse warping explained in section 4.2.2.4). Shown colored models are obtained by projecting warped images using [CCCS08]. But, for clarity sake, textures are not needed for HRTF calculation.

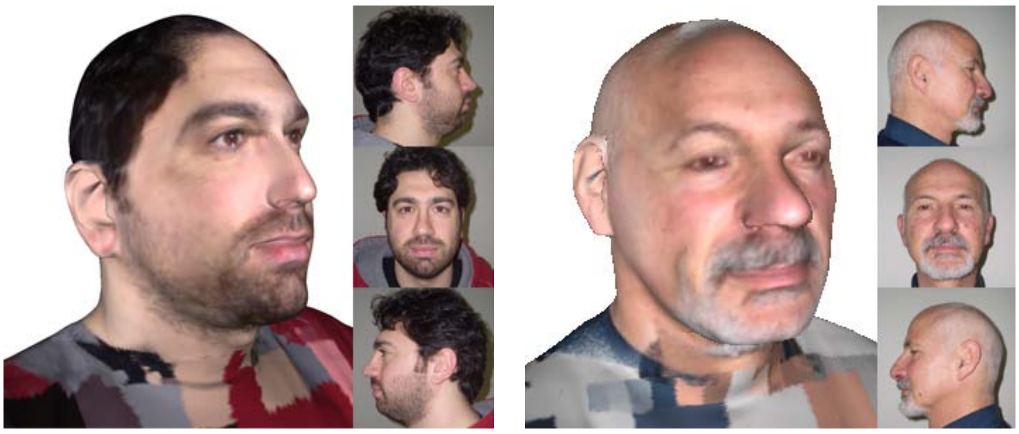


Figure 4.15: Two results of processed heads.

4.2.3 Results

Two results are shown in Figure 4.15. The entire system proves to be robust and quite fast: the overall time needed to produce the final 3D model, from input collection to model saving, it is less than ten minutes.
Although the visual resemblance of the obtained model is usually satisfactory, the main goal of the entire system was to be able to guarantee sufficient geometric accuracy. For this reason we performed some tests to compare 8 models obtained from photos to their corresponding laser scanned models. Moreover, preliminary HRTF calculation tests were performed on on reconstructed and laser scanned 3D heads, in order to obtain a comparison between the resulting simulations.

4.2.3.1 Geometric validation

A sub-millimetric precision in geometry reconstruction from photos is a results which is possible only under very particular and controlled conditions. In our case, since the input is provided by the user, and the starting dummy can be very different from the final target, the main goal is to be able to reproduce head features as much as possible. Hence, instead of using purely geometric comparison tools like [CRS98], we compared the results by taking into account two sets of measures, which could represent head features and their influence on the HRTF profile.
The first set was the position in space of several key-points, picked on the models. Results of comparison are shown in Table 4.1. The average error is usually less than 1 cm and the variance of data is not big. These values can be considered as satisfactory, especially considering the fact that the input data is solely two-dimensional and the scale factor can introduce inaccuracies.

	Average	Maximum	Variance
Nose	11.3	22.1	0.89
Chin	10.8	21.2	0.03
Left Eye	9.8	16.6	1.22
Right Eye	8.1	11.4	0.46
Left Mouth	11.4	22.2	2.83
Right Mouth	9.8	21.5	1.69
Left Lobule	8.4	13.5	1.89
Left Tragus	6.7	11.5	1.04
Right Lobule	8.3	14.0	1.30
Right Tragus	7.3	10.8	1.83

Table 4.1: Distance in mm between key-points of scanned and reconstructed model

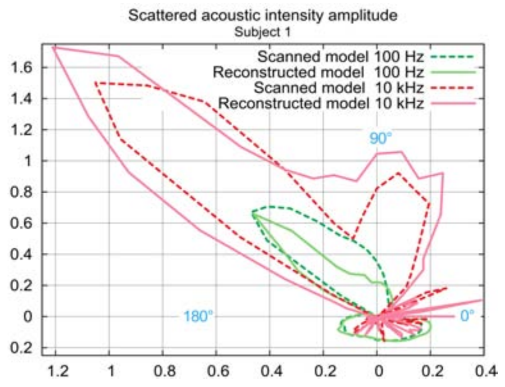
	Average	Maximum	Variance
Neck (d4)	5.6	15	0.10
Head Size (d9)	5.9	14.5	1.30
Head Size (d19)	3.2	6.7	0.47
Ear size (R) (d10)	3.7	6.3	0.50
Concha size (R) (d12)	1.8	3.4	0.04
Concha size (R)(d13)	0.8	1.3	0.005
Ear size (L) (d10)	3.8	8.2	0.58
Concha size (L) (d12)	2.1	3.8	0.33
Concha size (L) (d13)	1.1	2.5	0.24

Table 4.2: Difference in mm between distances indicated in [Lar01]

The second set of measures was extracted from [Lar01], where several ear and head measures were statistically analyzed in order to find which ones were more related to the changes in HRTF profile. We compared the set of 3D models using six measures (three for the head, three for the ears) which are indicated as very important in the conclusions of this work. Results are shown in Table 4.2. The difference between distances is often less than 5 mm, in particular the ear internal features seem to be preserved accurately.
An overall analysis of the data shows that, even if accuracy for some features (like ear size and tragus position) could be further improved, the error bound describes a very reliable system.

4.2.3.2 Preliminary assessment of HRTF simulations

A preliminary validation of HRTF simulation on 3D models was performed on couples of laser scanned and reconstructed 3D heads of the same subject.
To simulate HRTFs corresponding to the reconstructed geometry, we used a simplified boundary element approach leveraging on the Kirchoff approximation. The Kirchoff approximation allows for efficiently computing first order scattering off the reconstructed



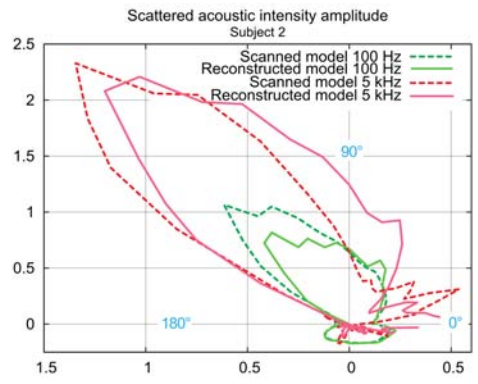


Figure 4.16: Two examples of polar plots for measurements on couples of scanned-reconstructed 3D heads.

mesh and can be efficiently implemented using programmable graphics hardware. Please refer to [TDLD07] for details.
We used this approach to compute the scattered field captured by two virtual microphones placed at the entrance of the left and right ear canal, less than 5 millimeters away from the surface of the mesh. Computed data are only a subset of a complete individual HRTF, but they provide enough information for a preliminary comparison. Two polar plots of left ear intensity-amplitude for our reconstruction approach compared to a laser scanned model are shown in Figure 4.16. The obtained data from preliminary calculations prove to be encouraging for a future use of 3D reconstructed models for HRTF calculation. A further stage of validation between reconstructed and measured in anechoic chamber HRTFs will provide more information. Moreover, it will be possible to further investigate the importance of each head feature to improve results and possibly further simplify the system.

4.3 Conclusions and future work

Our system automatically creates 3D head models from a small input set (five photos and some key-points indicated on them). The system integrates image processing techniques with a novel application of 3D morphing, based on a combination of several 2D deformations calculated in different camera spaces. The technique has been proven to be fast, robust and reliable, furthermore both geometric and preliminary HRTF validation are encouraging.
Future work to further improve the method includes:

- The implementation of more effective methods for face deformation (i.e. implementing part of the contribution of [Bla06]). This would probably lead to a better visual resemblance of the model, widening the application field of the proposed method to geometrically accurate avatar generation. In this case, hair extraction and visualization should be taken in account.

- Calculate warping by using the GPU. This could bring the whole computation time from minutes to seconds.
- Enriching the dummy library: it is currently composed by only ten models, but the best solution could be to select accurately a subset of 3D models from a wider set of 3D scanned heads.

In conclusion, the proposed system can be considered as a very promising method not only for individualized 3D-audio processing, but also for other applications which need accurate head geometry, produced from a few photographs.

Chapter 5

Final Remarks

In this thesis we propose a collection of novel techniques to improve efficiency, stability and visualization quality for interactive simulation of virtual cutting of deformable objects. We summarize the main contributions described in this thesis as follows:

- We presented a new data structure and algorithm called *Splitting Cubes* [PGCS09] to dynamically represent the external boundary of deformable object. Splitting cubes can be used in general to represent a surface embedded in a 3D deformation field which, in the specific case of a deformable object, is defined by the underlying physical model.
  One important advantage provided by the splitting cubes relies on its *independence* with respect to the underlying physical model used to encode the deformation field. Splitting cubes provides an implicit representation of the external boundary which can be dynamically modified to represent any topological modification, such as cuts or fractures, occurring during the simulation. Such implicit representation is encoded by a regular grid, whose cells represent the sub-portions of surface embedded. More precisely, each cell maintains a minimum set of information to derive the tessellation and the deformation of its sub-portion of surface.
  Due to this implicit representation, splitting cubes are *unconditionally stable* to topological modifications. Furthermore, *efficiency* is ensured by the fact that the amount of geometric primitives introduced by cuts or fractures is limited by the grid resolution.
  We also demonstrated how the Splitting Cubes can be successfully applied to mesh-free methods, which do not have an associated boundary representation on their own, by using the Point Based Method described in [MKN$^+$04] to compute the deformation field.

- We introduced a new method to model discontinuities on mesh-less methods called

Extended Transparency Method [PGCS09]. We illustrated how the Extended Transparency Method improves stability and efficiency with respect to previous method proposed in literature. Thanks to its formulation, this method can be easily implemented on the GPU, producing a considerable speedup.

- We introduced a new appearance-modeling paradigm for synthesizing the internal structure of a 3D model from photographs of a few cross-sections of a real object [POB+07]. When the internal surfaces of the 3D model are revealed (for example when it is cut, carved, or simply clipped) we are able to show the appropriate texture synthesized from the input photographs. Our texture synthesis algorithm is best classified as a morphing technique, which efficiently outputs the texture attributes of each surface point on demand. Our modeling paradigm, together with its implementation through our texture morphing algorithm, allows users to author 3D models that reveal highly realistic internal surfaces in a variety of artistic flavors. With respect to previous approaches this method can reproduce a large set of effects by capturing features at different scales. Furthermore, it can efficiently synthesize colors on demand so that it be integrated in a real-time simulation.

We also achieved two interesting additional results:

- We defined a new approach for interactive simulation of rope, including the possibility of performing complex knots [KPGF07] using Position Based Dynamics [MHHR07]. More precisely, rope dynamic is described by a set of constraints must be satisfied simultaneously during the simulation. With respect to previous methods proposed in literature, our method can be considered as a good compromise between physical accuracy and robustness.
- We presented a pipeline for reconstructing head models from a few photograph, to perform sound scattering calculation [DPT+08]. Our algorithm relies on two main contributions: first, we design the first pipeline to calculate personalized HRTF (for 3D sound rendering) from photos; second we define a method to reconstruct the entire head with particular attention to the morphology of the ears, as opposite to classical systems which mainly focus on facial features.

5.1 Future work

In this section we explore the directions for further investigation suggested by the results obtained in this thesis.

Surface description The key idea of the Splitting Cubes algorithm is to associate an ad hoc representation to the cut configuration of each cell, which in its current version is

a small triangle mesh. A natural extension could be to generalize the representation. This would allow to use other rendering strategies. For example we could consider a set of points rendered as splats [ZPvBG01, ZPvBG02] perform a ray cast in the fragment shader. These choices could be beneficial since they could naturally provide a level of detail representation, i.e. the work done for rendering the portion of surface inside a cell could be proportional to the number of pixels on screen it project to (while now is fixed with the number of triangles).
A second improvement could be to implement the Splitting Cubes in a multiresolution fashion to be able to move away from a fixed granularity of the cut. Obviously this approach would bring back the risk of excessive fragmentation and consequent performance loss that should be taken into account.

Physical model We could simulate the mechanical behavior of a deformable object by using the volumetric discretization provided by splitting cubes regular grid.
We can, for example, consider each cell as a continuous amount of material which can be simulated by using FEM. By considering the fact that their shape never degenerate to quasi zero volumes, the potential for instabilities is minimized.
Recently, Rivers and James [RJ07] presented the Fast Lattice Method, a shape matching method optimized for cubical cells. This method is unconditionally stable and easy to implement. We could consider each cell of the splitting cubes as a lattice cell of the Fast Lattice Method, by taking into account duplicating cubes when topological modifications occurs (according to the different connected components). The resulting framework will be unconditionally stable.

Internal appearance modeling It would be useful to reproduce other internal appearance attributes rather than simple color values. In the real world, if we cut an object, the revealed internal surface is not perfectly flat as the path defined by the cutting tool. Indeed the real surface often reveals micro-geometric structures whose shape depends on the object's constituent material and the tool used. This is especially true for organic objects like meat or wood. Then, an interesting way to improve our work could be to capture, reproduce and render efficiently such micro-geometric structure in real time.

5.2 List of Publications

- N.Pietroni, A.Giachetti, F.Ganovelli. *Robust segmentation of anatomical structures with deformable surfaces and marching cubes.* Poster session of **CARS** Computer Assisted Radiology and Surgery. Berlin **2005**

- N.Pietroni, A.Giachetti, F.Ganovelli. *Robust segmentation of anatomical structures with deformable surfaces and marching cubes.* In Proceedings of Workshop in Virtual Reality Interactions and Physical Simulations **VRIPHYS**, **2005**

- G.Turini, N.Pietroni, F.Ganovelli, R.Scopigno *Techniques for Computer Assisted Surgery.* **Eurographics Italian Chapter**, **2007**

- Nico Pietroni, Miguel A. Otaduy, Bernd Bickel, Fabio Ganovelli, and Markus H. Gross. *Texturing internal surfaces from a few cross sections.* **Computer Graphics Forum** (Special Issue - Eurographics), 26(3),637-644, **2007**

- Blazej Kubiak, Nico Pietroni, Fabio Ganovelli, and Marco Fratarcangeli. *A robust method for real-time thread simulation.* Proceedings of the ACM Symposium on Virtual Reality Software and Technology, **ACM VRST**, Newport Beach, California, USA, November 5-7, **2007**

- Giuseppe Turini, Nico Pietroni, Giuseppe Megali, Fabio Ganovelli, Andrea Pietrabissa, Franco Mosca *New Techniques for computer-based simulation in surgical training* International Journal of Biomedical Engineering and Technology **IJBET** special issue on HOF 2007 conference, in press. **2008**

- Matteo Dellepiane, Nico Pietroni, Nicolas Tsingos, Manuel Asselot, Roberto Scopigno *Reconstructing head models from photographs for individualized 3D-audio processing* **Computer Graphics Forum** (Special Issue - Pacific Graphics 2008), Volume 27, Number 7, page 1719 - 1727 - **2008**

- Nico Pietroni, Fabio Ganovelli, Paolo Cignoni, and Roberto Scopigno. *Splitting Cubes: a fast and robust technique for virtual cutting.* **The Visual Computer**, 25(3):227-239, **2009**.

Bibliography

[ADMT01] V. Ralph Algazi, Richard O. Duda, Reed P. Morrison, and Dennis M. Thompson. Structural composition and decomposition of hrtfs. In *Proc. IEEE WASPAA01*, pages 103–106, 2001.

[ARM06] Sven Andres, Niklas Röber, and Maic Masuch. Hrtf simulations through acoustic raytracing. Technical report, Otto v. Guericke University of Magdeburg, Germany, 2006.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 217–226, New York, NY, USA, 2001. ACM.

[Beg94] Durand R. Begault. *3D Sound for Virtual Reality and Multimedia.* Academic Press Professional, 1994.

[BG00] D. Bielser and M. Gross. Interactive simulation of surgical cuts. In Brian A. Barsky, Yoshihisa Shinagawa, and Wenping Wang, editors, *Proceedings of the 8th Pacific Graphics Conference on Computer Graphics and Application (PACIFIC GRAPHICS-00)*, pages 116–125, Los Alamitos, CA, October 3–5 2000. IEEE.

[BKF$^+$96] T. Belytschko, Y. Krongauz, M. Fleming, D. Organ, and W.K. Liu. Smoothing and accelerated computations in the element free galerkin method. *J. Comput. Appl. Math.*, 74(1-2):111–126, 1996.

[Bla97] J. Blauert. *Spatial Hearing : The Psychophysics of Human Sound Localization.* M.I.T. Press, Cambridge, MA, 1997.

[Bla06] Volker Blanz. Face recognition based on a 3d morphable model. In *FGR '06: Proceedings of the 7th International Conference on Automatic Face and Gesture Recognition*, pages 617–624, Washington, DC, USA, 2006. IEEE Computer Society.

[BLG94] T. Belytschko, Y.Y. Lu, and L. Gu. Element-free galerkin methods. *Internat. J. Numer. Methods Engrg.*, (37):229–256, 1994.

[BLM04] Joel Brown, Jean-Claude Latombe, and Kevin Montgomery. Real-time knot-tying simulation. *The Visual Computer*, 20(2-3):165–179, 2004.

[BMG99] Daniel Bielser, Volker A. Maiwald, and Markus H. Gross. Interactive cuts through 3-dimensional soft tissue. *Computer Graphics Forum*, 18(3):31–38, September 1999.

[BS01] Cynthia Bruyns and Steven Senger. Interactive cutting of 3D surface meshes. *Computers & Graphics*, 25(4):635–642, 2001.

[BSS07] Volker Blanz, Kristina Scherbaum, and Hans-Peter Seidel. Fitting a morphable model to 3d scans of faces. In *IEEE ICCV 2007*, pages 1–8, 2007.

[Buc98] John W. Buchanan. Simulating wood using a voxel approach. *Comput. Graph. Forum*, 17(3):105–112, 1998.

[Bur83] Peter J. Burt. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, April 1983.

[BV99] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3D faces. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, pages 187–194, Los Angeles, 1999. Addison Wesley Longman.

[BW98] David Baraff and Andrew P. Witkin. Large steps in cloth simulation. In *SIGGRAPH*, pages 43–54, 1998.

[CB05] Hui Chen and Bir Bhanu. Contour matching for 3d ear recognition. In *WACV-MOTION '05: Volume 1*, pages 123–128, Washington, DC, USA, 2005.

[CB07] Hui Chen and Bir Bhanu. Human ear recognition in 3d. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(4):718–737, April 2007.

[CCCS08] Marco Callieri, Paolo Cignoni, Massimiliano Corsini, and Roberto Scopigno. Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3D models. *Computers & Graphics*, 32(3):464–473, 2008.

[CDM$^+$02] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller, and Robert Jagnow. A procedural approach to authoring solid models. *ACM Transactions on Graphics*, 21(3):302–311, July 2002.

[CH02] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2):106–131, April 2002.

[Cha74] G. Chaikin. An algorithm for high speed curve generation. *Computer Graphics and Image Processing*, 3:346–349, 1974.

[CK02] Kwang-Jin Choi and Hyeong-Seok Ko. Stable but responsive cloth. *ACM Transactions on Graphics*, 21(3):604–611, July 2002.

[CRS98] Paolo Cignoni, C. Rocchini, and Roberto Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In Jessica Hodgins and John C. Hart, editors, *Proceedings of ACM SIGGRAPH 2003*, volume 22(3) of *ACM Transactions on Graphics*, pages 287–294. ACM Press, 2003.

[D'A01] N. D'Apuzzo. Human face modeling frommulti images. In *Proc. of 3rd Int. Image Sensing Seminar on New Dev. in Digital Photogrammetry, Gifu, Japan*, pages 28–29, 2001.

[DAA99] Richard O. Duda, Carlos Avendano, and V. Ralph Algazi. An adaptable ellipsoidal head model for the interaural time difference. In *Proc. IEEE (ICASSP)*, pages II:965–968, 1999.

[dB97] J. S. de Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *SIGGraph-97*, pages 361–368, 1997.

[DCA99] Hervé Delingette, Stephane Cotin, and Nicholas Ayache. A hybrid elastic model allowing real-time cutting, deformations and force-feedback for surgery training and simulation. In *CA*, pages 70–81, 1999.

[Del98] Herve Delingette. Towards realistic soft tissue modeling in medical simulation. Technical report, Inria Rhône-alpes, Grenoble, France, September 1998.

[DEL$^+$99] Julie Dorsey, Alan Edelman, Justin Legakis, Henrik Wann Jensen, and Hans Køhling Pedersen. Modeling and rendering of weathered stone. In Alyn Rockwood, editor, *Proceedings of the Conference on Computer Graphics (Siggraph99)*, pages 225–234, N.Y., August8–13 1999. ACM Press.

[DG01] Jean-Michel Dischler and Djamchid Ghazanfarpour. A survey of 3D texturing. *Computers & Graphics*, 25(1):135–151, 2001.

[DGF98] J. M. Dischler, D. Ghazanfarpour, and R. Freydier. Anisotropic solid texture synthesis using orthogonal 2D views. In David Duke, Sabine Coquillart, and Toby Howard, editors, *Computer Graphics Forum*, volume 17(3), pages 87–95. Eurographics Association, 1998.

[DKT95] Oliver Deussen, Leif Kobbelt, and Peter Tucke. Using simulated annealing to obtain good nodal approximations of deformable objects. In Dimitri Terzopoulos and Daniel Thalmann, editors, *Computer Animation and Simulation '95*, pages 30–43. Eurographics, Springer-Verlag, September 1995.

[DLTD08] Yue Dong, Sylvain Lefebvre, Xin Tong, and George Drettakis. Lazy solid texture synthesis. *Comput. Graph. Forum*, 27(4):1165–1174, 2008.

[DMTB96] D.Organ, M.Fleming, T.Terry, and T. Belytschko. Continuous meshless approximations for nonconvex bodies by diffraction and transparency. *Computational mechanics*, 18(3):225–235, 1996.

[dPP07] Maurizio de Pascale and Domenico Prattichizzo. The haptick library: A component based architecture for uniform access to haptic devices. *IEEE Robotics and Automation Magazine*, 14(4):64–75, 2007.

[DPT$^+$08] Matteo Dellepiane, Nico Pietroni, Nicolas Tsingos, Manuel Asselot, and Roberto Scopigno. Reconstructing head models from photographs for individualized 3d-audio processing. *Computer Graphics Forum (Special Issue - Pacific Graphics 2008 Proc.)*, 27(7):1719–1727, 2008.

[DSB99] Mathieu Desbrun, Peter Schröder, and Alan Barr. Interactive animation of structured deformable objects. In *Graphics Interface*, pages 1–8, June 1999.

[EF01] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 341–346. ACM Press / ACM SIGGRAPH, 2001.

[EGS03] Olaf Etzmuss, Joachim Gross, and Wolfgang Strasser. Deriving a particle system from continuum mechanics for the animation of deformable objects. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):538–550, October/December 2003.

[EL99] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *ICCV*, pages 1033–1038, 1999.

[EMP$^+$94] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, October 1994.

[EP94] I. M. Elfadel and R. W. Picard. Gibbs random fields, cooccurrences, and texture modeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:24–37, 1994.

[EWS96] Bernhard Eberhardt, Andreas Weber, and Wolfgang Strasser. A fast, flexible particle-system model for cloth draping. *IEEE Computer Graphics and Applications*, 16(5):52–59, September 1996 1996.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FKN80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. volume 14, pages 124–133, July 1980.

[FLCB95] Kurt W. Fleischer, David H. Laidlaw, Bena L. Currin, and Alan H. Barr. Cellular texture generation. In *SIGGRAPH*, pages 239–248, 1995.

[Flo03] Michael S. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[FOT04] Kouta Fujimura, Yasuhiro Oue, and Tomoya Terauchi. Improved 3d head reconstruction system based on combining shape-from-silhouette with two-stage stereo algorithm. In *ICPR '04: Volume 3*, pages 127–130, Washington, DC, USA, 2004.

[FTP03] Matthies Hermann-Georg Fries Thomas-Peter. Classification and overview of meshfree methods. Technical Report 2003-03, TU Brunswick, Germany, 2003.

[Gan01] F. Ganovelli. Animating cuts with on-the-fly re-meshing. In *EUROGRAPHICS 2001 Short paper session*, 2001.

[Gar84] Geoffrey Y. Gardner. Simulation of natural scenes using textured quadric surfaces. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 11–20, July 1984.

[Gar85] G. Y. Gardner. Visual simulation of clouds. In B. A. Barsky, editor, *SIGGRAPH '85 Conference Proceedings (San Francisco, CA, July 22–26, 1985)*, pages 297–303, 1985.

[Gar05] W.G. Gardner. Spatial audio reproduction: Towards individualized binaural sound. *National Academy of Engineering*, 2005.

[GCMS00] Fabio Ganovelli, Paolo Cignoni, Claudio Montani, and Roberto Scopigno. A multiresolution model for soft objects supporting interactive cuts and lacerations. *Comput. Graph. Forum*, 19(3), 2000.

[GD95] D. Ghazanfarpour and J. M. Dischler. Spectral analysis for automatic 3-D texture generation. *Computers & Graphics*, 19(3):413–422, May 1995.

[GD96] Djamchid Ghazanfarpour and Jean-Michel Dischler. Generation of 3D texture using multiple 2D models analysis. *Computer Graphics Forum*, 15(3):311–324, August 1996. ISSN 1067-7055.

[GG92] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer, 1992.

[GH97] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH'97*, pages 209–215, 1997.

[GM97] Sarah F. F. Gibson and Brian Mirtich. A survey of deformable modeling in computer graphics. Technical report, Mitsubishi Electric Research Laboratories, November 1997.

[GSX00] Baining Guo, Harry Shum, and Ying-Qing Xu. Chaos mosaic: Fast and memory efficient texture synthesis. Technical Report MSR-TR-2000-32, Microsoft Research (MSR), April 2000.

[HA04] R. Hassanpour and V. Atalay. Delaunay triangulation based 3d human face modeling from uncalibrated images. *Computer Vision and Pattern Rec. Workshop*, pages 75–75, 2004.

[Hag90] C Hagwood. A mathematical treatment of the spherical stereology. *NISTIR 4370*, 1990.

[HB95] D. J. Heeger and J. R. Bergen. Pyramid-based texture analysis/synthesis. In *ICIP*, pages III: 648–651, 1995.

[HJO$^+$01] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David Salesin. Image analogies. In *SIGGRAPH*, pages 327–340, 2001.

[HS88] N. J. Higham and R. S. Schreiber. Fast polar decomposition of an arbitrary matrix. Technical Report 88-942, Computer Science, Cornell University, Ithaca, NY 14853, 1988.

[HSO03] Kris K. Hauser, Chen Shen, and James F. O'Brien. Interactive deformation using modal analysis with constraints. In *Graphics Interface*, pages 247–256. CIPS, Canadian Human-Computer Commnication Society, A K Peters, jun 2003.

[HTK98] Koichi Hirota, Yasuyuki Tanoue, and Toyohisa Kaneko. Generation of crack patterns with a physical model. *The Visual Computer*, 14(3):126–137, 1998.

[Ian89] A. Iannarelli. Ear identification. *Paramount Publishing Company, Freemont, California*, 1989.

[IMT99] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. In *SIGGRAPH*, pages 409–416, 1999.

[ITF04] Geoffrey Irving, Joseph Teran, and Ron Fedkiw. Invertible finite elements for robust simulation of large deformation. In *Proceedings of the 2004 ACM SIGGRAPH Symposium on Computer Animation (SCA-04)*, pages 131–140, 2004.

[J.D98] I.Sinclair J.Demers, J.Boelen. Freedom 6s force feedback hand controller MPB technologies inc. *IFAC Workshop on Space Robotics.*, 1998.

[JDR04] Robert Jagnow, Julie Dorsey, and Holly Rushmeier. Stereological techniques for solid textures. *ACM Transactions on Graphics*, 23(3):329–335, August 2004.

[JDR08] Robert Jagnow, Julie Dorsey, and Holly Rushmeier. Evaluation of methods for approximating shapes used to synthesize 3D solid textures. *ACM Transactions on Applied Perception*, 4(4), January 2008.

[JHY$^+$05] D.L. Jiang, Y.X. Hu, S.C. Yan, L. Zhang, H.J. Zhang, and W. Gao. Efficient 3d reconstruction for face recognition. *J. of Pattern Recogn.*, 38(6):787–798, June 2005.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of Hermite data. *ACM Transactions on Graphics*, 21(3):339–346, July 2002.

[JM06] E. Jeges and L. Mate. Model-based human ear identification. *World Automation Congress, 2006. WAC '06*, pages 1–6, 24-26 July 2006.

[KAG$^+$05] Richard Keiser, Bart Adams, Dominique Gasser, Paolo Bazzi, Philip Dutré, and Markus Gross. A unified lagrangian approach to solid-fluid animation. In Marc Alexa, Szymon Rusinkiewicz, Mark Pauly, and Matthias Zwicker, editors, *Symposium on Point-Based Graphics*, pages 125–133, Stony Brook, NY, 2005. Eurographics Association.

[Kat01] B. Katz. Boundary element method calculation of individual head-related transfer function. part I: Rigid model calculation. *Journal Acoustical Soc. Am.*, 110(5):2440–2448, 2001.

[KB07] B. Katz and D.R. Begault. Round robin comparison of HRTF measurement systems: preliminary results. In *Proc. 19th Intl. Congress on Acoustics (ICA2007), Madrid, Spain*, 2007.

[KBSS01] Leif Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature sensitive surface extraction from volume data. In *SIGGRAPH*, pages 57–66, 2001.

[KEBK05] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Transactions on Graphics*, 24(3):795–802, July 2005.

[KFCO$^+$07] Johannes Kopf, Chi-Wing Fu, Daniel Cohen-Or, Oliver Deussen, Dani Lischinski, and Tien-Tsin Wong. Solid texture synthesis from 2d exemplars. *ACM Trans. Graph.*, 26(3):2, 2007.

[KLTW07] Vivek Kwatra, Sylvain Lefebvre, Greg Turk, and Li-Yi Wei. Example-based texture synthesis. In *ACM SIGGRAPH Course Notes*, 2007.

[KN06] Y. Kahana and P.A. Nelson. Numerical modelling of the spatial acoustic response of the human pinna. *Journal of Sound and Vibration*, 292(1-2):148–178, 2006.

[KPGF07] Blazej Kubiak, Nico Pietroni, Fabio Ganovelli, and Marco Fratarcangeli. A robust method for real-time thread simulation. In Aditi Majumder, Larry F. Hodges, Daniel Cohen-Or, and Stephen N. Spencer, editors, *VRST*, pages 85–88. ACM, 2007.

[KSE$^+$03] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics, SIGGRAPH 2003*, 22(3):277–286, July 2003.

[Lar01] Véronique Larcher. *Techniques de spatialisation des sons pour la réalité virtuelle*. Thèse de doctorat, Université Paris 6 (Pierre et Marie Curie), Paris, 2001.

[LC87] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[LD04] Yi-Je Lim and Suvranu De. On the use of meshfree methods and a geometry based surgical cutting in multimodal medical simulations. In *HAPTICS*, pages 295–301. IEEE Computer Society, 2004.

[Lew87] J. P. Lewis. Generalized stochastic subdivision. *TOG*, 6:167–190, 1987.

[LH05] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. *ACM Transactions on Graphics*, 24(3):777–786, July 2005.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *ACM Transactions on Graphics*, 25(3):541–548, July 2006.

[LLH04] Yanxi Liu, Wen-Chieh Lin, and James Hays. Near-regular texture analysis and manipulation. *ACM Transactions on Graphics*, 23(3):368–376, August 2004.

[LLY04] Tong-Yee Lee, Ping-Hsien Lin, and Tz-Hsien Yang. Photo-realistic 3d head modeling using multi-view images. In *ICCSA (2)*, pages 713–720, 2004.

[LMGC02] J. Lenoir, P. Meseure, L. Grisoni, and C. Chaillou. Surgical thread simulation. volume 12, pages 102–107. Modelling and Simulation for Computer-aided Medecine and Surgery (MS4CMS), 2002.

[LMT98] Won-Sook Lee and Nadia Magnenat-Thalmann. Head modeling from pictures and morphing in 3d with image metamorphosis based on triangulation. In *CAPTECH*, pages 254–267, 1998.

[LS81] P. Lancaster and K. Salkauskas. Surfaces generated by moving least squares methods. *Mathematics of Computation*, 37(155):141–158, July 1981.

[MBF04] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *ACM Transactions on Graphics*, 23(3):385–392, August 2004.

[McG08] Tim McGraw. Generalized reaction-diffusion textures. *Computers & Graphics*, 32(1):82–92, 2008.

[MHHR07] M. Muller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, April 2007.

[MHTG05] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM Transactions on Graphics*, 24(3):471–478, July 2005.

[MK97] Geoffrey J. McLachlan and Thriyambakam Krishnan. *The EM Algorithm and Extensions*. Wiley series in probability and statistics. JohnWiley and Sons, 1997.

[MKB+08] Sebastian Martin, Peter Kaufmann, Mario Botsch, Martin Wicke, and Markus Gross. Polyhedral finite elements using harmonic basis functions. *Computer Graphics Forum [Proceedings SGP]*, 27(5), 2008.

[MKN$^+$04] Matthias Muller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa. Point based animation of elastic, plastic and melting objects. In *Proceedings of the 2002 ACM SIGGRAPH Symposium on Computer Animation (SCA-02)*, pages 141–152, 2004.

[MMD$^+$02] Matthias Müller, Leonard McMillan, Julie Dorsey, Robert Jagnow, and Barbara Cutler. Stable real-time deformations. In Stephen N. Spencer, editor, *Proceedings of the 2002 ACM SIGGRAPH Symposium on Computer Animation (SCA-02)*, pages 49–54, New York, July 21–22 2002. ACM Press.

[MMO00] J.C. Middlebrooks, E.A. Macpherson, and Z.A. Onsan. Psychophysical customization of directional transfer functions for virtual sound localization. *Journal Acoustical Soc. Am.*, 108(6):3088–3091, 2000.

[MS94] C. Montani and R. Scopigno. Using marching cubes on small machines. *Graphical Models and Image Processing*, 56(2):182–183, March 1994.

[MZD05] Wojciech Matusik, Matthias Zwicker, and Frédo Durand. Texture design using a simplicial complex of morphable textures. *ACM Transactions on Graphics*, 24(3):787–794, July 2005.

[Nie03] Han-Wen Nienhuy. *Cutting in deformable objects*. PhD thesis, May 2003.

[NMK$^+$05] Andrew Nealen, Matthias Muller, Richard Keiser, Eddy Boxerman, and Mark Carlson. Physically based deformable models in computer graphics. *Eurographics State of the Art Report, Computer Graphics Forum*, 25(4):809–836, 2005.

[OBH02] James F. O'Brien, Adam W. Bargteil, and Jessica K. Hodgins. Graphical modeling and animation of ductile fracture. In John Hughes, editor, *SIGGRAPH 2002 Conference Proceedings*, Annual Conference Series, pages 291–294. ACM Press/ACM SIGGRAPH, 2002.

[OFTB96] D. Organ, M. Fleming, T. Terry, and T. Belytschko. Continuous meshless approximations for nonconvex bodies by diffraction and transparency. *Computational Mechanics*, (18):225–235, 1996.

[OH99] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *SIGGRAPH*, pages 137–146, 1999.

[ONNI03] Shigeru Owada, Frank Nielsen, Kazuo Nakazawa, and Takeo Igarashi. A sketching interface for modeling the internal structures of 3d shapes. In *In Proceedings of the 4th International Symposium on Smart Graphics (2003*, pages 49–57. Springer-Verlag, 2003.

[ONOI04] Shigeru Owada, Frank Nielsen, Makoto Okabe, and Takeo Igarashi. Volumetric illustration: designing 3D models with internal textures. *ACM Transactions on Graphics*, 23(3):322–328, August 2004.

[PB81] S. M. Platt and N. I. Badler. Animating facial expressions. *ACM Computer Graphics*, 15(3):245–252, August 1981.

[Pea85] D. R. Peachey. Solid texturing of complex surfaces. In B. A. Barsky, editor, *SIGGRAPH '85 Conference Proceedings (San Francisco, CA, July 22–26, 1985)*, pages 279–286, 1985.

[Per85] K. Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[Per91] P. Perona. Deformable kernels for early vision. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pages 222–227, 1991.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In Sheila Hoffmeyer, editor, *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)*, pages 465–470, New York, July 23–28 2000. ACMPress.

[PGCS09] Nico Pietroni, Fabio Ganovelli, Paolo Cignoni, and Roberto Scopigno. Splitting cubes: a fast and robust technique for virtual cutting. *The Visual Computer*, 25(3):227–239, 2009.

[PK08] M. Botsch M. Gross P. Kaufmann, S. Martin. Flexible simulation of deformable models using discontinuous galerkin fem. In *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 105–115, Dublin, Ireland, 2008. Eurographics Association.

[PKA$^+$05] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutré, Markus Gross, and Leonidas J. Guibas. Meshless animation of fracturing solids. *ACM Transactions on Graphics*, 24(3):957–964, July 2005.

[PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. In Jessica Hodgins and John C. Hart, editors, *Proceedings of ACM SIGGRAPH 2003*, volume 22(3) of *ACM Transactions on Graphics*, pages 641–650. ACM Press, 2003.

[pla] planetside. Terragen: Photorealistic schenery rendering software, more info on: http://www.planetside.co.uk/terragen/.

[PLK02] Jeff M. Phillips, Andrew M. Ladd, and Lydia E. Kavraki. Simulated knot tying. In *ICRA*, pages 841–846. IEEE, 2002.

[POB+07] Nico Pietroni, Miguel A. Otaduy, Bernd Bickel, Fabio Ganovelli, and Markus H. Gross. Texturing internal surfaces from a few cross sections. *Comput. Graph. Forum*, 26(3):637–644, 2007.

[PP93] K. Popat and R. W. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. In *Massachusetts Institute of Technology, Media Lab*, 1993.

[Pro95] Xavier Provot. Deformation constraints in a mass–spring model to describe rigid cloth behavior. In *Graphics Interface '95*, pages 147–154, May 1995.

[PS00] J. Portilla and E. P. Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision*, 40(1):49–70, October 2000.

[PW89] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. *ACM Computer Graphics*, 23(4):215–222, 1989.

[QY05] X. J. Qin and Y. H. Yang. Basic gray level aura matrices: Theory and its application to texture synthesis. In *ICCV*, pages I: 128–135, 2005.

[QY07] Xuejie Qin and Yee-Hong Yang. Aura 3D textures. *IEEE Trans. Vis. Comput. Graph*, 13(2):379–389, 2007.

[RJ07] Alec R. Rivers and Doug L. James. FastLSM: fast lattice shape matching for robust real-time deformation. *ACM Trans. Graph*, 26(3):82, 2007.

[RT01] Mark A. Ruzon and Carlo Tomasi. Edge, junction, and corner detection using color distributions. *IEEE Trans. Pattern Anal. Mach. Intell*, 23(11):1281–1295, 2001.

[Sch88] H. G. Schuster. *Deterministic chaos: An introduction.* VCH, Weinheim, 1988.

[SDF07] Eftychios Sifakis, Kevin G. Der, and Ronald Fedkiw. Arbitrary cutting of deformable tetrahedralized objects. In Michael Gleicher and Daniel Thalmann, editors, *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2007, San Diego, California, USA, August 2-4, 2007*, pages 73–80. Eurographics Association, 2007.

[SF95] E. P. Simoncelli and W. T. Freeman. The steerable pyramid: A flexible architecture for multi-scale derivative computation. In *ICIP*, pages III: 444–447, 1995.

[SFAH92] E P Simoncelli, W T Freeman, E H Adelson, and D J Heeger. Shiftable multiscale transforms. *IEEE Trans Information Theory*, 38(2):587–607, March 1992. Special Issue on Wavelets.

[SHGO02] Chen Shen, Kris K. Hauser, Christine M. Gatchalian, and James F. O'Brien. Modal analysis for real-time viscoelastic deformation. In *SIGGRAPH '02: ACM SIGGRAPH 2002 conference abstracts and applications*, pages 217–217, New York, NY, USA, 2002. ACM.

[SHGS06] Denis Steinemann, Matthias Harders, Markus Gross, and Gabor Szekely. Hybrid cutting of deformable solids. In *IEEE VR 2006*. IEEE, 2006.

[SKvW$^+$92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[SL04] S.Li and W.K. Liu. *Meshfree Particle Methods*. Springer, 2004.

[SOG06] Denis Steinemann, Miguel A. Otaduy, and Markus Gross. Fast arbitrary splitting of deforming objects. In Marie-Paule Cani and James O'Brien, editors, *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 63–72, Vienna, Austria, 2006. Eurographics Association.

[ST07] J. Spillmann and M. Teschner. CORDE: Cosserat rod elements for the dynamic simulation of one-dimensional elastic objects. In Dimitris Metaxas and Jovan Popovic, editors, *Symposium on Computer Animation*, pages 63–72, San Diego, California, United States, 2007. Eurographics Association.

[ST08] Jonas Spillmann and Matthias Teschner. An adaptive contact model for the robust simulation of knots. *Comput. Graph. Forum*, 27(2):497–506, 2008.

[TDLD07] Nicolas Tsingos, Carsten Dachsbacher, Sylvain Lefebvre, and Matteo Dellepiane. Instant sound scattering. In *Proc. of the Eurographics Symposium on Rendering*, 2007.

[THK98] A. Tanaka, K. Hirota, and T Kaneko. Virtual cutting with force feedback. In *the Virtual Reality Annual International Symposium, p. 71. IEEE Computer Society, Washington, DC, USA (1998)*, page 71, 1998.

[THM$^+$03] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In T. Ertl, B. Girod, G. Greiner, H. Niemann, H.-P. Seidel, E. Steinbach, and R. Westermann, editors, *Proceedings of the Conference on Vision, Modeling and Visualization 2003 (VMV-03)*, pages 47–54, Berlin, November 19–21 2003. Aka GmbH.

[TO99] G. Turk and J. F. O'Brien. Variational implicit surfaces. Technical report, Georgia Institute of Technology, 1999.

[TOII08] Kenshi Takayama, Makoto Okabe, Takashi Ijiri, and Takeo Igarashi. Lapped solid textures: filling a model with anisotropic textures. *ACM Trans. Graph.*, 27(3):1–9, 2008.

[Tur91] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *SIGGRAPH '91 Proceedings*, pages 289–298, 1991.

[Tur01] Greg Turk. Texture synthesis on surfaces. In *SIGGRAPH*, pages 347–354, 2001.

[Und70] E. E. Underwood. Quantitative stereology. 1970.

[Van98] Allen Van Gelder. Approximate simulation of elastic membranes by triangulated spring meshes. *Journal of Graphics Tools: JGT*, 3(2):21–41, 1998.

[Vis05a] VisualComputingLab. Interactive deformable objects library. more info on: http://idolib.sf.net. 2005.

[Vis05b] VisualComputingLab. Visualization and computer graphics library. more info on: http://vcg.sf.net. 2005.

[WAKW93] E. Wenzel, M. Arruda, D. Kistler, and F. Wightman. Localization using non-individualized head-related transfer functions. *J. Acoustical Soc. Am.*, 94(1):111–123, 1993.

[Wat87] K. Waters. A muscle model for animating three-dimensional facial expression. *ACM Computer Graphics*, 21(4):17–24, July 1987.

[WBD$^+$05] Fei Wang, Etienne Burdet, Ankur Dhanik, Tim Poston, and Chee Leong Teo. Dynamic thread for real-time knot-tying. In *WHC*, pages 507–508. IEEE Computer Society, 2005.

[WBG07] Martin Wicke, Mario Botsch, and Markus Gross. A finite element method on convex polyhedra. *Computer Graphics Forum*, 26(3):355–364, 2007.

[Wei02] Li-Yi Wei. *Texture synthesis by fixed neighborhood searching.* PhD thesis, Stanford University, 2002.

[Wei03] Li-Yi Wei. Texture synthesis from multiple sources. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Sketches & Applications*, pages 1–1, New York, NY, USA, 2003. ACM.

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 55–63, New York, NY, USA, 2004. ACM.

[WK91] Andrew Witkin and Michael Kass. Reaction-diffusion textures. volume 25, pages 299–308, July 1991.

[WL00] L. Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGraph-00*, pages 479–488, 2000.

[WL01] L. Y. Wei and M. Levoy. Order-independent texture synthesis. Technical report, 2001.

[WMW86] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, August 1986.

[Wor96] Steven P. Worley. A cellular texture basis function. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 291–294. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus H. Gross. Surface splatting. In *SIGGRAPH*, pages 371–378, 2001.

[ZPvBG02] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. EWA splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

[ZZV$^+$03] Jingdan Zhang, Kun Zhou, Luiz Velho, Baining Guo, and Heung-Yeung Shum. Synthesis of progressively-variant textures on arbitrary surfaces. *ACM Trans. Graph.*, 22(3):295–302, 2003.