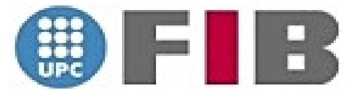


MASTER IN COMPUTING

(VISUALIZATION, VIRTUAL REALITY, AND GRAPHIC INTERACTION)

Research on Generic Interactive Deformable 3D Models

—Focus on the Human Inguinal Region—

Research Thesis for Master in Computing

Thesis Student:

Marc Musquera Moreno
[marc.musquera@upc.edu]

Thesis Director:

Antonio Susín Sánchez (MA1-UPC)
[toni.susin@upc.edu]

Thesis Reader:

Isabel Navazo Alvaro (LSI-UPC)
[isabel@lsi.upc.edu]

Spring 2008; v1.1

Acknowledgements

You will never find time for anything. If you want the time, you must make it.

- Charles Buxton

First, I would like to thank my Thesis Director Dr. Antonio Susin for guiding me through this awesome research project in physically-based animation. Many thanks also to Prof. Isabel Navazo for her help, interest, collaboration as Thesis Reader, in addition of her support in burocratic tasks during this project.

Also, I would like to thank the staff of the Master teachers, specially Profs. Carlos Andújar and Pere Pau Vázquez for their unvaluable help on voxelization and shader programming respectively, and my Master mates, Ricardo (I still remember we helping us each other with Mac peculiarities in front of PC), Quim and Jordi, for the good moments and friendly environment during all the Master.

My most sincere gratitude also to Laia, who has corrected from Toulouse, where she is achieving her PhD, the *state of the art* part in terms of making the text more comprehensive and easier to understand. Thanks also to Alba for doing the same with the introduction part.

And, overcoat, my most sincere gratitude for my three most closer people, who have supported me and have helped me improving my life while they were adapting their own to mine: my dear parents Joan and Montse, of course —I don't know what could I do without them—, and my particular jewel, Gemma, who has wanted to share with me these complex times, and have lived my good research moments as long as also supported me in the bad ones all the time. The love from you three have been essential for the success of this project.

Table of Contents

Acknowledgements I

Table of Contents III

List of Figures VII

List of Tables XI

List of Codes XIII

PROJECT'S FIRST CONTACT

1 Project Abstract 3
- 1.1 Overview . . . 3
- 1.2 Focusing on the research thesis aims . . . 4
  - 1.2.1 Interests and research fields . . . 4
  - 1.2.2 Final result . . . 5
  - 1.2.3 A non-interactive simulator . . . 5

2 Objectives and Research Contribution Overview 7
- 2.1 Motivations and interests . . . 7
  - 2.1.1 Physically-based computing animation . . . 7
    - 2.1.1.1 Computer Animation typologies . . . 7
    - 2.1.1.2 The 3D computer animation . . . 8
    - 2.1.1.3 Kinds of 3D computer animation. Focus on physically-based animation . . . 8
  - 2.1.2 Human musculature studies . . . 9
    - 2.1.2.1 Introduction to Human Body Modeling and Animation . 9
    - 2.1.2.2 Evolution of Human Animation . . . 10

2.1.2.3 Main goal for this project 11
2.1.3 Surgery simulation 12
2.1.3.1 Virtual surgery's brief overview 12
2.1.3.2 This project's approach and developed work 12
2.2 Research fields and contributions 13
2.2.1 Deformable models *over* static meshes 13
2.2.1.1 Objectives 13
2.2.1.2 Methodology Overview 13
2.2.2 Multiresolution meshes 14
2.2.2.1 Idea and motivation 14
2.2.2.2 A little brief scheme about remeshing steps 14
2.2.3 Surgery on a 3D object 15
2.2.3.1 A virtual scalpel for cutting 15
2.2.3.2 Proposal for this project 16

THE STATE OF THE ART

3 Volumetric Model from a Polygonal Mesh 19
3.1 Polygonal Meshes 19
3.1.1 Introduction to Polygonal Meshes. The Triangle Meshes 19
3.1.2 The Face-Vertex Mesh Format 20
3.2 Mesh Voxelization 20
3.2.1 The volumetric models. The voxelization process 20
3.2.1.1 Introduction to the voxel space 20
3.2.1.2 The 3D discrete topology 21
3.2.2 The Problem: Triangle VS Voxel 22
3.2.2.1 Introduction 22
3.2.2.2 The test's simplest case: gigantic meshes in low-grid voxelmaps 22
3.2.2.3 Test's generic case: voxelization of any mesh at any LOD. Alternative techniques 24
3.3 The *search tree* structures applied to voxelization 28
3.3.1 Reasons for using trees 28
3.3.2 The most common Search Trees Structures 28
3.3.2.1 Binary Space Partition Tree —BSP-Tree— 29
3.3.2.2 Kd-Tree 29
3.3.2.3 Quadtree (2D) and Octree (3D) 30
4 Polygonal Mesh Reconstruction: Multiresolution Meshes 31
4.1 Introduction to Multiresolution Meshes 31
4.1.1 The need of a multiresolution mesh capability 31

4.1.2 The IsoSurfaces 32
4.2 VoxelMap Refilling. The Model Carving 32
4.2.1 Space Carving 33
4.2.2 Voxel Carving 33
4.3 IsoSurface Extraction Algorithms 34
4.3.1 IsoSurface extraction as multiresolution mesh construction 34
4.3.2 Marching Cubes 34
4.3.2.1 Overview 34
4.3.2.2 Advantages and disadvantages 36
4.3.2.3 The isosurface-voxel intersection case LUT. Shortcomings 36
4.3.3 Marching Tetrahedron 37
4.3.3.1 Overview and shortcomings 37
4.3.3.2 The Marching Tetrahedra LUT 38

5 3D Object Model Animation/Deformation 41

5.1 Point-Oriented Deformation 41
5.1.1 The Point-Sampled Geometry 41
5.1.2 Point-sampled deformation 42
5.2 Geometric Deformation 43
5.2.1 Global or local deformation operators 43
5.2.1.1 Reminder: the jacobian matrix J_F of a function F 43
5.2.1.2 Method Overview 44
5.2.1.3 Global and local operators 44
5.2.1.4 Examples of global operator deformations 44
5.2.2 Free Form Deformation —FFD— 47
5.2.2.1 Introduction to FFD 47
5.2.2.2 Methodology details 48
5.2.2.3 Grid of Control Points 48
5.2.2.4 Capabilities and limitations 49
5.2.3 Interactive Axial Deformations —AxDf— 49
5.2.3.1 Method Overview 49
5.2.3.2 Algorithm Details 50
5.2.3.3 Advantages and lacks of AxDf 51
5.2.4 Deformation by Wires 51
5.2.4.1 Introduction and *wire* definition 51
5.2.4.2 Algorithm overview 52

6 Physically-Based Deformations 53

6.1 Introduction to deformation based on physics 53
6.1.1 Overview of time stepping deformation algorithms 53
6.1.1.1 Accuracy vs speed performance 53
6.1.2 A simple particle P as the deformation system 54
6.1.2.1 State Vector and Vector Field of a simple particle 54

6.1.2.2 The time step Δt. The numerical integrator concept. The Euler numerical integrator . . . 54
6.1.2.3 The integration system scheme . . . 55
6.1.2.4 The most used ODE integrators . . . 55
6.1.2.5 Accuracy and stability of numerical integrators . . . 56
6.1.2.6 Movement restrictions . . . 58
6.2 Dynamic Deformable Model Systems . . . 59
6.2.1 Introduction to deformable systems . . . 59
6.2.1.1 Definition of Particle System . . . 59
6.2.1.2 The deformable models as dependent-particle system. The continuum elasticity . . . 60
6.2.2 Points of View of a Physical Simulation. Solver Typologies . . . 61
6.3 Physical Simulation based on Lagrangian POV descriptions . . . 62
6.3.1 Mesh Based Methods . . . 62
6.3.1.1 Finite Element Method —FEM— . . . 62
6.3.1.2 Finite Differences Method . . . 64
6.3.1.3 Boundary Element Method . . . 64
6.3.1.4 Mass-Spring Systems . . . 65
6.3.2 Mesh Free Methods . . . 68
6.3.2.1 What are Mesh Free Methods . . . 68
6.3.2.2 Smoothed Particle Hydrodynamics —SPH— . . . 68
6.3.2.3 Meshless Deformations based on Shape matching . . . 69
6.3.3 The Hybrid Approaches . . . 69
6.3.3.1 Overview of the hybrid simulation approaches . . . 69
6.3.3.2 Pump it Up . . . 69
6.3.3.3 Mass-Spring with Volume Conservation . . . 70
6.4 Physical Simulation based on Eulerian POV descriptions . . . 71
6.4.1 Overview . . . 71
6.4.2 Fluid Simulation . . . 71
6.4.2.1 The Navier-Stokes equations for fluids . . . 71
6.4.2.2 The yielding idea of display and computation . . . 72
6.4.2.3 A small summary of variants and proposals . . . 72

7 Topological Changes on Polygonal Meshes **75**
7.1 Mesh Cutting or Mesh Splitting . . . 75
7.1.1 Introduction . . . 75
7.1.2 Motivations and applications . . . 76
7.1.2.1 Surgery simulation. Constraints and requirements . . . 76
7.1.2.2 Another applications of mesh splitting . . . 76
7.2 Mesh Dissection: Some Proposals . . . 76
7.2.1 Spring Framework Tools: Cutting Interaction . . . 76
7.2.1.1 The generic alrgorithm scheme . . . 77

7.2.1.2 Examples and applications 77
7.2.2 Finite Element Methodologies 78
7.2.3 Discontinuous FFD approach 78
7.2.4 Hybrid Cutting 79
7.2.5 Off-line virtual node algorithm based on replicas 80

Contributions and Results

8 External Used Libraries and Employed Hardware **85**
8.1 Mac Computers with MacOsX v10.5 "Leopard" 85
8.2 Used Developing Platforms 86
8.2.1 QT4.4 for GUI design with C++ as Native Language 86
8.2.2 `Matlab` for testings 87

9 Computer Graphics Programming **89**
9.1 OPENGL as the primary graphics specification 89
9.1.1 A Little Brief on Computer Graphics using OPENGL 89
9.1.2 API overview: The OPENGL graphical computing steps 90
9.1.2.1 Initial considerations; introduction 90
9.1.2.2 Drawing primitives 90
9.1.2.3 Vertex transformations; the matrix stack 92
9.1.2.4 Camera positioning and configuration; the viewport . . . 94
9.1.3 The Standard Graphics Pipeline 96
9.1.4 The OPENGL Extensions. The GLEW Library 97
9.2 The programmable graphics technology 98
9.2.1 The Shading Languages 98
9.2.2 A New Programming Paradigm 98
9.2.2.1 The GPU's Vertex and Fragment Processors 98
9.2.2.2 The New Programmable Graphics Pipeline 99
9.2.2.3 Geometry Shaders 99
9.2.3 OPENGL 2.0 and GLSL 100
9.2.3.1 Brief introduction to OPENGL 2.0 and GLSL 100
9.2.3.2 Data types and qualifiers 100
9.2.3.3 The most important implicit variables 102
9.2.3.4 A *shader* example: the *toon* shading 102
9.3 GPGPU: The New Computing Paradigma 104
9.3.1 The Programmable Graphics Pipeline, used for General Purpose Algorithms 104
9.3.1.1 The Frame Buffer Object —FBO— Extension 104
9.3.1.2 *General-Purpose Graphics Processor Unit* —GPGPU— 105
9.3.2 The nVidia CUDA approach 107

9.3.2.1 The next step in GPU computation 107
9.3.2.2 GPU as a multi-threaded coprocessor 107

10 Thesis *Software Suite*: (1) *MeshInspeQTor* 109
10.1 Application Overview 109
10.2 Loading 3D models 110
10.2.1 The OBJ file format specification 110
10.2.1.1 The geometry file *.obj 110
10.2.1.2 The material file *.mtl 111
10.2.2 The OBJ parsing with modelOBJ class 113
10.2.3 Automatic modelOBJ enhancements and improvements 114
10.2.3.1 Only triangular faces 114
10.2.3.2 Same number of vertices and normals: the per-vertex normals 114
10.2.3.3 Automatic generation of normals 115
10.2.4 IRI's own P3D format conversion to OBJ models 115
10.2.4.1 Dedicated GUI for file format conversion to OBJ format . 115
10.2.4.2 2D convex hulls for piecewise isosurface extraction merging 116
10.3 Displaying 3D models 118
10.3.1 Overview of MeshInspeQTor rendering capabilities. The Vertex Arrays 118
10.3.2 Rendering typologies 119
10.3.2.1 Solid object, wireframe and point set 120
10.3.2.2 Object Draft 122
10.3.3 Additional 3D model properties' visualization 123
10.3.3.1 Model's bounding box 123
10.3.3.2 Per-face normals 124
10.3.4 GPU dedicated algorithm for realistic Phong illumination model . . 125
10.3.4.1 Introduction to the Phong illumination model 126
10.3.4.2 The Phong reflection model shaders 127
10.4 3D model arbitrary plane culling 128
10.4.1 Introduction to the arbitrary plane culling 128
10.4.2 The *VirtualCutting* process 128
10.4.2.1 Defining the planes 128
10.4.2.2 How to process the plane culling 129
10.4.3 GPU dedicated algorithm for real-time VirtualCutter culling 132
10.5 Voxelization of polygonal meshes 134
10.5.1 Introduction of this functionality 134
10.5.2 The VoxelMap data structure. Format and access. 135
10.5.3 The octree as acceleration data structure for voxelization 135
10.5.4 The triangle-voxel intersection test 137
10.5.4.1 The 2D axis-aligned projection test 137
10.5.4.2 The cube space location in relation to the triangle 138

10.5.5 The voxel *vertizer* module . . . 139
10.5.6 Operations with mesh-generated voxelmaps based on the set theory 141
10.5.7 The Voxel Carving . . . 141
10.5.7.1 The standard *total refilling* strategy . . . 141
10.5.7.2 The thesis' innovative *hole preserving* algorithm . . . 142
10.6 Developed File Formats . . . 143
10.6.1 Scene *.SCN File Format . . . 143
10.6.2 VoxelMap *.VOX File Format . . . 144

11 Thesis *Software Suite*: (2) *ForceReaQTor* **145**
11.1 Application Overview . . . 145
11.2 From VoxelMap to 3D Mass-Spring System . . . 146
11.2.1 Overview of the main ForceReaQTor preprocess . . . 146
11.2.2 The conversion to a mass-spring system in detail . . . 147
11.2.3 Displaying the mass-spring system . . . 149
11.3 The Dynamic Force Data Structure . . . 150
11.3.1 Predefined deformations. The Timeline Simulation Specification . 150
11.3.1.1 The GUI configuration panel . . . 150
11.3.1.2 Independence between lists of dynamic forces and voxelmaps . . . 151
11.3.1.3 Data structure for dynamic forces . . . 151
11.3.2 Dynamic Factor Typologies . . . 152
11.3.2.1 Dynamic Forces . . . 152
11.3.2.2 Node Fixing . . . 152
11.3.2.3 Spring Cuttings . . . 152
11.4 The Deformation Simulator . . . 153
11.4.1 A thread-powered module; the FPS imposed rating . . . 153
11.4.2 The detailed deformation algorithm . . . 155
11.4.3 The Numerical Integrators . . . 155
11.4.3.1 Introduction to the integration algorithms . . . 155
11.4.3.2 Euler Integrator . . . 156
11.4.3.3 Verlet Integrator . . . 156
11.4.4 Force Accumulator . . . 156
11.4.4.1 Introduction to Force Accumulation . . . 156
11.4.4.2 Stretching and Bending . . . 160
11.4.4.3 Shearing . . . 161
11.5 Volume Preservation . . . 163
11.5.1 The need for volume preservation . . . 163
11.5.2 Fixing nodes for volume preservation . . . 164
11.5.3 ForceReaQTor's adopted solution: the *autostretching* . . . 164
11.6 Collision handling . . . 167
11.6.1 The need of collision handling during deformation . . . 167

11.6.2 The ForceReaQTor approach: spheres approximating cubes 167
11.6.2.1 Introduction 167
11.6.2.2 Collision testing 168
11.6.2.3 Collision processing 169
11.6.2.4 Adaptive acceleration for collision handling 170
11.7 Mass-Spring System Cuttings 172
11.7.1 Introduction to the spring cutting methodology 172
11.7.2 Repercussions on physically-based deformations 173
11.8 Accelerated Force Accumulator looping 174
11.9 Displaying the deformation system as a triangular mesh 175
11.9.1 The mass-spring system as a 3D non-rigid skeleton 175
11.9.2 A GPU-dedicated algorithm for the mesh rendering process 175
11.9.2.1 Overview of the shader program for FFD-renderization . 175
11.9.2.2 The handicap of passin the masspoints and hashing values to the GPU 176
11.9.2.3 The vertex shader algorithm 177
11.9.3 Autogenerated *skin* from the mass-spring *skeleton* 179
11.9.3.1 Introduction 179
11.9.3.2 ForceReaQTor and the deforming meshes 179
11.9.3.3 The wrongly parametrized triangular meshes 181
11.9.3.4 Inherent *pseudo*-FFD rendering adequation 183
11.9.3.5 Polygonal mesh cutting renderization capabilities 184
11.10 Developed File Formats 185
11.10.1 Dynamic Force List *.DFL File Format 185
11.10.2 VoxelMap *.VOX File Format 185

12 Tests and Results **187**
12.1 Introduction to this chapter 187
12.1.1 A brief about the developed software and the offered results 187
12.1.2 The medical-purpose three-dimensional models 188
12.1.3 Test structure, and given results 188
12.1.4 Testing Machine Specification 189
12.2 Introduction test over non medical models 189
12.2.1 Brief overview 189
12.2.2 Mesh voxelization test results 189
12.2.3 The Marching Cubes automatic mesh extraction 190
12.2.4 Lenghten and shorten forces over a model. 191
12.3 Test over medical models; the human inguinal muscle region 191
12.3.1 Brief overview 191
12.3.2 The *oblique* external muscle 193
12.3.3 The *transversus* muscle 194

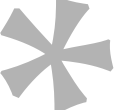
13 Project Conclusions and Future Work **195**
13.1 Contributions and Derived Conclusions 195
13.1.1 Final Conclusions 195
13.1.2 Research Contributions 196
13.1.2.1 Conservative mesh voxelization variant 196
13.1.2.2 New approach for an hybrid deformation system 196
13.2 Future Work 196
13.2.1 Persistent storage of the Marching Cubes extracted meshes 196
13.2.2 Innovative mesh splitting based on breaking voxel connectivity . . 197
13.2.3 More GPGPU-Focused Algorithms 197
13.2.4 *Haptic*-Oriented Product 198
13.2.4.1 The *workbench* prototype 198
13.2.4.2 The CAVE-oriented environment 199

Bibliography and Appendixes

A Bibliography **203**
A.1 Project's First Contact 203
A.2 The State of the Art 204
A.3 Contributions and Results 207
A.4 Conclusions and Future Work 209

B End-User *Software* Data Sheet: 'MeshInspeQTor' **211**
B.1 Product Overview 211
B.1.1 Highlights 211
B.1.2 Key Features 212
B.2 Product Details 212
B.2.1 *Models* tab: 3D Model Visualization 212
B.2.1.1 Setting the 3D scene 212
B.2.1.2 Controls and management 213
B.2.2 *VirtualCutters* tab: The Mesh Dissection Process 214
B.2.2.1 Setting a Virtual Cutters 214
B.2.2.2 Management 215
B.2.3 *Voxelizer* tab: The VoxelMaps, Volumetric Representations 215
B.2.3.1 Building a Voxelmap from a 3D scene 215
B.2.3.2 Basic Set Theory applied to Voxelization Method 216
B.2.3.3 Using the VoxelMap Navigator 217
B.2.3.4 Carving a VoxelMap 218
B.2.3.5 Management 220
B.3 Requirements and recommendations 221

C End-User *Software* Data Sheet: 'ForceReaQTor' **223**
C.1 Product Overview 223
C.1.1 Highlights 223
C.1.2 Key Features 224
C.2 Product Details 224
C.2.1 *Dynamic Force* tab: The 3D Dynamic Deformation Model settings . 224
C.2.1.1 A little introduction; the deformable skeletons and the dynamic force lists 224
C.2.1.2 The VoxelMap and the 3D non-rigid skeleton 225
C.2.1.3 Setting the Dynamic Force List 226
C.2.1.4 Controls and management 227
C.2.2 *Simulator* tab: The Real-Time Deformable Simulator 228
C.2.2.1 Deformable system visualization styles 228
C.2.2.2 Playing the simulation 228
C.2.2.3 The numerical integrators: Euler and Verlet 229
C.2.2.4 The frames per second —FPS— rate specification 229
C.2.2.5 Run-Time simulation parameters 230
C.2.2.6 Controls and management 230
C.3 Requirements and recommendations 231



List of Figures

Project's First Contact

1.1 A montage of a physicallu-based animation done with AERO simulator, a animation software whose web is www.aero-simulation.de. 3
1.2 Three steps of a dolphin mouth pulling with a $50N$ applied force —notice the yellow lines, representing the applied forces to the model—. The snapshots are from one of the two developed applications for this thesis, ForceReaQTor, that executes all the deformation simulations. 4

2.1 Examples of two and three-dimensional computer animation methodologies. 8
2.2 Some frameset examples of physically-based animation possibilities. . . . 9
2.3 The computer graphics standard human anatomy model: skin, skeleton and muscles. 9
2.4 The articulated stick human model scheme. 10
2.5 The multi-layered human body animation proposal by Scheepers et al in [mot2]. 11
2.6 The Surgery Simulation —or Virtual Surgery— can be an escellent surgeon training system. 12
2.7 The previously mentioned tensor field bunny model real-time deformations from [cnt1]. 13
2.8 The cow model'sQuadric-Error-Metric —QEM— simplification algorithm by Garland and Heckbert; source [cnt2]. 14
2.9 The Progressive Mesh —QEM— simplification algoroithm by Hughes Hoppe; source [cnt3]. 15
2.10 The virtual scalpel real-time interaction with a polygonal mesh; source [cnt4]. 15

The State of the Art

3.1 The wireframe essence of a polygonal mesh; here, the dyrtManArmTris. 19

3.2 A triangle mesh with face-vertex specification composing a cube. 20
3.3 Examples of three voxelizations of the model `dolphin`, executed by MeshInspeQTor software. 21
3.4 Voxel connectivity policies 21
3.5 2D relative position between a square and a triangle. 22
3.6 Testing $T \cap V$ by triangle-box test based on vertex containing checking; output by a Matlab code written for this thesis by its author. 23
3.7 Tipology of Bounding Boxes depending on their orientation; source [voxB] (edited). 24
3.8 The Gernot Hoffmann Triangle-Voxel intersection test. 25
3.9 The Gernot Hoffmann Triangle-Voxel intersection test results adquired by this document's author by his own Matlab programmed implementation of this proposal. 25
3.10 The first three iterations of the Sierpiński Fractal. 26
3.11 The Sierpiński Fractal `Matlab` plottings of the thesis' author own adaptation code for triangle area discretizations. 26
3.12 Scheme of a separator axis; notice that the normal projections of two meshes M_A and M_B are not intersecting; so, the two meshes are not intersecting, and thus, π_s is a separator plane. 27
3.13 A generic tree structure for being used in computer algorithm data structuration and search accelerations; in this example, **A** is the root of the tree, being **E**, **I**, **J**, **K**, **G**, **H**, **D** nodes without children, called leaves. Moreover, in the remarked subtree context, **C** is the root and **G**, **H** the leaves. 28
3.14 2D BSP construction based on a polygon, extracted from [vox8]. Notice that the satisfactory condition for stopping the BSP construction is that the two subspaces contain a convex polygon. 29
3.15 A point cloud based 2D Kd-Tree structure construction; source [vox9]. . . 29
3.16 The 3D Octree recursive subdivision; source [voxA]. 30

4.1 Multiresolution mesh of an airplane, at different LODs; source [mrs1]. . . . 31
4.2 Scheme of IsoSurface renderization as a triangular mesh. 32
4.3 Voxelization of the `sphere` model. Notice the emptiness in the voxelmap section, so the volumetric model is not filled but empty, like the original convex mesh. Output frameset from MeshInspeQTor. 32
4.4 The Space Carving algorithm in action; source [mrs4] (edited). 33
4.5 Detailed methodology for the Voxel Carving algorithm; source [mrs5] (edited). 33
4.6 Voxel Carving applied to a standard $64 \times 64 \times 64$ voxelization of the *torus* model. Since the final voxelmap is the intersection of the six voxel carving steps, the volumetric model of `torus` is filled. As usual, frameset created by the thesis purpose software MeshInspeQTor. 35
4.7 The 15 absolute cube-isosurface intersection situations; source [mrs8]. . . 36
4.8 An example of hole in a marching cubes generated mesh, as the result of an ambiguous case; source [mrs8] (edited). 37
4.9 The Marching Tetrahedra cell subdivision scheme. Each voxel will be treated as five independent tetrahedrons, so the method is more accurate and smooth with the discrete shape of volumetric unit than marching Cubes; source [mrs8]. 38
4.10 The Marching Tetrahedra cell subdivision scheme; source [mrs8]. 38

5.1 Hybrid system featured in [geo2], consisting in a hybrid geometry representation combining point clouds with implcit surface definitions. Note: the middle image is the colored sample density map. 41
5.2 Scheme of the contact handling framework developed in [geo3]. 42
5.3 Penalty and friction force processing for Γ_1 in front of a Γ_2 collision; source [geo3]. 43
5.4 Example of point-sampled object animation with the [geo4] method; the two left figures are representing the *phyxel-surfel* representation and the MLS reconstructed mesh. 43
5.5 The `cube` and `teapot` undeformed models. 45
5.6 The 'taper' effect on `cube` and `teapot` models; source [geoA]. 45
5.7 The 'twist' effect on `cube` and `teapot` models; source [geoA]. 46
5.8 The 'bend' effect on `cube` and `teapot` models; source [geoA]. 46
5.9 The 'vortex' deformation over a solid primitive; source [geo7]. 47
5.10 A complete deformation *animation* is shown in this couple of figures: (a) two local FFD are applied to the bar —only one is shown—, for transformating the bar onto a 'phone receiver', and (b) after, a global FFD bends the 'receiver'; source [geoC]. 47
5.11 There is a complete Free Form Deformation —FFD— executed over several objects (cubes and spheres) clearly embedded in the bounding flexible plastic; source [geoC]. 49
5.12 A `sphere` axis deformation: on the left there's the undeformed sphere with the defined axis, also undeformed; on the right, the sphere, deformed in an intuitive and consistent way with the axis deformation; source [geoD]. 50
5.13 The `horseshoe` axis deformation. Please notice the axis, now adapted to the object surface; thus, the deformation can be so complex as can be seen in the right image; source [geoD]. 50
5.14 An example of a facial modeling by 'wire' specification, and the next animation and deformation by 'wire' manipulation; source [geoE]. 51
5.15 Deformation of a point P to P_{def} by a wire W. The figure parameter p_R is the correspondence point, within R, of P; source [geoE]. 52

6.1 Architecture of an iterative integration system for animation by deformation; source [phy4]. 55
6.2 Stability scheme by ODE system plottings depending on the time step h; source [phy3]. 57
6.3 Intersection between 3D bounding volumes of 3D objects. 58
6.4 Force equilibrium in a particle system with connection restrictions; source [phy9]. 59
6.5 ODE numerical integration scheme for a single particle P. 59
6.6 Some ε elastic strain deformation parameters, from left to right: *Cauchy*, *Cauchy (simplified)* and *Green*; source [phyA]. 60
6.7 Visual results of the two object animation description point of views; source [phy5]. 61
6.8 A scheme of the approximate deformation $\tilde{u}(m)$ of a continuous deformation $u(m)$, done by the Finite Element Method; source [phy5]. 62
6.9 Hexahedral finite element volumetric representation of a topologically inconsistent mesh, ready for FEM application (notice that each hexahedron is one of the E_j subdivisions or elements); source [phy5]. 64

6.10 A typical mass-spring system three-dimensional representation; source [phy5]. 66
6.11 Effect of viscosity on a physically-based elastic deformations. Each figure has ten times higher viscosity than its left neighbour; source [phyC]. . . . 66
6.12 Here's the scheme of the three different internal forces for the 2D cloth mass-spring simulation model, including the connection affecting and a detailed visualization of the different force effects. 67
6.13 A cloth piece with four different hanging —relaxing— postures after a real-time simulation steps by shaking the cloth and sliding it over obstacles before allowing it to rest; source [phyD]. 67
6.14 Interface tension force SPH-based simulation between fluids of different *(i)* polarity, *(ii)* temperature diffusion and *(iii)* buoyancy; source [phy5]. . . . 69
6.15 Stable original shape recovering after a —literally— rigid and heavy deformation; [phy5]. 69
6.16 The Chen and Zeltzer finite element mesh and contraction approach. 70
6.17 A silver block catapulting some wooden blocks into an oncoming 'wall' of water; source [phy5]. 71
6.18 Sand-based bunny model deformation; source [phy5]. 72
6.19 A liquid drop falling into a liquid pool. In the six-figure frameset can be clearly noticed the three layers of the system: the Lagrangian layer *(blue)*, the bridge connection layer *(green)* and the Eulerian one *(red)*; source [phyB] (edited). 73
6.20 Here is the velocity field for a certain instant time of the above frameset. Notice the communication between the Lagrangian and Eulerian layers by the connection stage; source [phyB]. 73

7.1 Frameset for a fractured spherical shell; source [cut1]. 75
7.2 The interactive deformation with cutting capability algorithm scheme from [cut3]. 77
7.3 Virtual Scalpel cutting a 2-layered surface, top (left) and side (right) views; source [cut3]. 77
7.4 The cutting plane sequence implementation over a cube model; notice the cutting sequence from left to right; source [cut7] (edited). 78
7.5 A cut movement to a initially regularly triangulariez surface model with delaunay triangulation on the cut primitives; source [cut8]. 78
7.6 The interactive mesh cutting implemented by Sela, Schein & Elber in [cut5]. 79
7.7 The scheme for the primitive approximatting cutting process by the [cut2] method. 79
7.8 The interactive mesh cutting implemented by Steinemann et al in [cut2]. . 79
7.9 A complete frameset example of a volumetric mesh cutting by the [cut2] method. 80
7.10 High resolution cutting shell of the 3D volumetric complex `armadillo` model (380K tetrahedrons), by using the virtual node algorithm at [cut1]. 80
7.11 Tearing a plastically fragmentation deformation to the `armadillo`'s high resolution volumetric mesh; source [cut1]. 81

CONTRIBUTIONS AND RESULTS

8.1 Mainly used tools for the thesis' *software suite* design and implementation. 86
8.2 The Trolltech's QT4 versatile capabilities and framework diagram. 87
8.3 The Mathworks' `Matlab` mathematic environment logo. 87

9.1 The OPENGL logo by Silicon Graphics Inc. 89
9.2 The OPENGL primtive drawing styles; source [gpuE]. 91
9.3 The resulting rendered triangle from CPP code 2.3. 92
9.4 A scheme of the three basic geometric transformations: 'translation', 'scaling' and 'rotation', with the implicit correspondence between the transformation matrix and the OPENGL command. 92
9.5 The order of the multiplying applied geometric transformations is important: it's not the same a translation applied to a roation (left) that a rotation applied to a rotation (right); source [gpuE]. 93
9.6 The resulting scheme of the matrix stack CPP code 2.4; source [gpuE]. . . . 93
9.7 The coordinate system transformations from scene specification to scene screen displaying; this figure is taken directly from [gpuF]. 94
9.8 The two OPENGL available volume viewing specification and the camera positioning and orientation methods; figure sources from [gpuF]. 95
9.9 Overview of OPENGL method for display, on screen displays, geometric primitives —in the figure they have been markes as triangles due to the most common case— as pixels; source [gpu3]. 96
9.10 Detailed scheme of the OPENGL's Standard Graphics Pipeline; source [gpu4]. 97
9.11 The GPU's vertex and fragment processor managment tasks; source [gpu3]. 98
9.12 Detailed scheme of the Programmable Graphics Pipeline; source [gpu4]. . 99
9.13 The effects of two shaders applied to two models: the upper figures show the graphical results for a shader computing the Phong illumination model, and in the lower ones are featured the same models with the toon shading shader over them. This has been possible by the shader addition to this thesis' developed software MeshInspeQTor. 103
9.14 nVidia's CUDA is the next (theoretical) step on GPGPU computing. 107
9.15 The new nVidia CUDA-compatible GPU architecture; source [gpuD]. 107

10.1 Informal UML diagram of MeshInspeTor internal functionalities. 109
10.2 The schematic triangularization for generic polygonal faces —here, an hexagon— 114
10.3 Illumination contrasts between vertex's normal assigment in relation to the face comprising, or the normal assignment by the mean of normals per each vertex. 115
10.4 An example of internal oblique inguinal muscle, extracted by using the IRI tool; please notice the defficiences on the extraction —the model is rendered by remarking the polygon edges—. 115
10.5 The specific GUI within MeshInspeQTor for converting `IRI-P3D` format files to `OBJ`. 116
10.6 An example of a 2D convex hull, executed over a 2D point set. 117
10.7 A graphical scheme for an entire process of Jarvis March 118

10.8 MeshInspeQTor's main rendering offerings. Notice that the `dolphins.obj` model is really composed by three dolphins, each one clearly identified in the 'model list' at the top of the right panel. 121
10.9 MeshInspeQTor's 'draft' rendering style. Notice that, because of the *flat* face shading, the triangular faces are perfectly detailed and can be clearly observed. 123
10.10 Renderization of a model with its bounding box surrounding it, in MeshInspeQTor. 123
10.11 Renderization of a model with its per-face normals shown, in MeshInspeQTor. The model is rendered with the 'draft' style for a better per-face normal visualization. 124
10.12 Graphical scheme of Phong equation for the selftitled illumination model: here the light is white, the ambient and diffuse colors are both blue, and the specular color is white, reflecting almost all of the light hitting the surface, but only in very narrow highlights. The intensity of the diffuse component varies with the direction of the surface, and the ambient component is uniform (independent of direction). Figure extracted from [ami8]. 126
10.13 Differences between OpenGL standard illumination rendering and a Phong model implemented over the programmable GPU pipeline. Notice the more specular accuracy on the left part of the model, as well as the more delimited shaded zones in the right one. 127
10.14 Graphical scheme of a negative space located point p, and the $q \equiv p'$ point, being the p projection to the plane π. 130
10.15 Results example of applying a VirtualCutter to a generic three-dimensional model with MeshInspeTor. 131
10.16 An arm dissected by two VirtualCutters, featured on MeshInspeQTor. Please notice that the computer positive space is the intersection of all the positive spaces of all VirtualCutters. 132
10.17 The 'triangle-voxel' testing implemented in MeshInspeQTor, based on 2D axis projections and implicit parallelogram approximation of the triangle. 137
10.18 The 'triangle-voxel' testing conflict situations; notice the particular location and orientation of the triangle in relation to the cube. The test will validate as intern every external triangle with this particularity. 139
10.19 2D projection of the triangle-voxel postfiltering test based on space location of the eight cube —a square here— vertices in relation to the triangle plane —the thick plane on the figures—. 139
10.20 A clear examples of applied set theory; by considering the squares M_a and M_b as three-dimensional volumetric representations, the application on mesh voxelization is noticeable. 141
10.21 A $n \times n$ linear tracking —the $\mathcal{X}^+$ one indeed— of a cylinder voxel carving. 142

11.1 Informal UML diagram of ForceReaQTor internal functionalities. 145
11.2 The total conversion from a 3D model to its mass-spring system —the voxelmap is omitted but it can be deduced from the mass-spring system nodes. 149
11.3 The ForceReaQTor dynamic force configuration panel —*zoomed*—. 150
11.4 The dynamic force configuration panels for 'fixing' and 'cutting' classes. . 152
11.5 The ForceReaQTor implemented restricting forces, in a 2D scheme. 157

11.6 ForceReaQTor applying 'stretch' and 'bend' deformation effects to a $4\times4\times4$ cube after fixing the upper vertices and applying gravity force; notice that the combined deformation offers more equilibred results. 161
11.7 One of the four diagonals of a single cube. 162
11.8 A $4 \times 4 \times 4$ cube mass-spring system where it's applied 'shear' deformation forces together with some 'stretch' and 'bend' factors after applying gravity to the deformation model. 163
11.9 The shape/volume preserving example problem. 163
11.10 Fixing nodes as volume preservation solution example. 164
11.11 *Autostretching* as volume preservation solution example. 166
11.12 Scheme of the ForceReaQTor mass-point volumetric approximation buy using a sphere instead of a cube. Obviously, the sphere and the cube has the same center, and moreover, the sphere is tangent to all the cube faces. . . . 167
11.13 Scheme of sphere intersection; if $||c_2 - c_1|| \equiv L \leq D$, it must be ensured that the two spheres S_1 and S_2 are intersecting. 168
11.14 Scheme of the per-mass-point adapting collision testing. 170
11.15 Physically-based deformation with real-time cutting behaviour modification. 173
11.16 pseudo-FFD rendering process for deforming a triangular mesh w.r.t. its inner non-rigid skeleton —the mass-spring system indeed—. 175
11.17 The texture compression schemes for the desired shader data: the gray cells of the final matrices indicate unused and value-emptied cells. 176
11.18 ForceReaQTor deformable model rendering types: only the model skeleton —mass-spring system— or the model's skin —its triangular mesh, that lays over the mass-spring system—. 177
11.19 2D curve generation by the Marching Squares —2D Marching Cubes— technique. The example's threshold value for curve extraction is 5. 179
11.20 Example of ForceReaQTor autogenerated mesh of the highly featured $4 \times 4 \times 4$ cube; the normals are per-face configured, but by applying the Phong illumination GLSL program —section 10.3.4—, the renderization is robust. 180
11.21 Three-dimensional scheme of which voxels have to be checked for adjacency with a certain voxel, if it's desired to compute the isovalue of the LUF voxel vertex. 181
11.22 Example of ForceReaQTor extracted mesh of the 'left external oblique human muscle', wrongly extracted from the Visual Human Project with the `extractor` software provided by the IRI at UPC; notice the simplicity of the extracted mesh by using Marching Cubes —w.r.t. the original mesh—, but also its consistency in relation of the original. 182
11.23 An example of deformation force set applied to the $4\times4\times4$ cube Marching Cubes autoextracted mesh in ForceReaQTor. 184
12.1 The original `bunny.capped` polygonal mesh surface model. 189
12.2 The resulting voxelmaps from the `bunny.capped` voxelization processes. 190
12.3 The resulting marching cubes' autoextracted meshes from `bunny.capped` voxelmaps. 191
12.4 Lenghten (a,c,e,g) and shorten (b,d,f,h) force set deformation examples over the `dolphins` model. The rendering process has been at 25 FPS, with 'stretch' restrictive forces and Euler integrator. The forces are all $20N$ amount. 192

12.5 A deformation force set of the `obliqext` model, based on a dense $64\times64\times64$ voxelmap; here there is an abbuse of the `simacc` adaptative methodology for computing Force Accumulators. The integrator is the Verlet one using a 30 FPS animation, and the forces are $20N$ all 193
12.6 A deformation force set for the `transversus` model, based on a $32\times32\times32$ voxelmap; here can be seen the influence of a cutting on a dynamically deformable model. Here there is 30 FPS animation with a Verlet integrator with ‘stretching’ and ‘bending’; the force is $30N$ amount 194

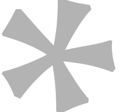
13.1 Scheme of a possible splitting for a marching cubes’ extracted mesh. . . . 197
13.2 Examples of ‘workbench-haptic’ virtual environment systems. 198
13.3 The CAVE virtual environment scheme. 199

Bibliography and Appendixes

B.1 MeshInspeQTor rendering the `dolphins` model with materials and phong algorithm illumination. Notice the three application panel functionalities: *models*, *virtualcutters* and *voxelizer*. 211
B.2 From up to down, wireframe, draft —with normal displaying; see below— and nodeset renderings of the `dolphins` model. Notice in the right panels the information about the `dolphin1` entity, already selected: 285 vertices composing 564 faces. 213
B.3 Here’s the `bunnycapped` model in dissection by a VirtualCutter. 214
B.4 The previous and final `dolphins`’ $<32\times32\times32>$ voxelmap automatic generation steps. 215
B.5 Two fish 3D models, `fish1` and `fish2`, will be used to obtain complex voxelizations by using basic set theory. Notice that the entity individual bounding boxes are really intersecting. 216
B.6 The $<32\times32\times32>$ `fish1` $\cap$ `fish2` automatic voxelization. Let’s see that the valid nodes only those that were valid on the both two voxelizations. 216
B.7 The $<32\times32\times32>$ VoxelMap equivalent to those voxels that are containing `fish1` but not `fish2`: so, the set theory operation is $[\texttt{fish1} \cap \overline{\texttt{fish2}}]$. 217
B.8 Voxelizer Navigator’s manipulation example. Notice the added *cube* on the upper left screen part, within the `torus` voxelmap. 217
B.9 A voxelmap finite range visible with the Voxelizer Navigator’s range visualization option. 218
B.10 The thesis carving purpose scene `sphecylcub` $<64\times64\times64>$ VoxelMap. Notice the emptyness of the cylinder inner space, so the contained objects —cube and sphere— are also empty and clearly visualizable. 218
B.11 Portion of the $<64\times64\times64>$ `sphecylcub` voxelization that shows the absolute emptyness of the standard 3D scene objects voxelization. 219
B.12 Space Carving policy selection dialog. 219
B.13 Space Carving done with ‘Total Refilling’ policy. Notice that all the inner zone corresponding to the more exterior voxelization is totally refilled. . . 220
B.14 Space Carving done with ‘Hole Preserving’ policy. Notice that the more exterior voxelization is refilled, but the inner ones are considered holes, so they are still empty. 220

B.15 MeshInspeQTor software and hardware requirements logos. 221

C.1 ForceReaQTor's simulation example by deformable forces to dolphin model. 223
C.2 An example of $< 16 \times 16 \times 16 >$ skeleton-node-modeling from sphere model voxelmap. 225
C.3 The preview of a previously specified dynamic force —yellow lines, maintaining the specified force vector direction, on the 'front right leg' zone—, applied to the cow skeleton. 226
C.4 The preview of a fixed nodes application —red remarked nodes, on the 'head' zone—, also to the cow skeleton. 226
C.5 The preview of a specified cutting —orange remarked lines, on the 'front left knee' zone—, applied to the cow skeleton too. 227
C.6 The ForceReaQTor's Marching Cubes extracted mesh rendering of a bunny. 228
C.7 The FPS error message thrown when the framerate cannot be achieved. . . 229
C.8 ForceReaQTor's fatal destiny: destabilized simulation. 230
C.9 ForceReaQTor software and hardware requirements logos. 231



List of Tables

PROJECT'S FIRST CONTACT

... ∅ ...

THE STATE OF THE ART

3.1 The progressive data increasing with Sierpiński Fractal adaptation . 27

4.1 Advantages and disadvantages of the Marching Cubes algorithm; extracted details from [mrs7]. 36

6.1 Iterative ODE integration estimated errors in relation to the numerical integrator. 56

CONTRIBUTIONS AND RESULTS

8.1 Used Mac computers' specification table. 85

9.1 Basic vectorial data types on GLSL 101
9.2 Basic texture data types on GLSL 101
9.3 Vectorial component indexes in relation to the data type conventions. 101
9.4 Most important 'uniform' automatically shader input variables. . . . 102
9.5 Most important 'attribute' automatically shader input variables. . . 102
9.6 Most important 'varying' automatically shader input variables. . . . 102

10.1 Table of supported tokens for the developed OBJ model loader. . . . 110

10.2 Table of supported tokens for the developed MTLLIB model loader. 112
10.3 Supported enumeration for the vertex array data block assignment. 119
10.4 Table of relative positions of a 3D point in relation to a plane. 130
10.5 Table of supported tokens for the developed OBJ model loader. . . . 143

11.1 Exemplification of the brutal increasing of mass-point cardinality on a 1/8-occuping deformable system by only changing the voxelmap grid size. 158
11.2 Adaptive neighbour selection for collision testing w.r.t. a mass-point moving direction. The 'selected neighbours' idetificators are corresponding to the featured ones in the figure 11.14. 172
12.1 Computational times for the 'voxelization' and 'voxel carving' processings. There have been tested the timings for two different voxelmap sizes: $32 \times 32 \times 32$ and $128 \times 128 \times 128$. 189
12.2 Computational times for the 'per-face' and the 'per-vertex' marching cubes proceedings. 190

Bibliography and Appendixes

B.1 MeshInspeQTor mouse controls. 214

C.1 ForceReaQTor mouse controls. 227
C.2 ForceReaQTor simulation hot key controls. 231

List of Codes

Project's First Contact

... ∅ ...

The State of the Art

3.1 *pseudo*: Triangle-box test by vertex checking. 24
3.2 *pseudo*: Triangle-box test based on Sierpinski fractal adaptation. . . 26

4.1 *pseudo*: The Marching Cubes isosurface extraction algorithm. 37

6.1 *pseudo*: vast algorithm for a deformation animation. 53
6.2 *pseudo*: Algorithm for FEM application to a problem P. 63
6.3 *pseudo*: Algorithm for mass-spring system deformation 65

7.1 *pseudo*: Interactive real-time cutting algorithm scheme. 77

Contributions and Results

9.1 *pseudo*: OpenGL animation scene drawing steps. 90
9.2 *cpp*: Drawing primitives with OpenGL. 91
9.3 *cpp*: A colorized triangle drawn with OpenGL. 92
9.4 *cpp*: An example of OpenGL matrix stack usage. 93

9.5 *pseudo*: Steps for executing a GLSL shader program. 100
9.6 *glsl*: The 'vertex shader' of toon shading. 103
9.7 *glsl*: The 'fragment shader' of toon shading. 103
9.8 *cpp*: Creation of a FRAMEBUFFER with a previous GLEW usage. . . . 104
9.9 *cpp*: Creation of a FRAMEBUFFER with a previous GLEW usage. . . . 105
9.10 *cpp*: Drawing the GPGPU quad, using **texture rectangles** —texture coordinates identical to pixel coordinates—. 106
9.11 *cpp*: Drawing the GPGPU quad, using **2D textures** —texture coordinates normalized to the range [0,1 106
9.12 *cpp*: Transferring the framebuffer content to a CPU array. 106

10.1 *pseudo*: OBJ file text specifying a simple cube. 111
10.2 *pseudo*: MTLLIB containing the specifications for 3 colors. 112
10.3 *pseudo*: OBJ file text specifying a simple cube with materials. 113
10.4 *pseudo*: Jarvis March for computing a 2D convex hull. 117
10.5 *pseudo*: Developed idea for merging isosurface portioned extractions. 118
10.6 *cpp*: Solid, wireframe and point-based rendering capabilities. 120
10.7 *cpp*: Draft rendering mode OPENGL commands. 123
10.8 *cpp*: Offline process for determining the normal factor for a single model. 124
10.9 *cpp*: Per-face normal renderization process. 125
10.10 *glsl*: Vertex shader of Phong reflection equation. 127
10.11 *glsl*: Fragment shader of Phong reflection equation. 128
10.12 *glsl*: Vertex shader of real-time VirtualCutter activation. 133
10.13 *glsl*: Fragment shader of real-time VirtualCutter activation. 134
10.14 *cpp*: Internal declaration of a voxelmap. 135
10.15 *cpp*: Internal declaration of an octree. 135
10.16 *cpp*: Expansion of an octree node. 136
10.17 *cpp*: Axis-aligned projection triangle-cube testing. 138
10.18 *cpp*: Getting voxel cell coordinates of every model vertex, and its offset w.t.r. this voxel's center. 140
10.19 *pseudo*: MeshInspeQTor scene file format. 144
10.20 *pseudo*: MeshInspeQTor voxelmap file format. 144

11.1 *cpp*: Internal data structure for mass-spring systems. 146
11.2 *cpp*: Voxelmap conversion to a mass-spring system: mass point generation. 148
11.3 *cpp*: Mass-spring system renderization adequation: node-spring generation. 149
11.4 *cpp*: Mass-spring system Vertex Array renderization code. 150

11.5 *cpp*: Dynamic Force internal data structure. 151
11.6 *cpp*: Dynamic Force Container internal data structure. 151
11.7 *cpp*: ForceSimulator main deformation thread execution. 154
11.8 *cpp*: ForceReaQTor per frame mass-spring renderization update. . . . 154
11.9 *cpp*: `ForceSimulator` internal handled data. 155
11.10 *cpp*: Euler numerical integration algorithm. 156
11.11 *cpp*: Verlet numerical integration algorithm. 156
11.12 *cpp*: Force Accumulator sequential force application. 158
11.13 *cpp*: Per-spring Force Accumulator calculus. 160
11.14 *cpp*: Longitudinal restrictive force application. 160
11.15 *cpp*: Stretch and Bend effect application algorithm. 161
11.16 *cpp*: Shear effect application algorithm. 162
11.17 *cpp*: Per-phantom-spring Force Accumulator calculus. 165
11.18 *cpp*: AutoStretching effect application algorithm. 166
11.19 *cpp*: Collision testing between two mass-points. 169
11.20 *cpp*: Euler collision handling function 169
11.21 *cpp*: Euler collision handling function. 170
11.22 *cpp*: Euler's adaptive neighbour selection for collision testing. . . . 171
11.23 *cpp*: Verlet's adaptive neighbour selection for collision testing. . . . 171
11.24 *cpp*: Internal data structure for spring cutting handling. 173
11.25 *cpp*: The data structure capable of accelerating the force accumulator loopings. 174
11.26 *pseudo*: Mass-point 'affected' flag configuration within Force Accumulator. 174
11.27 *glsl*: Vertex shader for rendering deformated triangular meshes. . . 179
11.28 *cpp*: The rough ForceReaQTor Marching Cubes algorithm. 184
11.29 *pseudo*: ForceReaQTor dynamic force list file format. 185

BIBLIOGRAPHY AND APPENDIXES

... ∅ ...

PART I

Project's First Contact

CHAPTER 1

Project Abstract

1.1 Overview

3D **polygonal meshes** are, as a simple overview, collections of **vertices** and *polygons* defined by vertex pair junctions, here called **faces**. They are, with no doubt, a **good and efficient discrete modelling approximation of a physically real object**, widely used in Computer Graphics, because this kind of objects are **representing the surface of an object** —not its volume; the objects are empty indeed—.

On the other side, the **simulation of animation or movement** in 3D Computer Graphics has been an **important research field** for some years ago, since the appearance of Pixar® entertainment films. Not only the modelling but also the **simulation of realistic movement is essential** for a good computing animation.

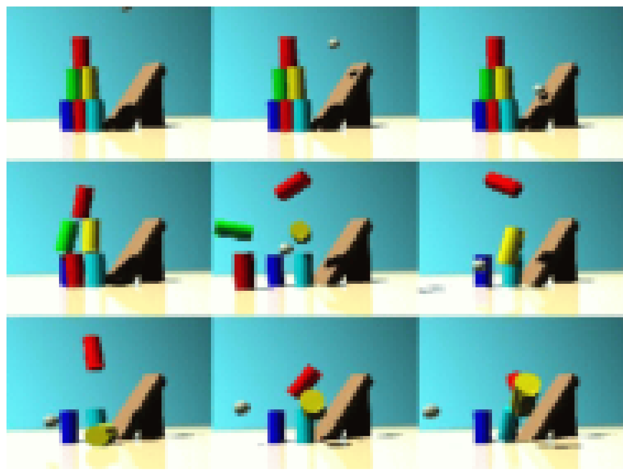


Figure 1.1: *A montage of a physicallu-based animation done with* AERO *simulator, a animation software whose web is* www.aero-simulation.de.

However, there are two different classes of computer animation, in relation to computational time —not to the methodology complexity—:

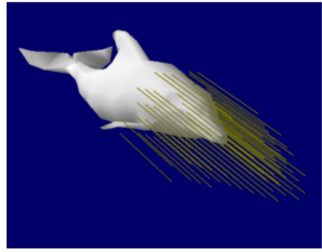
- **exact behaviour**, useful for awesome performance with no matters of time, specially coherent for computer animation films, due to there isn't real-time algorithms[1];
- **approximate behaviour**, not exact but approximately realistic, faster and more efficient. These techniques are the adequate for real-time applications like videogames or simulators, and the interesting ones for this thesis purposes;

1.2 Focusing on the research thesis aims

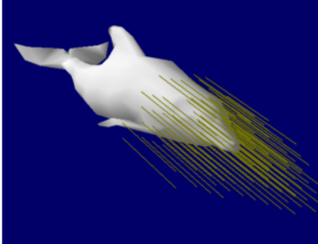
1.2.1 Interests and research fields

The goal of this project is **to research for real-time approximate methods of physically-based animation in conjunction with static polygonal meshes with the aim of deforming them and simulating an elastic behaviour for these meshes**. Because of this, in this project **it has been developed a *software suite*** capable of doing a lot of tasks, each one from different computer graphics research fields, conforming a **versatile capability project**; so, the developed applications can briefly:

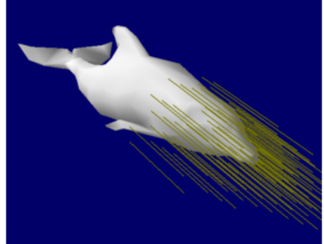
- **specify a volumetric representation** of any polygonal mesh, and will be able to be empty or entirely solid;
- building a **reconstruction** of any mesh, at any desired LOD, directly proportional on the volumetric representation;
- interact with any polygonal mesh by a **real-time physically-based deformable dynamic model** with run-time configurable parameters, based on deforming a 3D non-rigid **skeleton built from a volumetric representation**;
- **split** any polygonal mesh in the same way as a real solid object by a virtual scalpel;



(a) *Initial step, before any external force application. The* 3D *model is relaxed, no internal forces are activated.*



(b) *Deformation in curse; the restricting internal forces are not yet performing but the elasticity is activating them .*



(c) *Force totally applied; now the shape-managing restricting forces are avoiding an uncontrolled deformation.*

Figure 1.2: *Three steps of a* `dolphin` *mouth pulling with a* $50N$ *applied force —notice the yellow lines, representing the applied forces to the model—. The snapshots are from one of the two developed applications for this thesis,* ForceReaQTor*, that executes all the deformation simulations.*

[1] As also happens with the illumination methods like *raytracing*, which are incredibly realistic but too much computationally expensive for real-time applications.

1.2.2 Final result

All the above cited characteristics will be possible and fully-functional due to the two previously mentioned applications comprising an **entire dynamic simulation *software suite* based on physical laws**:

1. a **mesh visualizer** with an integrated volumetric representation generator;
2. a **dynamic model simulator** that allows force applying and a kind of virtual scalpel, and simulates a physically-based deformation behaviour to the entered 3D models;

So, though this software is **able to deform any kind of 3D objects with a shape-maintaining restricted elastic behaviour** —as it can be seen in figure 1.2, where a dolphin's mouth is the aim of a deformation force—, this interface will be used for a **simulation of a human anatomy, focused on the inguinal region**, by interacting with the polygonal meshes as approximation of the *Visual Human Project* gigantic data.

1.2.3 A non-interactive simulator

Pitifully, due to the lack of 2D desktop applications, **the simulator hasn't a run-time interactive user external force interface** with the simulator. This implies, it has been developed a **complete external force inducer** through a discretized time-line, that allows to create, to modify and to delete external forces as long as specifying cuttings or object fixed portions.

There was the intention of developping an interactive simulator within a workbench equipped with a haptic device, but it hasn't been possible by a matter of time.

CHAPTER 2

Objectives and Research Contribution Overview

2.1 Motivations and interests

2.1.1 Physically-based computing animation

2.1.1.1 Computer Animation typologies

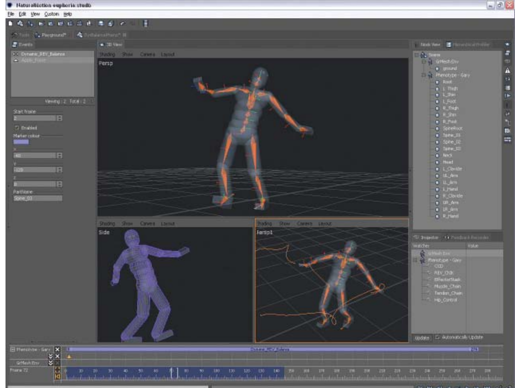
It's known that the generic concept of **animation** refers to show a set of subsequent static images that, displayed coherently fast, offers a **feeling of motion**. So, the computer animation field don't differ at all: the basic **computer animation** consists in an **automatic sequence of computer generated frames**. However, how this frame sequence differ each other is the problem of computer animation:

- the simplest solution is **programming explicitly each frame**, but it's at the same the most difficult way;
- there are other techniques about **specifying —parametrizing— the description of the scene** —composed by geometric objects—. Thus, the task of specifying all the image sequences is transformed to the **task of specifying how the scene changes through the time**.

Thanks to the last decade, when the first computer animation movies appeared and the visualization quality of the videogames has been increased exponentially, a **wide variety of computer animation** techniques have been created and used and improved. However, as it's said in [mot1], these animation techniques can be subdivided in two fields:

- the 2D techniques, mainly worried about **image manipulation**;
- the 3D techniques, focused overcoat in building huge three-dimensional worlds with also three-dimensional objects and characters, and the **way of interaction** between each other;

(a) *the classical emphsprite 2D animation.*



(b) *A 3D model with mobile articulation nodes.*

Figure 2.1: *Examples of two and three-dimensional computer animation methodologies.*

2.1.1.2 The 3D computer animation

The 2D techniques will be omitted because this research thesis is about the **animation using 3D techniques**, where the clearly different three steps are:

1. **model the scene**, describing and placing the different elements of the scene; for this task there is a great offer on modeling 3D tools as long as multiple 3D model data formats with their own specifications and data storage management;

2. **animate the scene**, specifying how the scene elements has to be moved in the world in relation to time —or a user actions—;

3. **render**, converting the instant scene description in screen displayed images;

It's true that, really, the first and last steps are not really an animator task, but **although the *model* stage can be avoided by taking previously built models, the *render* stage is really a must-have in computer graphics**, and computer animation is a branch of computer graphics —CG—. Hence, **this thesis will focus the *animation* and *render* stages**.

2.1.1.3 Kinds of 3D computer animation. Focus on physically-based animation

The task of **specifying a computer object animation is surprisingly difficult**. There are a lot of subtle details that have important weight for getting a coherent animation, and due to that, there are multiple developed techniques for developing three-dimensional animated scenes:

- **keyframing**, that uses inverse kinematics, or linear interpolation for moving objects following determined **mathematical equations**. It also receives the name of **geometric deformation**, because the 3D object is animated with a deformation due to geometric shape modification.

- **motion capture**, consisting on **position sensor capturing** within a real object, and after reproducing the different sensor movements in the modelled *clone* object nodes.

- **procedural methods**, based on using determined **iterative algorithms** that computing and generating the animation at any moment.

This thesis is focused on procedural methods, and concretely, in the *physically-based techniques*. These techniques make use of **physical laws —or a numeric approximation to those laws**[2] **to generate realistic motion**, and due to this, are the most engineeral and programming techniques in front of the more animator talent techniques.

Physycally-based simulation is obviously **realistic if the model is compatible to the physical characteristics** of the method (*e.g.*, particle systems or flexible surfaces), but the main disadvantage is the must-be **depth understanding of physical laws**.

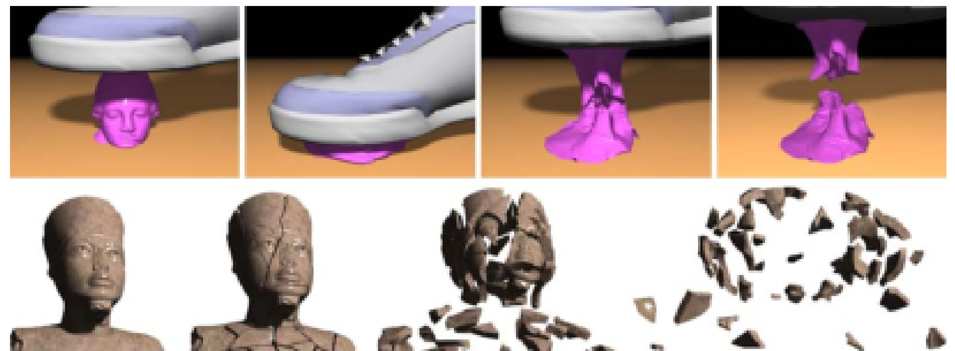
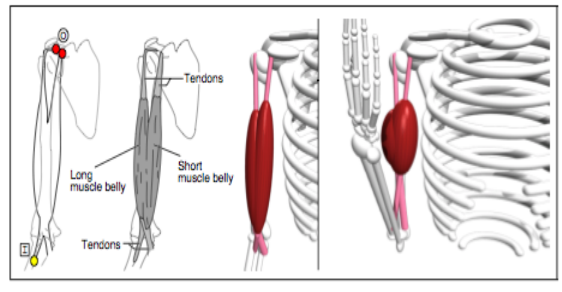


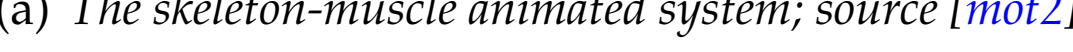
Figure 2.2: *Some frameset examples of physically-based animation possibilities.*

2.1.2 Human musculature studies

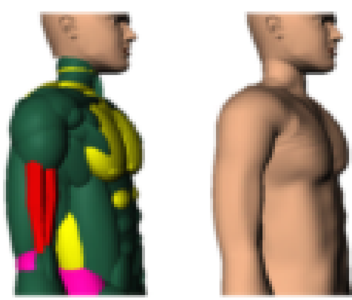
2.1.2.1 Introduction to Human Body Modeling and Animation

Since the 1970's, one of the most interesting areas of computer graphics have been the **human figure modeling and animation**. Due to the human body complexity, its movement simulation has an equivalent difficulty —the human body is capable to execute a huge number of movements—; it's known that **the human body is composed mainly by a conglomerate of skeleton, muscles, fat and skin**, as can be seen in the following computer graphics example system figure:





(a) *The skeleton-muscle animated system; source [mot2].*



(b) *Skin over skeleton-muscle; [mot3].*

Figure 2.3: *The computer graphics standard human anatomy model: skin, skeleton and muscles.*

[2] An approximation is maybe required in the cases —more common than the really desired— where the exact law model is too much complex for being computed in a reasonable computational time. It has to be noticed that the aim of this simulations is to offer a real-time simulation, so the physical behaviour has to be implemented in direct relation to the human smooth *continuous* vision, between 20 and 40 frames per second —FPS—, although a frame rate of 10 FPS is acceptable in term of *interactivity* instead of *real-time* simulation.

The more standard animated model is —according to [mot2] and [mot5]—:

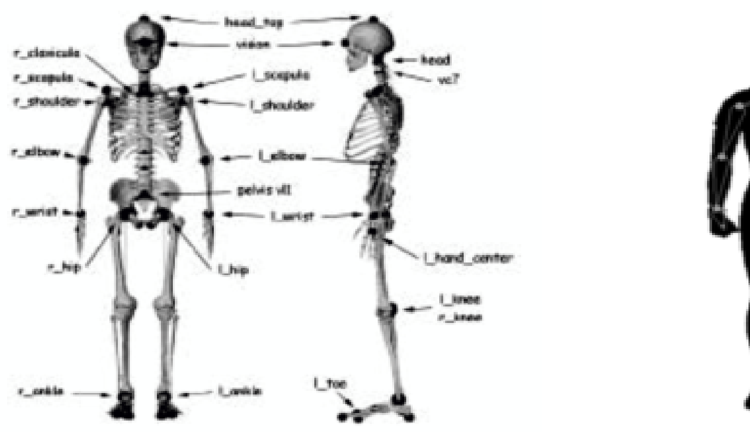
- the **skeleton**, the vertebrate body, is composed by a **hierarchical tree structure**, as it's used, for example, in [mot3] and [mot5], that links the bones between the others, with tree nodes on the articulations with information based on the predefined articulation mobility with respect to their degrees of freedom —DOF—;
- the **muscles**, independently each others, are composed by a *inner* skeleton covered by a polygonal mesh; thus, the skeleton is moved and the mesh follows the skeleton;
- the idea behind the **skin** is the same for the polygonal muscle mesh;
- and the **fat**, for computer graphics, is modeled into the skin model because the fat is a tissue located beneath the skin, *between* muscles and skin;

2.1.2.2 Evolution of Human Animation

The goal of the human animation has always been the looking for **realism of the behaviour movements**. hence, during the last 30 years there have been **a lot of approaches**, and obviously, the more the time passes, the more complexity these approaches have.

▷ ***Articulated Stick Figure Models***

So, the first approaches were involved in **articulated stick figure models** consisting in a hierarchical set or rigid segments —bones— connected at joints —articulations—; besides, **each joint could have up to three DOF** —*Degrees of Freedom*— to become a real human articulation.



(a) *The front and lateral views of the human skeleton, with the remarkes joints; source [mot3].*

(b) *An computerized stick articulated male; source [mot1].*

Figure 2.4: *The articulated stick human model scheme.*

▷ ***Surface Models***

Due to the **limitations** of the articulated stick model— —mainly the impossibility to perform some kind of movements like *twist*—, another approach appear: the **surface models**, composed by a skeleton surrounded by surfaces, made from planar or curved patches that simulate the skin. Thalmann and Thalmann developed in [mot6] the **Joint-dependent Local Deformation** animation algorithm —JLD—, a method based on a **local deformation operators set** that controls the evolution of these surfaces.

▷ ***Volume Models***

Additionally, there is another proposal: a model that approximate the structure of a 3D objet by a grid of volumetric primitives, such as ellipsoids or cylinders; these are the **volumetric models**[3]. However, they suffer from **non easy control management**, due to the huge number of comprising-primitives during the animation.

▷ ***Multi-layered Models***

In this approach, a **skeleton is the support for intermediate layers** that simulate the body —this layers are really an abstract concept of simulation for skin, muscles, or tissues—. Therefore, the animator only has to define the *connections* between the layers, because animation is an iterative process that:

1. motion specification at joints;
2. deformation according to the new positions;
3. mapping of the overlying layer vertices to the world space;
4. relaxing algorithm for adjusting the elastic equilibrium forces;

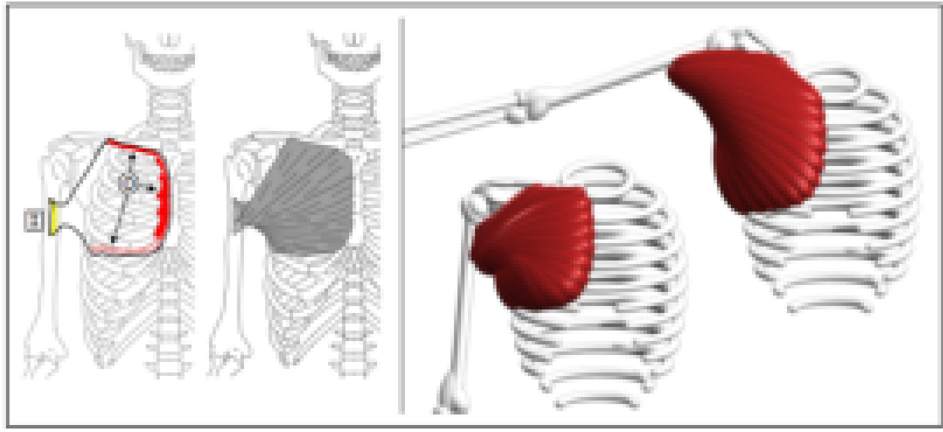


Figure 2.5: *The multi-layered human body animation proposal by Scheepers et al in [mot2].*

2.1.2.3 Main goal for this project

This **thesis' interests are focused on the muscle parts, concretely on their physical behaviour and movement simulation but besides, about non-anatomic movements but surgery interactive movements**. So, the skin, fat, and human skeleton will be omitted for being out-of-range.

Additionally, the developed animation model within the range of this thesis will be able to **deform any 3D model with the muscle's *viscuoelastic* behaviour**, because there will be some configurable parameters, and therefore, any 3D object will **be capable to be treated as a cuddly toy**.

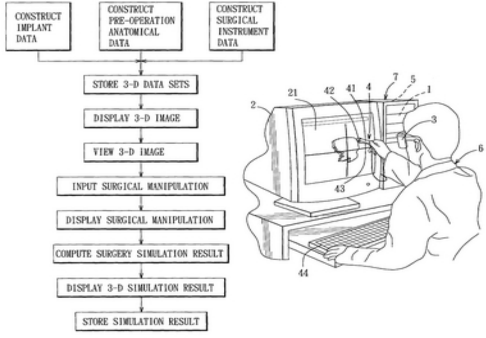
[3] And the VoxelMaps are the evolution of these volume models, since the cubes —*voxels*— can occupe all the object volume without intersecting each other.

2.1.3 Surgery simulation

2.1.3.1 Virtual surgery's brief overview

The computer-assisted surgery is one of the most interesting computer graphics — with robotics— research fields; but also the **surgery simulation is an interesting computer graphics field due to surgeon training tasks**.

(a) *The skeleton-muscle animated system at [mot9].*



(b) *A Patented Surgery Simulation System ([motA]).*

Figure 2.6: *The Surgery Simulation —or Virtual Surgery— can be an escellent surgeon training system.*

The usage of **3D interactive visualizations of important organs or anatomic parts can improve surgeons' understanding of complex human inner structures**. SO, multiple envinronments have been developed in the last years:

1. 2D desktop applications;
2. desktop-based 3D-based systems;
3. workstations and workbenches;
4. virtual reality environments, like the CAVE[4] environment;

These so different virtual systems can get responsed from a lot of user-application interfaces like keyboard, mouse, haptic devices, . . . it's obvious that **surgery is an inherent 3D task, so the 2D environment and/or devices are unsuitable**. However, developing an entire 3D virtual environment is a huge effort, out-of-range of this thesis.

2.1.3.2 This project's approach and developed work

In this thesis **the focus will be the physically-based simulation itself, and no device interaction has been developed**; so, there won't be any *software* implemented for workstations nor touch-sensing devices —*haptics*—.

However, the muscle's behaviour behaviour will be fully functional due to an **entire simulation software, developed for a 2D desktop *software suite***, that will feature a complete external force and cutting specification through a simulation time-line.

[4] Acronym of *Cave Automatic Virtual Environment*, it's an immersive virtual reality environment where projectors are directed to three, four, five or six of a room-sized cube walls. The name is taken from the Plato's Cave allegory, where a philosopher contemplates perception, reality and illusion.

2.2 Research fields and contributions

2.2.1 Deformable models *over* static meshes

2.2.1.1 Objectives

The first research field dealed with will be the **physically-based realistic animation for generic 3D models** —although the main idea is to use the results with medical models, concretely the **muscles of the human inguinal region**—.

More concretely, the aim of this thesis is to build a system capable of loading any 3D model and apply to it a **generic deformable system capable to perform the movements of a muscle**; that is:

1. the **contraction or elasticity of the model** when a force is applied, with an implied shorten or longed length;
2. the **relaxing and return to their former length and shape** after the applied forces;

Besides, according to anatomist theoretical and technical words, for achieving a generic deformation model, **only the gross muscle anatomy** —composed by all the organism muscles— will be treated, **rejecting the muscle microanatomy**, which comprises the structures of a single muscle.

2.2.1.2 Methodology Overview

The main idea is to **build a 2-layered model for each 3D model**, composed by:

- a 3D non-rigid skeleton as the underlying layer, based on the volumetric representation of the original 3D model;
- a polygonal mesh representing the surface model, that cannot be necessarily the original 3D mesh.

The 3D skeleton will be, as it has been said, **non-rigid**; that means, it will be able to receive elasticity and contraction forces, similar to the idea featured in [cnt1], where tensor fields are applied to skeleton nodes.

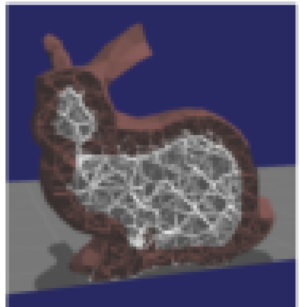


Figure 2.7: *The previously mentioned tensor field* `bunny` *model real-time deformations from [cnt1].*

Additionally, the polygonal mesh will be **mapped** onto the 3D skeleton, so it will be rendered according to the skeleton deformation.

So, an iterative procedural method based on physical laws, called **mass-spring system**, will be used, and each skeleton node will be literally connected to other direct neighbour nodes and will be **reacting to these neighbours own reactions**.

At the same time, a ***place and shape memory* will be included to all the skeleton nodes**, that will be useful for

- for the **relaxing return node movements** after force applying;
- for restricting, limiting and literally enclosing the possible node deformations when forces are affecting;

Additionally, a geometric displacement method, after a parametrized model place location, will manage the covering polygonal mesh —the original 3D model indeed— to deform it in direct proportion of the skeleton; thus, the visualizable **model will be deformed reacting in real-time to the 3D skeleton deformation**.

2.2.2 Multiresolution meshes

2.2.2.1 Idea and motivation

Another aim of this generic real-time animation software project, part of this thesis, if the **remeshing of 3D models** in direct proportion quality of the previously mentioned 3D non-rigid skeleton. The **reasons** for doing this are multiple:

1. having a **varying 3D object quality** in relation to the animation quality;
2. **improve a possible 3D model bad specification** —*i.e.*, bad poligonization, or possible uncoherent holes—;
3. a **better management of the model data**, due to the remeshing has been generated by a controlled algorithm.

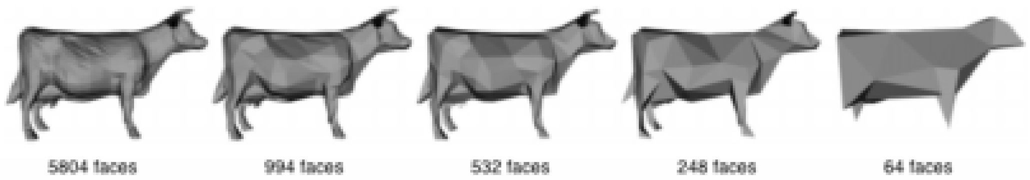


Figure 2.8: *The* COW *model'sQuadric-Error-Metric* —QEM— *simplification algorithm by Garland and Heckbert; source [cnt2].*

2.2.2.2 A little brief scheme about remeshing steps

For a remeshing generation of any 3D mesh, there is no special contribution, though is **not** int the way of the classical quadric-error-metric simplification algorithm by Garland and Heckbert, readable from [cnt2] nor the Hughes Hoppe progressive meshes from [cnt3]; the featured method is composed by **two steps**:

1. to **build a volumetric model** composed by constant-value arbitrary regular-sized boxes called *voxels* filled with the appropriate values for best 3D model approximation; this process receives the name of *voxelization*;
2. from this model approximation, a mesh extraction algorithm will be execute over that to obtain a **automatically generated mesh**.

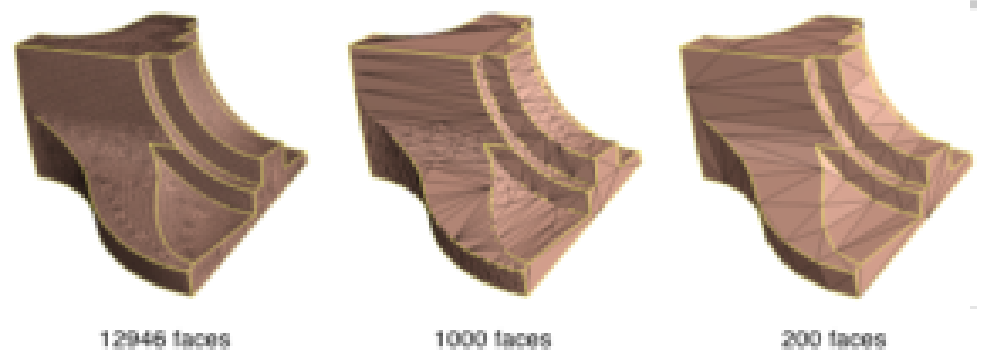


Figure 2.9: *The Progressive Mesh —QEM— simplification algoroithm by Hughes Hoppe; source [cnt3].*

2.2.3 Surgery on a 3D object

2.2.3.1 A virtual scalpel for cutting

In the last recent years, there have been some approachs for, apart from real-time model deformation, **cutting a 3D model in real-time computing**. That often **implies** this algorithmic steps (or at least some of them, depending on the proposal basis scheme):

- a **mesh retriangulation** because some mesh-comprising polygons will be subdivided into more little polygons, or if the polygon number increasing is forgiven, those affected by the cutting must be reconfigured;
- a **mesh reconnectivity**, due to the new blank space between some existing previously-connected polygons;
- the **feeling of interacting with a non-empty object** —a 3D model will be composed by a polygonal mesh representing only the model surface—, so a **polygon addition to the empty space will be necessary**;
- besides, if the *scalpel* is desired to be a virtual weighted object, two more implications have to be counted: the **scalpel collision detection with the object**, and a resulting **scalpel pushing force** from the cutting intention;

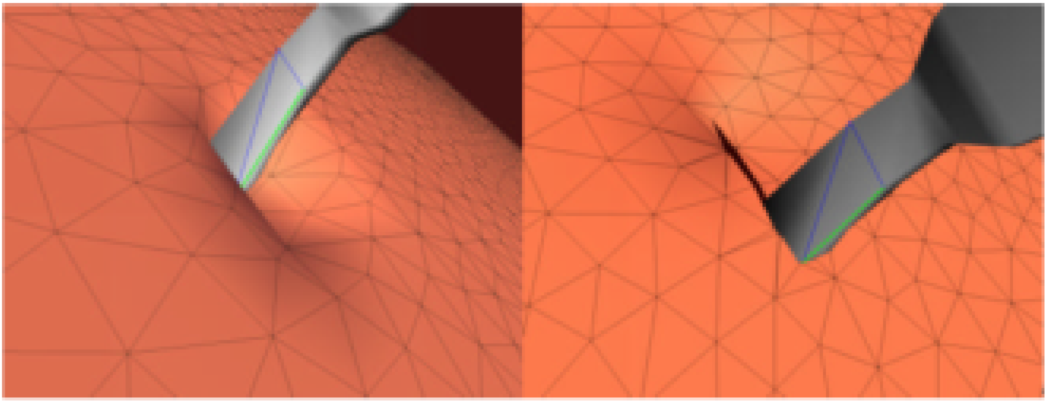


Figure 2.10: *The virtual scalpel real-time interaction with a polygonal mesh; source [cnt4].*

2.2.3.2 Proposal for this project

In spite of these great proposals, a new contribution is tried to feature in this project. **Based again on the 3D model's volumetric skeleton**, the idea is **to cut the skeleton node connections instead of model's primitives**; thus, with a **remeshing of the broken zone** —with a security perimeter volume also remeshed and reconnected to the global model— the cuttings will appear **always accurate in relation to the —approximate— volumetrical topology**.

In other words, this thesis offers an interactive cutting method that:

- it's **based on the 3D skeleton mesh**, basis of every project's deformation method;
- it **doesn't split any primitive, nor really cuts the model**;
- the **cutting action is skeleton-connectivity affected**, so the mesh only has to be restructured, and besides **does implie**:
  - **no** polygon number increasing,
  - **no** primitive subdivision,
  - and **no** difficult recomputations.

PART II

The State of the Art

CHAPTER 3

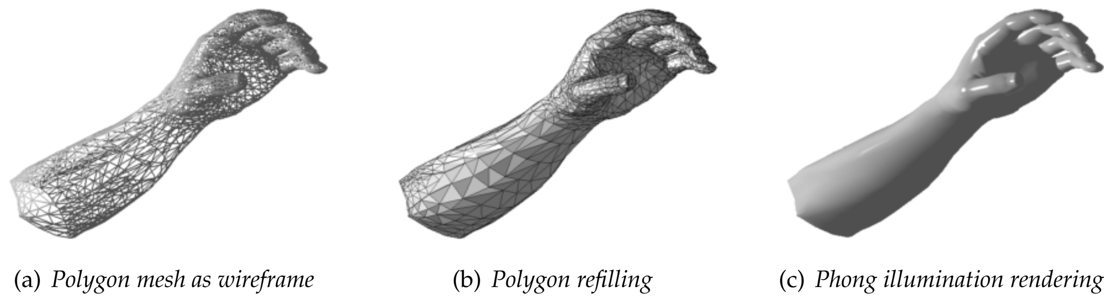
Volumetric Model from a Polygonal Mesh

3.1 Polygonal Meshes

3.1.1 Introduction to Polygonal Meshes. The Triangle Meshes

A **polygonal mesh** is the basic shape that we use to build our 3D model, or better said, its **surface**, parametrized by a collection of **polygons** —composed by a set of vertices that indicate the edges of every polygon—. On the other side, it's known that a mesh is any of the open spaces in a net or network; that means, an **interstice**. In front of this apparent contradiction, more explanation is needed.

The reason a polygonal mesh is called a *mesh*, even though the form looks **solid** when we do a mesh rendering, is that the previously cited **surface is actually a wireframe** (a mesh of interconnected wires) that gives the surface its shape; even, this object (assuming it's convex) is **empty**. In the following figure can be seen this phenomenon, with snapshots from one of the two applications developed for this thesis, MeshInspeQTor:



(a) *Polygon mesh as wireframe* (b) *Polygon refilling* (c) *Phong illumination rendering*

Figure 3.1: *The wireframe essence of a polygonal mesh; here, the* `dyrtManArmTris`.

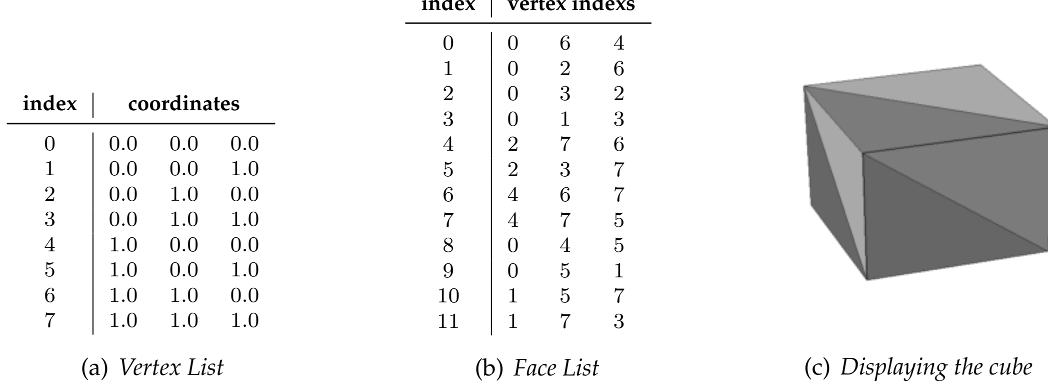
Since the collection of polygons forming the mesh are called **faces**, it's obvious that every polygon must define a **plane**. Because of this, the most used polygons are the **triangles** —three vertices always define a plane—.

3.1.2 The Face-Vertex Mesh Format

There are a lot of mesh specifications, as commented in [vox1], but the most common one is the **face-vertex mesh format**. It consists in two lists:

1. a **list of vertices**, consisting in the coordinates of every vertex;
2. a **list of faces**, consisting in a set of indexes, indicating which vertices are forming the cited face —polygon—;

Here is a simple example of a *cube* with the face-vertex specification, where every face is a triangle —a cube is composed by 8 vertices and 6 quads (12 triangles)—:



index	coordinates		
0	0.0	0.0	0.0
1	0.0	0.0	1.0
2	0.0	1.0	0.0
3	0.0	1.0	1.0
4	1.0	0.0	0.0
5	1.0	0.0	1.0
6	1.0	1.0	0.0
7	1.0	1.0	1.0

(a) *Vertex List*

index	vertex indexs		
0	0	6	4
1	0	2	6
2	0	3	2
3	0	1	3
4	2	7	6
5	2	3	7
6	4	6	7
7	4	7	5
8	0	4	5
9	0	5	1
10	1	5	7
11	1	7	3

(b) *Face List*

(c) *Displaying the cube*

Figure 3.2: *A triangle mesh with face-vertex specification composing a cube.*

3.2 Mesh Voxelization

3.2.1 The volumetric models. The voxelization process

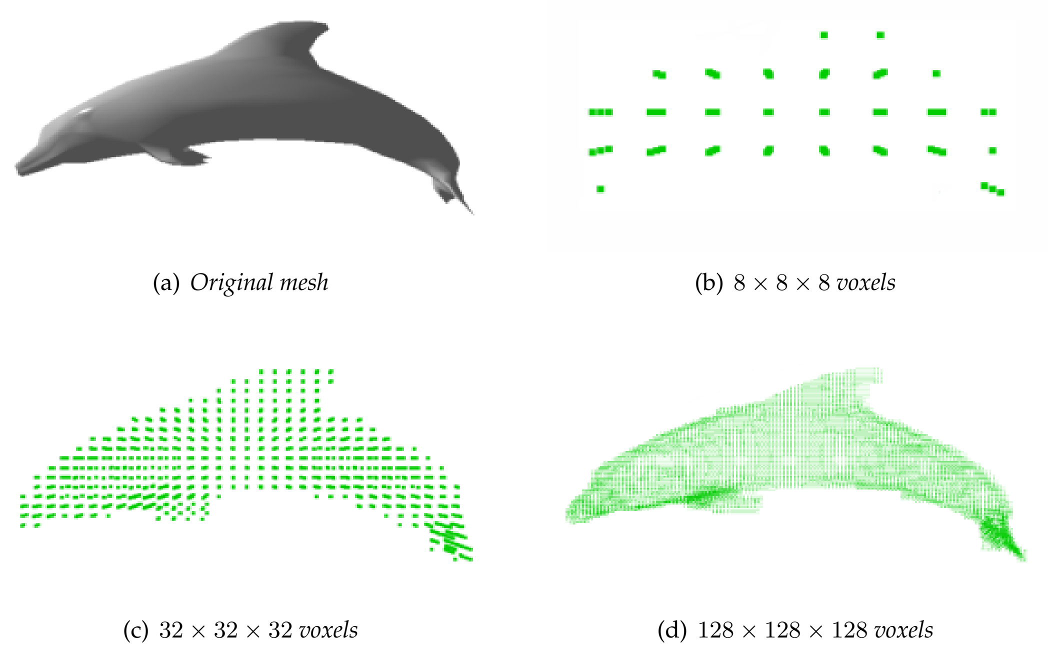
3.2.1.1 Introduction to the voxel space

Let's divide the **computer graphics space into a grid of** $n \times n \times n$ **tangent boxes** containing the volumetric space inside every box, or *cube*. Then the space has been transformed to a voxel space, or **voxelmap**, and every box is called **voxel**, abbreviation for *volume element*. Conceptually, the 3D voxel is the 3D counterpart of the 2D pixel, and in the same way, exactly like a **voxel** is the **quantum unit of volume** —it has a numeric value (or values) associated with some measurable properties of the polygonal mesh scene—.

So, through a parallelism with the process of making a photograph of the real world[5], it's called **voxelization**, according to [vox2], the process of transformating a **geometric representation (a polygonal mesh, or a parametric surface) into a voxelmap** where the associated voxel values are representing the *best* approximation of the scene model within the discrete voxel space.

[5] More accurate and technical, the more adequate counterpart of voxelization is the **rasterization** —screen drawing by pixels— of 2D geometric objects, although voxelization process does not render the voxels but generates a discrete digitization of a continuous 3D object.

Here can be seen three different voxelizations of the same model, `dolphin`; notice the ascendent accuracy as long as the number of voxels in the grid increase:



(a) *Original mesh* (b) $8 \times 8 \times 8$ *voxels*

(c) $32 \times 32 \times 32$ *voxels* (d) $128 \times 128 \times 128$ *voxels*

Figure 3.3: *Examples of three voxelizations of the model* `dolphin`*, executed by MeshInspeQTor software.*

3.2.1.2 The 3D discrete topology

Like any discretization process, voxelization is not infallible; that means, the voxelization can result in the **existence of valid voxels considered invalid ones, which can make harder the voxel entity identification** with respect to the voxelmap. For solving this problematic situations, a **connectivity postfilter** is needed.

This filter **passes through the voxel space**, or voxelmap, and analyses the connectivity of different voxels considering **three possible adjacency settings**; hence, one voxel is connected to another one by following these three policies from [vox3]:

- 6-adjacency, where two voxels are considered linked if they are adjacent by any entire face;
- 18-adjacency, where the connectivity is accepted for the previous 6 voxels, and also their upper and lower ones;
- 26-adjacency, where the entire surrouding voxels for each voxels are candidates;

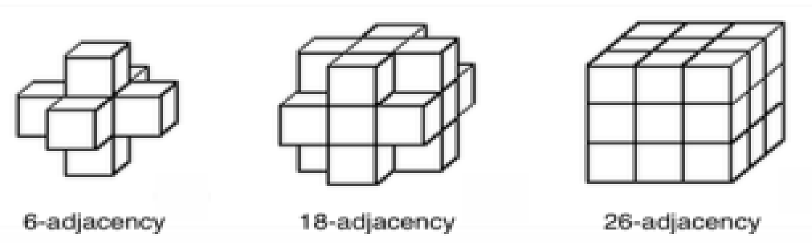


Figure 3.4: *Voxel connectivity policies*

3.2.2 The Problem: Triangle VS Voxel

3.2.2.1 Introduction

Given a mesh M, composed by a set of faces F, and a voxelmap X, the voxelization process $\mathcal{V}_X(M)$ that modifies X will set valid voxels —that is, voxels that are intersecting any face composing the mesh— to *true* boolean value, and invalid voxels to *false*; so, the final result is a **discrete approximation by cubes within X of the mesh M**, and it implies that, supposing M a convex mesh (thus, empty), **there will be invalid voxels outside as long as inside the voxelization** of M, $\mathcal{V}_X(M)$.

More accurately, focusing in the **intersection test between a voxel V and a triangle** T[6], both two in $\mathbb{R}^3$, their relative position is given by the **existence of a finite and delimited coplanar zone between them**.

This is, two conditions must be accomplished, one more restrictive than the previous:

1. the T's containing plane must intersect the voxel —cube— V:

$$T \cap V \neq \emptyset \iff T \subset \pi \in \mathbb{R}^3 \;\wedge\; \pi \cap V \neq \emptyset$$

2. T —or a piece of it— must intersect V:

$$\boxed{Q \equiv \pi \cap V \implies T \subseteq Q}$$

Indeed, the problem is equilavent to its 2D analogue discussion: coplanar triangle-square relative position. In the next figure the four relative positions between a triangle and a square are featured:

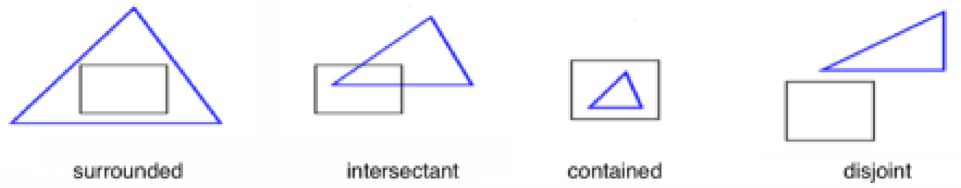


Figure 3.5: 2D *relative position between a square and a triangle.*

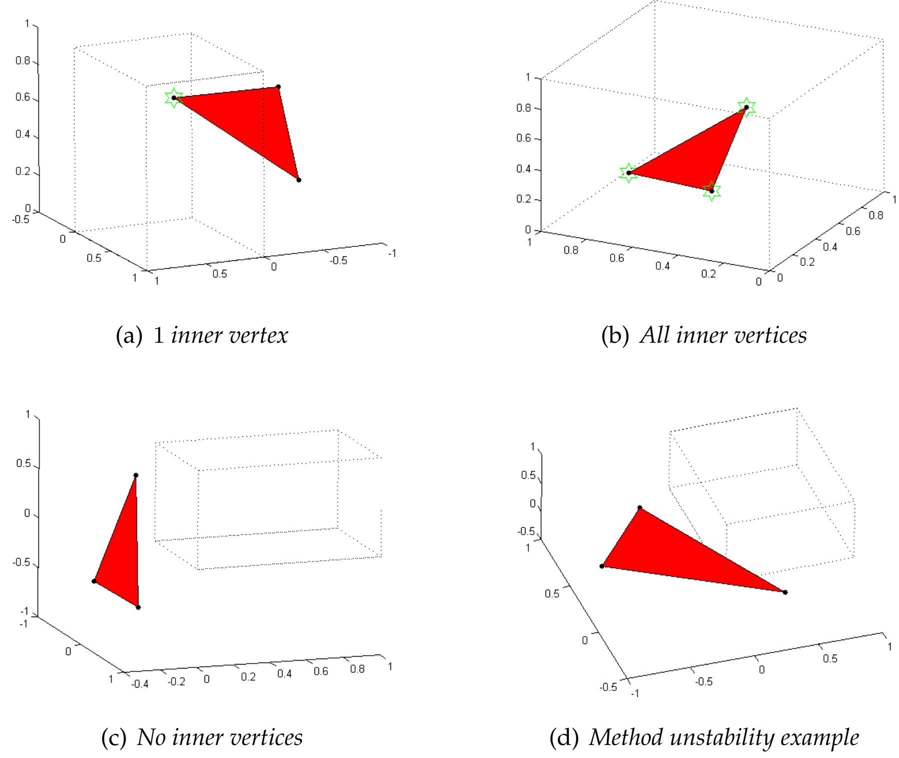
3.2.2.2 The test's simplest case: gigantic meshes in low-grid voxelmaps

When a gigantic mesh M, composed by $n \gg 10000$ faces, is voxelized over a voxelmap V with a grid of big voxels, it can be assumed that, if the mesh is well-formed, **every triangle will be tiny with respect to the voxel face area**; that implies that:

- the *surrounded* relative position at figure 3.5 between T and V is impossible;
- each triangle T will be included only in 1-level connectivity adjacent voxels;
- if all the triangles are adjacent to any other triangle, there won't be valid voxels with no mesh vertex;

Therefore, the triangle-voxel intersection in that case is very simple: **the algorithm only has to test if there are some triangle vertices within the voxel**.

[6] From now, faces will be triangles for a more simple point of view of the problem.



(a) 1 *inner vertex* (b) *All inner vertices*

(c) *No inner vertices* (d) *Method unstability example*

Figure 3.6: *Testing $T \cap V$ by triangle-box test based on vertex containing checking; output by a Matlab code written for this thesis by its author.*

This method is the most powerful and fast, because the checking is direct or quasi-direct, depending on the **voxel bounding box typology**, as there is in the figure 3.7:

- **Axis-Aligned Bounding Box** AABB: all the voxel faces are parallel to one of the canonical axis, so with the minimum and maximum bounding box coordinates; so **the i-th vertex of a triangle t is within the cube if its (x, y, z) coordinates are all in the box range**:

$$T = t_1 t_2 t_3 \rightarrow \ t_i \subset V \ \equiv \left\{ \begin{array}{l} t_{i_x} \in [V_{min_x}, V_{max_x}] \\ t_{i_y} \in [V_{min_y}, V_{max_y}] \\ t_{i_z} \in [V_{min_z}, V_{max_z}] \end{array} \right. \implies \exists i \in \{1,2,3\} \text{ t.q. } t_i \subset V \implies T \cap V$$

- **Object-Oriented Bounding Box** OOBB: knowing that, given the plane π equation and a point P, the plane-point relative position is able for knowing the plane normal with this calculus:

$$\pi : ax + by + cz + d = 0, \ P = (p_x, p_y, p_z) \implies \left\{ \begin{array}{lcc} \pi(P) > 0 & \rightarrow & \textit{upper} \\ \pi(P) > 0 & \rightarrow & \textit{contained} \\ \pi(P) > 0 & \rightarrow & \textit{lower} \end{array} \right. ;$$

  so, **the testing is done by checking if, for all the six cube faces, any vertex of a triangle is intern**.

However, **if any of the three previous conditions is not fulfilled** (*e.g.*, the longest triangle edge is greater than the middle of voxel edge), **this method is too unstable**.

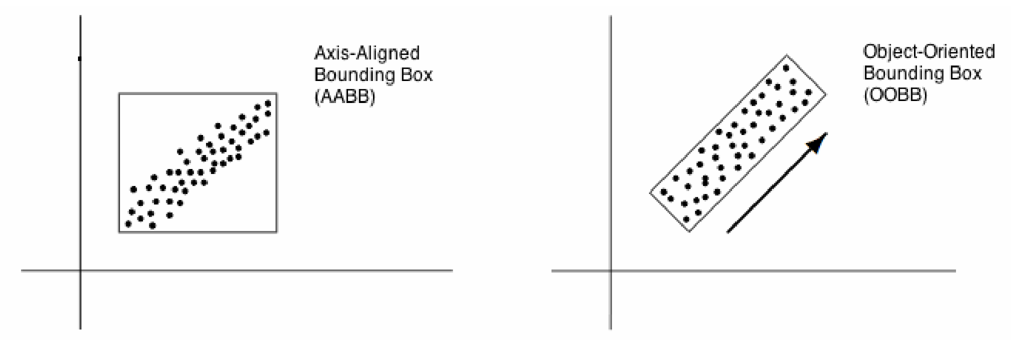


Figure 3.7: *Tipology of Bounding Boxes depending on their orientation; source [voxB] (edited).*

Here's the **pseudocode for the triangle-box —AABB assuming— test by vertex checking**; at the figure 3.6 on previous page there are some examples from the same code written in Matlab, including an unstable test where an inner triangle is considered extern.

```
1  function triboxAABB_vertexcheck( Cube C, Triangle T )
2  returns bool
3      Vertex cubemin = getCubeMinorVertex( C );
4      Vertex cubemax = getCubeMajorVertex( C );
5      int innerverts = 0;
6      for k= 1 to 3 do
7          Vertex vert = T.getVertex(k);
8          if between( vert.x, [cubemin.x, cubemax.x] and ...
9             between( vert.y, [cubemin.y, cubemax.y] and ...
10            between( vert.z, [cubemin.z, cubemax.z] then
11             innerverts = innerverts + 1;
12         endif
13     endfor
14     if( innerverts > 0 ) return true;
15     else return false;
16     endif
17 endfunction
```

PSEUDO **code 3.1:** *Triangle-box test by vertex checking.*

3.2.2.3 Test's generic case: voxelization of any mesh at any LOD. Alternative techniques

Due to the huge amount of mesh pecualiarities, it cannot be supposed that vertex-voxel testing will be always valid, so a generic case is needed. Let's present some of the different alternatives.

▷ ***Greedy Algorithm***

The first approach, the **greedy algorithm**, is computationally **prohibitive**:

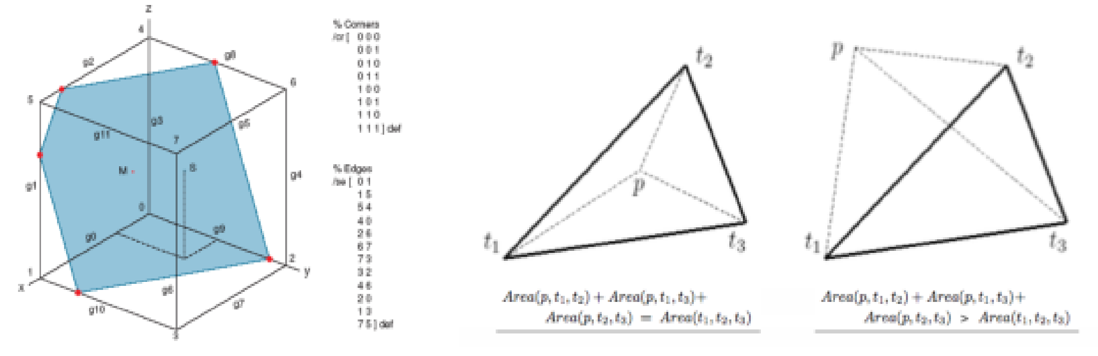
1. to find the Q area mentioned in the section 3.2.2.1, on the page 22, then, the intersection zone between $(\pi \supset T) \cap V$;
2. if Q is not null, check if the triangle, or a part of it, is in Q: $T \subset Q$;

because the first step implies:

- find the six intersections between $(\pi \supset T) \cap f_i,\ i = 1..6$, being f_i the i-th face of V;
- store the polygon generated by the six lines;

▷ *Intersection between triangle and voxel edges*

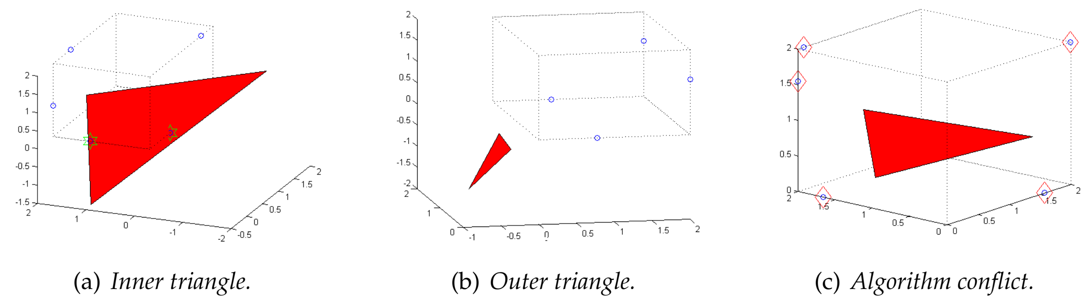
Gernot Hoffmann, in [vox4], presents an alternative to greedy algorithm, to give more efficiency on the same calculus: his algorithm **doesn't find the intersection between $\pi \supset T$ and the six voxel faces but with its twelve edges**, as can be seen in the next figure. Finally, it's easy to check a possible intersection between the two areas, the P intersection polygon one, and the T one.



(a) *Plane-cube intersection test at [vox4].* (b) *Triangle-Point container area method (intern, extern)*

Figure 3.8: *The Gernot Hoffmann Triangle-Voxel intersection test.*

However, this method is **still unstable** due to those cases where a voxel's inner triangle doesn't intersect any cube's edge, as it can be seen in these Matlab plots obtained by this project's student own code implementation of this method.



(a) *Inner triangle.* (b) *Outer triangle.* (c) *Algorithm conflict.*

Figure 3.9: *The Gernot Hoffmann Triangle-Voxel intersection test results adquired by this document's author by his own Matlab programmed implementation of this proposal.*

▷ *Sierpiński's Fractal for Triangle Area Discretization*

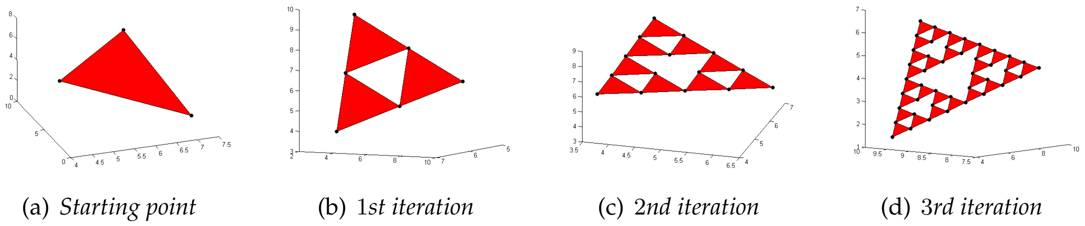
This **fractal**[7] was introduced in 1915 by the polish mathematic Wacław Sierpiński, and consists, as said in [vox5] and [vox6], in this **iterative triangle subdivision**, that starts with an equilater triangle:

1. draw three **lines that connect the midpoints of the triangle's sides**. The result is a centered white triangle;

2. the original triangle is now divided into **three smaller triangles**, and for each one, repeat again;

[7] A Fractal is a semi-geometric object with a peculiar structure: it's repeated whatever the scale you use to represent it.

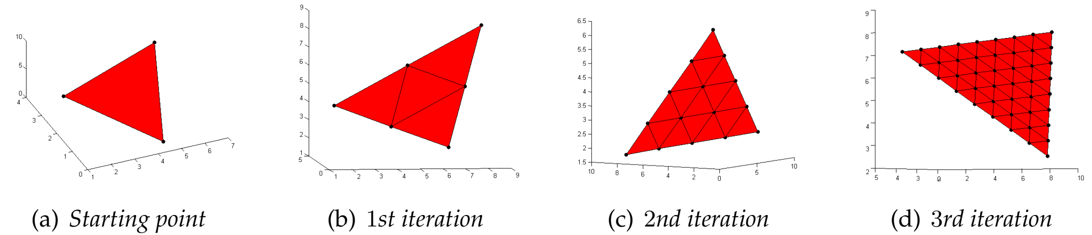
Here you can see the starting point and the first three iterations, made from an arbitrary —non equilater— triangle with a `Matlab` code written by this thesis author for research testing purposes:



(a) *Starting point* (b) 1*st iteration* (c) 2*nd iteration* (d) 3*rd iteration*

Figure 3.10: *The first three iterations of the Sierpiński Fractal.*

This fractal is able to, indeed, be used for **discretizing the triangle area**, by **modifying the fractal method with not rejecting the centre triangle but also counting with it for post subdivisions**, hence, the final result is a coherent and **equitative discretization of the triangle area**:



(a) *Starting point* (b) 1*st iteration* (c) 2*nd iteration* (d) 3*rd iteration*

Figure 3.11: *The Sierpiński Fractal* `Matlab` *plottings of the thesis' author own adaptation code for triangle area discretizations.*

Since the main problem for the generic voxelization case is the abstract level of the triangle area, this fractal can be used for implementing the fast algorithm decribed in 3.2.2.2, on page 22; that is, **execute the vertex-voxel test for every vertex as result of each triangle area discretization from a mesh**, with a code like the following one:

```
1  function triboxAABB_sierpinski( Cube C, Triangle T, ...
2                                  int level )
3  returns bool
4      { 'level' indicated the algorithm recursive steps }
5      Set<Triangle> AST = getAdaptedSierpinski( T, level );
6      for k = 1 to AST.length() do
7          if( triboxAABB_vertexcheck( C, AST.elem(k) ) then
8              return true;
9          endif
10     endfor
11     return false;
12 endfunction
```

PSEUDO **code 3.2:** *Triangle-box test based on Sierpinski fractal adaptation.*

However, there are **two great disadvantges** for this methodology, **both focused on the number of desired fractal iteration levels**:

- what is the most **adequate iteration limit level per each triangle**, due to, although it's a good discretization of the total triangle area, it's not sure that if the triangle intersects a voxel, there will be some discretization vertices within the cube;
- as long as the iteration level increases, the number of triangles also increases; consequently, the **number of vertices per original triangle grows very much**, as it's shown on this table:

# iteration	0	1	2	3	...	n
# triangles	1	4	16	64	...	4^n
# vertices	3	6	15	46	...	$\sum_{i=0}^{\alpha_n} i, \begin{cases} \alpha_0=2 \\ \alpha_i=\alpha_{(i-1)}+(\alpha_{(i-1)}-1) \end{cases}$

Table 3.1: *The progressive data increasing with Sierpiński Fractal adaptation*

▷ ***Separating Axis Theorem Derived Approach***

Tomas Akenine-Möller featured in 2001, on [vox7], a fast testing for 3D triangle-cube overlap checking. This method is based on the **separating axis theorem**, saying that **two meshes, M_A and M_B, are not intersecting if and only if there is a plane π_s separating them, one in each "side" of the plane**:

- let π_s be a considered separator plane of two meshes M_A and M_B;
- let n_{π_s} the π_s normal rect; if the two mesh projections over n_{π_s} don't intersect, this line is called **separator axis**; *ergo*, π_s is a **separator plane**.

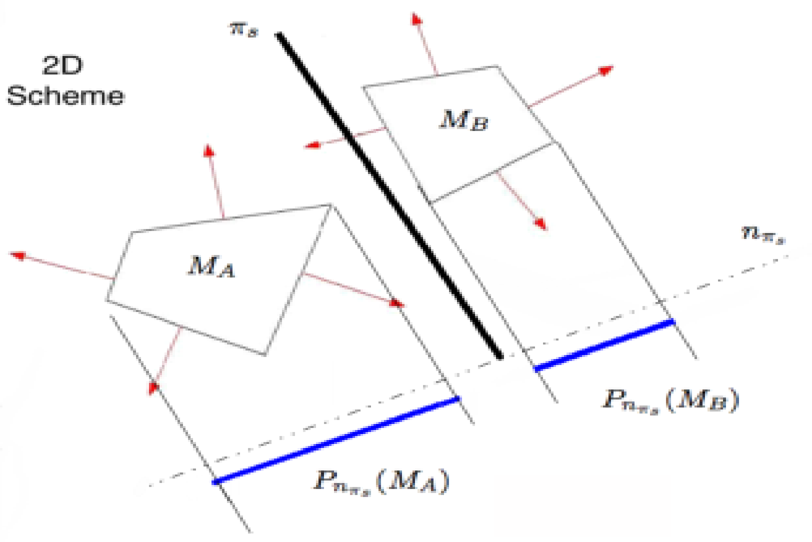


Figure 3.12: *Scheme of a separator axis; notice that the normal projections of two meshes M_A and M_B are not intersecting; so, the two meshes are not intersecting, and thus, π_s is a separator plane.*

Besides, Akenine-Möller proofs that, for a **3D convex meshes**[8] there is a more focuses theorem; **two convex meshes C_A and C_B are disjoint if and only if one of these three axis are tipified as separator axis:**

[8] The most common 3D meshes that model a really physical object.

- perpendicular to any C_A face, or to any C_B face;
- parallel to the cross product between any C_A edge and any C_B edge;

The concrete algorithm checks the disjoint of a triangle and a voxel in 13 axis:

1. **4 axis**: the three cube face normals and the triangle one;
2. **9 axis**: the cross product between a cube edge and a triangle edge;

3.3 The *search tree* structures applied to voxelization

3.3.1 Reasons for using trees

When the voxelization process is needed, it's obvious to think in a voxel-discretized space; thus, **the straightforward solution is to build a regular grid of voxels**, for executing the 3D rasterization that is the base of the mesh voxelization. It's simple and effective, but **terribly inefficient, since to it's not an adaptive method**. The general solution approach is, then, the adaptive nature provided by the **search tree structures**. Let's see the most famous ones in versatility and widely usage, because of the **disjoint space-subdivision characteristics**.

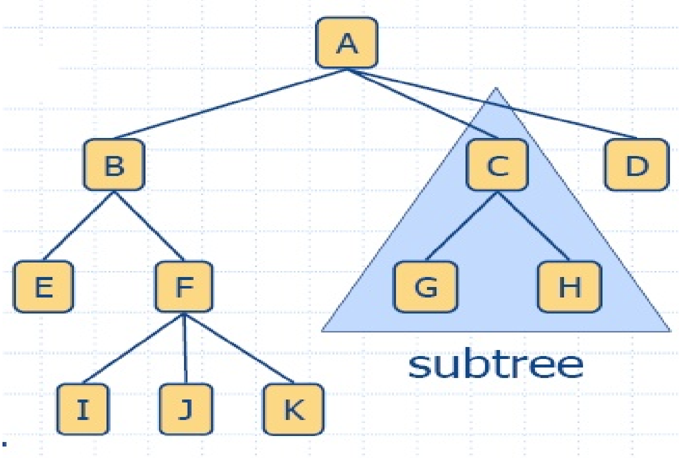


Figure 3.13: *A generic tree structure for being used in computer algorithm data structuration and search accelerations; in this example,* **A** *is the root of the tree, being* **E**, **I**, **J**, **K**, **G**, **H**, **D** *nodes without children, called leaves. Moreover, in the remarked subtree context,* **C** *is the root and* **G**, **H** *the leaves.*

3.3.2 The most common Search Trees Structures

The main advantage of this kind of structures is its adaptability: **the deeper is the voxel-associated tree node, the minor size will have**. That means, **if a huge voxel is considered all valid or invalid, it's a valid decision for every real grid voxel contained in this huge voxel**. More technically, the tree construction algorithm for each kind of tree will be a **recursively scene space subdivision into 2 more subspaces, until a satisfactory condition is achieved**.

Besides, this tree structures can offer the same concepts in 3D (voxels) as well as in 2D (pixels), so for better explanations, some structures will be explained in 2D.

3.3.2.1 Binary Space Partition Tree —BSP-Tree—

It's the **most generic case** of all the tree structures for voxelization purposes, and each recursive subdivision step consists in **specifying the better splitting plane —a line in the 2D case— that cuts the node scene space into 2 subspaces that simplifies the coherence of the scene**; therefore, for voxelization purposes, this plane should be the most valid-invalid node classificatory: **the aim will be to obtain the greatest possible nodes with no-mesh, or mesh-containing; and, in the second case, the *submeshes* should be convex**.

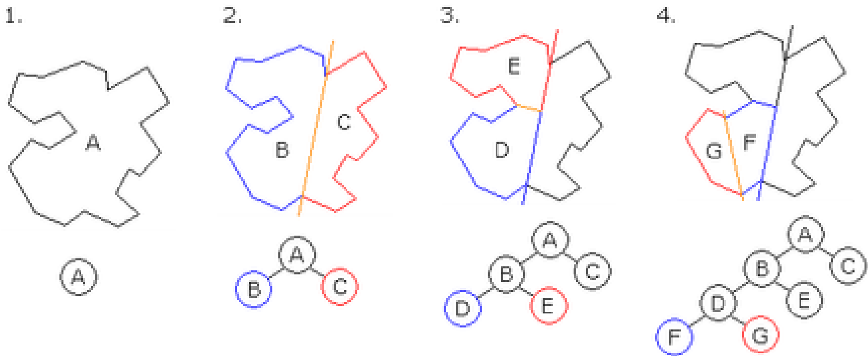


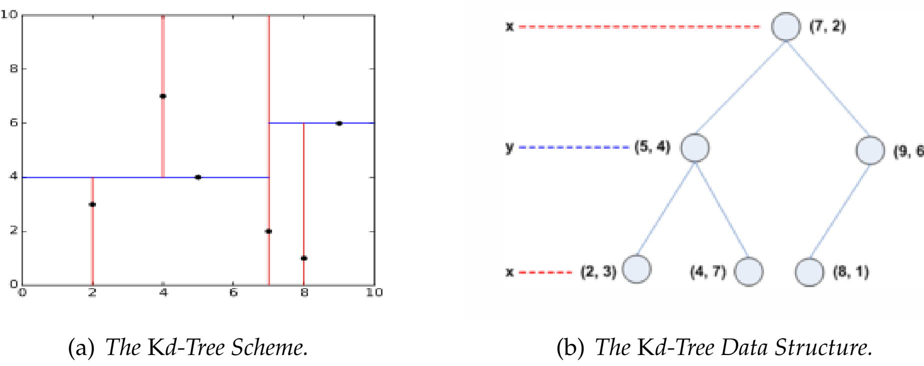
Figure 3.14: 2D BSP *construction based on a polygon, extracted from [vox8]. Notice that the satisfactory condition for stopping the* BSP *construction is that the two subspaces contain a convex polygon.*

3.3.2.2 Kd-Tree

A Kd-Tree is a *subclass* of the BSP-Tree. In the BSP case, the splitting plane could be any plane $\in \mathbb{R}^3$, while in this case, **the space cutting planes will always be axis-aligned** to the canonical planes $< \mathcal{XY} >$, $< \mathcal{YZ} >$ or $< \mathcal{XZ} >$.

The next figure shows the 2D Kd-Tree built structure from a point cloud; the plane-choosing policy is the following:

1. in each recursive splitting step, the plane will be **parallel to every canonical axis in a cycle**: $< \mathcal{XY} >, < \mathcal{YZ} >, < \mathcal{XZ} >, < \mathcal{XY} >, \ldots$
2. the characteristic value of the plane will be decided by the **median of the points being put into the Kd-Tree**, with respect to their coordinates in the axis being used.



(a) *The Kd-Tree Scheme.* (b) *The Kd-Tree Data Structure.*

Figure 3.15: *A point cloud based* 2D *Kd-Tree structure construction; source [vox9].*

There is a Kd-Tree subclass called **Bin-Tree**, where the second policy step is modified by the **middle space node**.

3.3.2.3 Quadtree (2D) and Octree (3D)

With no doubt, **the easiest and most widely used of all the search tree models**. The Octree structure creation is based on a **recursive space equitative subdivision in eight equal subboxes for each box** —or subdivision in quads in the textsc2d case—, with no matter about the space data or properties; that is, each node will have eight sons —four in the case of the Quadtree—.

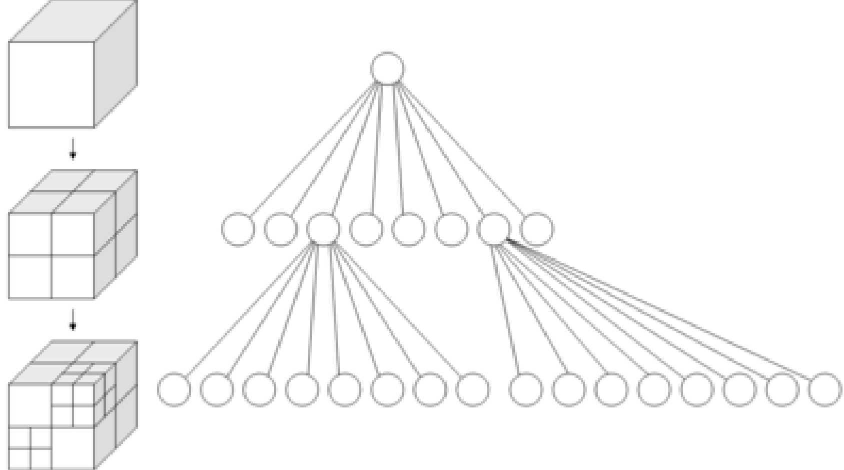


Figure 3.16: *The* 3D *Octree recursive subdivision; source [voxA].*

CHAPTER 4

Polygonal Mesh Reconstruction: Multiresolution Meshes

4.1 Introduction to Multiresolution Meshes

4.1.1 The need of a multiresolution mesh capability

In [mrs2], the authors focuss their work in progressive meshes, a thesis out-of-range research field; in this document it's noticed the importance of the multiresolution meshes described in section 2.2.2 —page 14—. As said there, **multiresolution meshes are the basis for generating geometric objects at different levels of detail —LODs—**.

The use of the *multiresolution* term is due to the **direct relation of the resulting approximate polygonal mesh with the space cell density** (*e.g.*, the grid size of a voxelmap).

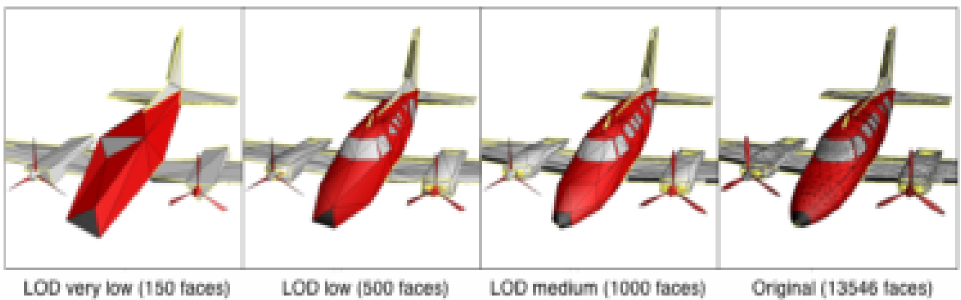


Figure 4.1: *Multiresolution mesh of an airplane, at different* LODs*; source [mrs1].*

Also, Hughes Hoppe, famous for his multiresolution contributions with the *progressive meshes* ([mrs2]), remarks the importance of this kind of meshes due to:

- less triangles for the rendering process means **more efficiency** but not necessarily loss of mesh quality at human's eye (for gigantic meshes overcoat);
- **easier interaction with the mesh**, for data transmission or morphing transitions (like deformations);

4.1.2 The IsoSurfaces

An **isosurface**, as said in [mrs3], is a **3D surface whose points are constant scalar values within a volume of space**; it usually takes the form of a $f(x, y, z) = 0 \in \mathbb{R}^3$ function for easier computation, and it's the most popular form of volume dataset visualization **that can be rendered as a polygonal mesh**, which can be drawn on the screen efficiently, using reconstructive mesh algorithms computing as valid or invalid the isosurface values with respect to an arbitrary **threshold value**.

Once all the isosurface scalar values are extracted, a mesh is extrapolated from the **frontier voxels**, voxels connected to inner and outer voxels at the same time —that is, **voxels containing partially the original objects—**.

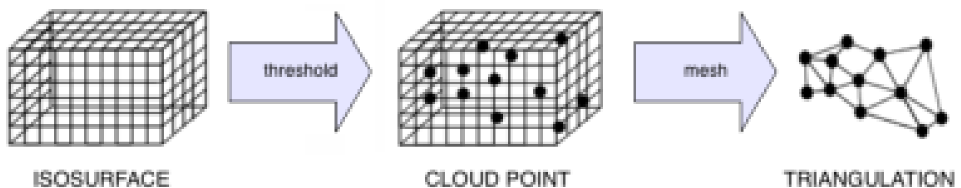
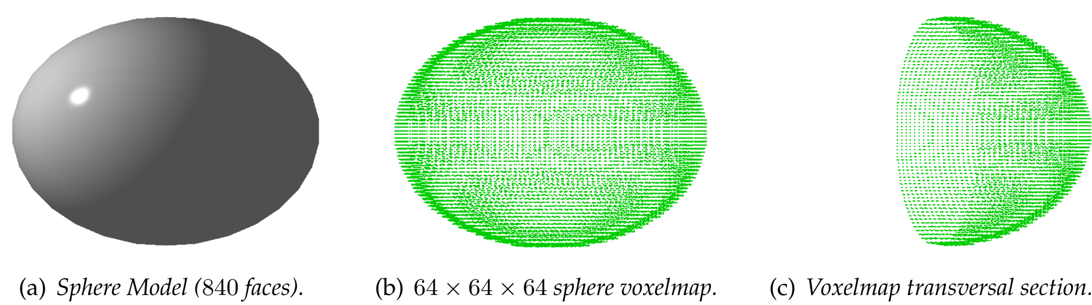


Figure 4.2: *Scheme of IsoSurface renderization as a triangular mesh.*

4.2 VoxelMap Refilling. The Model Carving

In section 3.1.1, on the page 19, it is said that a polygonal mesh is representing only the surface of an object. As a consequence, if the mesh is convex, the synthesized object will be empty, and then, executing this mesh's voxelization will result in an **empty voxelization —that is, with inner holes due to the inexistence of inner parts of the models—**. In other words, **in the resulting voxelmap from a convex mesh voxelization, there won't be inner voxels**: only outer voxels in the two disjoint subspaces split by the frontier voxels generated by voxelization.



(a) *Sphere Model (840 faces).* (b) $64 \times 64 \times 64$ *sphere voxelmap.* (c) *Voxelmap transversal section.*

Figure 4.3: *Voxelization of the* `sphere` *model. Notice the emptiness in the voxelmap section, so the volumetric model is not filled but empty, like the original convex mesh. Output frameset from* MeshInspeQTor.

To solve this problems there are a singular approach named **carving**, with the original focus on 3D voxelmap reconstruction of a 3D real scene using multiple-angle camera captures —**space carving**—, and an adapted version to a voxelmap directly —**voxel carving**—. This methods are **able to refill potential voxelmap holes**.

4.2.1 Space Carving

In [mrs4], Kiriakos Kutulakos and Steven Seitz feature a new approach relative to **computing the 3D shape of an unknown, arbitrarily-shaped scene from multiple photographs taken at known but arbitrarily-distributed viewpoints**.

They include the concept of **photo hull** by analyzing the relation between all the input photographs; the **photo hull**, computed directly from the scene photographs, subsumes all the 2D shapes from all the photographs for, with a probably-correct method, the **space carving**, extrapolating all the 3D shapes of the scene.

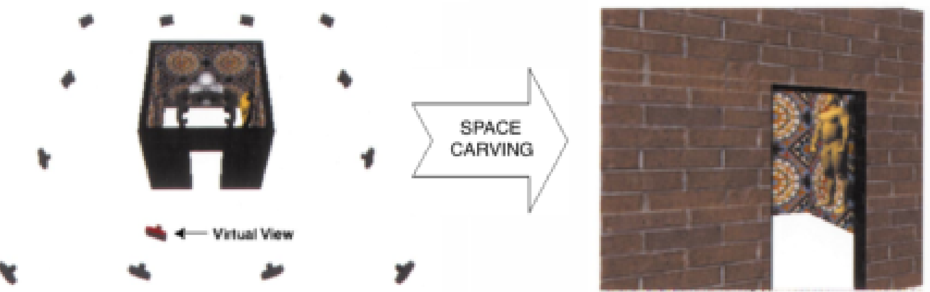


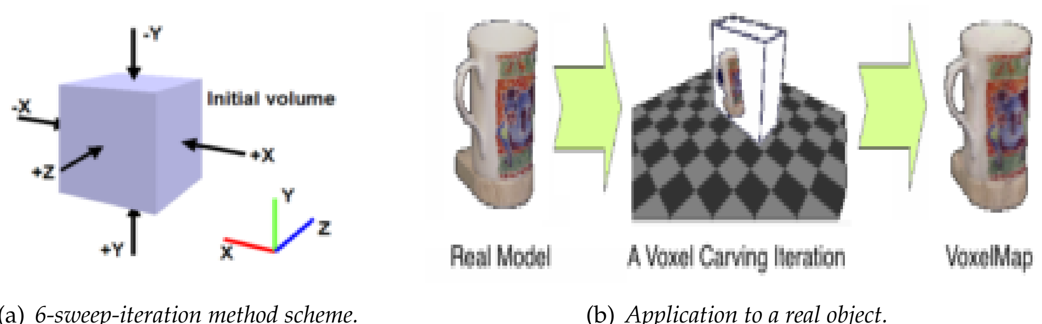
Figure 4.4: *The Space Carving algorithm in action; source [mrs4] (edited).*

Widely, the photo hull is actually a volumetric model, and **space carving modifies —*carves*— an empty voxel space to adapt it to the photo hull**, which the authors call *"satisfy the photo-consistency"*.

4.2.2 Voxel Carving

Some time later, Sainz, Bagherzadeh and Susin got inspired by Kutulakos work and presented in [mrs5] an **improved approach of space carving, called voxel carving, based on carving a bounding volume using an inner data similarity criterion**.

This technique takes the advantages of **performing six disjoint carvings by sweeping along the positive and negative direction of each axis of the object bounding box**. Notice that in each step, only the visible cameras relative to the sweeping plane are enabled. Finally, the resulting voxelmap is the **intersection of the six intermediate carved voxelmaps**; the result is obviously **satisfactory for convex models**.



(a) *6-sweep-iteration method scheme.* (b) *Application to a real object.*

Figure 4.5: *Detailed methodology for the Voxel Carving algorithm; source [mrs5] (edited).*

In the next page there's a **complete visual guide of the six voxel carving steps** applied to the voxelization of a `torus` mesh. Let's notice the clear difference of the carved torus voxelmap (down) with respect to the uncarved one (up), where only a standard mesh voxelization has been executed.

4.3 IsoSurface Extraction Algorithms

4.3.1 IsoSurface extraction as multiresolution mesh construction

Far away from **volume rendering** techniques, that are **out of the range** of this thesis, the aim of this section is to **introduce a few techniques**, the most famous ones, for **generating a fast extrapolated triangular mesh from an isosurface**.

It's a computational field **commonly used for medical purposes** —as really aimed this thesis—, since isosurfaces are widely created from a set of three-dimensional CT scan, allowing the visualization of internal organs, bones, or other structures. However, this thesis only concerns preloaded polygonal meshes; thus, the 3D isosurface generation from this medical images is, again, out of the range of this project.

Nevertheless, the main idea of these algorithms, due to the efficient mesh generation provided by two techniques, called **Marching Cubes** and **Marching Tetrahedra**[9], is **to build a multiresolution clone of any mesh, calling one of these methods by passing them the isosurface for the mesh voxelmap**.

4.3.2 Marching Cubes

4.3.2.1 Overview

Originally proposed by Lorensen and Cline in their 1984 paper [mrs6], it consists basically —and without entering into implementation details— in:

1. with the support of a look up table —LUT— of all the possible unit voxel tipologies with their triangulations (precalculated in relation to a real isosurface offset), it **generates the most coherent connectivity inter slices**[10];

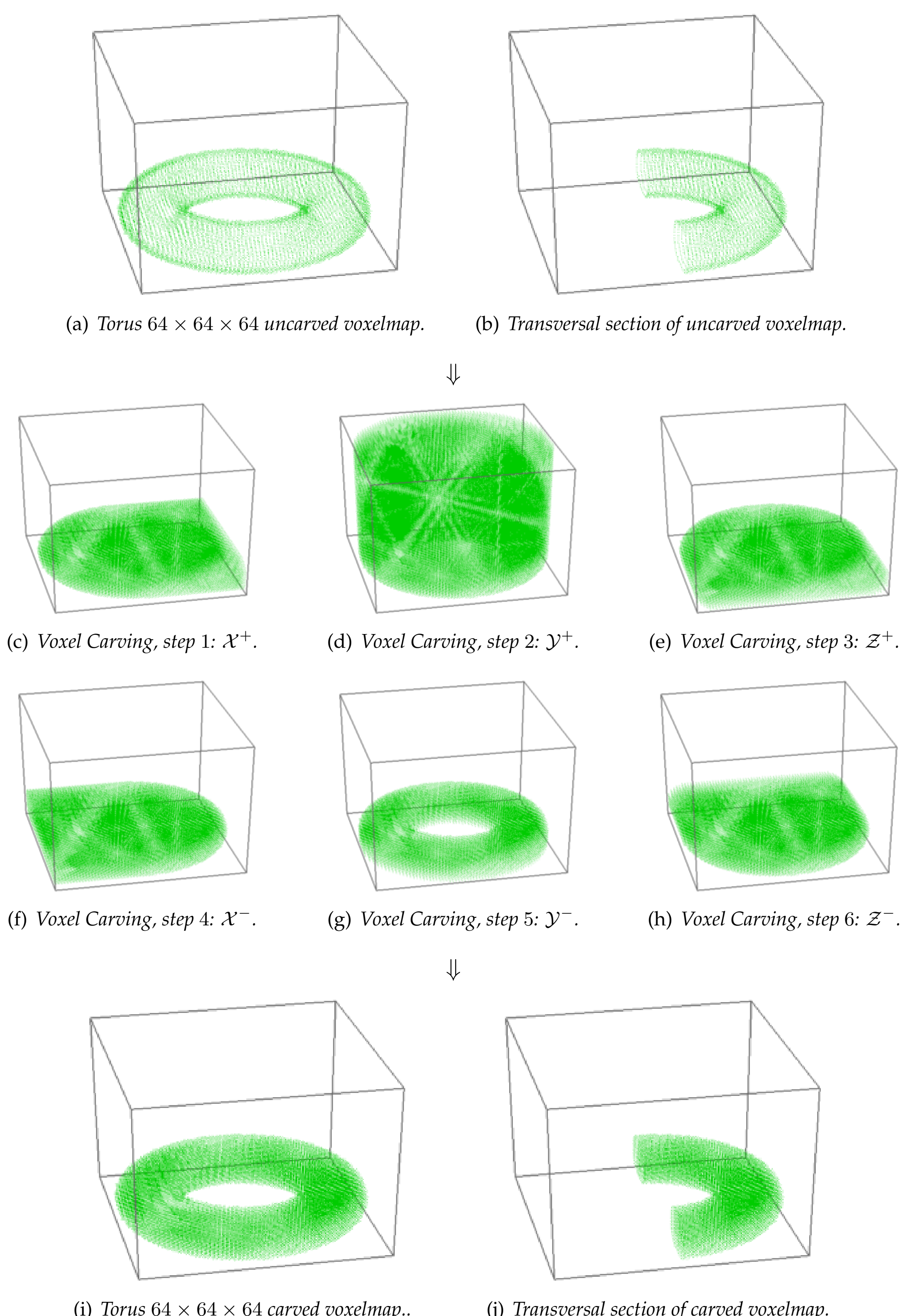
2. after that, it processes de 3D data and, **using linear interpolation, calculates the current triangle vertices**;

3. for a convenient mesh shading, the **normals are calculated from the original isosurface data gradient**;

With this method, **all the voxel locations where the isosurface passes through are determined, so it's possible to generate triangles joining this isosurface-voxel intersection points**. The final surface is generated when all the triangles are generated and coherently joined.

[9] Although the second is really a variation of the first, considered a standard approach to the isosurface extraction problem from a volumetric dataset.

[10] This LUT, that will deserve its own section later at the page 36, , is only for frontier voxels, that is, voxels partially included in the original isosurface. In other words, a voxel is tipified as frontier if there are some voxel vertices considered inner the isosurface, and the rest of vertices are considered outer.



(a) *Torus* $64 \times 64 \times 64$ *uncarved voxelmap.* (b) *Transversal section of uncarved voxelmap.*

$\Downarrow$

(c) *Voxel Carving, step* 1: $\mathcal{X}^+$. (d) *Voxel Carving, step* 2: $\mathcal{Y}^+$. (e) *Voxel Carving, step* 3: $\mathcal{Z}^+$.

(f) *Voxel Carving, step* 4: $\mathcal{X}^-$. (g) *Voxel Carving, step* 5: $\mathcal{Y}^-$. (h) *Voxel Carving, step* 6: $\mathcal{Z}^-$.

$\Downarrow$

(i) *Torus* $64 \times 64 \times 64$ *carved voxelmap..* (j) *Transversal section of carved voxelmap.*

Figure 4.6: *Voxel Carving applied to a standard* $64 \times 64 \times 64$ *voxelization of the* torus *model. Since the final voxelmap is the intersection of the six voxel carving steps, the volumetric model of* `torus` *is filled. As usual, frameset created by the thesis purpose software* MeshInspeQTor.

4.3.2.2 Advantages and disadvantages

Advantages		Disadvantages	
A1	High resolution, voxelmap size depending.	**D1**	Possible holes in the generated mesh from the isosurface.
A2	Easy and fast rendering.	**D2**	Final quality of the mesh is strongly isosurface-depending; and the isosurface is threshold-segmented so the original data must be excellent to avoid problems.
A3	Good quality meshes with low cost (CPU time).		

Table 4.1: *Advantages and disadvantages of the Marching Cubes algorithm; extracted details from [mrs7].*

4.3.2.3 The isosurface-voxel intersection case LUT. Shortcomings

When the algorithm is analyzing the **cube-isosurface intersections** for a voxel, there are 256 **possible intersections**; however, they are reduced to 128 by complement situations[11], and by rotations, this number is reduced again, obtaining **15 absolute possible intersections**, shown in the following figure:

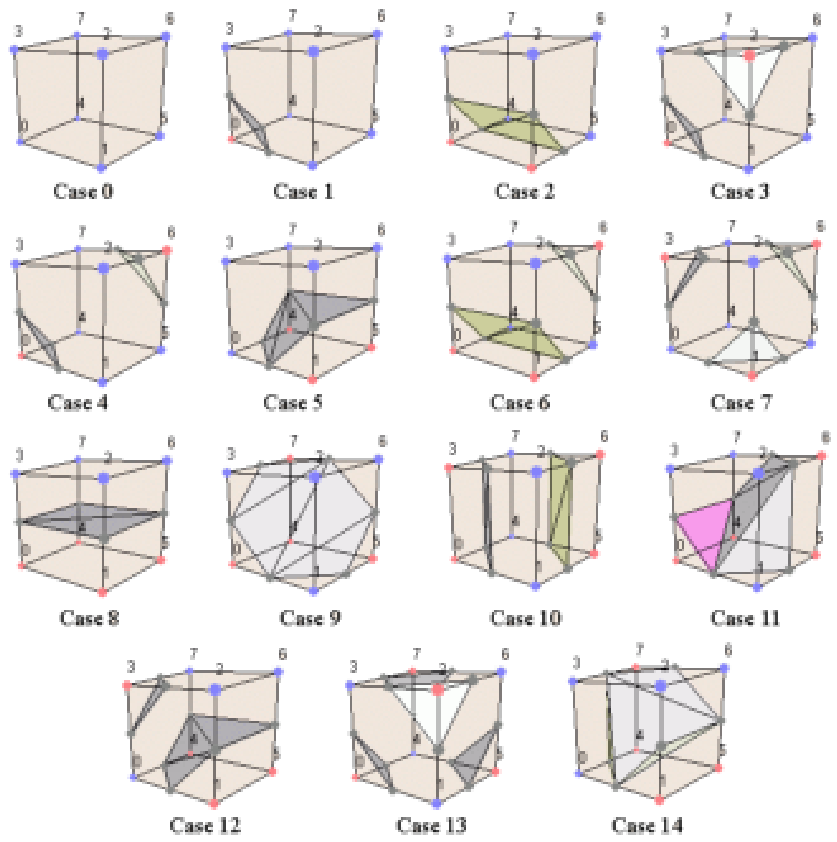


Figure 4.7: *The 15 absolute cube-isosurface intersection situations; source [mrs8].*

[11] Inverting the marked intersection points within the voxel, and the triangle normals must be inverted too for coherence.

However, there are some **ambiguous cases where, in case of choosing a bad cube-isosurface intersection type, a mesh hole can be produced**, as the remarked table 4.1(**D1**) disadvantage tells; there are some solution approaches, like those who can be found on [mrs9] and [mrsA]. A clear example of hole can be seen in the following figure:

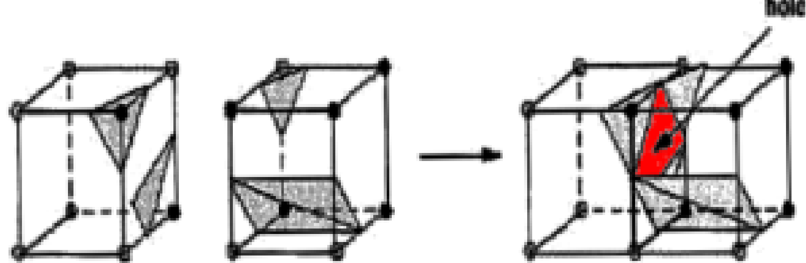


Figure 4.8: *An example of hole in a marching cubes generated mesh, as the result of an ambiguous case; source [mrs8] (edited).*

Nevertheless, it's a good algorithm and widely used for automatic generation of triangular meshes from an isosurface —*e.g.*, taken from tomographies and medical scanners properly—; so its pseudocode should be something like that:

```
1  function marchingcubes( VoxelGrid VG, Isosurface S )
2  returns TriangularMesh
3      Set<Point> SP;
4      foreach Voxel V in VG do
5          { marks the voxel as inside or outside in
6            relation to the isosurface volume }
7          VG.setStatus( V, S );
8          foreach Edge E in V do
9              { an edge is an isosurface boundary
10               if it has an inner vertex being
11               outer the other, from isosurface }
12             if isBoundary( E, S ) then
13                 SP.add( middlepoint(E) );
14             endif
15         endforeach
16     endforeach
17     TriangularMesh M;
18     M = generateMeshByPointSetTriangulation( SP );
19 endfunction
```

PSEUDO **code 4.1:** *The Marching Cubes isosurface extraction algorithm.*

4.3.3 Marching Tetrahedron

4.3.3.1 Overview and shortcomings

As explained previously in the introduction of this section, there is a **Marching Cubes method variant, called Marching Tetrahedra**. Its approach is based on the same Marching Cubes philosophy but here, **each voxel will be subdivided in five tetrahedrons**.

This technique can produce a **smoother generated mesh than the Marching Cubes resulting one**, but it's also true that due to this tetrahedron subdivision, an increasing number of triangles is produced, and **it's possible to have too much polygons where it isn't needed**.

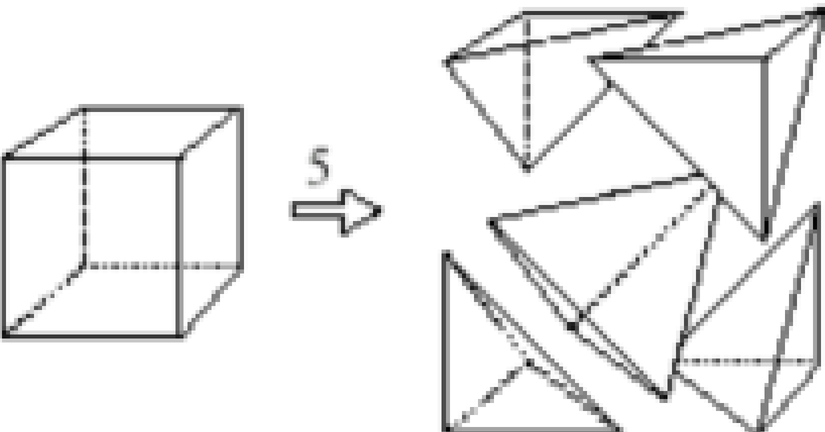


Figure 4.9: *The Marching Tetrahedra cell subdivision scheme. Each voxel will be treated as five independent tetrahedrons, so the method is more accurate and smooth with the discrete shape of volumetric unit than marching Cubes; source [mrs8].*

As it can be deduced, **the algorithm steps are the same that the Marching Cubes method**, but taking into acount that **the planar facet approximation to the isosurface is calculated for each tetrahedron independently**. The facet vertices are determined by linearly interpolating the place where isosurface cuts the edges of the tetrahedron, as in Marching Cubes.

4.3.3.2 The Marching Tetrahedra LUT

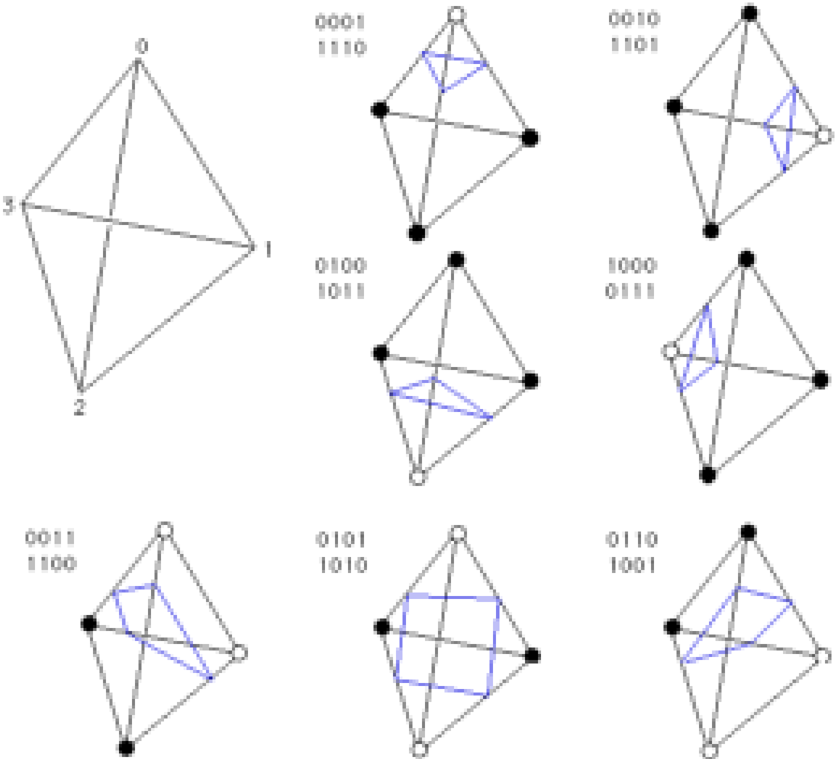


Figure 4.10: *The Marching Tetrahedra cell subdivision scheme; source [mrs8].*

The figure on the previous page illustrates the tetrahedron-isosurface possible intersections, after complementation and rotation reducements[12]. The hollow and filled circles at the vertices of the tetrahedron indicate that the vertices are on different sides of the isosurface.

The most important advantage with respect to its original method, Marching Cubes, is that **this triangulation intersection LUT has no ambiguous situations**. But, as it has been said, the number of generated triangles will be much greater than the Marching Cubes generated mesh would have.

[12] There is a configuration left. The case not illustrated is where all the vertices are either above of below the isosurface; then no facets are generated.

CHAPTER 5

3D Object Model Animation/Deformation

5.1 Point-Oriented Deformation

5.1.1 The Point-Sampled Geometry

There is a recent branch, appeared since 2002 due to Zwicker et al's **Pointshop3D** [geo1] proposal; it's a system for interactive and appearance editing of 3D **point-sampled geometry**; that is, a **geometry with no vertex connectivity information**.

Based on a novel interactive **point cloud parameterization** and an **adaptive remeshing method**, this system permits to **transfer the 2D image editing capabilities to an irregular 3D point set**. Unlike 2D pixels, the absence of connectivity gathered with the irregularity of the point-sampled patterns are the reasons for this system, like a **3D photo editing software**. The idea behind that is purely based on irregular **3D points as the image primitives**. Briefly, the process is, as seen in the autoexplanatory example figure above, **to spread 2D image pixels over 3D surface pixels** —*surfels*—.

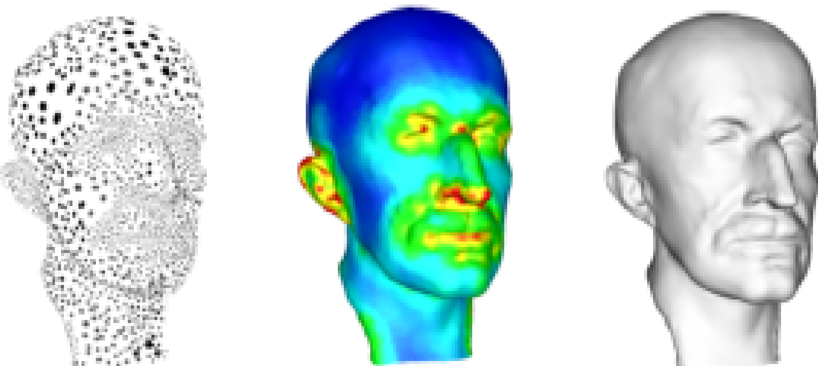


Figure 5.1: *Hybrid system featured in [geo2], consisting in a hybrid geometry representation combining point clouds with implcit surface definitions. Note: the middle image is the colored sample density map.*

One year later, in 2003, Pauly et al proposed in [geo2] a **free-form shape modeling framework for point-sampled geometry** that takes benefit of the minimum consistency requirements of point clouds for developing an *on-the-fly* surface restructuration, **supporting geometric deformations** in interactive time. In the figure above there's an example of a reconstructed `planck` 3D model mesh from a point cloud.

5.1.2 Point-sampled deformation

In [geo3] there's an approach for **collision detecting and dynamical deformation response**, based on **continuum mechanics**, everything done on 3D objects where the **volume as well as the representing surface are point-based sets**. Developed by Müller et al from their previous work [geo4], it offers a complete animation framework schematized in the following figure.

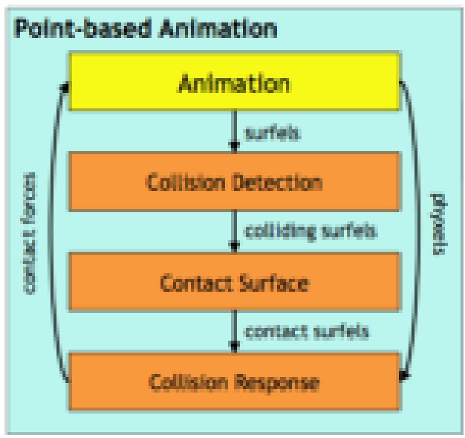


Figure 5.2: *Scheme of the contact handling framework developed in [geo3].*

In more technical terms, in [geo3] a point-based object, called Γ_k, is defined as a **cloud of n *surfels*** s_k —surface elements—, **animated over a set of m *phyxels*** p_k —points, physical elements— with position $x_k \in \mathbb{R}^3$. So, an **overview** of this point-based animation system, in relation to this nomenclature, should be the following:

- **dynamic simulation**; the scheme for all m sample points —and for each time step Δt— dynamic simulation loop is something like this, extracted from [geo4] quasi-directly:

$$\underset{(displacements)}{\mathbf{u}_t} \Longrightarrow \underbrace{\left[\underset{(derivatives)}{\nabla \mathbf{u}_t} \rightarrow \underset{(strains)}{\varepsilon_t} \rightarrow \underset{(stresses)}{\sigma_t} \rightarrow \underset{(forces)}{\mathbf{f}_t} \right]}_{\text{spatial derivatives by Moving Least Squares —MLS— procedure}} \Longrightarrow \underset{(integration)}{\mathbf{u}_{t+\Delta t}}$$

  where the MLS procedure is the **most common position interpolation in mesh free algorithms**[13], consisting in, according to [geo5], the calculation of a **biased weighted least squares** measure towards the region around the point at which the reconstructed value is requested.

[13] A mesh-free is a mesh composed by a set of unorganized point samples.

- **collision detection**; here the goal is to find new ***surfels* inside of any of the other** n ***surfel*-bounding objects for a collision between** Γ_1 **and** Γ_2; that is, to find a contact surface $\Phi_{1,2}$ lying in the intersection volume $V_{1,2} = \Gamma_1 \cap \Gamma_2$. The *surfels* of $\Phi_{1,2}$ are called *contact surfels*, and for each of them, there's a **penalty and friction force** to be computed, like in the following figure[14]:

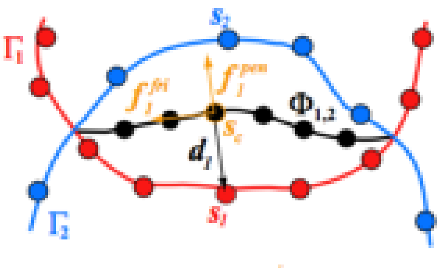


Figure 5.3: *Penalty and friction force processing for* Γ_1 *in front of a* Γ_2 *collision; source [geo3].*

This research field is **totally outside this thesis**, so the details of the algorithms won't be explained and studied deeply. However, it's an entire and innovative method of rendering, deformation and animation, so it has deserved a **special mention about its basis and capabilities**. Thus, in the following figure there's a complete example of Max Planck bust where **elastic, plastic, melting and solidifying** animation models are shown: .

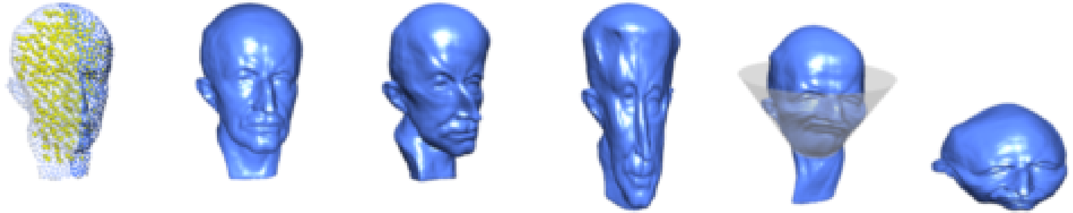


Figure 5.4: *Example of point-sampled object animation with the [geo4] method; the two left figures are representing the* phyxel-surfel *representation and the* MLS *reconstructed mesh.*

5.2 Geometric Deformation

5.2.1 Global or local deformation operators

5.2.1.1 Reminder: the jacobian matrix J_F of a function F

First of all, a **jacobian matrix** must be defined as a reminder: assuming $F : \mathbb{R}^n \to \mathbb{R}^m$ a function, $n, m \in \mathbb{N}^+$, F is given by m real-value component functions:

$$\begin{aligned} F :\ \mathbb{R}^n &\longrightarrow \mathbb{R}^m \\ x &\longmapsto y = F(x) \end{aligned}$$

$$\Downarrow$$

$$x \in \mathbb{R}^n \equiv \big(x_1, x_2, \ldots, x_n\big) \implies y \in \mathbb{R}^m \equiv \big(y_1(x), y_2(x), \ldots, y_m(x)\big)$$

[14] The details won't be explained here; for more information, please read [geo3].

So, if the **partial derivatives** of all these functions $y_1, \ldots, y_m$ really exist, they can be **organized** in an m**-by-**n **matrix** called the **Jacobian matrix of** F, in this way:

$$J_F = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

5.2.1.2 Method Overview

In 1984, Barr offered in [geo7] a **hierarchical solid modeling operations for simulating transformations of geometric objects**, like *twisting*, *bending* or *tapering*, for constructing complex figures from simpler ones. The idea behind this was to process more complex forms by **calculating the transformed per-vertex position and normal vectors from the original ones** according to the jacobian matrix J_F of a **transformation function** F. Besides, this transformation matrices can be joined in a set, so this technique can build easily an animation, like a **keyframe animation**[15].

The transformation function F has to be defined as an $\mathbb{R}^3$ **endomorphism**

$$\begin{aligned} F : \mathbb{R}^3 &\longrightarrow \mathbb{R}^3 \\ p &\longmapsto p' = J_F \cdot p \end{aligned}$$

so to apply F to an object model composed by n vertices $v_i,\ i = 1..n$, the algorithm is as simple as **applying the jacobian transformation matrix** J_F **to every vertex** v_i.

5.2.1.3 Global and local operators

This deformation operators defined by Barr had, besides, the possibility of being **global or local, depending on the affected 3D object zone**, and how the position and normal vectors have to be computed:

- **global** if it modifies the global space point coordinates; that means, every model-comprising vertex position —and normal, indeed—.
- **local** if it only modifies the solid tangent space[16].

5.2.1.4 Examples of global operator deformations

The main innovation of this proposal was the **possibility of altering the transformation while it was being applied to an object**. Let's show some of the most common geometric transformations, as specified in [geoA]: *taper*, *twist*, and *bending* effects. It's important to pay attention on that **all these geometric transformations make use of the own point position** p **for calculating the transformed —deformed— one**; so, this will be a real object transformation.

[15] See section 2.1.1.3 on page 8 for more details.

[16] A tangent space associated to a n-manifold object is a n-dimensional space that contains all the tangent vectors to a certain manifold point. A n-manifold is a mathematical space in which every point has a neighborhood composing a n-euclidean space; thus, lines are 1-manifold, planes are 2-manifold, a sphere is also 2-manifold. . .

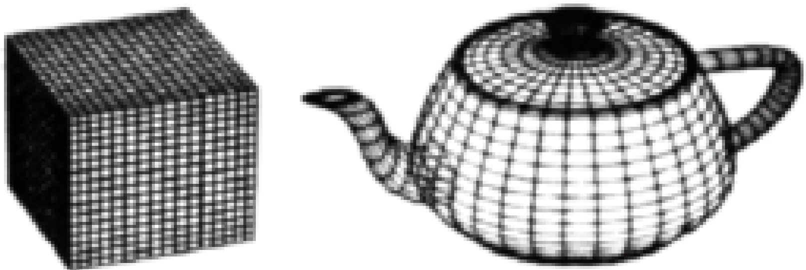


Figure 5.5: *The* `cube` *and* `teapot` *undeformed models.*

These two models, `cube` and `teapot`, are widely and commonly used; so, they will be the affected models that will *receive* the specified geometric transformations. Notice that **the two models are very vertex-sampled** —*e.g.*, the cube has more than eight (the minimum necessary) vertices— for convenient deformations[17].

▷ ***'Taper' geometric deformation***

As defined in [geoB], a **taper deformation** is a scaling transformation whose effect is a **smooth object scalability along a chosen axis** —in this case, the $\mathcal{X}$ axis, but can easily be extended to the other ones—. There's a parametrization set for this deformation: **user-defined scalar-to-scalar threshold values**, x_0 and x_1. By manipulating these threshold values, the deformation is controlled.

$$J_{taper}(p) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & s(p_x) & 0 \\ 0 & 0 & s(p_x) \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} ; \Longrightarrow s(x) = \begin{cases} 1 & x \leq x_0 \\ 1 - 0.5\left(\frac{x-x_o}{x_1-x_o}\right) & x_0 < x < x_1 \\ 0.5 & x_1 \leq x \end{cases}$$

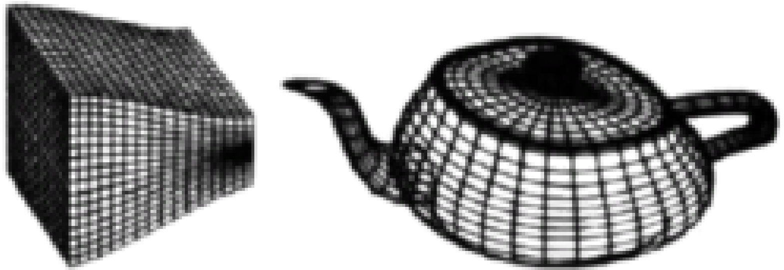


Figure 5.6: *The 'taper' effect on* `cube` *and* `teapot` *models; source [geoA].*

▷ ***'Twist' geometric deformation***

A **twist deformation** is essentially a **progressive rotation along a direction axis** —in the standard specification, the axis is one of the three canonic axis, in this case the $\mathcal{Z}$ one—. The **user-defined parameterization set** includes the **maximum rotation angle** and **two threashold values** again, as in *taper*.

[17] With only eight vertices, a cube couldn't be smoothly deformed, since there are a little amount of vertices to be transformed into new ones. Nevertheless, with so much vertices, a smooth deformation is possible, as can be seen in the next figures.

$$J_{twist}(p) = \begin{pmatrix} \cos\left(r(p_z)\right) & -\sin\left(r(p_z)\right) & 0 \\ \sin\left(r(p_z)\right) & \cos\left(r(p_z)\right) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}; \Longrightarrow s(x) = \begin{cases} 0 & z \leq z_0 \\ \theta_{max}\left(\frac{x-x_o}{x_1-x_o}\right) & z_0 < z < z_1 \\ \theta_{max} & z_1 \leq z \end{cases}$$

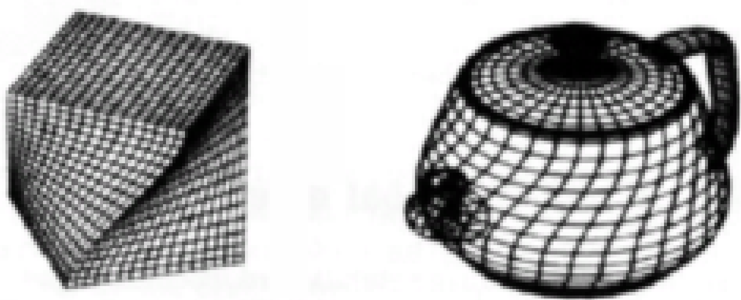


Figure 5.7: *The 'twist' effect on* `cube` *and* `teapot` *models; source [geoA].*

▷ ***'Bend' geometric deformation***

In this last mentioned deformation —from a huge number of possible geometric ones—, the object is bent over an specified axis. In this deformation, the user can set the **bending axis**, the **bending angle and radius** and the **bending start vertex**.

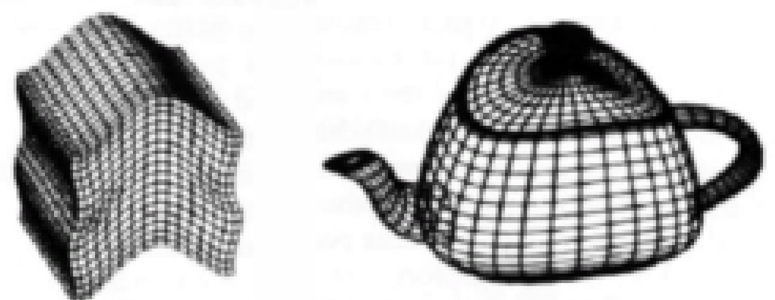


Figure 5.8: *The 'bend' effect on* `cube` *and* `teapot` *models; source [geoA].*

▷ ***'Vortex' geometric deformation***

This is a *twist* deformation variant. It's known that **Any spiral motion with closed streamlines**[18] **is a vortex flow**; thus, specifying the vortex deformation is simply like taking the *twist* matrix and implementing an α vortex function, like the following:

$$J_{vortex}(p) = \begin{pmatrix} \cos\left(\alpha(p)\right) & -\sin\left(\alpha(p)\right) & 0 \\ \sin\left(\alpha(p)\right) & \cos\left(\alpha(p)\right) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}; \Longrightarrow \begin{cases} \alpha(p) = r(p_z) \cdot e^{(-p_x^2+p_y^2)} \\ \Downarrow \\ s(x) = \begin{cases} 0 & z \leq z_0 \\ \theta_{max}\left(\frac{x-x_o}{x_1-x_o}\right) & z_0 < z < z_1 \\ \theta_{max} & z_1 \leq z \end{cases} \end{cases}$$

[18] The *streamline* is a family of curves that are instantaneously tangent to the flow's velocity vector.

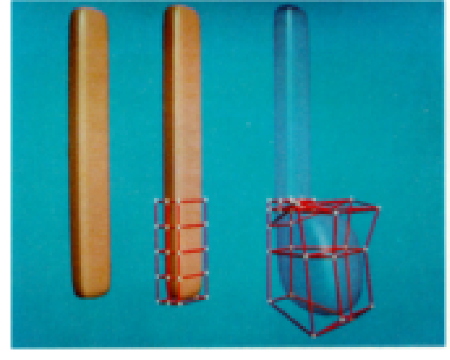
Figure 5.9: *The 'vortex' deformation over a solid primitive; source [geo7].*

5.2.2 Free Form Deformation —FFD—

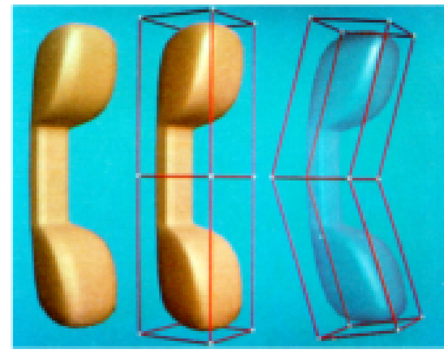
5.2.2.1 Introduction to FFD

Apart from Barr proposal in 1984, two years later, Sederberg and Parry presented in [geoC] a method for **deformating any type of surface primitives, with global as well as local definitions**. And, as Barr's [geo7] proposal, this technique can be used easily for simulation purposes by a **keyframe animation**.

The basic idea is **assuming any 3D object to be deformable**, no matter its shape or type. Then, a **flexible *plastic* bounding box** is built for surrounding all the objects wished to deform. Now, by imposing a **local coordinate system on the flexible *plastic* bounding parallelepiped region** —as shown in figure 5.11(b) on the page 49—, **any original scene point has its correspondence on this coordinate system**, whatever shape the *flexible plastic* surrounding box had.



(a) *A local FFD deformation. Please notice the 'flexible plastic' deformation (red-lined).*



(b) *The global FFD deformation after object's symmetric local FFD executions.*

Figure 5.10: *A complete deformation* animation *is shown in this couple of figures: (a) two local FFD are applied to the bar —only one is shown—, for transformating the bar onto a 'phone receiver', and (b) after, a global FFD bends the 'receiver'; source [geoC].*

5.2.2.2 Methodology details

Mathematically, the FFD approach is based on a **tensor field product trivariate Bernstein polynomial**. As a reminder, we will define the **Bernstein polynomies**: let $f : [0,1] \rightarrow \mathbb{R}$ a continuum function, its n-th Bernstein polynomy, called $\mathcal{B}(n, f, x)$ or $\mathcal{B}_n(, f, x)$ is defined as

$$\mathcal{B}(n, f, x) \equiv \mathcal{B}_n(, f, x) = \sum_{k=0}^{n} f(k/n) \binom{n}{k} x^k (1-x)^{(n-k)}, \quad \text{where } \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

So, defining the **parallelepiped's local coordinate system** as $< x_0, (\mathcal{S}, \mathcal{T}, \mathcal{U}) >$; where x_o is the origin, and $\mathcal{S}, \mathcal{T}, \mathcal{U}$ the three axis vectors[19]. Hence, any point $X \in \mathbb{R}^3$ has an (s, t, u) triplet associated as coordinates in this system such that:

$$X = x_0 + s\mathcal{S} + t\mathcal{T} + u\mathcal{U}, \quad \text{where} \begin{cases} s = \dfrac{\mathcal{T} \times \mathcal{U} \cdot (X - x_0)}{\mathcal{T} \times \mathcal{U} \cdot \mathcal{S}} \\ t = \dfrac{\mathcal{S} \times \mathcal{U} \cdot (X - x_0)}{\mathcal{S} \times \mathcal{U} \cdot \mathcal{T}} \\ u = \dfrac{\mathcal{S} \times \mathcal{T} \cdot (X - x_0)}{\mathcal{S} \times \mathcal{T} \cdot \mathcal{U}} \end{cases}$$

for **any point X interior to the parallelepiped**,

$$0 < s < 1, \;\; 0 < t < 1, \;\; 0 < u < 1;$$

however, in case of having a **prismatic bounding box** —a prism is a particular parallelepiped case—, like in the figure 5.11, it's known that **each axis is the resulting cross product of the other two**:

$$\begin{cases} \mathcal{S} = \mathcal{T} \times \mathcal{U} \\ \mathcal{T} = \mathcal{S} \times \mathcal{U} \\ \mathcal{U} = \mathcal{S} \times \mathcal{T} \end{cases} \Longrightarrow \begin{cases} s = \mathcal{S} \cdot (X - x_0) \,/\, \| \mathcal{S} \| \\ t = \mathcal{T} \cdot (X - x_0) \,/\, \| \mathcal{T} \| \\ u = \mathcal{U} \cdot (X - x_0) \,/\, \| \mathcal{U} \| \end{cases}$$

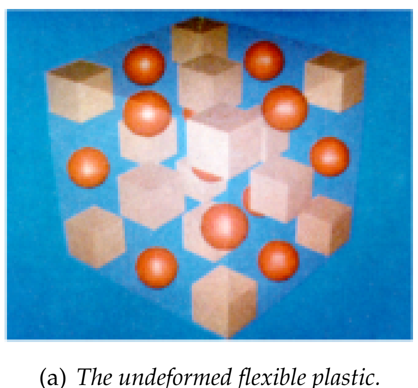
5.2.2.3 Grid of Control Points

Once the local coordinate system for the flexible parallelepiped has been specified, it is necessary to establish a **regular grid of control points** P_{ijk} for the parallelepiped. Hence, **the deformations will be defined by moving some control points, and then the inner scene will follow the movements** by the (s_r, t_r, u_r) triplet calculus for each point X_r of the scene —see figures 5.11(c) and 5.11(d) for a graphical explanation—.

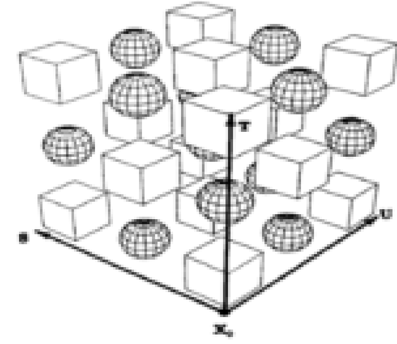
When the **regular control points grid of size** $l \times m \times n$ is defined, like in the figure 5.11(c) —where the grid is $1 \times 2 \times 3$-sized—, there will be $l+1$ planes in $\mathcal{S}$ direction, $m+1$ in the $\mathcal{T}$ one, and $n+1$ planes in the direction of $\mathcal{U}$. So, the P_{ijk} **control point positions** can be easily found by this calculus:

$$P_{ijk} = x_0 + \frac{i}{l} \cdot \mathcal{S} + \frac{j}{m} \cdot \mathcal{T} + \frac{k}{n} \cdot \mathcal{U}$$

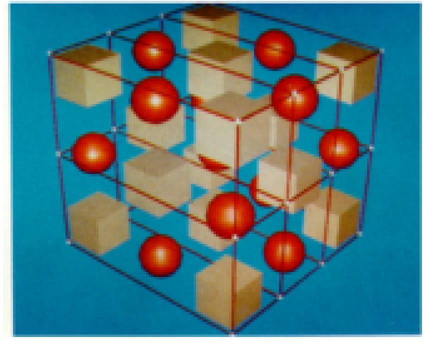
[19] And, as it can be deduced, in the case of AABB —*Axis Aligned Bounding Box*, view section 3.2.2.2 for more info— *flexible plastic*, they are parallel to the canonic axis $(\mathcal{X}, \mathcal{Y}, \mathcal{Z})$.



(a) *The undeformed flexible plastic.*



(b) *The local 'coordsys'* $< x_0, (\mathcal{S}, \mathcal{T}, \mathcal{U}) >$.



(c) *The undisplaced control points.*

(d) *The control points in a deformed position.*

Figure 5.11: *There is a complete Free Form Deformation —FFD— executed over several objects (cubes and spheres) clearly embedded in the bounding flexible plastic; source [geoC].*

5.2.2.4 Capabilities and limitations

The FFD proposal is very **useful for 3D animation** due to the possibility of creating discretized deformation steps by a **keyframe interpolation of control points**; however, **there are some deformations not reachable yet**, like blendings, because the starting object shape doesn't allow FFD to change the object's orientation.

5.2.3 Interactive Axial Deformations —AxDf—

5.2.3.1 Method Overview

Lazarus, Coquillart and Jancène proposed another 3D object deformation method in [geoD], similar —in behaviour— to FFD: **based on a 3D axis user-definition, the movement of the axis generates the object deformation in the same way**. More technically:

1. the user defines a 3D axis —inside or outside the object, it doesn't matter—. This **axis can have any shape**, not only a simple line;

2. now, the user changes the shape of the axis, and **the axis deformations will be automatically passed to the object** —or objects—;

This method, called AxDf —*Axial Deformations*—, offers the possibility of applying the same deformation axis to more than one objects; however, the **passed deformations to far objects may be problematic or even badly done** due to the implicit nature of the method.

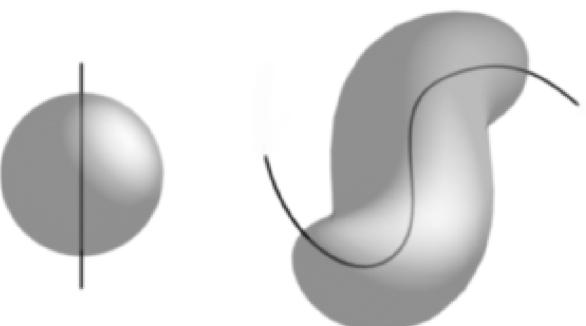


Figure 5.12: *A `sphere` axis deformation: on the left there's the undeformed sphere with the defined axis, also undeformed; on the right, the sphere, deformed in an intuitive and consistent way with the axis deformation; source [geoD].*

5.2.3.2 Algorithm Details

This method's problem is **to find the way to pass the axis deformations to the object**. For managing this obstacle, the authors propose a three-step process:

1. **each *object vertex* V has to be attached to one *axis point* A_V, and its local coordinates have to be computed**; for doing this, the authors implement the closest-vertex-point strategy, though they warn that it may be **conflictive** for complex axis[20].

2. **the deformed transformation for finding *axis point* A'_V —homologous to A_V— is used to compute the deformation of the *object vertex*'s (x, y, z) local coordinates**; the direct transformation doesn't need any explanation, but the problem of finding the local coordinate frames between A_V and A'_V is solved with the **Frenet frame**[21].

3. in order to localize the to-be-deformed area by this local coordinate system, a **zone of influence** is defined for finding the desired 3D space portion.

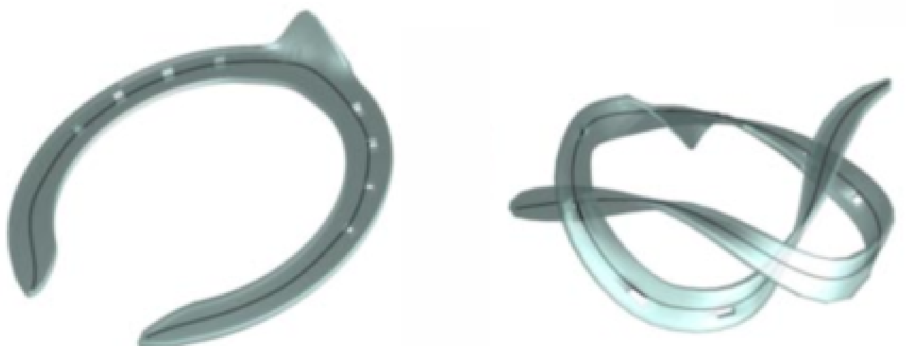


Figure 5.13: *The `horseshoe` axis deformation. Please notice the axis, now adapted to the object surface; thus, the deformation can be so complex as can be seen in the right image; source [geoD].*

[20] *e.g.*, two —or more— *axis points* at the closest distance from a *object vertex*.

[21] For each point of a 3D curve, the Frenet frame is represented by three ortogonal unitary vectors: *tangent*, *normal* and *binormal* of the curve at that point. A binormal unit vector B associated to a curve point is defined as the cross product of the tangent vector T and the normal one N. A non-unique normal at curve point is obviously a huge problem

5.2.3.3 Advantages and lacks of AXDF

- since **the deformation is object-independent**, it can be re-used as much as wanted, and besides, **it can be applied to any kind of modeled objects**;
- it can be combined with other deformation techniques, such FFD;
- a set of subsequent axis can be used for a **keyframing object-deforming animation**.

However, the technique isn't infalible, due to a **lack of ambiguity in the two steps of deformation process**: *(i)* it can be **more than one** *axis point* candidate for being associated to an *object vertex*, and *(ii)* a **3D curve hasn't a unique normal at a single point**, so the coordinate frame following the Frenet method is not a good solution. Nevertheless, **for simple axis, it's a successful method**.

5.2.4 Deformation by Wires

5.2.4.1 Introduction and *wire* definition

Related to the previous axial deformations, there is an approach from 1998 by Singh and Fiume at [geoE], where they **bring geometric and deformation modeling joined together by using a collection of *wires* as a representation of the object surface, and a directly manipulated deformation primitive at the same time**.

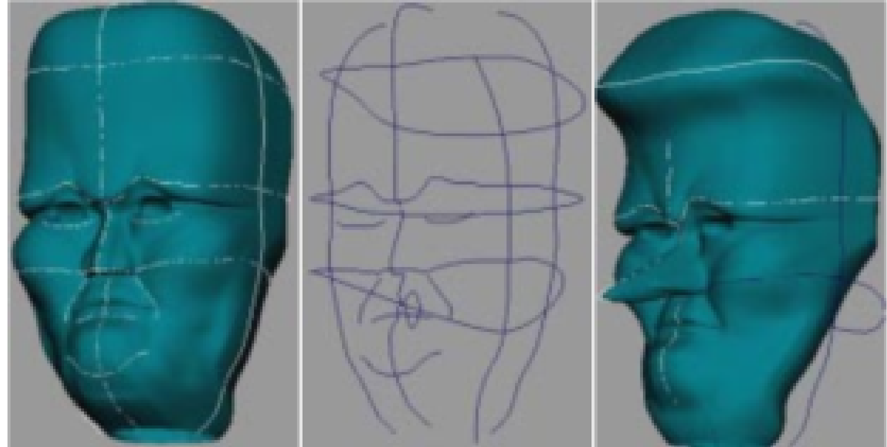


Figure 5.14: *An example of a facial modeling by 'wire' specification, and the next animation and deformation by 'wire' manipulation; source [geoE].*

The authors define a *wire* as a **curve whose manipulation deforms the surface of an associated object near the curve**. Its specification is a tuple $< W, R, s, r, f >$, where:

- W and R are free-form parametric curves —the *wire* curve and the *reference* curve respectively; W will be deformed and R will be useful for comparatives and transformations, and can be swapped within the method for complex calculum—,
- s is a scalar that controls radial scaling around the curve,
- r is another scalar defining an influence radius around the curve,
- and $f : \mathbb{R}^+ \rightarrow [0, 1]$ is the *density function*, and must be at least C^1 and monotonically decreasing.

5.2.4.2 Algorithm overview

Briefly, when an object is bound to the wire, the method follows these steps:

1. the **wire's domain of influence** is indicated by a r-radial surface offset around the reference curve R;
2. using f, the within-offset object points are computed to find its **deformation influence**;
3. W is manipulated, and every undeformed point object will be **deformed if its deformation influence is great**; next, a W-R-*swapping* is executed, in order to have as reference the last deformation;

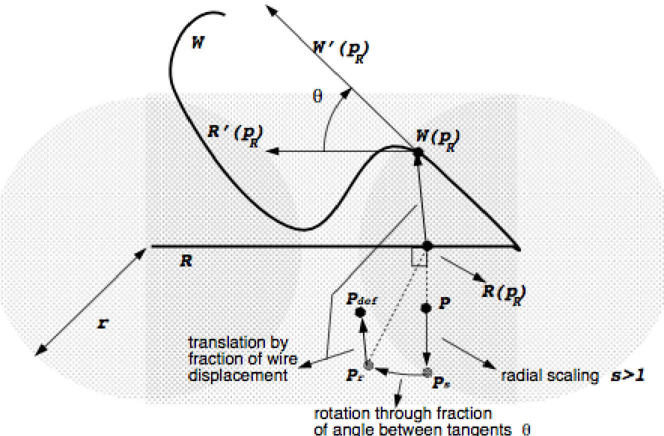


Figure 5.15: *Deformation of a point P to P_{def} by a wire W. The figure parameter p_R is the correspondence point, within R, of P; source [geoE].*

CHAPTER 6

Physically-Based Deformations

6.1 Introduction to deformation based on physics

6.1.1 Overview of time stepping deformation algorithms

6.1.1.1 Accuracy VS speed performance

Nowadays, some **precise and complex simulation methods based on physics and dynamics** have been developed from engineering[22]. These deformation-based animation algorithms **are non-real-time because of their complexity, since they are so sofisticated**; however, as Thomas Jacobsen says in [phy1], for an **interactive use, real-time is the primary concert, so the accuracy is not as important as three another handicaps**:

- **belieavability**, properly parameterized for immersive feeling improvements;
- **stability**: a physical deformation method isn't good if the object seems to drift, or vibrates, or if the model simply explodes by an iteration's result divergence;
- **execution time**, because the available computational time is really a little portion of the *single-frame timing*. From now, this timing will be called **time step**.

```
1  time_step = sys_clock();
2  while true do
3      time_step = sys_clock() - time_step;
4      foreach Object O in Scene S do
5          S.O = deformObject( S.O, time_step );
6          drawObject( S.O );
7      endforeach;
8  endwhile
```

PSEUDO **code 6.1:** *vast algorithm for a deformation animation.*

[22] And almost of them are always composed by an iterative method due to the computer graphics iterative rendering integration —each animation frame must be rendered subsequently from the previous one—.

6.1.2 A simple particle P as the deformation system

Before entering the complex analysis and deformation management and handling methodologies for deformation models, a **list of concepts and strategies** will be defined now **as if the entire system was a simple particle**, called P. So, according to Susin's *Volume Modeling Course* at [phy2], there are **two concepts** to be taken in account: the *state vector*, and the *time step associated to a numerical integrator*.

6.1.2.1 State Vector and Vector Field of a simple particle

▷ ***State vector*** $x(t)$

The P **position** through the time-line will be defined by its **state vector** $x(t)$, which can contain several parameters: *position*, *velocity*, *previous position*, *mass*... Besides, the **time-dependent evolution of the state vector is given by the following ODE**[23]:

$$\dot{x} = \frac{dx}{dt} = f\big(x(t), t\big)$$

where velocity $\dot{x}$ is the space variation in relation to the time; so, by knowing the initial P position, the **initial value problem** is about to find the P position at any time step.

The **state vector dimension** will, at least, be the same dimension of the space where P is moving along. So, for **the most physically-based deformation system, implementing the *Newton's* 2nd *law***, the velocity will have to be introduced as a variable (since acceleration a is the second order derivative of the position x, *ergo* $a \equiv \ddot{x}$:)

$$a = \frac{F}{m} \implies \ddot{x} = f(x,t) \rightarrow \left\{ \begin{array}{l} \dot{x} = v \\ \dot{v} = f(x,t) \end{array} \right. \implies \begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} \implies \begin{pmatrix} v \\ f(x,t) \end{pmatrix};$$

now the system **dimension is** $2n$, being n the dimension of $x \in \mathbb{R}^n$.

▷ **ODE *vector field***

The $\dot{x}$ velocity can be interpreted as the $x(t)$ tangent line's slope at instant t. In other words, a vector that indicates the direction. Thus, for each space coordinates where the ODE is defined, there will be a **director vector**. The set of these *plane point-director vector* is called **ODE vectorial field**.

6.1.2.2 The time step Δt. The numerical integrator concept. The Euler numerical integrator

Almost every particle P's **path-tracking method is based on a time discretization**, so the particle is going along its trajectory by small time steps Δt, called here **integration steps**. Thus, the next step position will be defined as

$$x\big(t + \Delta t\big) = x(t) + \Delta t f(x,t);$$

so it's **easy to generate subsequent configurations** $x(0), x(\Delta t), x(2\Delta t), \ldots, x(n\Delta t)$. Additionally, following the previously mentioned **iterative scheme**, the resulting equation is the **ODE integration's Euler method**:

$$x_{n+1} = x_n + \Delta t f(x_n, t_n)$$

[23] Acronym of *Ordinary Differential Equation*. An ODE is an equation that involves a function and its derivatives, such that a n-order ODE has the form $F\big(x, y, y', \ldots, y^{(n)}\big) = 0$.

6.1.2.3 The integration system scheme

The numerical integrators have been finally defined, and also the common approach of Newton's 2nd law for their resulting convergence. But, additionally, as said in [phy4], there are **another two important factors** for any physically-based deformation iterative system, **external to the system itself and of stochastic nature**:

1. **collision handling**; a collision can be between **two objects**, or **self-collisions** between different parts of the same object; besides, the object can be **rigid or non-rigid**, and the collision handling can derive to one of two kinds: **elastic or inelastic**;

2. **constraint management**; a deformable object can be configured with some constraints or restrictions; *e.g.*, volume preservation, maximum deformation angle, elasticity or damping limit values, ... and all those constraints must be managed and solved if anyone is violated;

This two factors will be studied later, on the next pages, in more detail. Nevertheless, now in the next figure there is a **scheme of a simple iteration on a numerical integration system focused on physically-based deformations**; please notice all the stages: the integration step, the solver due to the Newton's 2nd law applying to the system, and constraint and collision management[24].

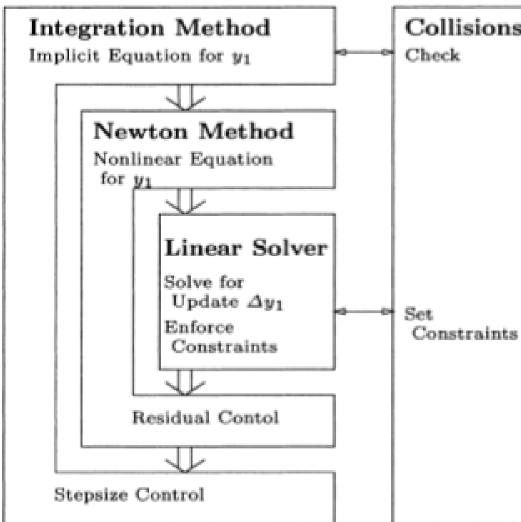


Figure 6.1: *Architecture of an iterative integration system for animation by deformation; source [phy4].*

6.1.2.4 The most used ODE integrators

Apart from the Euler method, there are other numerical integrators; the most common, however, are the following:

- **Modified Euler Method, or Runge-Kutta** 2; the idea is to improve the main time step by an intermediate step; in more tecnhical terms, the formula is as follows:

$$x_{n+1} = x_n + \Delta t\, f\Big(x_n + (\Delta t/2) f(x_n, t_n),\, t_n + (\Delta t/2)\Big)$$

[24] The scheme indicates clearly that collisions and constraints are external factors of the system; that is, stochastic events to be handled that modify extremely a particle's state vector.

- **Runge-Kutta** 4; following the strategy of the Runge-Kutta 2 method, but now there are 4 subiterations:

$$x_{n+1} = x_n + (1/6)\big(k_1 + 2k_2 + 2k_3 + k_4\big) \Longleftarrow \begin{cases} k_1 &= \Delta t f\big(x_n,\, t_n\big) \\ k_2 &= \Delta t f\big(x_n + (k_1/2),\, t_n + (\Delta t/2)\big) \\ k_3 &= \Delta t f\big(x_n + (k_2/2),\, t_n + (\Delta t/2)\big) \\ k_4 &= \Delta t f\big(x_n + k_3,\, t_n + \Delta t\big) \end{cases}$$

- **Verlet**; this is an approach featured on [phy1] by Thomas Jakobsen; he calls this method a *velocityless* integrator due to the **velocity approximation by the difference between the actual and the previous position**[25]:

$$\begin{cases} x_{new} &= x_{old} + v_{old}\Delta t \\ v_{new} &= v_{old} + a_{old}\Delta t \end{cases} \Longrightarrow \begin{cases} x_{new} &= (2x_{old} - x_{prev}) + a_{old}\Delta t \\ x_{prev} &= x_{old} \end{cases}$$

6.1.2.5 Accuracy and stability of numerical integrators

Each possible numerical integrator has to **find the equilibrium between accuracy** —the accumulated error in a certain step in relation to the real value— **and stability** —the integrator must always make the system converge to the correct solution— **in contrast to the time step**.

▷ ***Accuracy***

The **accumulated error** of an iterative step cannot be calculated because the real value isn't available, but **can be estimated in function of the time step Δt and the numerical integrator used**; *e.g.*, the already cited Euler method has an error of $O\big((\Delta t)^2\big)$.

Hence, there are **two possible strategies** to improve the particle path calculus:

1. **decrease the time step**; this will improve the precision but the computation step number will increase and besides physics-dedicated-time per frame will be reduced[26], so the method modification maybe can not be possible;

2. **to change the numerical integrator**, with a greater error order $O\big((\Delta t)^k\big),\ k \geq 3$; in the following table are shown the error measure estimations of the most common numerical integrators

Numeric Integrator	Estimated Error
Euler	$O\big((\Delta t)^2\big)$
Euler Modified (Runge-Kutta 2)	$O\big((\Delta t)^3\big)$
Runge-Kutta 4	$O\big((\Delta t)^5\big)$
Verlet	$O\big((\Delta t)^2\big)$

Table 6.1: *Iterative* ODE *integration estimated errors in relation to the numerical integrator.*

[25] In other words, this method doesn't store the position and velocity values but the previous and actual position; by this, a linear and energy-constant approximation of the velocity is assumed, but the number of perations is dramatically reduced.

[26] Here it's studied only a particle P, but when there is a particle system, this computation step is transformed to a equation system, and then the time step is very important.

▷ ***Stability or Convergence***

The most important requirement on a ODE deformation system like the studied in this chapter is, with no doubt, its **stability**. So, is defined a **stiff ODE** the system who **always converges —and quickly— to an equilibrium state**, whatever it is. Thus, an integration scheme that is yielding a **bounded solution** —due to a suitable numerical method— is called **stable**.

The stability of a system is extremely **related to the time step**. There is a clear example of that in the following figure about the state vector $\dot{x} = -kx$. Notice that the system is **oscillating** for a time step $h > 1/k$, and is completely **unstable** for time steps $h > 2/k$:

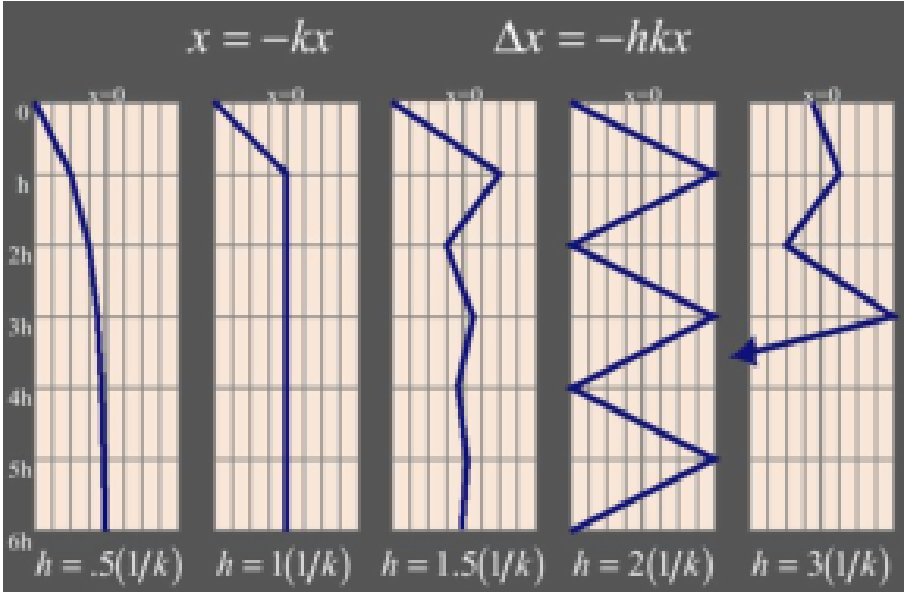


Figure 6.2: *Stability scheme by* ODE *system plottings depending on the time step h; source [phy3].*

However, there are some proposals for solving possible unstablements. For example, by using **adaptative time step handling**, and always **selecting the greater time step possible**, the stability will be increased[27]. Nevertheless, since computation requires to establish a native minimum time step, the stability is not always warranted.

Another good solution is considering an **implicit methodology**, where **all the possible slope changings** are taken into account; indeed, the idea is to find the **mean of slopes**

$$x(t + \Delta t) \; = \; x(t) \; \Delta t \Big[(1 - \lambda) f(x, t) \; + \; \lambda f(x + \Delta x, t + \Delta t) \Big];$$

it's important to notice that, for $\lambda = 1$, here's the **Euler implicit method**:

$$x_{n+1} \; = \; x_n \; + \; \Delta t f\big(x_{n+1}, t_{n+1}\big)$$

and the way for solving it is **isolating** x_{n+1}.

[27] This is because a big time step makes the system becoming unstable, as seen in figure 6.2; however, a very small time step may be stable but *(i)* not computable, or great-error-accumulating, so the system won't be considered stable due to converging to invalid solutions.

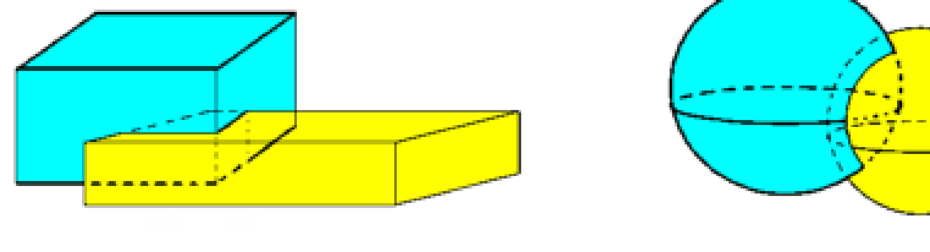
6.1.2.6 Movement restrictions

It has been seen the path-tracking and management of a single particle P within the space, but with no counting the **possible interactions and restrictions**, *e.g.*, interaction between P and the other space objects. This restrictions **can provoque extremal particle path modifications**.

▷ ***Collisions***

Two or more particles can **collide between them** if they are in the same animation space. At this moment, a **stochastic process** has occurred, and the path-tracking is being terribly affected, so the **involved particles will modify their trajectory according to an external factor, and this affair must be managed by the deformation system after the solver integration step**, as can be seen in the figure 6.1, on page 55. Additionally, there are two extremal kinds of collisions, and of course, the intermediate types: from a perfect inelastic [28] colliding stucking until a 180° direction-turning by a perfect elastic[29] one.

The **collision management must be robust and efficient** in the same level, so for a **collision between two generic 3D objects** this task is really **impossible**. Because of that, the collision checking is often done by reducing an object to its **bounding box** or even its **bounding sphere**, geometrically easier[30].



(a) *Intersection between bounding boxes*

(b) *Intersection between bounding spheres*

Figure 6.3: *Intersection between* 3D *bounding volumes of* 3D *objects.*

Let's see now the geometrical computation for **checking the intersection between two bounding spheres** S_1 **and** S_2, defined by their respectives center C, radius r and velocity vector $\vec{v}$:

$$\begin{cases} S_1 = (C_1, r_1, \vec{v_1}) \\ S_2 = (C_2, r_2, \vec{v_2}) \end{cases} \Longrightarrow \begin{cases} C_1(t) & = & C_1 + \vec{v_1}t \\ C_2(t) & = & C_2 + \vec{v_2}t \\ r_{max} & = & \max(r_1, r_2) \end{cases}$$

$$\Downarrow$$

$$\boxed{\begin{array}{llll} \| C_2(t_i) - C_1(t_i) \| & \leq r_{max} & \rightarrow & \textbf{intersection in } t_i \textbf{ time instant} \\ \| C_2(t_i) - C_1(t_i) \| & > r_{max} & \rightarrow & \textbf{no intersection in } t_i \end{array}}$$

[28] An inelastic collision is a collision such that the involving objects suffer deformations; ideally, the two objets will be stuck from that moment.

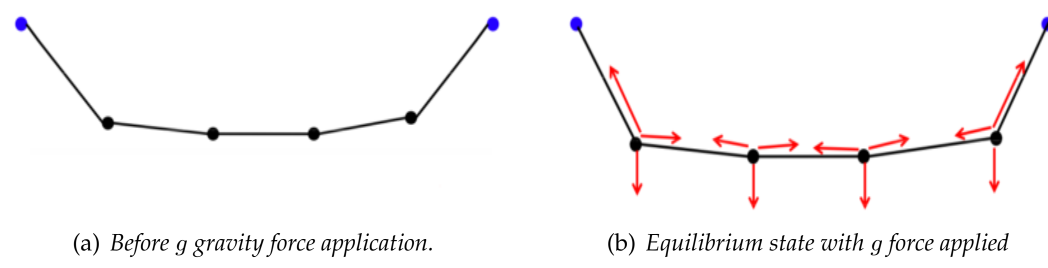
[29] A collision is considered elastic if the kinetic energy and the linear momentum is conserved for the implied objects.

[30] A sphere is specified by its center and its radius using the equation $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$, while a prismatic box is defined by 6 planes, each one delimited by its intersections with the others.

▷ ***Constraints***

A system constraint is based on **particle forced behaviours**, like connection with other particles —*e.g.*, a delimited distance range with respect to another particle, or the same relative position between them—, or determined surface passing-through —*e.g.*, a forced friction with floor, or an elastic behaviour in relation to another particle or object—.

In the following figure there is a set of particles with two constraints: *(i)* **connections** between subsequent particles —generating a kind of spring—, and *(ii)* **delimited length range** for all these *springs*. Notice that, when a *gravity* **force is applied, the constraints are maintained** —red arrows are remarking the gravity as long as the constraining forces— **by a deformation (red image) of the original relaxing shape (left image)**.



(a) *Before g gravity force application.*

(b) *Equilibrium state with g force applied*

Figure 6.4: *Force equilibrium in a particle system with connection restrictions; source [phy9].*

6.2 Dynamic Deformable Model Systems

6.2.1 Introduction to deformable systems

6.2.1.1 Definition of Particle System

A single **particle** P is usually displayed as a graphical point or sphere, and **stores a set of attributes**, including its vector state and another *object* parameters such as temperature, mass, lifetime, … These attributes are, in essence, the **numerical definition of the particle behaviour in front of dynamic deformations and movements over time**, computed by a solver for each animation time step.

A **solver** will be an **algorithm that executes the time step integration for any particle** P**, given its attributes, its state vector and, of course, the time step**. A solver is **directly related to a numerical integrator**, so each integrator within the table 6.1, on page 56 will must have its own solver:

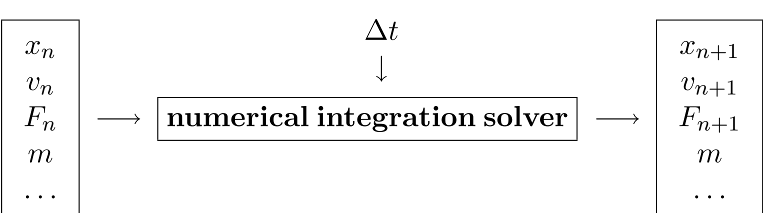


Figure 6.5: ODE *numerical integration scheme for a single particle P.*

A **particle system**, then, will be defined as a **set of finite particles, each one with its own attributes and state vectors**; and besides, can be two inherent natures: **independent** —in other words, omitting inter-particle forces— or **dependent** —with inter-particles forces or another system constraints—:

- a **system with independent particles** are ideal for *waterfalls*, or *fountains*, or *splashes*, due to this animations requires only a convincing and fast animation, and because of the huge number of particles and their stochastic behaviour of that the inter-particle forces are not required;
- however, for more stiff animations, a **system with dependent particles** is the most accurate. The interaction between particles really evolve through the timeline, so particle system —being itself a single entity and not a set of entities (particles)— complex geometry and topological changes are easily managed using this approach;

6.2.1.2 The deformable models as dependent-particle system. The continuum elasticity

Since the **independent particle systems can be computed by using the methodology explained until now** —that is, each particle as a system itself, as in the previous section—, this type of particle systems will be ommitted from now, **the attention will be focused on the dependent particle systems**. So, a three-dimensional deformable object is tipycally defined by:

1. its **undeformed shape**, composed by a continous connected discrete set of points $M \subset \mathbb{R}^3$, where each point $m \in M$ is called the **object property at this point**.
2. a set of **material parameters** that define **how the deformation occurs** under external force applications.

Hence, the 3D object will be deformes when a force is applied over it. How is deformed will be **determined implicitly** by the **displacement vector** $u(m)$, that leads any point from its undeformed position m to the deformed $x(m)$ one. Since new locations $x(m_i)$ are defined for all $m_i \in M$, a **vector field on** M **is defined by the set of** $u(m_i)$ **displacement vectors**.

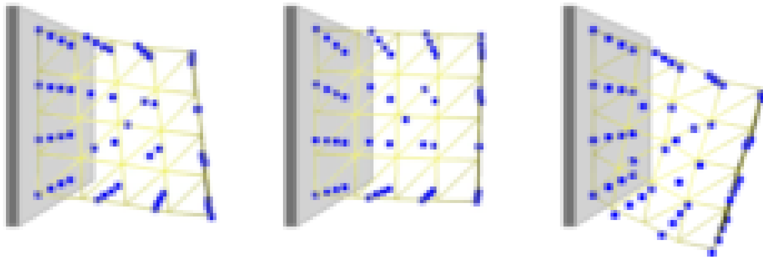


Figure 6.6: *Some ε elastic strain deformation parameters, from left to right:* Cauchy, Cauchy (simplified) *and* Green; *source [phyA].*

From $u(m)$ is computed the called **elastic strain**, named ε —a dimensionless quantity representing a **constant displacement field**—, since it's computationally measured in terms of spatial variations of $u(m)$:

$$m \in M \subset \mathbb{R}^3 \longrightarrow u(m) = \begin{pmatrix} u \\ v \\ w \end{pmatrix} \Longrightarrow \nabla u = \begin{pmatrix} \mathcal{U}_x & \mathcal{U}_y & \mathcal{U}_z \\ \mathcal{V}_x & \mathcal{V}_y & \mathcal{V}_z \\ \mathcal{W}_x & \mathcal{W}_y & \mathcal{W}_z \end{pmatrix}, \mathcal{A}_{(x|y|z)} \equiv \frac{\partial a}{\partial(x|y|z)}$$

The most common strains in computer graphics are the left and right deformation resultings from figure 6.20:

- **Cauchy's linear tensor**: $\varepsilon_C = 1/2 \cdot \left(\nabla u + |\nabla u|^T\right)$;
- **Green's non-linear tensor**: $\varepsilon_G = 1/2 \cdot \left(\nabla u + |\nabla u|^T + |\nabla u|^T \cdot \nabla u\right)$;

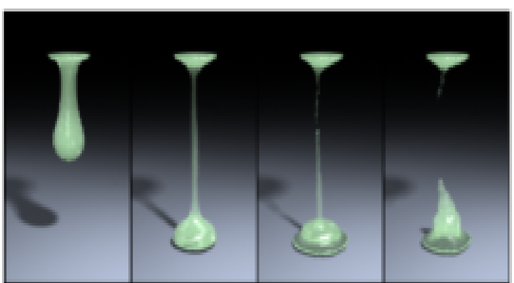
The **Cauchy simplified** won't be explained, because it's an experimental proposal featured on a prior work of [phyA] and it's out of this project range. However, it's a good approach due to its advantages in multiresolution dynamic behaviour.

6.2.2 Points of View of a Physical Simulation. Solver Typologies

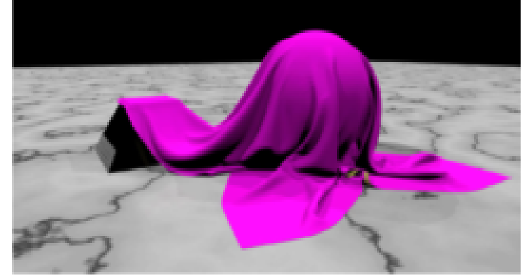
As it's said in [phy5], there are **two general points of view** of describing an object desired to be physically-simulated, or better said, **describing how an object has to be discretized to work with it numerically**:

1. the **Eulerian point of view** describes an object as a **stationary** set of points and calculates how the material properties stored at the stationary point set are being modified over time.
2. the **Lagrangian point of view**, on the other hand, describes an object as a set of **moving** points —called *material coordinates*— that tracks a trajectory and changes their position over time.

No one of the models is the best for any given application. Instead, a **huge amount of parameters and considerations** need to be taken into account, such as *(i)* model representations, *(ii)* physical parameter range, *(iii)* computational requirements of the simulation —real-time, interactive, off-line, ...—, *(iv)* topological changes, ...



(a) *Example of an Eulerian-based fluid animation.*



(b) *Example of a Lagrangian-based cloth simulation.*

Figure 6.7: *Visual results of the two object animation description point of views; source [phy5].*

So, **in the next two sections the two points of view —POV— will be explained**, following a little bit of the teachings of [phy5] with addings from another publications, a real **survey** on physically-based deformation methodologies. However, **this project is focused on the Lagrangian point of view, so only a brief of the Eulerian one will be found here**.

Additionally, **this thesis is about real-time —or interactive at least— physically-based deformations**; that implies, some explained methods in the next sections are inherently offline, so that proposals won't be detailed, or even not explained. This decision is made due to not making a pure clone of [phy5].

6.3 Physical Simulation based on Lagrangian POV descriptions

6.3.1 Mesh Based Methods

6.3.1.1 Finite Element Method —FEM—

This is one of the most popular methods for **solving Partial Differential Equations —PDE[31]— on irregular grids**. This kind of methodology can be used due to the vision of an **object as a discretized connected volume in an irregular mesh**. As a little overview, it can be said that FEM is a method that **approximates a continuous deformation for a pointset** $m \in M \subset \mathbb{R}^3$**, defined by a displacement vector field** $u(m)$**, to a sum of linear approximative functions** $\tilde{u}(m)$ **already defined within a set of finite elements**.

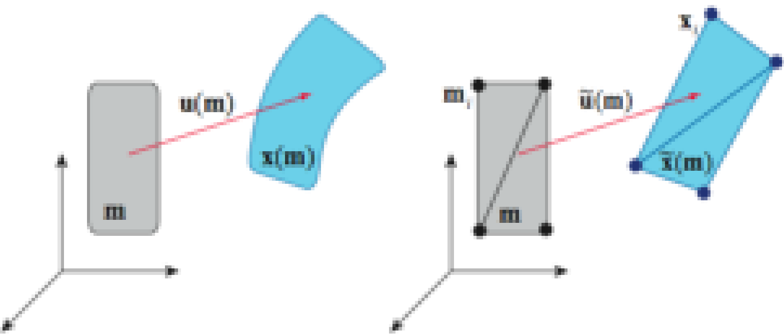


Figure 6.8: *A scheme of the approximate deformation* $\tilde{u}(m)$ *of a continuous deformation* $u(m)$*, done by the Finite Element Method; source [phy5].*

More concretely, the FEM method **gives real solution's continuous approximations** due to it's expressed in terms of **3D local basis function** $N_j(r,s,q)$ combinations —in general, interpolation functions—. So, for this thesis' main problem, the **generic physically-based 3D particle system deformation case for** $u(m)$ **and its implicit temporal dependency** —due to the iterative numerical integrators—, the **solution** is

$$u(m,t) \;\equiv\; u(r,s,q,t) \;=\; \sum_j q_j(t)\, N_j(r,s,q),$$

being $q_j(t)$ some time-depending scalar weights for every approximative function.

▷ ***Continuum mechanics provides the solution***

The PDE that leads the dynamic elastic material deformation is based on the **continuum mechanics** and the formulation is given by

$$\rho\ddot{x} \;=\; \nabla\sigma + f \;\Longrightarrow\; \nabla\sigma \;=\; \begin{pmatrix} \sigma_x^{xx} & \sigma_y^{xy} & \sigma_z^{xz} \\ \sigma_x^{yx} & \sigma_y^{yy} & \sigma_z^{yz} \\ \sigma_x^{zx} & \sigma_y^{zy} & \sigma_z^{zz} \end{pmatrix}, \; \sigma_{(x|y|z)}^{a(x|y|z)} \equiv \frac{\partial\sigma(a,(x|y|z)}{\partial(x|y|z)}$$

[31] The Partial Differential Equations are a type of mathematical differential equations involving an unknown function(s) of several independent variables and their respective partial derivatives w.r.t. those variables. This kind of equation is used to formulate some physical phenomena —like fluids or elasticity—, and various of them can interestingly have idential PDE formulations since they are governed by the same underlying dynamics.

where the conceptual parameters of this equation are:

- ρ is the material density,
- f are the set of external applied forces —gravity, collisions, ...— to the model,
- $\nabla\sigma$ is the divergence operator, given by a transformation from the 3×3 stress tensor to the 3-vector defined in the equation;

▷ FEM ***application to a deformable model***

As it's said in [phy2], to apply the Finite Element Method to a certain problem, defined by the PDE P, consists in executing this schematic algorithm:

```
1     function applyFEM( Problem P, Set<BasisFunction> N )
2     returns Set<SolutionFEM>
3         Set<Domain> D = domainDecomposition( P );
4         Problem A = approximateProblem( P, N );
5         WeakForm W = getWeakForm( A, N );
6         Set<SolutionFEM> S = solveProblemByFEM( W, N );
7         return S;
8     endfunction
```

PSEUDO code 6.2: *Algorithm for* FEM *application to a problem* P*.*

So, the schematic proceeding is composed by a four-step method, more detailed —in relation to the 3D model deformation and not in generic calculus— now:

1. **Domain Decomposition**; here the deformable model has to be subdivided into subdomains E_j called *elements*; in the 3D space case, these elements will be **tetrahedrons or rectangular prisms** —cubes or *voxels*—.

2. **Approximation Functions**; that is, the basis functions $N_j(r,s,q)$ choosing step. Generally, this functions are **interpolation polynomies** —Lagrange, Hermite, ...—, but they can also be B-splines, Fourier Series or even *wavelets*[32]. The more high order would be the basis functions, the more exact solution —w.r.t. the continuous deformation— will get, but also more reference points will must be taken and more computational time will be needed.

3. **Problem's Weak Form**. Given a PDE $\mathcal{P}$, it has to be expressed in its **differential or weak form**; that is, the aim is to find solutions for the —a priori infinitely— discretized initial problem,

$$\int_{E_j} v(r,s,q)\,\mathcal{P}\,dr\,ds\,dq \;=\; 0,$$

where $v(r,s,q)$ is an arbitrary weight function. Tipically, the most common approach for this weight functions is the **Galerkin method**, that assumes the local basis functions as the weighted ones: $v(r,s,q) = N_j(r,s,q)$. Now the procedure for computing each element E_j won't be detailed here; *c.f.* [phy2, section 9].

[32] There won't be a reminder of all these mathematical curves, series, polynomies and formulations, because they are really within project out-of-range.

4. **Element Joining and Resolution**. Now it's only the problem solving left; it's important to notice that the **connection between the elements must be taken in account**, so a certain point can be within more than one E_j. Therefore, the **global problem is approached as a k-order equation system, being k the number of reference points**.

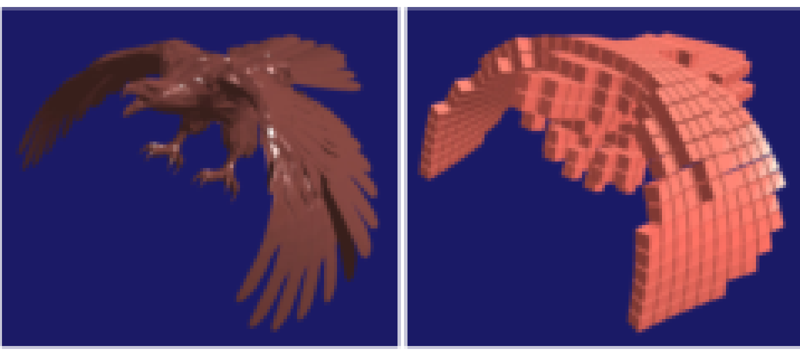


Figure 6.9: *Hexahedral finite element volumetric representation of a topologically inconsistent mesh, ready for* FEM *application (notice that each hexahedron is one of the E_j subdivisions or elements); source [phy5].*

6.3.1.2 Finite Differences Method

Basically, as [phy5] tells, this method is a simplification of the FEM by using a **regular spatial grid** of elements. It's **easier** than FEM (in implementation as long as in conceptualization), but it has two **important and inherent drawbacks**: approximate a model boundary by a regular node mesh is difficult, and besides, the **local approximations used in FEM are not available here**.

Due to that, a **dynamic strategy** is adopted, as said in [phy2], so the system will be able to have **two states: transitory and stationary**, the last one referring to a minimum energy state, understanding the **energy gradient as the opposite of the force**:

$$f = -\nabla E(x,y,z) \implies \left(f_x = \frac{\partial E}{\partial x},\ f_y = \frac{\partial E}{\partial y},\ f_z = \frac{\partial E}{\partial z}\right);$$

then, the **dynamics of deformable models can be computed from the potential energy stored in the eastically deformed body**; and for its regularly spatial grids, the **deformation energy is defined as weighted matrix norm of the difference between the metric tensors of the deformed and the original model shapes**.

6.3.1.3 Boundary Element Method

Abbreviatted with BEM, this is also an interesting alternative to the standard FEM approach: while in FEM the computations are done on all the model's volume —the inner parts, indeed—, **in the BEM the process is involving only the model's surface**.

This method yields in an extreme advantage: the three-dimensional volume computation problem is turned now onto a **two-dimensional surface problem**; however, there are some too much important drawbacks for omitting them:

1. it works successfully **only for homogeneic models** —that is, models whose interior is composed by a homogeneous material—;
2. **topological changes are difficult to handle** because only surface is computed;

6.3.1.4 Mass-Spring Systems

Object of multiple research projects, these are the **simplest and most intuitive** of all deformable models. The reason is based on its starting point: instead of generating a space discretization from a PDE specification —such as in the previous methods—, here **the starting point is directly a discretized model**. And this discretized model, as the system name describes, **this model simply consists of mass points connected together by a network of massless springs**.

At any instant time t, the system state is defined by a **state vector for each mass point** $p_i, i = 1..n$:

- position x_i and velocity v_i
- its force f_i, computed due to *(i)* its spring connection with its neighbours, called **internal forces**, and *(ii)* **external forces** like gravity, friction, and so on. Due to that, the global sytem forces are

$$F_{global} = F_{internal} + F_{external}$$

```
1   function mass-springIteration( MassSpring MS,
2                                  TimeStep TS,
3                                  Set<Force> EF )
4   returns MassSpring
5       foreach Model M in MS do
6           M = applyExternalForces( M, EF );
7           M = applyMassSpringInternalForces( M );
8           M = doNumericalIntegrator( M, TS );
9       endforeach
10      return MS;
11  endfunction
```

PSEUDO code 6.3: *Algorithm for mass-spring system deformation*

▷ ***Newton mechanics provide the solution***

To allow deformations, it will be assumed that the springs that connect mass points together are elastic, so the **internal forces derived from this spring elasticity tend to maintain a fixed distance** —configured initially as the starting, or relax, distance— **between the connected particles**, so when a deformation is applied, the internal forces try to avoid this deformation.

So, the real system deformation is lead by **Newton mechanics**, concretely the Newton's second law, like a system composed by a single particle —see the first section of this chapter for more information—. It's really coherent because the **mass-spring system is considered a n-set of single particle systems, with constraints in terms of connection with other particles**.

Hence, for each particle there will be a Newton's second law ODE formula, according to [phy2, section 7.1]

$$\ddot{P}_i = \frac{F}{m_i} \implies \begin{pmatrix} \dot{x} \\ v \end{pmatrix} = \begin{pmatrix} v \\ F/m_i \end{pmatrix};$$

▷ ***Architecture of a mass-spring system***

Tipycally, a mass-spring system is configured such that the **mass points are regularly spaced in a lattice, connected together** along the twelve edges of each hexahedron by ***structural* springs**. Additionally, each *cell* —that means, empty space between eight mass points comprising a perfect cube; in other words, a kind of *voxel*— **masses on opposite corners** are also connected together by ***shear* springs**.

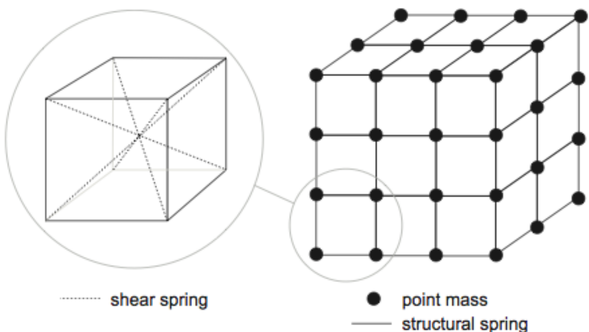


Figure 6.10: *A typical mass-spring system three-dimensional representation; source [phy5].*

▷ ***Longitudinal and shear force formulation according to Newton mechanics***

The springs are commonly modeled with **linearly elastic** behaviour; therefore, the acting force on a mass point i generated by a spring connection with another mass point j is the following:

$$f_i = k_s\big(|x_{ij}| - l_{ij}\big) \cdot \frac{x_{ij}}{|x_{ij}|}, \quad f_j = -f_i$$

where

- x_{ij} is the distance difference between the mass points i and j —$(x_j - x_i)$—,
- $|x_{ij}|$ is the euclidean distance in relation to the mass point positions x_i and x_j,
- k_s is the **spring's stiffness** —that is, the elasticity coefficient, arbitrary—,
- and l_{ij} is the initially fixed spring length.

Additionally, physical bodies are never perfectly elastic, they dissipate deformation energy due to the **damping effect**; so, a dampening coefficient is **used over the difference between the implied mass point velocities** to give a **viscoelastic behaviour** to the deformation:

$$f_i = k_d\big(v_j - v_i\big), \quad f_j = -f_i$$

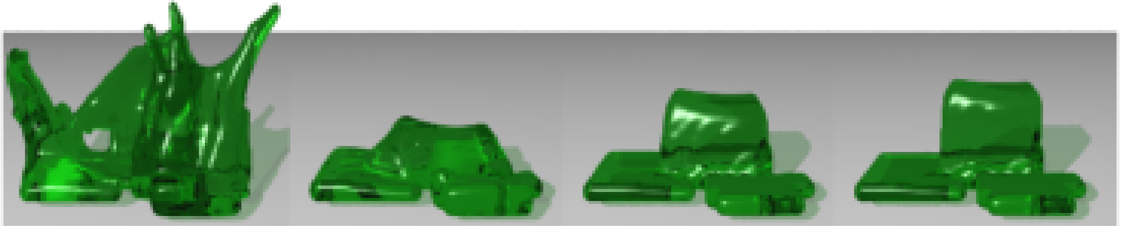


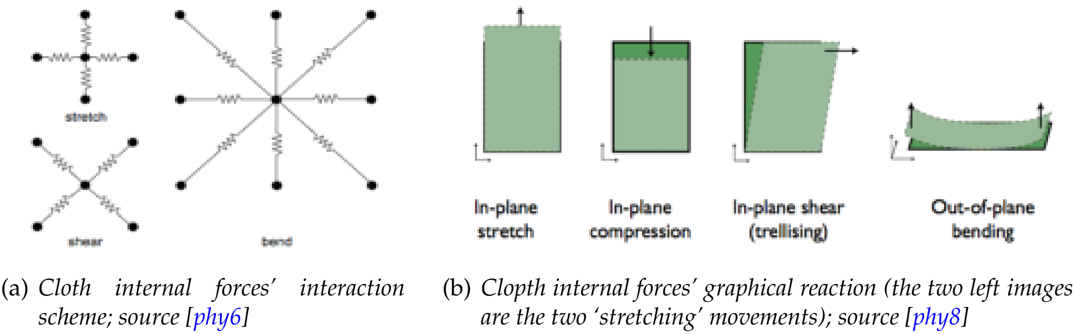
Figure 6.11: *Effect of viscosity on a physically-based elastic deformations. Each figure has ten times higher viscosity than its left neighbour; source [phyC].*

▷ *Clothing simulation*

One of the most interesting research contributions about mass-spring particle systems is the **cloth simulation**. Interestingly, Xavier Provot introduced in 1995 on [phy7] a **2D physically-based model for animating cloth objects directly derived from the elastically deformable mass-spring model**, and due to the unrealistic behaviour when a small region is victim of a huge number of high stresses, his method is also **inspired on dynamic inverse procedures**.

While a great amount of proposals, variants, improvements and algorithm accelerations have been featured since the Provot approach, they won't be discussed nor explained. For more information, be referred directly to [phy9], a good **survey on cloth simulation** by Yongjoon Lee.

Basically, the Provot cloth model is based on a regular 2D grid of mass points, each one connected with its neighbours with elastic springs; that is, a **2D mass-spring model, and even maintaining the same Newton-mechanics-based internal force**. Additionally, he adds, apart from the longitudinal internal force activation, here called **stretching**, two more forces: **shear** and **bending**, that can be seen in the next figure in a self-explanatory interaction scheme with graphical reaction.



(a) *Cloth internal forces' interaction scheme; source [phy6]*

(b) *Clopth internal forces' graphical reaction (the two left images are the two 'stretching' movements); source [phy8]*

Figure 6.12: *Here's the scheme of the three different internal forces for the 2D cloth mass-spring simulation model, including the connection affecting and a detailed visualization of the different force effects.*

So, by executing the mass-spring generic iterative deformation animation, some cloth pieces can be deformed by **applying external forces and fixing some mass points**, as features the next figure:

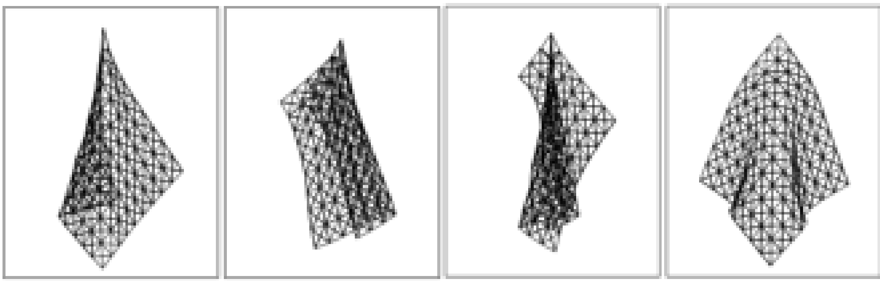


Figure 6.13: *A cloth piece with four different hanging —relaxing— postures after a real-time simulation steps by shaking the cloth and sliding it over obstacles before allowing it to rest; source [phyD].*

6.3.2 Mesh Free Methods

6.3.2.1 What are Mesh Free Methods

The Mesh Free Methods enter in clear contrast with the previous Mesh Based Methods; **while all the Mesh Based ones yield over a explicit connectivity and a *surface* boundary, the Mesh Free are more similar to point-based animation**. Indeed, the point-based physically-based animation is found here —see the first section of the previous chapter for a point-based geometric animation explanation—.

In this subsection a **rough explanation of some of the most famous and commonly used Mesh Free Methods** will be able to be found; however, if more detail is desired, or due to bibliographical references, please read the survey at [phy5].

6.3.2.2 Smoothed Particle Hydrodynamics —SPH—

Abbreviated with the SPH letters, this is a technique where **discrete and smoothed particles are used to calculate approximate values of needed physical quantities** and their spatial derivatives. In essence, a function A is, as said in [phy5], is interpolated at a position $x \in \mathbb{R}^3$ from its neighbors by using a summation interpolant:

$$A(x) \;=\; \sum_j m_j \cdot \left(\frac{A(x_j)}{\rho_j}\right) \cdot W(r,h),$$

where the interpolation parameters are:

- m_j and ρ_j are the particle p_j's mass and density respectively;
- $W(r,h)$ is a smoothing kernel with the following properties —or restrictions—:

$$\int W(r,h)dr = 1, \qquad \lim_{h\to 0} W(x,h) = \delta(x),$$

  being $r = |x - x_j|$ and $\delta(x)$ the Dirac function.

A finite differences methodology, or even a node grid, is not necessary. Indeed, this method is based on assigning an **initial density to each particle**, and then, the continuity equation —mass conservation— is then used for computing the density variation through the timeline:

$$\dot{\rho}_i \;=\; -\rho \nabla v_i,$$

where ∇v_i is called *divergence of particle i's velocity*, and can be approximated to

$$\nabla v_i \approx \frac{1}{\rho_i} \sum_{j \neq i} m_j (v_j - v_i) \cdot \nabla W(r,h);$$

now the equations of motion can be solved by deriving forces, after finding the relation between the density variation and the resulting internal force. Here's an example of SPH applied to an elastic solid melting to a fluid.

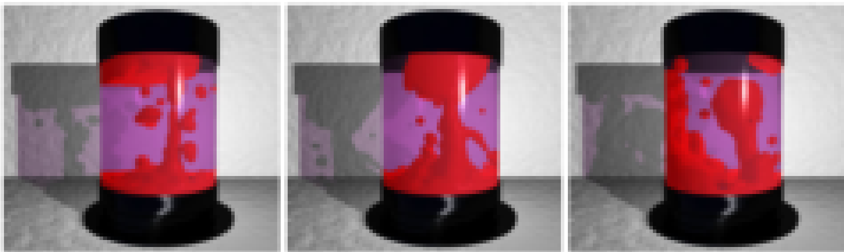


Figure 6.14: *Interface tension force* SPH*-based simulation between fluids of different* (i) *polarity,* (ii) *temperature diffusion and* (iii) *buoyancy; source [phy5].*

6.3.2.3 Meshless Deformations based on Shape matching

Proposed by Müller et al in 2005, this is a meshless method to animate deformable objects in an iterative algorithm such that, **at every time step, the original pointset configuration is fitted to the actual point cloud** using shape matching techniques for point clouds with correspondence.

The nodes of a volumetric mesh are treated as **mass points and are animated as a simple particle system with no connectivity** —only the shape matching and point correspondence techniques—. This is a very **suitable method for original shape recovering after a heavy deformation**.

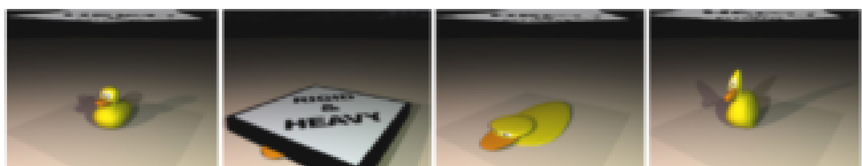


Figure 6.15: *Stable original shape recovering after a —literally— rigid and heavy deformation; [phy5].*

6.3.3 The Hybrid Approaches

6.3.3.1 Overview of the hybrid simulation approaches

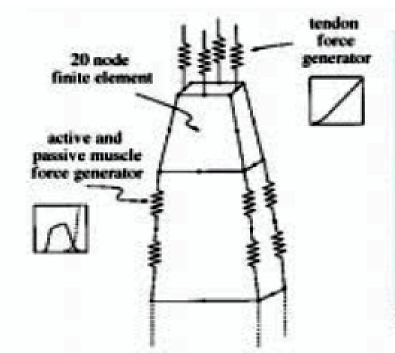
In the previous chapter, deformation algorithms based on geometric transformations have been treated, instead of physically-accomplishing laws. However, **there are some approaches that are really joining these two deformation methodologies: by using a bridge layer, a 3D is animated with geometric and physically-based deformation simulations**. Thus, in this section a few methods will be featured and explained —there have been another proposals in the recent years but they will be omitted—.

6.3.3.2 Pump it Up

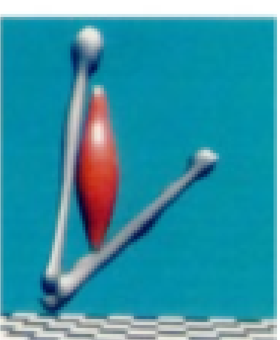
Proposed by Chen and Zeltzer, it's a hybrid approach focused on **tissue deformation**:

- a **finite element method** —FEM— analysis is performed on a **prismatic bounding box** embedding the muscle —or generic 3D object indeed—;
- the FEM resulting deformation is mapped onto the object surface mesh following a **Free-Form Deformation** —FFD— principle.

The object will be assumed to be composed by an **homogeneous, incomprenssible, linear isotropic material**, so the contraction force reaction is simulated by applying **bio-mechanically-based processes** on the mesh. The only problem with this method is that is **too expensive to be considered for interactive** applications.



(a) *Schematic hybrid model based on* FEM *mesh and* FFD *transformation; [phyE]*



(b) *Muscle contraction example (relaxed in left figure, shorten in right one); source [phyE]*

Figure 6.16: *The Chen and Zeltzer finite element mesh and contraction approach.*

6.3.3.3 Mass-Spring with Volume Conservation

Promayon and Baconnier features a previous proposal by themselves in [phyF, section 1.4] about the **volume conservation constraint** on physically-based deformations. The control of the volume is **necessary** in order to **simulate the incompressibility of 3D objects** —like some human organs— as well as to control the volume variation of other scene objects, everything **in run-time**.

The main advantage of this proposed method is that **the volume can be preserved constant during deformation in real-time without using an iterative process**, in contrast to almost all other Lagrangian approaches.

So, the idea behind this method is to consider a 3D object composed by a **triangular mesh with n points $p_1, \ldots, p_n$, that will be used as a particle system**. Let:

- V_0 the initial volume, also called the relaxing volume;
- and $V(p_1, \ldots, p_n)$ a volume function that describes the actual volume parameterized by the particle system positions;

If, in a deformation step, the volume is deformed, this method is able to **find the particle-respective correct displacements to maintain the correct volume in the current $p_1, \ldots, p_n$ system positioning**, by solving the following equation system:

$$\begin{cases} p_{1_{corrected}} &= p_1 + \lambda \left(\dfrac{\partial V(p_1, \ldots, p_n)}{\partial p_1} \right) \\ \vdots & \\ p_{n_{corrected}} &= p_1 + \lambda \left(\dfrac{\partial V(p_1, \ldots, p_n)}{\partial p_n} \right) \\ V(p_1, \ldots, p_n) &= V_0 \end{cases}$$

6.4 Physical Simulation based on Eulerian POV descriptions

6.4.1 Overview

This section, as mentioned before, will **not be an exhaustive reference for Eulerian techniques**; nevertheless, this simulation descriptions deserves a special mention because it's a different point of view, and it is really being used for physically-based animation.

The main drawback in relation to the Lagrangian methodologies is that **the boundaries of the object are not being explicitly defined because the point set is stationary**, so it can be understood as a heavy world of voxels. But, **this kind of representation makes the Eulerian approach ideal for simulating fluids**, since a fluid doesn't have explicit boundaries.

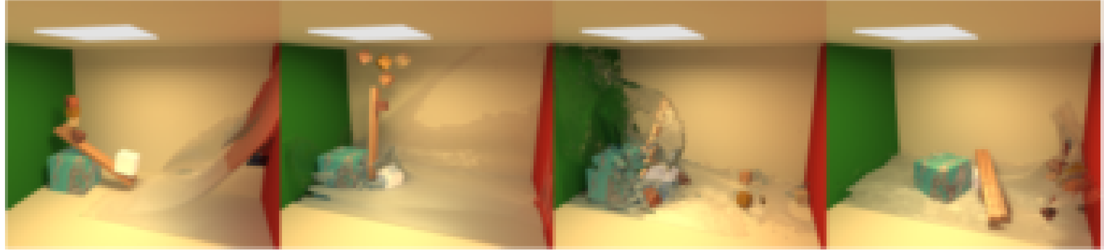


Figure 6.17: *A silver block catapulting some wooden blocks into an oncoming 'wall' of water; source [phy5].*

6.4.2 Fluid Simulation

6.4.2.1 The Navier-Stokes equations for fluids

The Eulerian approach for simulating fluids —in CG— was featured and become popular due to the Foster and Metaxas paper series between 1996 and 1997 —see [phy5, section 6] for more bibliographical references—. Their proposal **solves the Navier-Stokes fluid equations on a regular voxel grid and uses finite difference formulation** for all the equations in a single step.

So, as a reminder, there are two **Navier-Stokes equations**, representing, respectively, the mass conservation and the incomprensible fluid momentum:

$$\begin{cases} \text{(eq1)} & \rightarrow & \nabla u & = & 0 \\ \text{(eq2)} & \rightarrow & u_t & = & -(u \cdot \nabla)u + \nabla(vu) - \nabla p + f; \end{cases}$$

where the several variables are the following concepts:

- u_t is the time-derivative vector field for the **fluid velocity**;
- p is the **scalar pressure** —assumed a constant density, implicitly absorbed into the pressure term for simplicity—;
- f is the **body force** vector field per mass unit;

then, in this Eulerian formulation the **results are stored in a grid of cells** —similar to a grid of voxels—. It's important to be noticed that the position x is not computed nor stored, since **the grid positions remain fixed**.

6.4.2.2 The yielding idea of display and computation

Based on Foster and Metaxas approaches, **a liquid is** displayed as a height field, or in other words, **a massless particle set**.

These massless particles are quite different from the material coordinate particles from the Lagrangian simulation because of the *path tracking*: here, **the particles are passively moved into the fluid's velocity field, and their velocity is interpolated from the grid**[33].

6.4.2.3 A small summary of variants and proposals

It has been said that the Eulerian approach is used in fluid simulation. It's true, but there's a **commonly used analogy with some non-fluid materials**: the multi-particle materials, like *sand*, *smoke* or even *viscoelastic materials*.

Due to this, some variants and proposals have been proposed in the recent years; a little overview of some of them is given now —for a more detailed explanation and bibliographical references, please go to [phy5, section 6]—:

- **smoke** is simulated by Foster and Metaxas by defining a **temperature field** on the grid, and using that temperature for defining a floating thermal nature and using it, the turbulence motion is generated; Stam, on the other hand, defines smoke focusing their approach in a grid subdivision and defining a **scalar density field** that can be used by process quantities of smoke within the same environment —grid—.

- Zhu and Briudson's approach takes **sand** modeled as a fluid by using Lagrangian simulation for moving the particles and Eulerian simulation for interpolating velocities.

Figure 6.18: *Sand-based bunny model deformation; source [phy5].*

[33] As a vast alternative explanation, and not-much-exact overview but more clear, the moving execution is done by a *stationary cell swapping* between massless particles, or between massless particles and empty cells.

- Goktekin et al generates **viscoelastic materials** —materials that have both solid and liquid properties at the same time, which behave in a similar way to the honey (but not exactly)— by adding elastic terms to the basic Navier-Stokes equations.

- Suárez and Susin developed in [phyB] —a reference is needed because this approach has not been included [phy5]— another hybrid development for fluid simulation composed by two layers: an Eulerian-based underlayer and a Lagrangian-based *surface* layer for gigantic fluids where the more complex phenomena occur on the surface. These two layers are based, respectively, on the MAC —*Marker and Cell*— method, not explained here, and the Lagrangian SPH —*Smoothed Particles Hydrodynamic*—, in more detail on section on page 68.

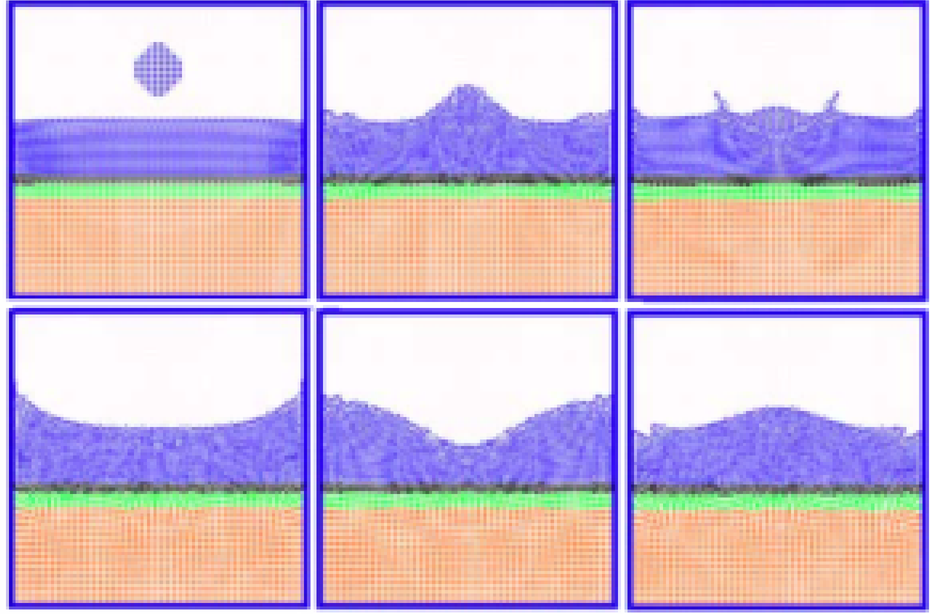


Figure 6.19: *A liquid drop falling into a liquid pool. In the six-figure frameset can be clearly noticed the three layers of the system: the Lagrangian layer* (blue)*, the bridge connection layer* (green) *and the Eulerian one* (red)*; source [phyB] (edited).*

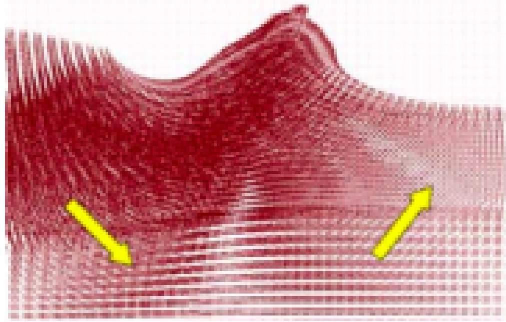


Figure 6.20: *Here is the velocity field for a certain instant time of the above frameset. Notice the communication between the Lagrangian and Eulerian layers by the connection stage; source [phyB].*

CHAPTER 7

Topological Changes on Polygonal Meshes

7.1 Mesh Cutting or Mesh Splitting

7.1.1 Introduction

As said in [cut1], there is an **intimate connection between the Lagrangian simulation model of a deformable solid and its discretization**; because of that, research in Lagrangian simulation can be classified into three categories: *(i)* **mesh generation** —omitted in this thesis—, *(ii)* **mesh simulation** —the main goal of this project indeed—, and finally, *(iii)* **mesh alteration** while its simulation. This section is about this last category.

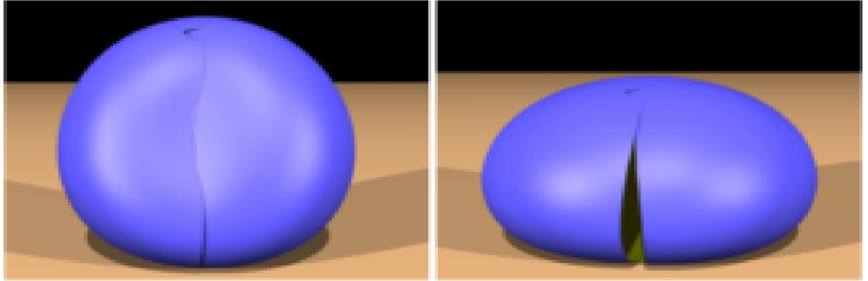


Figure 7.1: *Frameset for a fractured spherical shell; source [cut1].*

While some approaches have been features in the last decades —with the virtual environment appearance overcoat—, there are **three main approaches** for Lagrangian mesh material splitting:

1. **element cracking with connectivity preservation** by arbitrary stretching;
2. **mesh splitting** along element boundaries;
3. **continuuous remeshing** based on the dissection zones;

and the first two can, a priori, produce **visual artifacts** on the volumetric mesh.

7.1.2 Motivations and applications

7.1.2.1 Surgery simulation. Constraints and requirements

As it can be decuded from section 2.1.3, on page 12, **surgical simulation has become a popular application in computer graphics researching**, and within this field, one of the most essential parts is the **cutting of soft, deformable tissues**. So, since volumetric meshes —tetrahedrons or cubes— are a popular representation for volumetric objects, **methods to impose topological changes on this kind of meshes are required**.

The **mesh cutting**, also called **mesh splitting**, is the computer graphics methodology for cutting a volumetrical mesh, and it's a **non-trivial problem, because it involves topological changes**; due to that, **two extremal constraints** are required:

- the cutting process should **not create badly shaped elements**; if it's not guaranteed, numerical instabilities during deformation calculation will surely appear;
- **well incision approximation**; the user defined cut trajectory should be closely approximated for **realistic appearance**;

Almost proposed methods in the recent years are only concentrating on one of these restrictions, because the **two both are great research problems**. Besides, **most of the methods are not real-time** because of the complex model reconfiguration.

7.1.2.2 Another applications of mesh splitting

Although the surgery is the main goal of this research field, **many simulation problems include fractures as a critical component** —and a fracture has to be considered, of course, as a mesh topological change—:

- metallic structures' simulation: in harsh environments they are particularly susceptible to be corrosived or to be object of material failures;
- sculpting and modeling with computer aided design applications;
- tearing of textiles;
- ...

7.2 Mesh Dissection: Some Proposals

7.2.1 Spring Framework Tools: Cutting Interaction

Bruyns and Montgomery, in [cut3], features an interesting approach focused on **cutting geometric primitives depending on intersected polygon or polihedral type**. The final result is a **complete and generic rea-time framework for cutting and splitting single and multiple surface objects as well as hybrid and volumetric meshes using virtual tools with single and multiple cutting surfaces** —from scalpels to scissors—. The complete article is very versatile and offers real-time solutions for multiple cases; in this section it will be only mentioned in a generical overview. For a more detailed explanation please do a revision of [cut3].

7.2.1.1 The generic alrgorithm scheme

Basically, this method is based on a state *tags* for every intersected primitive, as follows the scheme shown in the figure 7.2: the framework will only apply a cutting action to those primitives **no longer in collision**.

```
1  while true do
2      foreach Primitive P in Model M do
3          if( isCollisionedByScalpel( P ) ) then
4              discern P.statetag
5                  is NO_COLLISION so
6                      P.statetag = START;
7                  is START so
8                      P.statetag = UPDATE;
9              enddiscern
10         else
11             if( P.statetag == UPDATE ) then
12                 P.statetag = MOVE;
13             endif
14         endif
15     endforeach
16     applyCuttingToMOVEPrimitives( M );
17 endwhile
```

PSEUDO code 7.1: *Interactive real-time cutting algorithm scheme.*

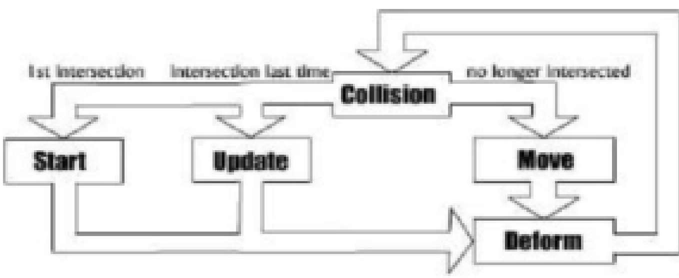


Figure 7.2: *The interactive deformation with cutting capability algorithm scheme from [cut3].*

7.2.1.2 Examples and applications

This framework is capable of cutting so much tuypes of 3D models: from multi-layered surface objects, as in the figure 7.3, simulating volume, until real volumetric models using tetrahedron primitives.

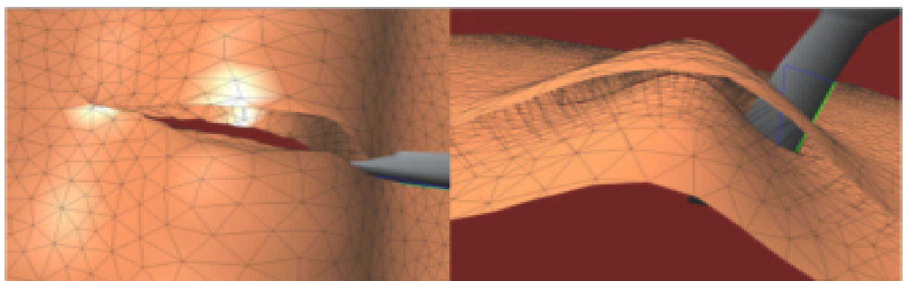


Figure 7.3: *Virtual Scalpel cutting a 2-layered surface, top (left) and side (right) views; source [cut3].*

7.2.2 Finite Element Methodologies

At [cut7] and [cut8] there are offered **two proposals** based two both on a **finite element method—FEM— deformable model**:

1. the first one is considering the cutting tool having a triangular shape, so a single **cutting range is delimited by a finite *triangle***. For more than one cuttings, the *triangle*-model collisions are stored as a set, and they are computed —displaced— sequentially over a volumetric model composed by tetrahedrons, with no node repositioning;

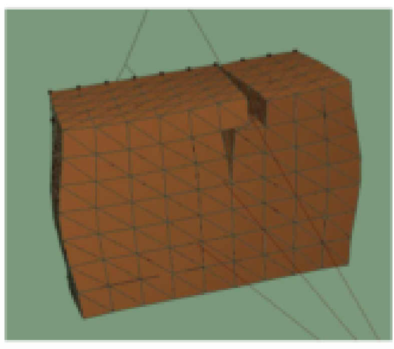
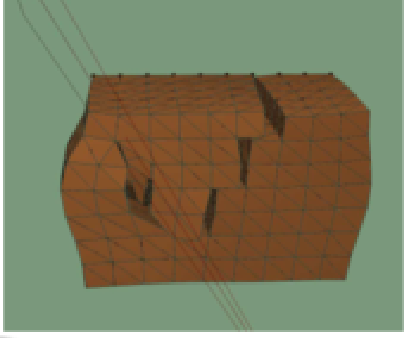



Figure 7.4: *The cutting plane sequence implementation over a cube model; notice the cutting sequence from left to right; source [cut7] (edited).*

2. the second one is, roughly speaking, about a **Delaunay retriangulation process** to the new subdivided zones; but this approach is only working on surface or multi-layered models;

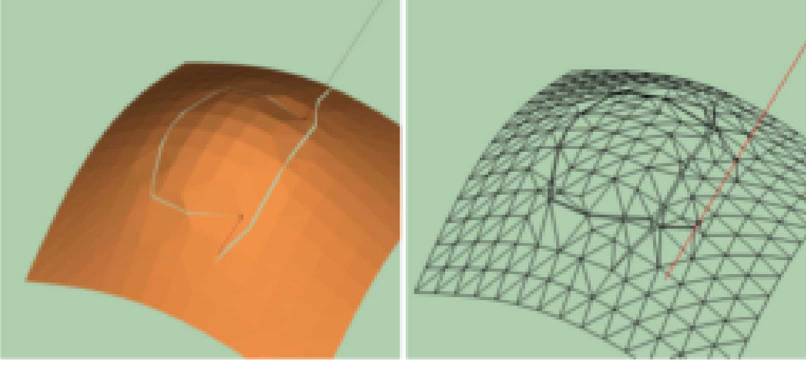
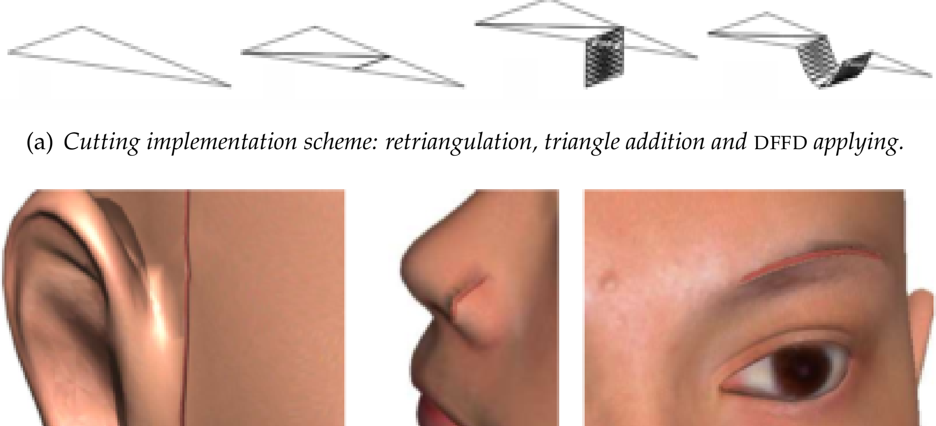


Figure 7.5: *A cut movement to a initially regularly triangulariez surface model with delaunay triangulation on the cut primitives; source [cut8].*

7.2.3 Discontinuous FFD approach

In [cut5], Sela, Schein & Elber offers a proposal based on ***Discontinuous Free Form Deformation* —FFD, a special non-physical-based deformation model—, applied to an added inner triangle set simulating the non-emptyness of the object**; in other words, a kind of strip that hides the empty object zone, better understood by watching the figure 7.6 on the next page:



(a) *Cutting implementation scheme: retriangulation, triangle addition and* DFFD *applying.*

(b) *Facial cutting examples with this* DFFD *additional deformation model.*

Figure 7.6: *The interactive mesh cutting implemented by* Sela, Schein & Elber *in [cut5].*

7.2.4 Hybrid Cutting

[cut2] features a **polygonal remeshing** of the model, proposed by Steinemann et al, where the **cutting line is approximated by a sharped line** that passes through the closer nodes; then a **node duplication** is executed and every triangle is subdivided according this sharped line;

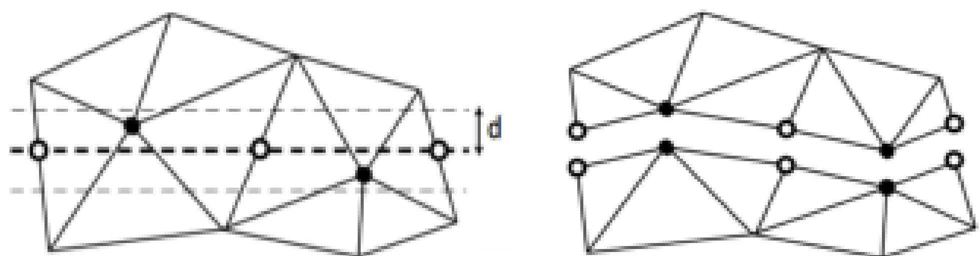
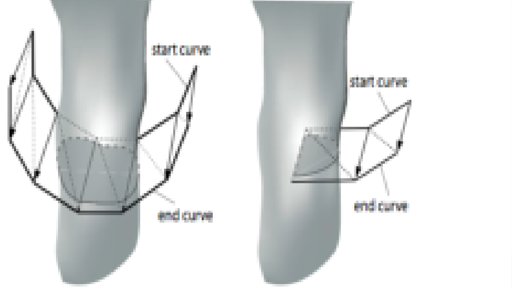
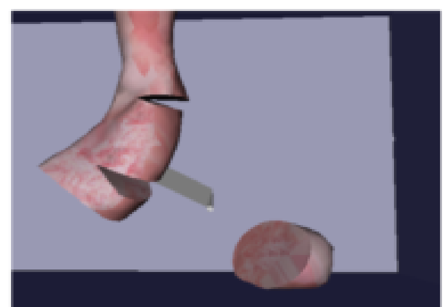
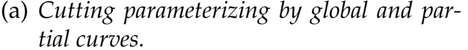


Figure 7.7: *The scheme for the primitive approximatting cutting process by the [cut2] method.*

This method allows, besides, to define cutting curves by its stationary points —globally thrpugh the model as long as only partially—, as can be seen in the following figure:





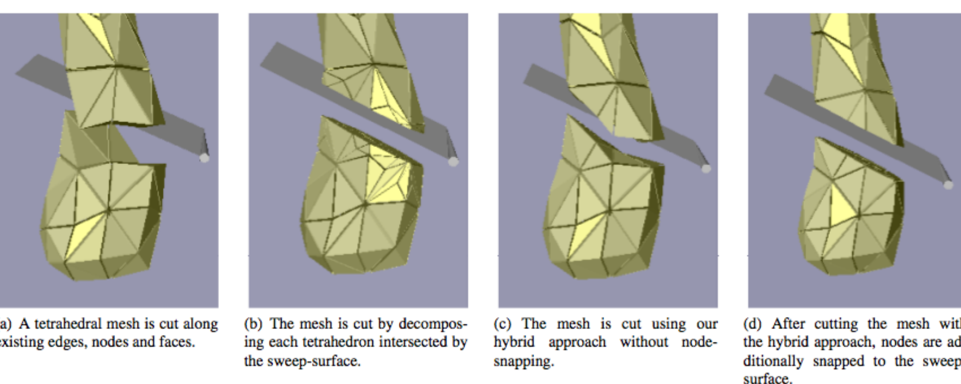


(a) *Cutting parameterizing by global and partial curves.*

(b) *Visual method example of partial cuts.*

Figure 7.8: *The interactive mesh cutting implemented by* Steinemann et al *in [cut2].*

Thiu method is called **hybrid** by the authors because it **combines the topological update by subdivision with adjustments of the existing topology**, avoiding the creation of small or degenerate tetrahedral elements.



(a) A tetrahedral mesh is cut along existing edges, nodes and faces.
(b) The mesh is cut by decomposing each tetrahedron intersected by the sweep-surface.
(c) The mesh is cut using our hybrid approach without node-snapping.
(d) After cutting the mesh with the hybrid approach, nodes are additionally snapped to the sweep-surface.

Figure 7.9: *A complete frameset example of a volumetric mesh cutting by the [cut2] method.*

7.2.5 Off-line virtual node algorithm based on replicas

There is in [cut1], by Molino et al, an **off-line** —it can result in hours of execution— algorithm that **allows so much complex processes like model cutting or even model fragmentation to volumetric tetrahedral models**, as it can be seen in the figures below:

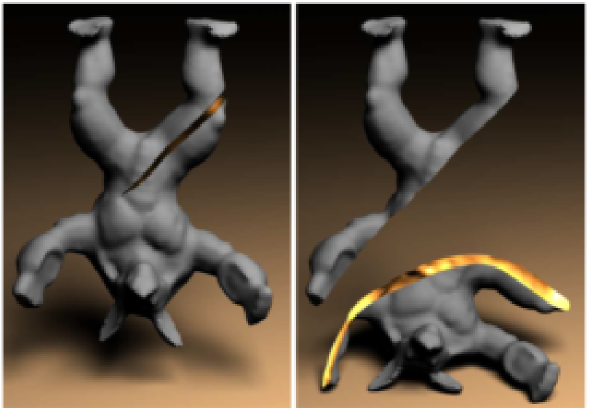


Figure 7.10: *High resolution cutting shell of the* 3D *volumetric complex* `armadillo` *model* (380K *tetrahedrons), by using the virtual node algorithm at [cut1].*

Briefly, the method works with cutting, and more generically with **3D model material fragmentation**, as follows; it's important to know that this method really **alleviates many shortcomings of traditional Lagrangian simulation techniques for meshes with changing topology**:

- the material within an element is **fragmented by creating several replicas of the element** and assigning a portion of real material to each replica;
- this replica-comprising elements contain both real material and empty regions — due to there are replicas that are empty space—. The missing material, so, is contained in another copy (or copies) of this element;

- finally, the method provides the DOF[34] required to simulate the partially or fully fragmented material in a fashion consistent with the embedded geometry.

Due to this versatile, complex and generic **algorithm focused on material fragmentation**, it can be possible to generate framesets like the following, with an exploding animation of the `armadillo` volumetric model:

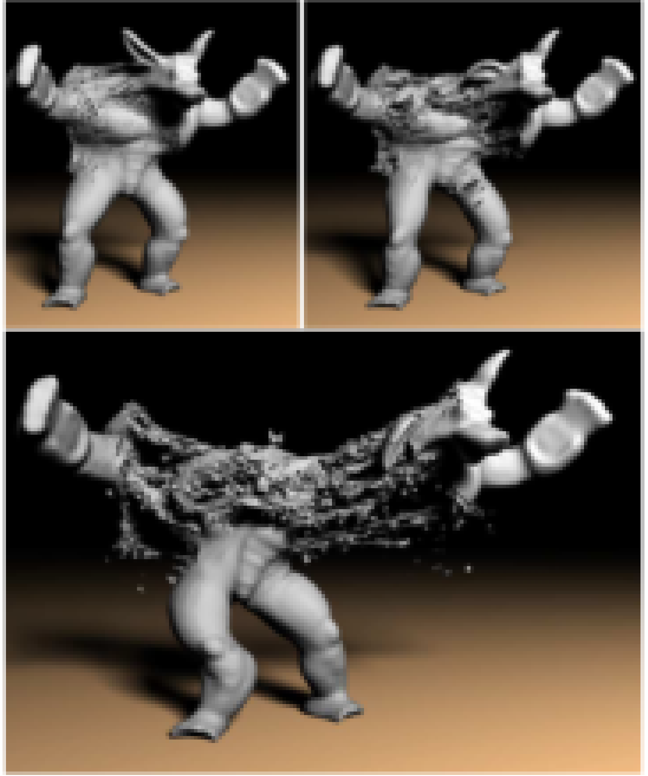


Figure 7.11: *Tearing a plastically fragmentation deformation to the `armadillo`'s high resolution volumetric mesh; source [cut1].*

[34] Acronym of *Degree of Freedom*.

PART III

Contributions and Results

CHAPTER 8

External Used Libraries and Employed Hardware

8.1 Mac Computers with MacOsX v10.5 “Leopard”

In the development of this Master Thesis codes and *software suite* composed by the two programs, MeshInspeQTor and ForceReaQTor, only **Mac Computers** have been used, with the **latest version of MacOsX operating system**, the 10.5 one, called *Leopard*.

Here are the following **specifications for the two used machines** —only the most importants in relation to this thesis focus—:

	iMac	**MacBook Pro**
CPU model	2.4GHz Intel Core 2 Duo	2.2GHz Intel Core 2 Duo
CPU RAM	2GB 667(Mhz) DDR2 SDRAM	2GB 667(Mhz) DDR2 SDRAM
Frontside bus freq.	800MHz bus	800MHz bus
Hard-disk capacity	500GB SATA@7200 rpm	160GB SATA@5400 rpm
Monitor Display	Built-in 20-inch (viewable) widescreen TFT active-matrix liquid crystal display.	MacBook Pro 15-inch new TFT LED technology widescreen display.
GPU model	ATI Radeon HD 2600 PRO	nVIDIA GeForce 8600M GT
GPU RAM	256MB GDDR3 SDRAM	128MB GDDR3 SDRAM

Table 8.1: *Used Mac computers’ specification table.*

8.2 Used Developing Platforms

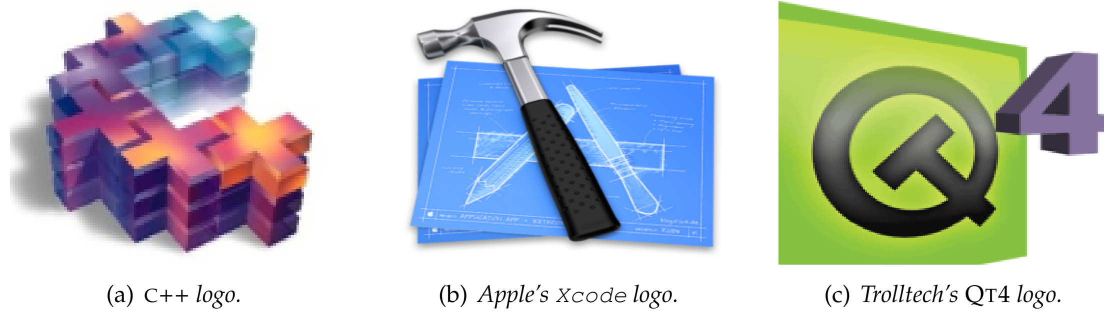
8.2.1 QT4.4 for GUI design with C++ as Native Language

The chosen programming language for all the software components is the structured-programming language named C++ because of its **modularity capabilities** and the possibility of the **generic programming**:

- **Every implemented algorithm and technique**, from the mentioned in the past chapter, has been isolated in an **independent module, with the minimum relation between other modules** (*e.g.*, the mesh class is not directly related to the voxelization class, while they have generic data input and output for relating them coherently).

- **Structure inheritance and overloading operators have also been used in the code design** for a more clear and intuitive operations with the complex data transparently, for avoiding unchained errors.

On the MacOsX systems, there is a complete IDE[35] called `Xcode`, that has been used for deploying the two application components of this master thesis developed *software suite*, MeshInspeQTor and ForceReaQTor.



(a) C++ *logo.* (b) *Apple's* `Xcode` *logo.* (c) *Trolltech's* QT4 *logo.*

Figure 8.1: *Mainly used tools for the thesis'* software suite *design and implementation.*

Also, for Graphical User Interface —GUI— design and implementation, the Trolltech's cross-platform QT library[36] (in its version 4.4) has been the chosen one, for its multiple platform compatibilities (including MacOs X, Linux and Windows). QT is a **complete development framework** for C++[37] that features:

- non-standard C++ extensions that produces native C++ code in precompilation;
- obviously, all the needed GUI components and resources;
- *thread* management for parallel computing;
- other non-used and non-GUI features, like SQL database access, XML parsing or NETWORK support, and others...

[35] Acronym of *Integrated Development Environment*, defines the software capable of design, implement, compile and execute applications. It normally consists of a source code editor, a compiler and/or interpreter, build automation tools, and (usually) a debugger. Sometimes a version control system and various tools are integrated to simplify the construction of a *Graphical User Interface* —GUI—.

[36] `http://trolltech.com/products/qt/`.

[37] Although there are existing bindings for other languages like C#, JAVA, Python, or PHP.

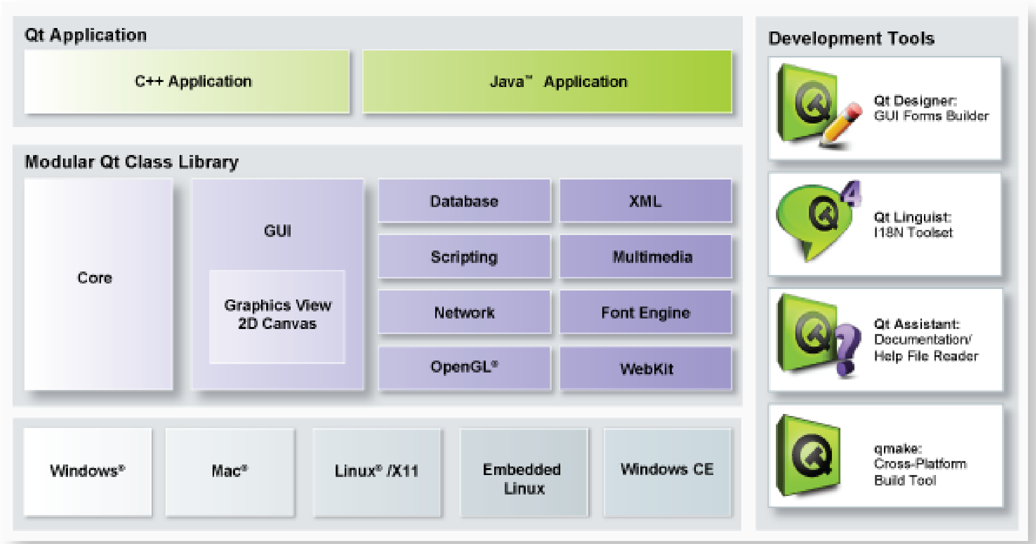


Figure 8.2: *The Trolltech's* QT4 *versatile capabilities and framework diagram.*

8.2.2 `Matlab` for testings

Parallel to the C++ complex *software suite* design and implementation, for some graphic techniques a `Matlab` previous testing application is developed, and its $R2007a$ version has been the used one.

Figure 8.3: *The Mathworks'* `Matlab` *mathematic environment logo.*

`Matlab`[38] is a powerful **numerical computing environment** with its own programming language, the called `m-code`. Created by `The MathWorks`, `Matlab` features, apart from other awesome characteristics:

- easy matrix manipulation[39] and data structs —no pointers!— management;
- plotting of functions and data;
- an optional toolbox interface with the `Maple` symbolic engine, allowing access to computer algebra capabilities.

Thus, some voxelization methods, or even FFD technique, have been **tested firstly on a selfmade `Matlab` applications**, because its versatile plotting capability permits an easy data structure visualization and besides, **potential math-programming errors like bad array pass-throughs or bad designed mathematic operations are completely avoided**.

[38] `http://www.mathworks.com/products/matlab/`.

[39] Really, `Matlab` is the acronym of *Matrix Laboratory*, and within it, the float matrix $\mathcal{M}_{m \times n} \in \mathbb{R}^2, \;\; i,j > 0 \in \mathbb{N}$ is a native data type, in a clear difference with other languages like textscc++ where the native data type is the scalar.

CHAPTER 9

Computer Graphics Programming

9.1 OPENGL as the primary graphics specification

9.1.1 A Little Brief on Computer Graphics using OPENGL

As it's said in [gpu1], OPENGL, the acronym of *Open Graphics Library*, is a **standard specification** developed in 1992 by Silicon Graphics Inc —SGI— and defines a **cross-language cross-platform API**[40] **for producing 2D and 3D graphics**. The mentioned interface permits, by calling simple primitives and setting some parameters and configurations, to draw complex three-dimensional scenes.

As a simple API overview, it's said that **OPENGL is a specification**; that implies that it's actually a **document that describes the must-fulfilled functions and procedures**. From this *spec*, hardware vendors[41] *have to* create *spec*-compatible implementations.

Figure 9.1: *The* OPENGL *logo by Silicon Graphics Inc.*

Hence, there exists so much vendor-supplied implementations of OpenGL —making use of graphics acceleration hardware for achieving a good specification compatibility— for **multiple systems: MacOs X, Windows, Linux**, many Unix platforms, and so on.

[40] Acronym of *Application Programming Interface*, is a set of declarations of procedures that external software, like operating systems, libraries or service provides to support requests made by computer programs.
[41] The greatest vendors are at this moment nVidia and ATI, with no doubt, since it's the GPU hardware instead of the CPU the executing chip.

9.1.2 API overview: The OPENGL graphical computing steps

9.1.2.1 Initial considerations; introduction

This section is **not** pretending to be a complete **tutorial** for OPENGL computer graphics programming; the aim of the following explanations is to give a brief of the computation style of this API, because **on the next sections, this knowledgments will be so necessary**. An important thing, however, is to notice that all the OPENGL methods always begin with the `gl` prefix.

The classical rendering action with OPENGL hasn't any complication, but if there's an **animation** wanted to be shown, now appears the problem: this process **requires a smooth sequence of *final* animation frames**[42], and due to this, the **double buffering** technique is available. The *drawing* occurs on the background buffer, and **once the scene is completed, it's brought to the front** —displayed on the monitor display—.

```
1  { this process is done forever }
2  while true do
3      resetBackgroundBuffers();
4      setCameraViewing();
5      { now the user point of view of the scene is
6        set: 'camera' position and orientation, and
7        vision perspective configuration }
8      foreach Object O in Scene S do
9          { animation + rendering steps }
10         animateObject( S.O );
11         drawObject( S.O );
12     endforeach;
13     swapBuffers();
14     { by swapping buffers, the scene displaying
15       is automatic }
16 endwhile
```

PSEUDO code 9.1: OPENGL *animation scene drawing steps.*

This thesis is about the line 10, `animateObject()`, so this line is eclipsed by all this document. Also the first and last commands, `resetBackgroundBuffers()` and `swapBuffers()` will be omitted. **This section, hence, will treat accurately the other two commands —lines 4 and 11 respectively—**: `setCameraViewing()` and the own drawing process `drawObject( Object O )`, based on [gpuE] and [gpuF] tecahings.

9.1.2.2 Drawing primitives

For the drawing process, OPENGL needs to **specify where start and where the primitive generation commands finish**. Since the geometric primitives, whatever they are, are com posed by **vertices**, this will be the geometric primitive coherent definition; so, **any primitive will be a sequence of 3D or 2D vertices**. Additionally, there are some **different primitive styles**, as the figure 9.2 shows. The constant must be declared at the primitive drawing starting command.

[42] A *final* frame is a totally generated and rendered image; *e.g.*, a sequence of two triangles is rendered by painting each triangle sequentially, and no animator wants an animation showing one and two triangles intermitently.

```
1     glBegin( GL_PRIMITIVE_STYLE );
2         glVertex3f( [...] ); // v0
3         glVertex3f( [...] ); // v1
4         glVertex3f( [...] ); // v2
5         [...]
6         glVertex3f( [...] ); // vN
7     glEnd();
```

CPP code 9.2: *Drawing primitives with* OPENGL.

In this generic code for OPENGL primitive drawing, the `glBegin()` parameter, that receives the name of `GL_PRIMITIVE_STYLE` can be one of the following constants:

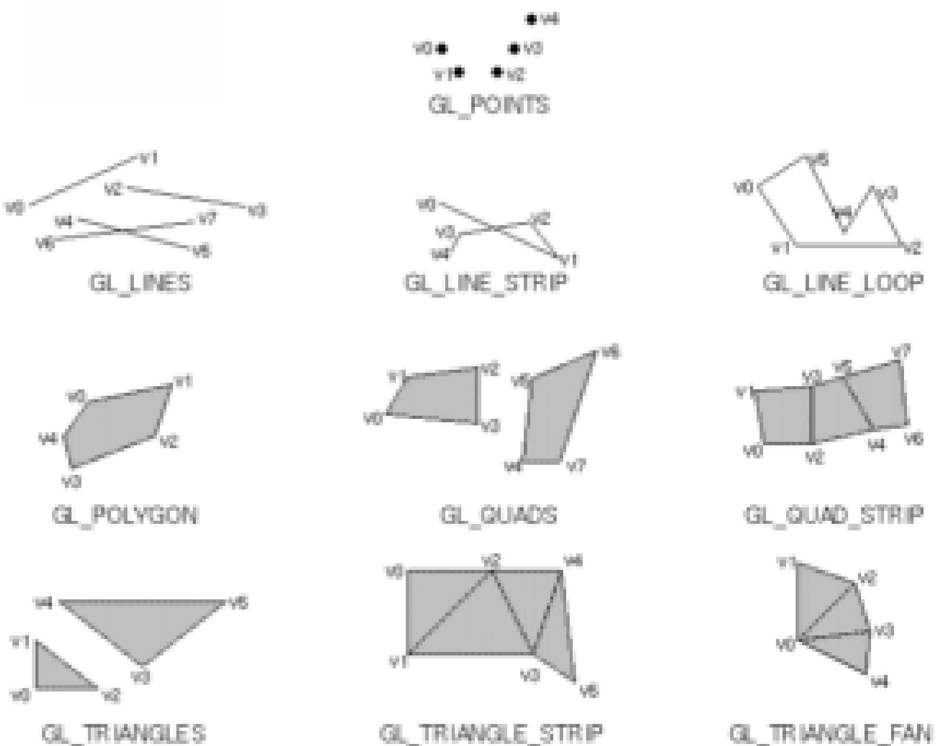


Figure 9.2: *The* OPENGL *primtive drawing styles; source [gpuE].*

Besides, there's the **possibility of integrating vertex color definition** with a vertex-coords previous command that specifies the RGB color codification —RGBα if transparencies are enabled—. OPENGL lighting effects are not telled here, but the **primitive colorizing due to the vertex colors** is done by a code like the following:

```
1     glBegin( GL_TRIANGLES );
2         glColor4f( 0.0, 0.0, 1.0, 1.0 );
3         glVertex3f( 0.0, 0.0, 0.0 );
4         glColor4f( 1.0, 0.0, 0.0, 1.0 );
5         glVertex3f( 3.0, 0.0, 0.0 );
6         glColor4f( 0.0, 0.0, 1.0, 1.0 );
7         glVertex3f( 1.5, 3.0, 0.0 );
8     glEnd();
```

CPP code 9.3: *A colorized triangle drawn with* OPENGL.

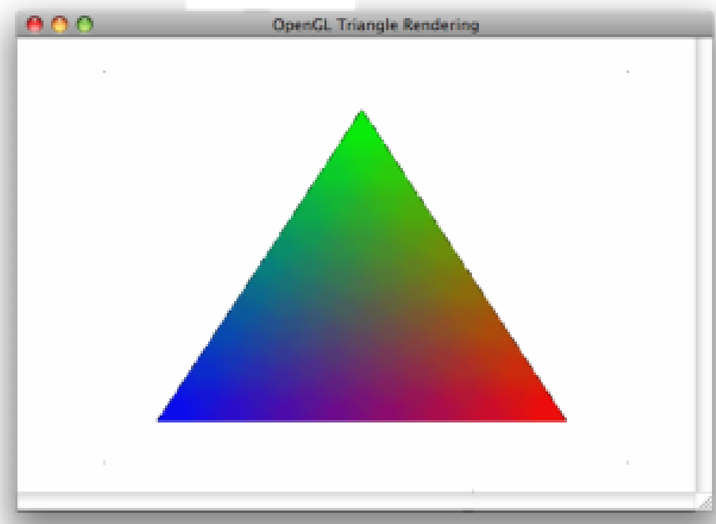
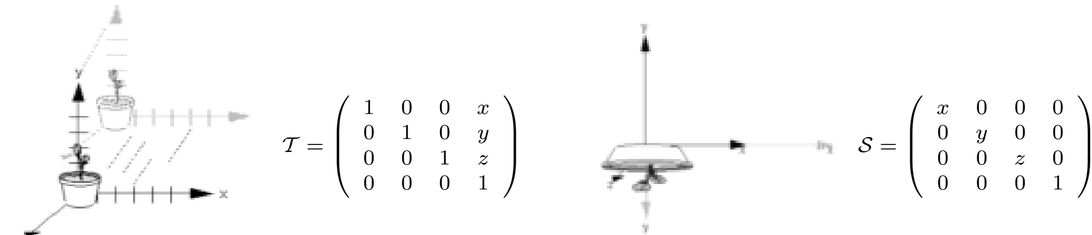


Figure 9.3: *The resulting rendered triangle from* CPP *code* 2.3.

It's not the only way to specificy primitives; there's also the **vertex buffers** and the **display lists**, but they won't be detailed here.

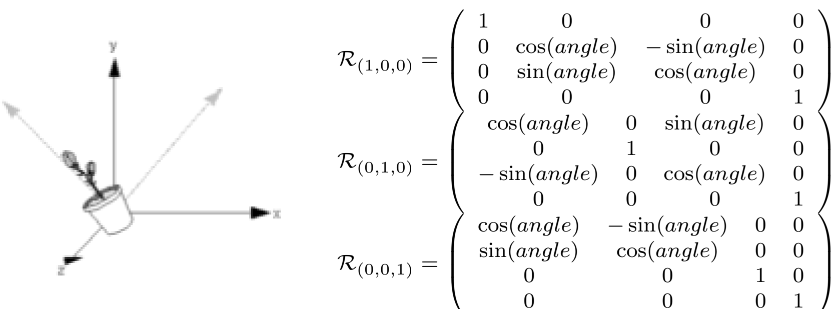
9.1.2.3 Vertex transformations; the matrix stack

The most simple animation of geometric primitives on OPENGL is based on **geometric 3D transformations**, concretely three of them: *translation*, *rotation* and *scaling*. Let's view on the following two figures the **OPENGL command correspondence with its geometric matrix transformation**:



$$\mathcal{T} = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathcal{S} = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) *The 'translation'* `glTranslatef( x, y, z ).` (b) *The 'scaling'* `glScalef( x, y, z ).`



$$\mathcal{R}_{(1,0,0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(angle) & -\sin(angle) & 0 \\ 0 & \sin(angle) & \cos(angle) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{R}_{(0,1,0)} = \begin{pmatrix} \cos(angle) & 0 & \sin(angle) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(angle) & 0 & \cos(angle) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathcal{R}_{(0,0,1)} = \begin{pmatrix} \cos(angle) & -\sin(angle) & 0 & 0 \\ \sin(angle) & \cos(angle) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(c) *The three Euler 'rotation' matrices, by* `glRotatef( angle, x, y, z ).`

Figure 9.4: *A scheme of the three basic geometric transformations: 'translation', 'scaling' and 'rotation', with the implicit correspondence between the transformation matrix and the* OPENGL *command.*

All of these **transformations are able to be concatenated simply by calling the desired transformations subsequently**, and, as it passes when you define the order of matrix transformation product items, **the order of applied transformations is also important in OPENGL**:

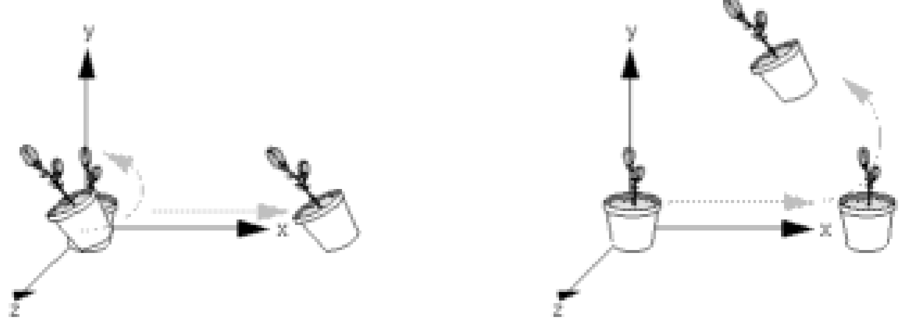


Figure 9.5: *The order of the multiplying applied geometric transformations is important: it's not the same a translation applied to a roation (left) that a rotation applied to a rotation (right); source [gpuE].*

Because of this, OPENGL features the **matrix stack** to avoid conflicts between set of transformation sets; *e.g.*, you want two objects translated by the same transformation by rotated by different rotations. Let's view the following code, and the resulting scheme:

```
1      glPushMatrix();
2          glTranslatef( 10.0, 10.0, 0.0 );
3          drawRedTriangle();
4          glPushMatrix();
5              glTranslatef( 20.0, 10.0, 0.0 );
6              /* this (20,10,0) translation is
7                 added to the previous (10,10,0) */
8              drawBlueTriangle();
9          glPopMatrix();
10     glPopMatrix();
11     /* no translation is applied to this triangle */
12     drawGreenTriangle();
```

CPP code 9.4: *An example of* OPENGL *matrix stack usage.*

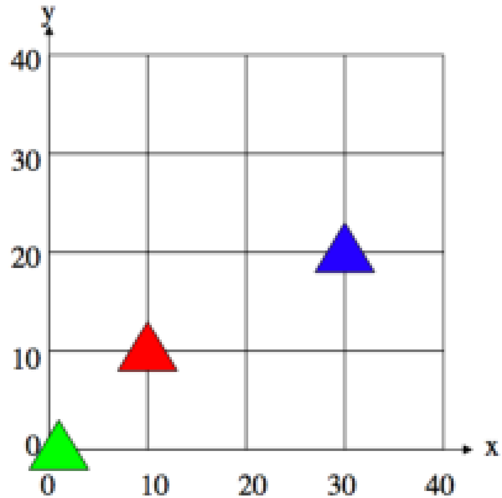


Figure 9.6: *The resulting scheme of the matrix stack* CPP *code* 2.4; *source [gpuE].*

9.1.2.4 Camera positioning and configuration; the viewport

In OPENGL, sa it's said on [gpuF], there are two important coordinate systems to be counted, due to their own importance:

1. the first is the **world coordinate system**, also called **object coordinate system**; this is used for defining the drawn scene —therefore, the coordinates used with the `glVertex3f` method—.

2. and the second one is the **eye coordinate system**; that means, the scene with a local coordinate system where the origin is the **center of the *camera*** —in other words, the scene point-of-view origin—.

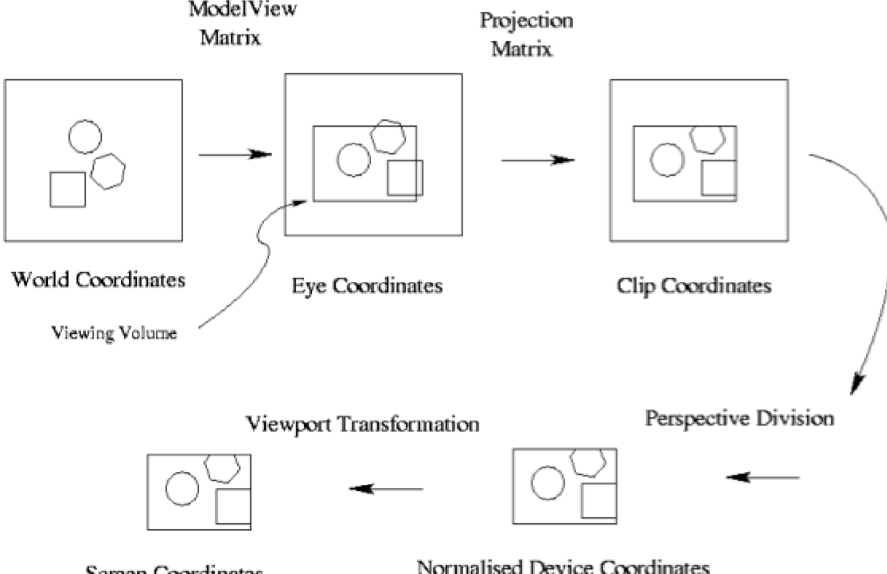


Figure 9.7: *The coordinate system transformations from scene specification to scene screen displaying; this figure is taken directly from [gpuF].*

▷ ***The Volume Viewing***

Is in the **eye** system where the **volume viewing** is defined. Briefly, a volume viewing is a ***box* that contains the scene that will be displayed**, and is defined in *eye coordinates* due to **the *box*'s comprising-planes are specified from the scene point-of-view**. And it's important to remark again that any object outside this *box* will be drawn but not shown[43].

Additionally, OPENGL allows the programmer to specify **two kinds of volume viewings projections** —or scene-containing boxes—:

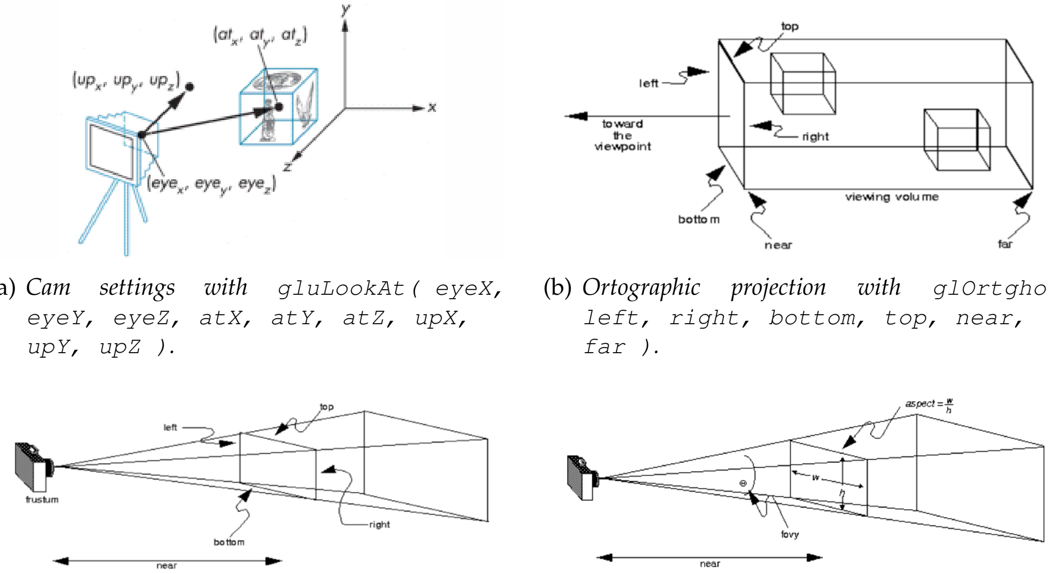
- a **ortographic** —or **axonometric**, or **parallel**— projection uses a **parallelepiped** for a constant-area volume box;

- a **perspective** projection will be defined with a **frustum**, or **truncated pyramid**[44];

[43] If there's only an object portion within the volume viewing, only that potion will be displayed; the rest will be *clipped* —see the second and third blocks on the figure 9.7—.
[44] This is an interestig approach for simulating the real *photocamera* viewing, including a little 3D space deformation due to the non-parallelepipedic nature of the volume viewing.

Additionally, the **camera —or the *eye*— must be positioned and oriented**, so it must be specified by **three parameters** $\in \mathbb{R}^3$:

1. its **position**, where the *eye* will be placed;
2. its **focusing direction**, tipically is equivalent to the geometric center of the volume viewing;
3. its **view up vector** —VUV—, or in other words, the vector to be considered the *up* direction[45];



(a) *Cam settings with* `gluLookAt( eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ ).`

(b) *Ortographic projection with* `glOrtgho( left, right, bottom, top, near, far ).`

(c) *Perspective projection with* `glFrustum( left, right, bottom, top, near, far ).`

(d) *Perspective —alt.— projection with* `gluPerspective( fovy, aspect, near, far ).`

Figure 9.8: *The two* OPENGL *available volume viewing specification and the camera positioning and orientation methods; figure sources from [gpuF].*

▷ *The Viewport*

So, there's also a last important technical part to be counted, the **viewport**, or **the display area where the scene will be drawed from the specified camera**. By default, the viewport is placed over all the display area, but **it's possible to define more than one viewport** on the same display —like in a 3D modeling software—. By doing this, there is the following OPENGL method:

```
glViewPort( lowerLeftCornerX, lowerLeftCornerY,
            viewportWidthInPixels, viewportHeightInPixels ).
```

[45] For a more clear explanation, if the VUV is the $(0,1,0)$ vector the viewing is equivalent to our standing vision of the world —since on computer graphics, the $\vec{\jmath}$ direction is floor-perpendicular due to a convention of understanding the third coordinate as a extension of the classical 2D coordinate system—, but if it's $(-1,0,0)$, the vision will be similar to see a vertical photograph in an horizontal orientation.

9.1.3 The Standard Graphics Pipeline

In a more detailed vision, OPENGL is considered a **state machine**[46], and as it's said in [gpu1], serves **two main purposes**:

1. to gather in a single API the complexities of interfacing multiple (and different) graphics accelerator hardwares;

2. to hide the differing capabilities of graphics hardware by specifiyinc a standard (using software emulation if needed);

The **basic operation** of OPENGL's state machine, that receives the name of **Standard Graphics Pipeline**, is **to receive primitives —points, lines and polygons— and to convert them into pixels shown in the screen display**, in the following way —without entering in precise details—:

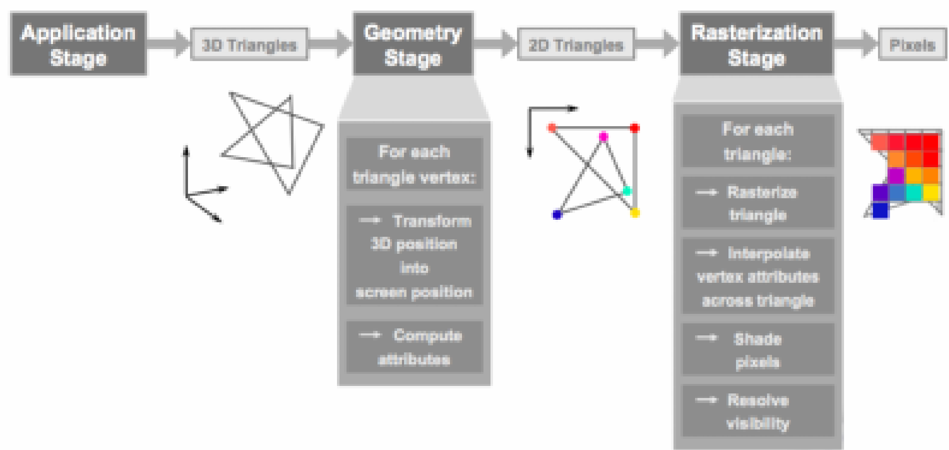


Figure 9.9: *Overview of* OPENGL *method for display, on screen displays, geometric primitives —in the figure they have been markes as triangles due to the most common case— as pixels; source [gpu3].*

Nevertheless, there are a lot of parameters for setting the receivement of the primitives, or configuring how this pipeline processes these primitives. **Each stage of the pipeline performed a fixed function and is configurable**, and at the **next page's more detailed pipeline scheme** this is better explained, supported by a **little brief, inspired by [gpu5], of the most important —ordered by number— pipeline steps**; the rest of the steps are omitted because their detailed description is not necessary:

(1) the geometric **primitives are send** from CPU to OPENGL state machine[47];

(2) the **vertex processing** stage, also called T&L —acronym of *Transformation & Lighting*—, because now is when **each primitive vertex is treated independently** for computing its final 3D position due to the possible modelling and projetion matrices, and its correct illumination is also applied here;

[46] That is, a model of behavior composed of a finite number of states, transitions between those states, and actions. It's also called *Finite State Machine* —FSM—.

[47] Nowadays, every graphics accelerator hardware —*Graphics Processor Unit*, or GPU— contains the OPENGL standard graphics pipeline implemented; so, since the pipeline is hardware-executed, this step is equivalent to send the primitives from CPU to GPU.

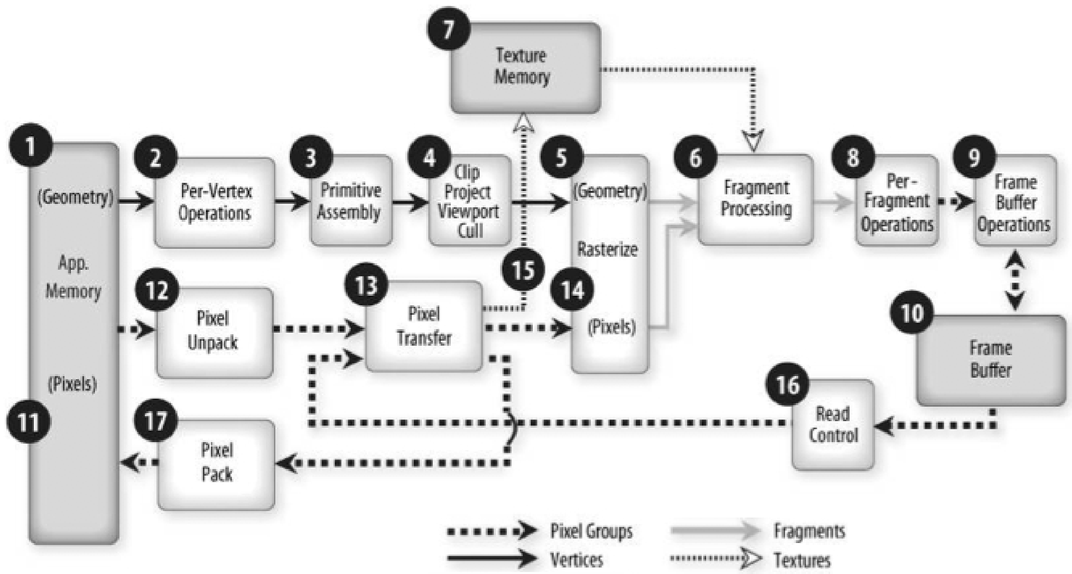


Figure 9.10: *Detailed scheme of the* OPENGL*'s Standard Graphics Pipeline; source [gpu4].*

(3) in **primitive assembly** the vertex data is collected into **complete primitives**;

(4) the **clipping** stage, when **the visible part of the primitive is decided** in relation to the scene viewing volume;

(5) the **rasterization** stage manages **decompose primitives into smaller units corresponding to pixels** in the destination frame buffer —data buffer that contains the final screen display drawing—.

(8) **per-fragment processing**, where fragments are submitted to a set of fairly simple test operations to decide which fragment —potential pixel— stays as pixel in the frame buffer[48];

(14) here is the **rasterization** process of the final screen pixel stream, read from application memory; only a few steps more and the data will be shown on ths screen[49].

9.1.4 The OPENGL Extensions. The GLEW Library

The OPENGL *spec* allows individual vendors to provide **additional functionalities for relaxing restrictions or adding new capabilities to OPENGL**. These functionalities are known as **extensions**, and each vendor has an alphabetic abbreviation used for naming the necessary new functions and constants (*e.g.*, NV for nVidia or ATI for ATI).

However, if more than one vendor implements the same extension, its abbreviation is transformed to EXT —*generic extension* abbreviation—, and besides, if the OPENGL Architecture Review Board accepts the extension for being a accepted extension in OPENGL standard *spec*, the abbreviation will be ARB.

[48] Obviously, passing from 3D space to 2D projection, there will be, a priori, more than one candidate fragment to place the same frame buffer pixel.

[49] However, only a frame it's shown on the screen; for displaying a moving image, this process must be executed a lot of times per second —getting the *Frames Per Second* FPS measure—, with little changes for each frame to get a smooth animation.

So, since each vendor is different, and even there are multiple graphics accelerator models for each vendor, **a availability test is required before using an extension**; each system has its own availability mechanism, but **cross- platform external libraries like GLEW[50] —acronym of *GL Extension Wrapper*— simplify the process**.

9.2 The programmable graphics technology

9.2.1 The Shading Languages

From the year 2002, there appeared some proposals for a new kind of programming languages: the **real-time shading languages**; that is, **a language used to determine the final surface properties of an object or image**, or more technically said, **a language that provides code overlapped within graphics standard pipeline**[51].

From the three proposals, the nVidia's CG —acronym of *C for Graphics*—, the Microsoft's HLSL —*DirectX High-Level Shading Language*—, and the OPENGL ARB's GLSL —*OpenGL Shading Language*—, the last is the chosen one, due to:

- appeared in 2003, it's part of the OPENGL 2.0 API, so it's independent of Microsoft's DirectX[52] specification;
- it's very similar to nVidia's CG in syntax, though different in semantics;
- it's included to an application by including the extensions `GL_ARB_vertex_shader` and `GL_ARB_fragment_shader`.

9.2.2 A New Programming Paradigm

9.2.2.1 The GPU's Vertex and Fragment Processors

For achieving this **new programming paradigm focused on specific graphics data management**, the hardware vendors had to build specific GPU architecture that were supporting that. Thus, the **vertex processors** and the **fragment processors** were born, and **there are multiple processors for each type within a single GPU**:

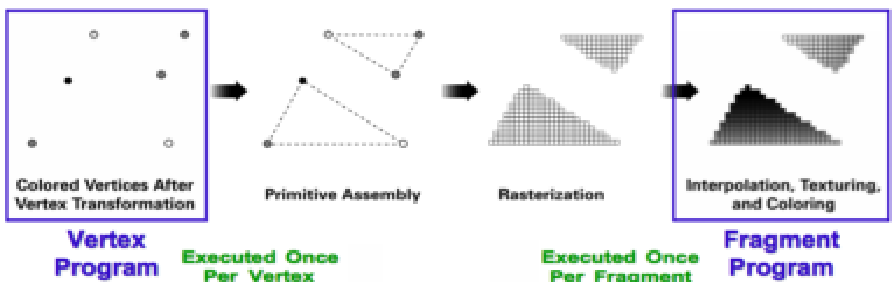


Figure 9.11: *The GPU's vertex and fragment processor managment tasks; source [gpu3].*

[50] http://glew.sourceforge.net/

[51] The name of *shading language* is received because the initial aim of this new programming paradigm was the real-time shade and shadow generation.

[52] It's a collection of APIs for handling miultimedia tasks on Microsoft platforms. Direct3D is the concrete API that can be compared to OPENGL with respect to concepts and aims.

(1) the **vertex processor** is the responsible of computing the **final 3D vertex position**, with the peculiarity of **knowing nothing about other vertices** —thus, this processor is found before the primitive assembler—;

(2) on the other side, the **fragment processor** has the mission of defining **each fragment —potential pixel— final color**, from illumination, material and texture properties;

obviously, **there's no direct vertex-fragment computing proportion**, because a single triangle —3 vertices— can be placed over a lot of fragments. Nevertheless, a **vertex shader can pass data to all its derived fragment shaders** by the **varying variables**.

9.2.2.2 The New Programmable Graphics Pipeline

So, the old **per-vertex operations** and **fragment processing** stages, from the standard graphics pipeline —figure 9.10—, now have been replaced by the **vertex processor** and **fragmenrt processors** respectively, as can be seen in the next figure:

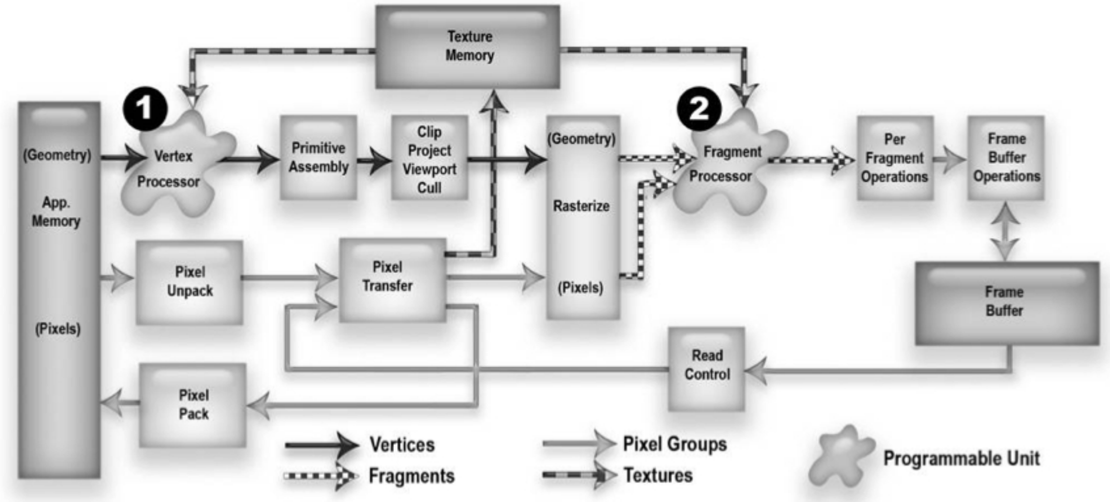


Figure 9.12: *Detailed scheme of the Programmable Graphics Pipeline; source [gpu4].*

The GLSL shader programs —vertex and fragment ones— loading process is possible by a special OPENGL functions for sending to GPU:

- the shader codes to the adient processors;
- the shader input data structures, common for **every** shader execution, called **uniform variables**;

however, as it's done for the OPENGL section, there won't be an API tutorial, but an overview of these kind of technologies.

9.2.2.3 Geometry Shaders

In the last year, a new kind of GPU processor unit has been introduced with **Shader Model 4.0** of Microsoft DirectX 10 specification: the **geometry processor**, able to execute the new **geometry shaders**. Although it's really a DirectX approach, OPENGL is able to use it by the extension, though it will likely be incorporated into the upcoming version 3.0 standard *spec*.

Geometry shaders are executed after vertex shaders, and their input is a whole primitive —that is, vertex-relation information, and possibly adjacency information too—. These shaders can emit zero or more primitives as the output, and they will be rasterized —*ergo*, these generated primitive fragments will be passed to the pixel shader—.

Nevertheless, this thesis has been focused into the **Shader Model 3.0**; hence, this kind of shaders are not able to use, and only the vertex and fragment ones will be programmed for determined tasks.

9.2.3 OPENGL 2.0 and GLSL

9.2.3.1 Brief introduction to OPENGL 2.0 and GLSL

As in OPENGL's section 9.1.2, on 90, **this won't be a detailed tutorial for the GLSL shading language in connection with the OPENGL specification** 2.0; however, here it can be encountered a brief vision of the basic philosophy about *shader programming*.

With entering to no deep details of OPENGL code for invoquing a GLSL program loading to the GPU, it's important that any GLSL SHADER must

- to write the `gl_Position` variable if it's a vertex shader;
- to write the `gl_FragColor` variable —or to return by a `discard` command if the actual fragment has to be rejected— if it's a fragment shader;

additionally, the **steps for executing a shader program —composed by a vertex shader and a fragment shader—** are the following, and they have to be done by any OPENGL application that wants to use GPU programming:

```
1  { this process is executed once, at the
2    application beginning }
3  ShaderCode vs = loadShader( VERTSHADER_GLSLCODE );
4  ShaderCode fs = loadShader( FRAGSHADER_GLSLCODE );
5  ShaderProg glslprog = createGLSLProgram( vs, fs );
6  [...]
7  { the glsl program must be selected for being
8    executed before the scene drawing commands }
9  useGLSLProgram( glslprog );
10 setUniformParams( glslpros, Set<UniformParam> );
11 renderScene();
```

PSEUDO code 9.5: *Steps for executing a GLSL shader program.*

9.2.3.2 Data types and qualifiers

The GLSL is a computer graphics programing language

- oriented to be **executed on GPU**;
- focused on **performing geometric computing** ;

due to this curious nature and also due to GPU's architecture is vectorial, GLSL basic data type is a $\mathbb{R}^3$ vector[53] and 1D, 2D and 3D textures.

[53] With 4-coord compatibility because of possible geometric transformation sets applied to a vector or a position.

Just like that, the next table shows the **basic data types allowed by glsl** —it also accepts structs, but as the specification format is analogous to the ANSI C one, they will be ommitted here—:

Data Format	Dimension 1	2	3	4	2×2	3×3	4×4
int	int	ivec2	ivec3	ivec4			
float	float	vec2	vec3	vec4	mat2	mat3	mat4
bool	bool	bvec2	bvec3	bvec4			

Table 9.1: *Basic vectorial data types on* GLSL

Texture Format	Dimension N	N×N	N×N×N
normalized	sampler1D	sampler2D	sampler3D
shadow	sampler1DShadow	sampler2DShadow	
unnormalized	sampler1DRect	sampler2DRect	

Table 9.2: *Basic texture data types on* GLSL

Besides, each vectorial data formats —plus the texture elements—, have **indexed elements**: they are the two, three or four implicit elements; all of these four-index sets are capable to index every data format, but **due to the *position*, *color* and *texture coordinate* convention, the coherent indexes are**:

Convention	Indexes			
position/vector	x	y	z	w
color codification	r	g	b	a
texture coordinates	s	t	p	q

Table 9.3: *Vectorial component indexes in relation to the data type conventions.*

Additionally, the *shader* **global variables** can be defined with **three different prefixes** that are determining their characteristics:

- **`uniform`**: vertex and fragment shader input data from OPENGL application —that is, from CPU—; vertex set global read-only;
- **`attribute`**; the same as `uniform` but only for vertex shader; read-only;
- **`varying`**; fragment shader input data, interpolated from vertex shader[54]; read-only for fragment shader, read-write for vertex shader;
- **`const`**; not really a kind of gobal variable, it specifies a shader-local compile-time constant with a predetermined value;

[54] Because, as a reminder, it has to be noticed that each vertex shader execution derives onto various fragment shader processings.

9.2.3.3 The most important implicit variables

The OPENGL 2.0 constructs an automatic bridge with GLSL, and uses it for sending **some *uniform* and *attribute* automatically defined values** when a *shader program* is called for execution. The total number of *uniform* and *attribute* sent variables is huge and **involves a lot of graphical parameters and configurations**. **The most important** of them are typeset in the following tables, though all the variables can be found at [gpuG]:

Uniform Variables	
mat4	gl_ModelViewMatrix
mat4	gl_ProjectionMatrix
mat4	gl_ModelViewProjectionMatrix
mat4	gl_NormalMatrix
float	gl_NormalScale

Table 9.4: *Most important 'uniform' automatically shader input variables.*

Attribute Variables	
vec3	gl_Vertex
vec3	gl_Normal
vec4	gl_Color
vec4	gl_SecondaryColor
vec4	gl_MultiTexCoord0..7

Table 9.5: *Most important 'attribute' automatically shader input variables.*

Varying Variables	
vec4	gl_FrontColor
vec4	gl_TexCoord[]

Table 9.6: *Most important 'varying' automatically shader input variables.*

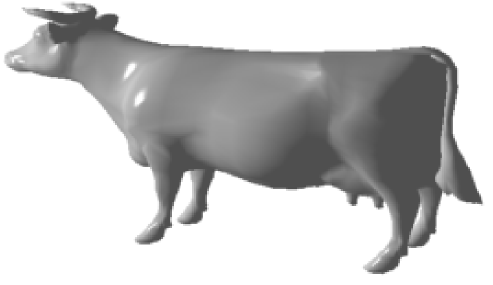
9.2.3.4 A *shader* example: the *toon* shading

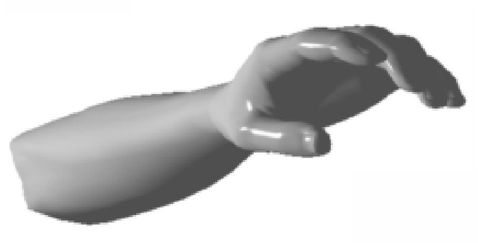
```
1   varying vec3 normal, lightDir;
2   void main()
3   {
4       vec3 vVertex = vec3( gl_ModelViewMatrix *
5                            gl_Vertex );
6       lightDir = vec3( gl_LightSource[0].position.xyz -
7                        vVertex );
8
9       lightDir = normalize( lightDir );
10      normal = gl_NormalMatrix * gl_Normal;
11      gl_Position = ftransform();
12  }
```

GLSL **code 9.6:** *The 'vertex shader' of toon shading.*

```
varying vec3 normal, lightDir;
void main()
{
    float intensity;
    vec4 color;
    vec3 n = normalize(normal);

    // intensity is the relativization of
    // light w.r.t. the vertex normal
    intensity = dot( lightDir, n );

    // toon effect w.r.t. the intensity
    if( intensity > 0.95 )
        color = vec4( 1.0, 1.0, 1.0, 1.0 );
    else if( intensity > 0.5 )
        color = vec4( 0.6, 0.6, 0.6, 1.0 );
    else if( intensity > 0.25 )
        color = vec4( 0.3, 0.3, 0.3, 1.0 );
    else
        color = vec4( 0.2, 0.1, 0.1, 1.0 );

    gl_FragColor = color;
}
```

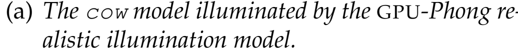
GLSL code 9.7: *The 'fragment shader' of toon shading.*

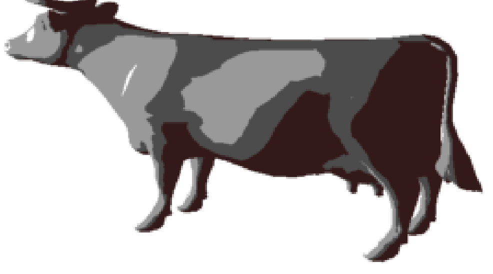




(a) *The* `cow` *model illuminated by the* GPU*-Phong realistic illumination model.*

(b) *The* `dyrtManArmTris` *model illuminated by the* GPU*-Phong realistic illumination model.*





(c) *The* GPU*-implemented toon shading algorithm over* `cow` *model.*

(d) *The* GPU*-implemented toon shading algorithm over* `dyrtManArmTris` *model.*

Figure 9.13: *The effects of two shaders applied to two models: the upper figures show the graphical results for a shader computing the Phong illumination model, and in the lower ones are featured the same models with the toon shading shader over them. This has been possible by the shader addition to this thesis' developed software MeshInspeQTor.*

9.3 GPGPU: The New Computing Paradigma

9.3.1 The Programmable Graphics Pipeline, used for General Purpose Algorithms

Although the vertex and fragment processors are built for executing only graphics-oriented tasks, **it's possible to use these multi-processor capabilities offered by a GPU for computing general tasks**, in conjunction with an existent OPENGL extension called **Frabe Buffer Object** —FBO—, able to be includes to an OPENGL graphics program by the invocation of the `GL_EXT_framebuffer_object` constant.

9.3.1.1 The Frame Buffer Object —FBO— Extension

As it's said on [gpu7], the FBO is an OPENGL extension that allows flexible **off-screen rendering, including rendering to a texture**[55]. Thus, image filters and post processing effects are able to be implemented for CPU algorithms as long as for GPU shaders.

The following code lines create a FRAMEBUFFER with previous usage of GLEW —see section 9.1.4, on the page 97—, by a **2D texture binding and its association with the FBO**:

```
1  unsigned int fbo, tex;
2
3  void setFBO()
4  {
5      glewInit();
6      if( !glewIsSupported( "GL_EXT_framebuffer_object" ) )
7      {
8          std::cout << "EXT_framebuffer_object_unsupported"
9                    << std::endl;
10         exit( -1 );
11     }
12     // generate texture
13     glGenTextures( 1, &tex );
14     glBindTexture( GL_TEXTURE_2D, tex );
15     glTexImage2D( GL_TEXTURE_2D, 0, innerFormat,
16                   texSize, texSize, 0,
17                   texFormat, GL_FLOAT, 0);
18     // create fbo and attach texture to it
19     glGenFramebuffersEXT( 1, &fbo );
20     glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbo );
21     glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
22                                GL_COLOR_ATTACHMENT0_EXT,
23                                GL_TEXTURE_2D, tex, 0 );
24 }
25
26 void drawScene()
27 {
28     glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, fbo );
29     // primitive generation from now; if the normal
30     // rendering is wanted, please type the follow:
31     // glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, 0 );
32 }
```

CPP code 9.8: *Creation of a* FRAMEBUFFER *with a previous* GLEW *usage.*

[55] That means, the generated frame won't appear onto screen display but stored into a CPU-read texture.

9.3.1.2 *General-Purpose Graphics Processor Unit* —GPGPU—

Due to the usability of this versatile extension, **it's possible to use theoretical textures as data matrix structures, accessible from the vertex and fragment shaders** and then, take advantage of the parallel computing capabilities of a GPU for **using the graphics unit as a little supercomputer**.

Dominik Göddeke in his web [gpu8] offers **three advanced GLSL tutorials for using shader capabilities for executing general-purpose algorithms** using the nVidia CG language. His teachings, in a simple overview of [gpu9], **are focusing on the general purpose methods and algorithms by using GPU potence with the aid of the *shaders***. He tells that there are three basical steps on performing that:

▷ ***Textures as data arrays***

For a CPU, the native data layout is the 1D array, but for a GPU the native one is the 2D one —but the available graphics memory is a layout size limit—. Besides, on the CPU we define the array elements as **array indices**, but on the GPU they are **texture coordinates**[56]. So, you must convert any ND-array to a 2D-array in this way: **a N-length 1D array is mapped into a texture of $\sqrt{N} \times \sqrt{N}$ texels**[57] **for single float values, or $\sqrt{N/4} \times \sqrt{N/4}$ texels for RGBA formats, always assuming N as a power-of-two scalar value.**

▷ ***Shaders as Kernels***

The **shaders** have to be taken as **kernels**, so the vertex and fragment processors compute general purpose algorithms to texture data in parallel within the GPU[58];

▷ ***One-to-one mapping from array index to texture coordinates***

The **drawing** process is equivalent to a parallel computing by using the FBO extension and render-to-texture capabilities. Due to this, a similar code to the one shown on code 9.8 will be necessary. However, by doing this, this methodology must be preceeded by a **specific special projection that maps *model-coordinate* 3d space to 2D *screen-display-coordinate* space by a $1:1$ mapping between wished rendered pixels and accessed data texels**; a functional code would be like this:

```
1  void enableOneToOneMapping()
2  {
3      glMatrixMode( GL_PROJECTION );
4          glLoadIdentity();
5
6          gluOrtho2D( 0.0, texSize, 0.0, texSise );
7      glMatrixMode( GL_MODELVIEW );
8          glLoadIdentity();
9          glViewport( 0, 0, texSize, texSize );
10 }
```

CPP code 9.9: *Creation of a* FRAMEBUFFER *with a previous* GLEW *usage.*

[56] And these coordinates are storing four-element tuples, referring the four color channels: *red, green, blue, alpha* (RGBA)—.
[57] A texel is a texture quantum unit; texel is an abbreviation form of *texture element*.
[58] So, the contained data must be independent in relation to each other texture element.

Besides, since the general purpose algorithm will be executed by a *fragment shader*, so **a fragment shader is executed for each data element we stored in the target texture**. Therefore, the idea is **make sure that each data item is computed uniquely into a fragment**, and obviously, by a unique *fragment shader*.

Then, given our projection and viewport settings one-to-one mapped, this is simply accomplished by **drawing a simple quad that covers the whole viewport —with texture coordinates correctly assigned—**, with one of the next codes, depending on used texture style:

```
1      glPolygonMode( GL_FRONT, FL_FILL );
2      glBegin( GL_QUADS );
3          glTexCoord2f( 0.0, 0.0 );
4              glVertex2f( 0.0, 0.0 );
5          glTexCoord2f( texSize, 0.0 );
6              glVertex2f( texSize, 0.0 );
7          glTexCoord2f( texSize, texSize );
8              glVertex2f( texSize, texSize );
9          glTexCoord2f( 0.0, texSize );
10             glVertex2f( 0.0, texSize );
11     glEnd();
```

CPP code 9.10: *Drawing the* GPGPU *quad, using* ***texture rectangles*** *—texture coordinates identical to pixel coordinates—.*

```
1      glPolygonMode( GL_FRONT, FL_FILL );
2      glBegin( GL_QUADS );
3          glTexCoord2f( 0.0, 0.0 );
4              glVertex2f( 0.0, 0.0 );
5          glTexCoord2f( 1.0, 0.0 );
6              glVertex2f( texSize, 0.0 );
7          glTexCoord2f( 1.0, 1.0 );
8              glVertex2f( texSize, texSize );
9          glTexCoord2f( 0.0, 1.0 );
10             glVertex2f( 0.0, texSize );
11     glEnd();
```

CPP code 9.11: *Drawing the* GPGPU *quad, using* **2D** ***textures*** *—texture coordinates normalized to the range [0,1*

Additionally, for **transferring data from GPU's FBO to a CPU's texture**, after a rendering step the programmer should add this code:

```
1  void enableOneToOneMapping()
2  {
3      glReadBuffers( GL_COLOR_ATTACHMENT0_EXT );
4      float* data = new float[texSize*texSize];
5      glReadPixels( 0, 0, texSize, texSize, texFormat,
6                    GL_FLOAT, data );
7  }
```

CPP code 9.12: *Transferring the framebuffer content to a* CPU *array.*

Then, `data` will contain the shader program complete computation calculum. Obviously, you can use `data` as shader input, and create a **hardware *looping* algorithm**.

9.3.2 The nVidia CUDA approach

9.3.2.1 The next step in GPU computation

As acronym of *Compute Unified Device Architecture*, CUDA is a C-precompiler that allow programmers to **use the C programming language to code general-purpose algorithms for execution on a CUDA-compatible nVidia GPU** —that is, from the 8000 series—.

(a) *The nVidia logo.* (b) *The CUDA logo.*

Figure 9.14: *nVidia's CUDA is the next (theoretical) step on GPGPU computing.*

9.3.2.2 GPU as a multi-threaded coprocessor

Because of the new CUDA approach, nVidia has created the **new vision of a GPU: a highly multi-threaded coprocessor**; now, a GPU is viewed as a computing device that:

1. is a **coprocessor** of the host —CPU— with its own DRAM;
2. runs many threads in parallel due to its multiple processors[59];
3. each GPU processor is an independent **kernel**, and each one is computing data-parallel portions of an application, with **no data intersections**;.

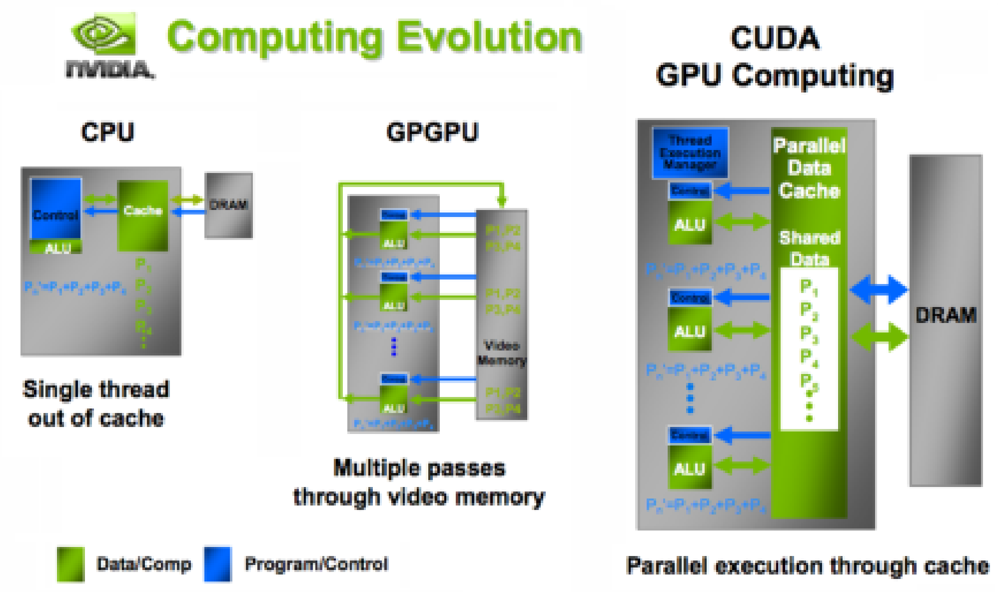


Figure 9.15: *The new nVidia CUDA-compatible GPU architecture; source [gpuD].*

[59] Since the 8000 series, nVidia GPU doesn't have vertex processors and fragment processors but a unified architecture capable of adapting to vertex or fragment requirements in relation to the resource demands.

It's an incredebly **fast and robust** paradigm although they are like threads because of a great difference between the CPU-classic threads (the real ones, in a multi-core processors) and the GPU ones:

- in front of CPU threads, **the GPU ones are extremely lightweight**, and there is little creation overhead;
- a multi-core CPU with many threads is inefficient, but a **GPU is really efficient when is managing hundreds of them**.

However, due to **nVidia technology compatibility only**, and being more conformatble on GLSL shading algorithms, CUDA has been rejected for this thesis development.

CHAPTER 10

Thesis Software Suite: (1) MeshInspeQTor

10.1 Application Overview

As has been said in multiple times along the past chapters of this document, the **developed ideas of this Thesis** have been included in a ***software suite* composed by two applications**; and one of them is this one: MeshInspeQTor, capable of: *(i)* **loading three-dimensional models** stored in OBJ file format as well as IRI[60]'s own P3D format, *(ii)* **displaying them in multiple ways** as well as *(iii)* culling them **model dissection** based on user-defined plane culling, and *(iv)* **building model volumetric representations**.

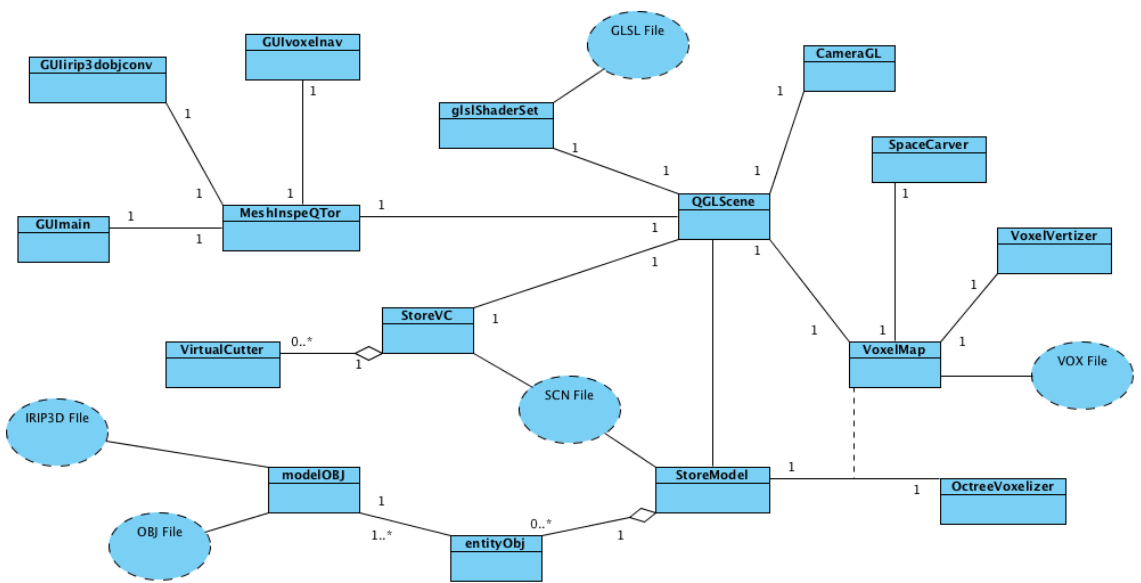


Figure 10.1: *Informal* UML *diagram of MeshInspeTor internal functionalities.*

[60] *Institut de Robòtica Industrial*, at Universitat Politècnica de Catalunya.

So, on the previous page there's an informal UML[61] diagram that contains the main concepts and algorithmic ideas and entities. The **following sections will be focused on the three MeshInspeQTor capabilities**, and related to this UML diagram.

10.2 Loading 3D models

10.2.1 The OBJ file format specification

Developed by *Alias—Wavefront Technologies* for its Advanced Visualizer animation package, the OBJ format is an **open —public— specification for geometry definition**, adopted by multiple 3D graphics application vendors. It's based on a simple data that represents 3D geometry, such *(i)* vertex position, *(ii)* texture coordinates, *(iii)*, normal directions, and *(iv)* the face set that makes each model polygon, **in the *face-vertex* way**, as defined on section 3.1.2, on page 20.

10.2.1.1 The geometry file *.obj

Called *object files*, the OBJ format can be in ASCII —text— mode or in binary mode —in this case, the file extension is `*.mod` instead of `*.obj`—. Moreover, this format **supports both polygonal and free-form geometry —that is, different curve and surface parametrizations—**[62].

However, the parser developed for this thesis is only **capable of loading the OBJ polygonal subset**, enough to load the most commonly available *object files*. The following table shows the **supported tokens** for this thesis' developed loader[63], as [ami3] tells:

Token Format	Token Description
`# some text`	A comment until the end of the line.
`v float float float`	Single vertex space position; inherently, the first specified vertex will have index 1, and the following ones will be numbered sequentially.
`vn float float float`	A normal direction; numbered like the vertices.
`vt float float`	A texture coordinate —in 2D—.
`f int int int ...`	A polygonal face, indexing only the vertices.
`f int/int int/int ...`	Another face, indexing vertices and *texcoords*.
`f int/int/int ...`	A vertex/*texcoord*/normal indexed face.

Table 10.1: *Table of supported tokens for the developed OBJ model loader.*

In the OBJ specification there is no maximum number of vertices per polygon; however, the **`modelOBJ` class, contained in MeshInspeQTor, will always triangularize every non-triangle face** for better performance and flatting-ensurance.

[61] *Unified Modelling Language*, a standardized general-purpose modeling language about software engineering. UML uses a set of graphical notations techniques for create abstract models of specific systems.
[62] See [ami4] for a complete OBJ format file specification.
[63] Indeed, the parser is closely similar to the SUN JAVA 3D OBJ loader.

On the next code it's listed the **OBJ specification for a simple cube**; notice that the set of vertices contains eight positions and the normal set, only six directions —the face normal ones—, and, because there is no face specification for only vertex/normal puspose, the texture coordinate is empty:

```
# cube vertex list (8 vertices)
v   -0.500000   -0.500000    0.500000
v   -0.500000    0.500000    0.500000
v    0.500000    0.500000    0.500000
v    0.500000   -0.500000    0.500000
v    0.500000   -0.500000   -0.500000
v    0.500000    0.500000   -0.500000
v   -0.500000    0.500000   -0.500000
v   -0.500000   -0.500000   -0.500000

# cube normal list (6 faces, 6 normals)
vn   0.000000    0.000000    1.000000
vn   0.000000    0.000000   -1.000000
vn  -1.000000    0.000000    0.000000
vn   1.000000    0.000000    0.000000
vn   0.000000    1.000000    0.000000
vn   0.000000   -1.000000    0.000000

# cube face list (6 squares -4 vertices per face-,
# notice the texture coordinate emptyness indicating
# no texture coordinate available.
f   4//1    3//1    2//1    1//1
f   8//2    7//2    6//2    5//2
f   1//3    2//3    7//3    8//3
f   5//4    6//4    3//4    4//4
f   3//5    6//5    7//5    2//5
f   5//6    4//6    1//6    8//6
```

PSEUDO code 10.1: *OBJ file text specifying a simple cube.*

Although the number of vertices and normals won't be, a priori, the same, **modelOBJ will force the same number of them, by calculating the mean of all normals inciding each vertex** —useful for obtaining a more realistic illumination model, the **Gouraud Shading**, as explained in this chapter's future sections—.

10.2.1.2 The material file *.mtl

The geometry description of a model in OBJ file is done, but it's an important factor left: the material properties' definition. In other words, *with which color will be rendered a OBJ model*.

Because of that, there is the **MTLLIB file format specification**. MTLLIB is the abbreviation of *Material Library*, and is an independent file with extension *.mtl —assuming its location is the same folder than the OBJ file, although if it doesn't exist the model loading is not affected[64]—, containing the **material properties —in RGB format— of several illumination material factors**:. In the next page's table are shown the tokens supported by the MTLLIB file format specification, as featured in [ami5]:

[64] In case of parser don't find the material file, the model is also loaded, and the rendering is done with standard material properties.

Token Format	Token Description
`# some text`	A comment until the end of the line.
`newmtl <materialname>`	Sets a new material; subsequent materials with the same name will override the previous following ones will be numbered sequentially.
`Ka float float float`	Ambient color (default: $[0.2, 0.2, 0.2]$).
`Kd float float float`	Diffuse color (default: $[0.8, 0.8, 0.8]$).
`Ks float float float`	Specular color (default: $[1.0, 1.0, 1.0]$).
`illum int`	0 to disable lighting, 1 for non-specular lighting, and 2 for full lighting. The default value is 2.
`Ns float`	Shininess (clamped to $[0.0, 128.0]$). 0.0 is the default value.
`map_Kd filename`	Texture map path.
`d float`	Transparency (clamped to $[0.0, 1.0]$). The default value is 1.0 (opaque).
`Tr float`	Same as `<d float>`.

Table 10.2: *Table of supported tokens for the developed* `MTLLIB` *model loader.*

In the following codes there will be found the `MTLLIB` for *red*, *blue* and *yellow* diffuse colors, and how the previous `cube.obj` file is modified to use this colors by:

1. loading the `MTLLIB` file;
2. and specifying **face groups**, each with its diffuse color, with the token `g <groupname>`.

it's important to notice that, if in any material definition, there's some parameters left, they won't be parametrized with the default values, as well as if the entire material is not found —or even the `*.mat` file is not found—.

```
1  newmtl yellow
2      Ka      0.4000  0.4000  0.4000
3      Kd      0.8000  0.8000  0.1000
4      Ks      0.5000  0.5000  0.5000
5      illum   2
6      Ns      60.0000
7
8  newmtl red
9      Ka      0.4000  0.4000  0.4000
10     Kd      0.9000  0.3000  0.1000
11     Ks      0.7000  0.7000  0.7000
12     illum   2
13     Ns      60.0000
14
15 newmtl cyan
16     Ka      0.4000  0.4000  0.4000
17     Kd      0.0000  0.7000  0.8000
18     Ks      0.7000  0.7000  0.7000
19     illum   2
20     Ns      60.0000
```

PSEUDO **code 10.2:** `MTLLIB` *containing the specifications for 3 colors.*

```
mtllib  yellowredcyan.mtl

# cube vertex list (8 vertices)
v   -0.500000   -0.500000    0.500000
v   -0.500000    0.500000    0.500000
v    0.500000    0.500000    0.500000
v    0.500000   -0.500000    0.500000
v    0.500000   -0.500000   -0.500000
v    0.500000    0.500000   -0.500000
v   -0.500000    0.500000   -0.500000
v   -0.500000   -0.500000   -0.500000

# cube normal list (6 faces, 6 normals)
vn   0.000000    0.000000    1.000000
vn   0.000000    0.000000   -1.000000
vn  -1.000000    0.000000    0.000000
vn   1.000000    0.000000    0.000000
vn   0.000000    1.000000    0.000000
vn   0.000000   -1.000000    0.000000

# cube face list (6 squares -4 vertices per face-,
# notice the texture coordinate emptyness indicating
# no texture coordinate available.
g yellow_faces
usemtl yellow
f   4//1    3//1    2//1    1//1
f   8//2    7//2    6//2    5//2
g red_faces
usemtl red
f   1//3    2//3    7//3    8//3
f   5//4    6//4    3//4    4//4
g cyan_faces
usemtl cyan
f   3//5    6//5    7//5    2//5
f   5//6    4//6    1//6    8//6
```

PSEUDO code 10.3: *OBJ file text specifying a simple cube with materials.*

10.2.2 The `OBJ` parsing with `modelOBJ` class

MeshInspeQTor contains a **parser/container class of `OBJ` models**, called `modelOBJ`, that is really a minor version of the class `ModelOBJ` —published on the IRI's `CVS` some time ago by this thesis' author—. This class contains another classes, comprising all a complete `OBJ` model storage; **every `modelOBJ` is composed by**:

- a `OBJModelMatLib`, abstraction of a `MTLLIB`; it contains all the materials in several `MaterialOBJ` objects.
- a set of `OBJModelGroup`, composed by multiple instances of `OBJModelFace`.

Then, the **vertex, normal and texture coordinates are commonly stored in `modelOBJ`**, so every `OBJModelGroup` can access these sets from their local face indexings. For using these data, the user has to extract every `OBJModelGroup` with its associated global data, and this action can be done by getting a **`entityOBJ`, understood as a *mini*-`modelOBJ` because it's an autonomous three-dimensional storage data extracted from a `modelOBJ`**, containing, for any group: *(i)* its vertices, normals and texture coordinates, *(ii)* its material information, and *(iii)* its face set.

10.2.3 Automatic `modelOBJ` enhancements and improvements

Indeed, the `modelOBJ` parsing process is not limited to read the data from the `OBJ` text file. As it has been said previously, **there are some internal processes done by this parsing** —done *offline*, when data is loaded into memory, and directly usable once processed without reprocessings—, due to a better performance and better data managing, like **normal adequation and triangularization of generic polygonal faces**.

10.2.3.1 Only triangular faces

The most common type of polygonal face is the **triangle**: three points are the minimum size for specifying a polygon, and it will sure be flat. However, the `OBJ` format supports polygona faces of any size, with no limit, so **the parser will triangularize non-triangular faces** following the method shown in the next figure:

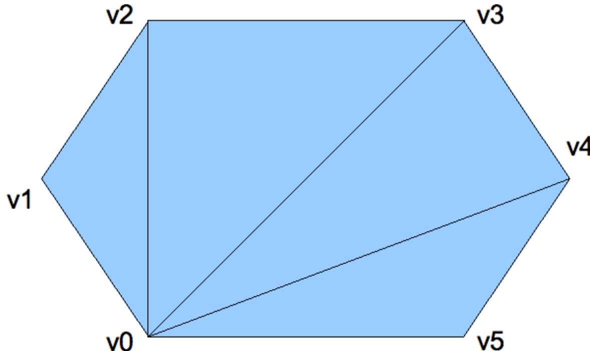


Figure 10.2: *The schematic triangularization for generic polygonal faces —here, an hexagon—.*

Obviously, there are better methods that **generate triangles from the first indexed vertex from any non-triangular face** —like the **Delaunay triangulation**—; however, a priori, this will be the major face (in relation to number of vertices) because almost 3D models are already based on triangles, so this approximation is:

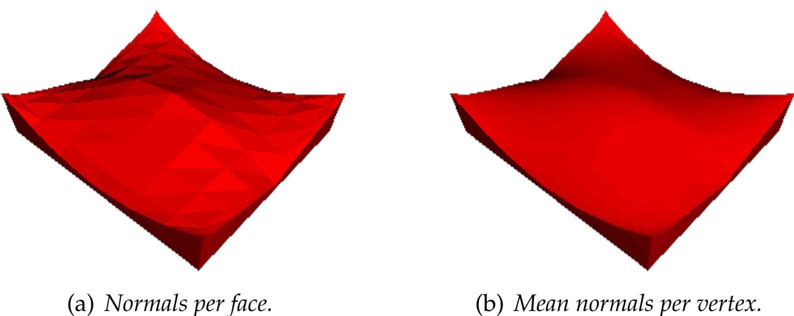
1. **fast**, because this strategy **doesn't require** to create another **vertices**;
2. assuming all the faces being **convex**, all the triangles will be surely **non-intersecting**;

10.2.3.2 Same number of vertices and normals: the per-vertex normals

The **`OBJ` specification supports different number of vertices and normals**, so *(i)* a certain normal can be assigned to more than one vertex, and *(ii)* the same vertex can have different normal directions, because the same vertex will be in more than one face. Thus, the **flat shading** will be probably sharped, as in the figure 10.3(a) —on the next page—.

So, for a better shading rendering, like in the figure 10.3(b), the way is so simple as **compute one normal per vertex by calculating the mean of all normals assigned to that vertex**; in more technical words, **changing the normals per face $\vec{n}_{f_i}$, assigned to the faces' vertices $v \in f_i$, to unitary normals per vertex $\vec{n}_{v_j}$**:

$$\vec{n}_{v_j} = \frac{1}{k_j} \sum_{i=1}^{k_j} \vec{n}_{f_i}, \quad f_i \supset v_j, \ \forall v_j \in \mathbb{R}^3$$



(a) *Normals per face.* (b) *Mean normals per vertex.*

Figure 10.3: *Illumination contrasts between vertex's normal assigment in relation to the face comprising, or the normal assignment by the mean of normals per each vertex.*

10.2.3.3 Automatic generation of normals

Moreover, this methodology can be used for **generating normals for those OBJ models not containing them** —not necessary for defining geometry but so much important for a realistic model display—, by **calculating all the per-face normals, and then executing the previous formula for every vertices within the geometry**.

10.2.4 IRI's own P3D format conversion to OBJ models

10.2.4.1 Dedicated GUI for file format conversion to OBJ format

Some time ago, Eva Monclús —researcher at IRI— told me how to extract three-dimensional models from the Visual Human project[65], available from this University. From a previous degree project implementation, done by another student, a 3D isosurface extractor of the different human organs, listed in a database, is available; however, the **quality of that extractions is poor, due to two reasons**, explained on the next image:

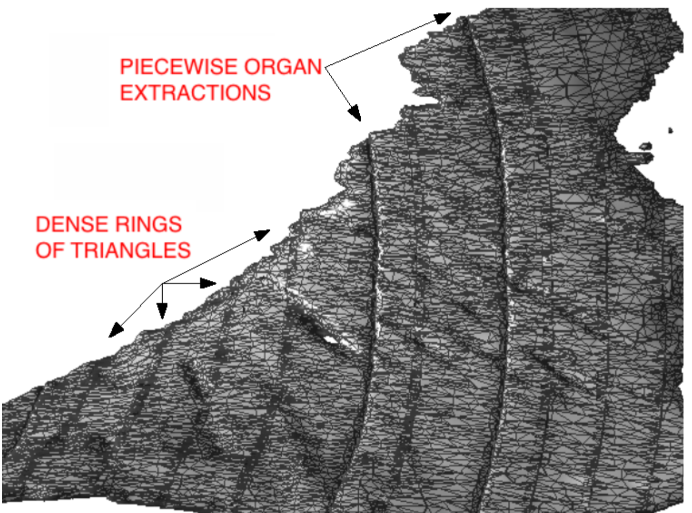


Figure 10.4: *An example of internal oblique inguinal muscle, extracted by using the* IRI *tool; please notice the defficiences on the extraction —the model is rendered by remarking the polygon edges—.*

[65] The Visible Human Project, as can be seen in [ami6], was a anatomical project promoted by NLM —*United States' National Library of Medicine*— in 1986 for creating a complete, anatomically detailed, 3D representations of the normal male and female human bodies, by acquisition of transverse CT —*Computed Tomographies*—, MR —*Magnetic Ressonances*— and cryosection images from adequate corpse sectionings.

- **some organs are too huge for being extracted at once** because of memory overflow problems, so they have to be extracted at piecewise, with the inherent loss of quality —because the 3D model is built from an isosurface—.
- the **isosurface extraction algorithm is no good**; there are great defficiences in the triangular meshing, like dense *rings* of triangles.

Nevertheless, this extractor generates a three-dimensional model based on a text format called P3D, with **no relation to the existent Lisp-based format for representing general 3D models**. This IRI-P3D format —named ust like that for provoquing a differenciation with the another P3D format file— is a own specification for IRI purposes, and modelOBJ supports a conversion from P3D to OBJ, by a dedicated MeshInspeQTor GUI:

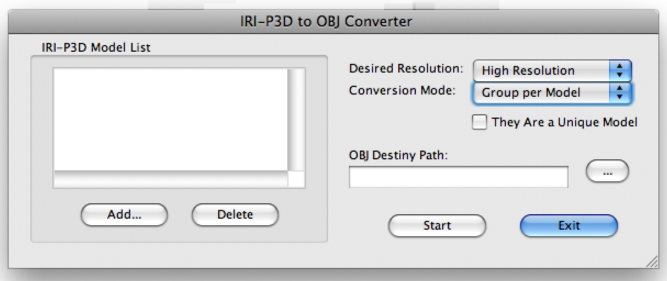


Figure 10.5: *The specific* GUI *within MeshInspeQTor for converting* IRI-P3D *format files to* OBJ*.*

This IRI-P3D format wont be detailed here; however, the GUI —and, of course, the **implemented converter**— will be roughly explained:

1. since the human organ will be stored into several P3D files —or maybe because the user desired to convert more than a human organ at once—, a **multiple P3D file selector** is available on the left of the gui, as well as a textbox that supports the path for the **single OBJ file destiny path** where all the files will be stored.
2. before clicking on start button, there are some parametrizations of the conversion that must be explained:
   - **desired resolution**: one single P3D file contains **three different resolutions** for the same three-dimensional representation of a isosurface extraction. The selection will be applied to every P3D file listed in the box when its conversion starts.
   - **conversion mode**: as mentioned before, OBJ models support face groups to separate different entities within the same 3D model. Hence, this option permits the user to **establish if every P3D is wished to be trated as an independent group within the OBJ, or all must be combined in a unique group**.

10.2.4.2 2D convex hulls for piecewise isosurface extraction merging

Additionally, on the MeshInspeQTor GUI for P3D->OBJ conversion there's a *checkbox* entitled ``They Are A Unique Model''. This checkbox **enables —or disables— a experimental —not fully functional— preprocess that tries to merge in a better way the human organs badly generated by a piecewise extraction**.

For doing this, the process assumes that the **P3D files are subsequently selected, from left to right, and by generating a 2D convex hull for the more extreme zones** remarked as *Piecewise Organ Extractions* at figure 10.4[66]; then, **computing inter-convex-hull relations between vertices, some vertex mergings are processed**. The results are shown, indeed, in the figure 10.4.

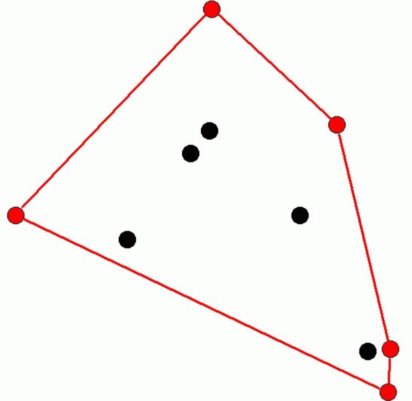


Figure 10.6: *An example of a 2D convex hull, executed over a 2D point set.*

So, the convex hull problem is the following: *given a planar set S of n points $p_i \in \mathbb{R}^2$, **compute** the vertices of $CH(S)$, sorted in **counterclockwise** order*. There are a lot of proposals, like **Graham Scan**, **Jarvis March**, or the **Quickhull** —a variant of quicksort sorting algorithm for computing convex hulls—, but the taken in account for this project was the Jarvis March, consisting in something similar like this:

```
1  function jarvis-march( Set<Point2D> S )
2  returns Stack<Point2D>
3      Point2D p0 = pointWithMinimumY( S );
4      Stack<Point2D> H;
5      H.push( p0 );
6      minpoint = getPointWithMinAngle( H.top(), ...
7                                       S.delete(H.top()) );
8
9      { sort the point set w.r.t. the angle related to
10       the entered point, from monimum to maximum angle }
11     Set<float> vecAng = getPolarAnglePointRelated( S, p0 );
12     Set<Point2D> vecSortedS := sortByAng( S, vecAng );
13     H.push( vecSortedS.first() );
14
15     { the stack will be filled with the minimum angle
16       point sequence until the reached point will be p0 }
17     S.insert( p0 );
18     while H.top() != p0 in vecSortedS do
19         minpoint = getPointWithMinAng( H.top(), ...
20                                       S.delete(H.top()) );
21         H.push( vecSortedS.first() );
22     endwhile
23     return H;
24 endfunction
```

PSEUDO code 10.4: *Jarvis March for computing a 2D convex hull.*

[66] It's possible to compute a valid 2D convex hull for a extremal zone of a isosurface extraction generated mesh because all the vertices of these zones have one of the coordinates with the same value.

A **graphical scheme of Jarvis March** is given in the next figure for a better feeling about the algorithm:

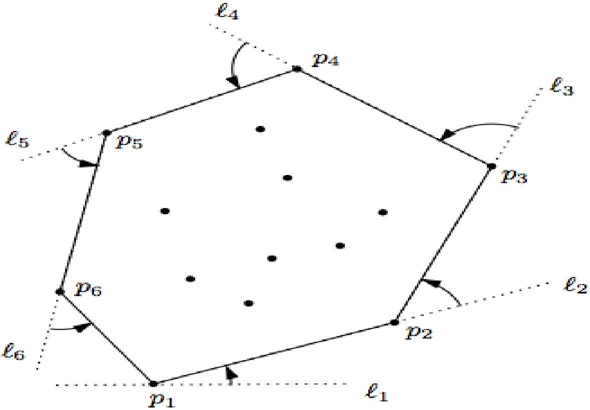


Figure 10.7: *A graphical scheme for an entire process of Jarvis March*

so the developed algorithm for piecewise Visual Human isosurface extractions is done by the next code within the `Graph2D` class, a generic module for operating with 2D pointsets:

```
1  function jarvis-march( Mesh<Point3D> MA, Mesh<Point3D> MB )
2  returns Mesh<Point3D>
3      { assumed MA is adjacent and left to MB }
4      Set<Point3D> XMA = getGreaterXPoints( MA );
5      Set<Point3D> XMB = getLowerXPoints( MB );
6      { XMA and XMB have the same 'x' coordinate,
7        so it can be omitted for having a 2D set }
8      Set<Point2D> CHA = getConvexHull( XMA );
9      Set<Point2D> CHB = getConvexHull( XMB );
10
11     { mixCHulls merges any CHA point with the
12       most euclidean-distant closer point of CHB }
13     Set<Point2D> CHmix = mixCHulls( CHA, CHB );
14
15     { Mmixed is generated by combining the sets
16       MA-XMA, MB-XMB (basic set theory), and CHmix
17       with 'x' coordinate the mean(XMA,XMB) 'x' }
18     Mesh<Point3D> Mmixed = ...
19         generateMesh( MA, MB, XMA, XMB, CHmix );
20     return Mmixed;
21 endfunction
```

PSEUDO **code 10.5:** *Developed idea for merging isosurface portioned extractions.*

10.3 Displaying 3D models

10.3.1 Overview of MeshInspeQTor rendering capabilities. The Vertex Arrays

Once the three-dimensional models have been charged into memory, MeshInspeQTor is able to **display them in several ways**, with the aim of offering the best model displaying capabilities.

So, by using the OPENGL technique of **Vertex Arrays**, the 3D model rendering is faster than with single primitive callings with the

```
glBegin( ... ); glVertex3f( ... ); ... glVertex3f( ... ); glEnd();
```

(n vertices)

methodology explained on section 9.1.2.2, on page 90; as mentioned in [ami7], with the `glBegin->glEnd` philosophy, **a lot of OPENGL calls are needed for specifying even a simple geometry**. However, with **vertex arrays**, the geometry associated **data is placed into arrays** stored in the OPENGL client —CPU— address space.

These blocks of data may then be used to **specify multiple geometric primitives with a single OPENGL command**:

```
glDrawElements( enum mode, sizei count, enum type, void *indices );
```

where `mode` can receives the same modes as `glBegin`, `count` specifies the length of `indices` —composed it by the geometry indexes—, and `type` specifies the type of data of `indices`: `GL_FLOAT`, `GL_INT`, ans so on.

Therefore, the supported OPENGL storing client arrays are the following six, which must be enabled/disabled by the command `glEnableClientState(enum array)` or `glDisablelCientState(enum array)`:

Array Enumerator	Size	Supported Types
EDGE_FLAG_ARRAY	1	boolean
TEXTURE_COORD_ARRAY	1, 2, 3, 4	short,int,float,double
COLOR_ARRAY	3, 4	(u)byte,(u)short,(u)int,float,double
INDEX_ARRAY	1	ubyte,short,int,float,double
NORMAL_ARRAY	3	byte,short,int,float,double
VERTEX_ARRAY	2, 3, 4	short,int,float,double

Table 10.3: *Supported enumeration for the vertex array data block assignment.*

10.3.2 Rendering typologies

Once the vertex arrays have been introduced[67], let's go with the **different rendering typologies** offered by MeshInspeQTor, with a snapshot of the resulting rendering a three-dimensional object —in this case, a dolphin— together with its respective code.

The different kinds of rendering include:

1. a **solid** object;
2. a **wireframe** model, similar to the solid representation but without filled faces, only the connection edges betwen the vertices will be painted;
3. a **point** based representation, based on only rendering the vertex positions without connection edges;

[67] There's another rendering methodology, called *Display Lists*, but in this thesis this technique has been rejected; the *Vertex Arrays* give good results for this project's purposes.

4. a different style called **draft**, based on a mix of the two first renderings, but without real illumination —the aim of this display style is to show the real geometry of a 3D model, so the faces (with normal per face) as well as the edges between vertices are clearly featured—.

10.3.2.1 Solid object, wireframe and point set

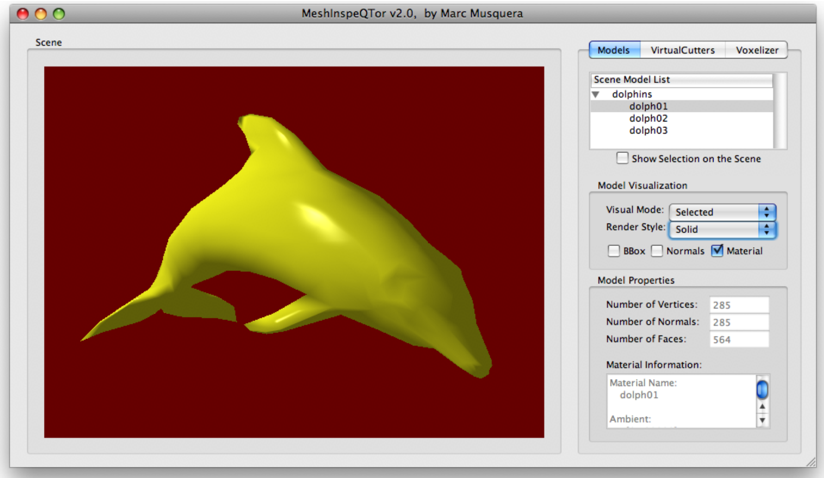
There are three basically distinct ways of model rendering, but for OPENGL they are only differing in some parameters before sending de geometry[68]; basically, there is a variable called `visualization` that manages which renderization style is desired — please notice that in the following code the material properties and colors are omitted—:

```
1  int elementsdrawed = GL_TRIANGLES;
2  switch( visualization )
3  {
4      case STYL_SOLID:
5          // the polygons will be rendered solid & filled
6          glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
7          break;
8      case STYL_WIREF:
9          // only the polygon edges will be painted
10         glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
11         break;
12     case STYL_NODES:
13         // the primitive won't be a triangle but a point
14         elementsdrawed = GL_POINTS;
15         glPointSize( pointsize );
16         break;
17 }
18
19 // there are 3 components for vertex position and
20 // normal direction because we are in R3 space, and
21 // there are 3 indexes per face because every model
22 // face is a triangle; so:
23 //     float* vecvertices = new float[nfaces*3];
24 //     float* vecnormals  = new float[nfaces*3];
25 //     int* vecvertices = new int[nfaces*3];
26 glEnableClientState( GL_VERTEX_ARRAY );
27 glEnableClientState( GL_NORMAL_ARRAY );
28     glVertexPointer( 3, GL_FLOAT, 0, vecvertices );
29     glNormalPointer(    GL_FLOAT, 0, vecnormals );
30     glDrawElements( elementsdrawed, nfaces*3,
31                     GL_UNSIGNED_INT, vecfaces );
32 glDisableClientState( GL_NORMAL_ARRAY );
33 glDisableClientState( GL_VERTEX_ARRAY );
```

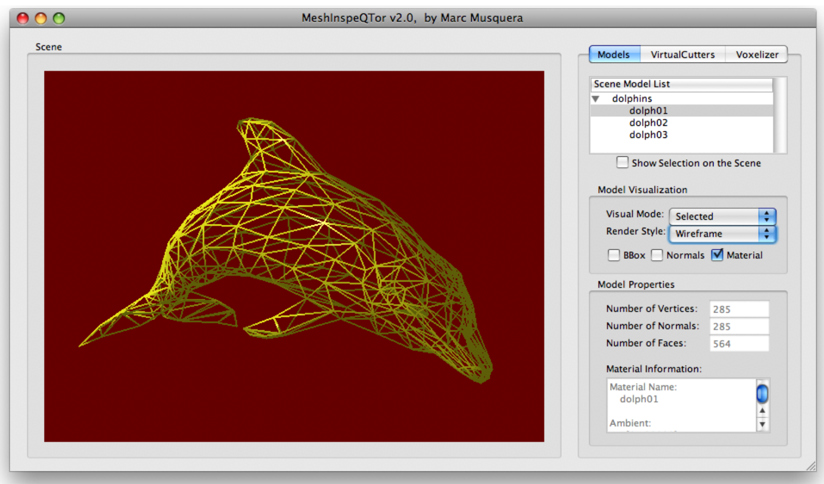
CPP code 10.6: *Solid, wireframe and point-based rendering capabilities.*

On the **next page figures, there can be seen these three types of rendering**, within the MeshInspeQTor GUI for a deeper feeling about the application. Please notice that the rendering can be easily set by a *combobox* in the middle of the right panel.

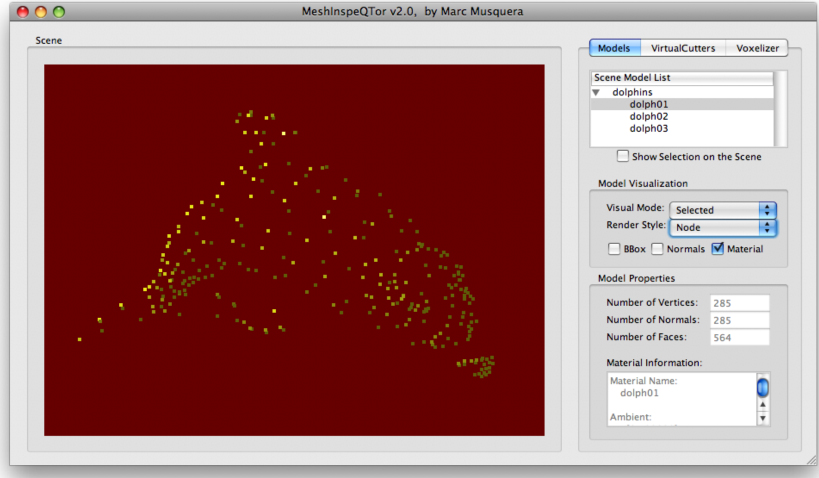
[68] It's not really true; for point set rendering the sent geometry is different from solid and wireframe geometry, because the vertices must be sent only once, while for polygonal rendering, each vertex must be sent once per face comprising. However, for simplifying the task, the geometry will be the same for polygonal and point-based rendering.



(a) *The 'solid' rendering style.*



(b) *The 'wireframe' rendering style.*



(c) *The 'node' point-based rendering style.*

Figure 10.8: *MeshInspeQTor's main rendering offerings. Notice that the* `dolphins.obj` *model is really composed by three dolphins, each one clearly identified in the 'model list' at the top of the right panel.*

10.3.2.2 Object Draft

As it has been said before, there's another rendering type, with the aim of featuring a non-nice vision but a **visual scheme of the model structure**; in more technical words, offering a **solid representation of a model that at the same time schematizes its face composition**.

For doing this, this rendering is composed by **two rendering iterations**, a *wireframe* one, with **smooth shading**[69] and another based on *solid* representation, with **flat shading**[70]. Moreover, the material properties are disabled and the faces are rendered using a quasi-white material color —for a quasi-black color for the polygon edges—.

In the following code there are the **commands for achieving this kind of renderization**; pay special attention on shading configuration and specific coloring settings —then, the original material properties disabling— for the two rendering iterations:

```
1  // the data array blocks are, of course, the same that
2  // the used with the previous renderization methods.
3
4  // first iteration:
5  // wireframe renderization with quasi-black
6  // material properties and smooth shading.
7  glShadeModel( GL_SMOOTH );
8  glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
9  glColor4f( 0.3, 0.3, 0.3, 1.0 );
10 glEnableClientState( GL_VERTEX_ARRAY );
11 glEnableClientState( GL_NORMAL_ARRAY );
12     glVertexPointer( 3, GL_FLOAT, 0, vecvertices );
13     glNormalPointer(    GL_FLOAT, 0, vecnormals );
14     glDrawElements( GL_TRIANGLES, nfaces*3,
15                     GL_UNSIGNED_INT, vecfaces );
16 glDisableClientState( GL_NORMAL_ARRAY );
17 glDisableClientState( GL_VERTEX_ARRAY );
18
19 // second iteration:
20 // solid renderization with quasi-white
21 // material properties and flat shading.
22 glShadeModel( GL_FLAT );
23 glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
24 glColor4f( 0.8, 0.8, 0.8, 1.0 );
25 glEnableClientState( GL_VERTEX_ARRAY );
26 glEnableClientState( GL_NORMAL_ARRAY );
27     glVertexPointer( 3, GL_FLOAT, 0, vecvertices );
28     glNormalPointer(    GL_FLOAT, 0, vecnormals );
29     glDrawElements( GL_TRIANGLES, nfaces*3,
30                     GL_UNSIGNED_INT, vecfaces );
31 glDisableClientState( GL_NORMAL_ARRAY );
32 glDisableClientState( GL_VERTEX_ARRAY );
```

CPP code 10.7: *Draft rendering mode* OPENGL *commands.*

[69] This is the illumination model based on *per-vertex normal* instead of *per-face* normal. This mode is supported here because, as mentioned in section 10.2.3.2 on page 114, MeshInspeQTor computes always the per-vertex normals when an OBJ file is loaded into memory.

[70] This is the illumination model with per-face normal as its basis. This type of shading is enabled although the normals are normalized per-vertex, adn the final result is lower than the smoothed one, but for the purpose of this kind of rendering is ideal.

Then, as done in the previous section, here's an **example figure featuring the cited *draft* renderization methodology**, using the same model that with the another methods:

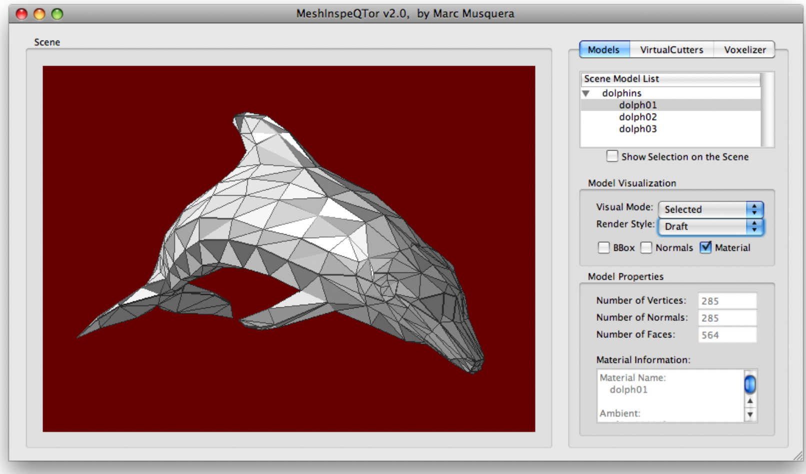


Figure 10.9: *MeshInspeQTor's 'draft' rendering style. Notice that, because of the* flat *face shading, the triangular faces are perfectly detailed and can be clearly observed.*

10.3.3 Additional 3D model properties' visualization

10.3.3.1 Model's bounding box

The MeshInspeQTor parser also computes, for every model group read from an OBJ file, its own **bounding box**; that is, an **axis aligned prism that envolves all the model**; it's stored simply with two vertices $\in \mathbb{R}^3$: *(i)* **pmin**, containing the minimum (x, y, z) coordinates of all the group vertices —with the three coordinate values indepedendent w.r.t. the vertex where found—, and *(ii)* **pmax**, containing the maximum ones.

The code for rendering the bounding box of a model will be omitted because follows the **same strategy for painting a square given its minimum and maximum coordinates —though here there are six *wireframe* squares to be painted, one for each cube face—**; but an example will be shown in the next figure, where a bunny is sourrounded by its bounding box —notice that the bounding box isn't involved on lighting parameters—.

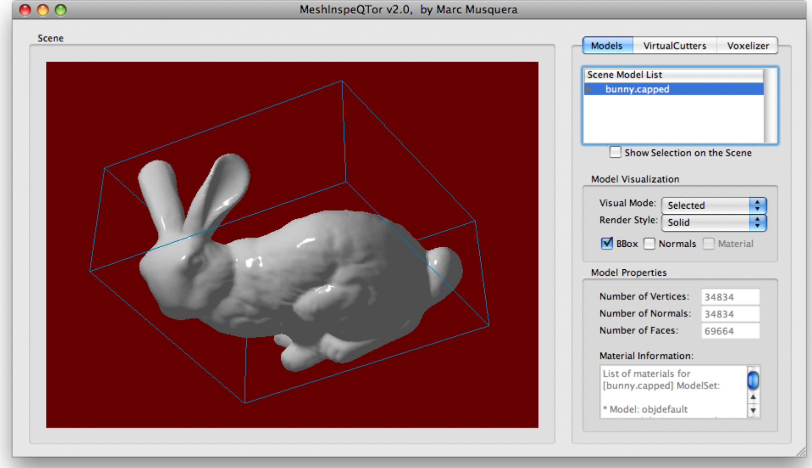


Figure 10.10: *Renderization of a model with its bounding box surrounding it, in MeshInspeQTor.*

10.3.3.2 Per-face normals

Another capability of MeshInspeQTor is to **show the per-face normals** by panting a non-affecting-lighting **red segments from the center of every face in direction of its normal**, as in the following figure:

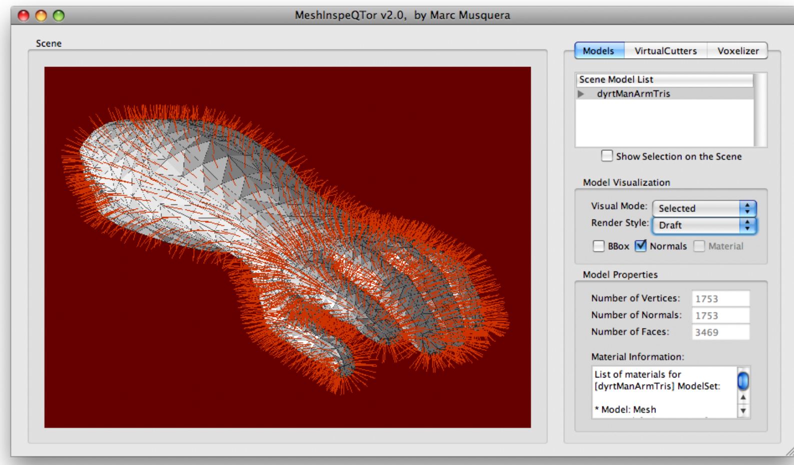


Figure 10.11: *Renderization of a model with its per-face normals shown, in MeshInspeQTor. The model is rendered with the 'draft' style for a better per-face normal visualization.*

For achieving this functionality, there's an **offline process** that computes a float value called `normalfactor` that is the length of this normal-representing segment, in relation to every model[71]. Here's the code for this process:

```
float determineNormalFactor( float* vecvertices,
                             int* vecfaces, int nfaces )
{
    float factor = 0.0;
    for( int i=0; i<vp.nfaces; i+=3 )
    {
        int idx[3];
        for( int x=0; x<3; x++ )
            idx[x] = vecfaces[i+x]*3;

        // this function returns the sum of the three
        // triangle edge lengths: in other words, the
        // triangle perimeter.
        factor +=
            getTriangPerimeter( &vecvertices[idx[0]],
                                &vecvertices[idx[1]],
                                &vecvertices[idx[2]] );
    }

    // normalization of the sum of the perimeters
    factor /= 2.0*nfaces/3.0;
    return factor;
}
```

CPP code 10.8: *Offline process for determining the normal factor for a single model.*

[71] Thus, the normals will always be shown on the screen with a nice size in relation to the own size of the model —its faces, indeed—.

Then, once the `normalfactor` is calculated when the `OBJ` file is read, the **normal renderization** implies a code just like the next —notice that is a run-time process, so the rendering is done by a set of `glVertex` callings—:

```
void renderNormals( float* vecvertices,
                    float* vecnormals, float factor,
                    int* vecfaces, int nfaces )
{
    glDisable( GL_LIGHTING );
    glColor4f( 0.8, 0.2, 0.0, 1.0 );
    glBegin( GL_LINES );
        for( int i=0; i<nfaces*3; i+=3 )
        {
            // for every face, there are computed two
            // values: final_vertex (center of the face)
            // and final_normal (mean of all the vertex-
            // composing face normals)
            float final_vertex[3];
            float final_normal[3];
            for( int x=0; x<3; x++ )
            {
                final_normal[x] = 0.0;
                final_vertex[x] = 0.0;
                for( int z=0; z<3; z++ )
                {
                    int idx = vecfaces[i+z]*3+x;
                    final_normal[x] += vecnormals[idx];
                    final_vertex[x] += vecvertices[idx];
                }

                final_normal[x] /= 9.0; //3verts*3coords
                final_vertex[x] /= 9.0; //3verts*3coords
            }

            // the normal segment is computed by the
            // final_vertex and another vertex in the
            // normal direction from final_vertex, at
            // normalfactor distance
            final_normal = normalize( final_normal );
            for( int x=0; x<3; x++ )
                final_normal[x] = final_vertex[x] +
                                  factor*final_normal[x];

            glVertex3fv( final_vertex );
            glVertex3fv( final_normal );
        }
    glEnd();
    glEnable( GL_LIGHTING );
}
```

CPP **code 10.9:** *Per-face normal renderization process.*

10.3.4 GPU dedicated algorithm for realistic Phong illumination model

Although all this features, MeshInspeQTor displays realistically three-dimensionals by an **automatically located white lighting directed to the center of all the displayed models**, and the GPU is the responsible of computing the **Phong illumination model**, more realistic than the OPENGL *pseudo*-Phong illumination model.

10.3.4.1 Introduction to the Phong illumination model

Published by Bui Tuong Phong in his 1973 Ph.D., this is an **empirical model of local illumination**, which describes the way a surface is lit as a **combination of the diffuse lighting of rough surfaces with the specular reflection of shiny surfaces**.

For defining the Phong equation, there are two concepts to be defined first of all, as mentioned in [ami8]: *(i)* the light components, and *(ii)* the material properties[72]:

- **light parameters**: i_s and i_d, where are its RGB specular and diffuse intensitiy components[73];
- **material parameters**: k_s, k_d and k_a, being the specular, diffuse —also called **Lambert term**— and ambient reflection constants, and α being the material shininess constant —defining the specular shining spot—;

But that's not all. More several directions involved in the equation must be defined, for a good understanding of the reflection model adopted by Phong:

- L as the direction vector from a certain surface point P **towards the light source**;
- N as the surface **normal** in P;
- R as the perfectly **reflected ray** of light, with P as its source point;
- V as the **direction from** P **towards the viewer** —that is, the direction from P to the camera position—;

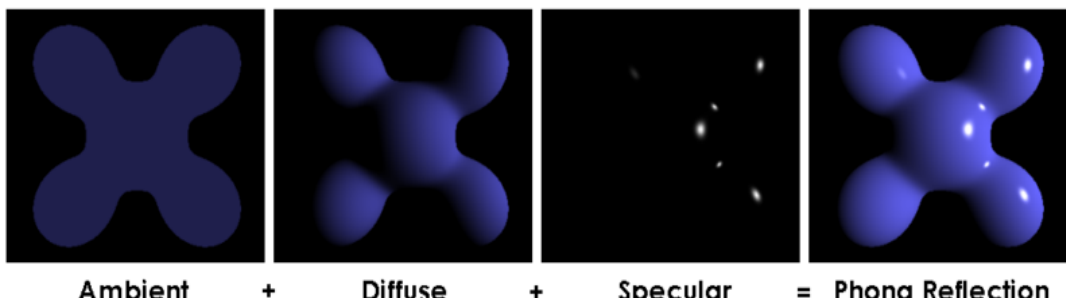


Figure 10.12: *Graphical scheme of Phong equation for the selftitled illumination model: here the light is white, the ambient and diffuse colors are both blue, and the specular color is white, reflecting almost all of the light hitting the surface, but only in very narrow highlights. The intensity of the diffuse component varies with the direction of the surface, and the ambient component is uniform (independent of direction). Figure extracted from [ami8].*

Once all these parameters have been defined, the **Phong realistic —without non interaction with other objects, only with the light sources— shading** is given by the called **Phong reflection model** that follows the next formula, being I_p the intensity received by a surface point P

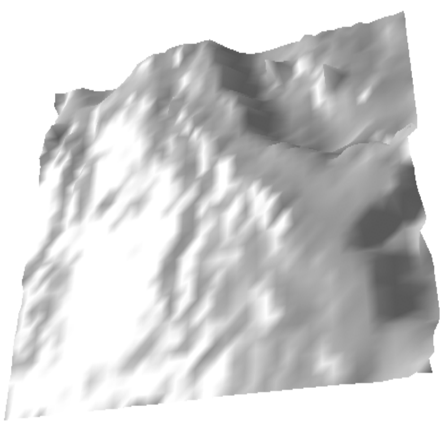
[72] Close similar to the OBJ three-dimensional format specification material properties, explained on section 10.2.1.2.

[73] Also, it's common to specify too an ambient intensity component i_a, product of the common intensity components of all the scene light sources.

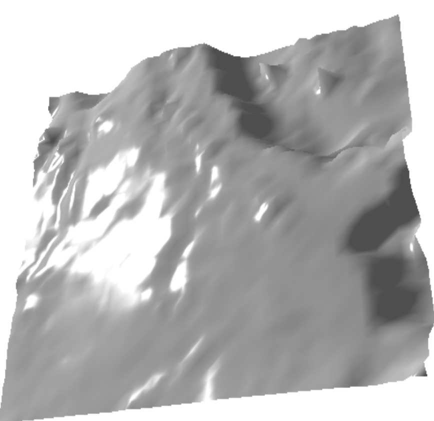
$$I_P = i_a k_a + \sum_{\forall \text{light}} \left(i_d k_d LN + i_s k_s (RV)^{\alpha} \right)$$

10.3.4.2 The Phong reflection model shaders

So, by charging two shaders —vertex and fragment; see section 9.2.2 on page 98 for more information—, the **Phong Illumination Model have been executed through the programmable pipeline of the GPU**.



(a) OPENGL's *standard illumination model.*



(b) *Phong illumination model executed in* GPU.

Figure 10.13: *Differences between* OPENGL *standard illumination rendering and a Phong model implemented over the programmable* GPU *pipeline. Notice the more specular accuracy on the left part of the model, as well as the more delimited shaded zones in the right one.*

The strategy is based, as has been already said, in two communicated shaders:

1. the **vertex shader** computes, per each vertex, the vectors L, N and V of the Phong reflection equation;

2. the **fragment shader** receives this computed data and process, per each fragment —potential pixel—, the phong equation reflection; in this way, a **more accurate illumination model is obtained**, better than the OPENGL standard one;

```
1  varying vec3 normal, lightDir, eyeVec;
2
3  void main()
4  {
5      normal = gl_NormalMatrix * gl_Normal;
6      vec3 vVertex = vec3( gl_ModelViewMatrix * gl_Vertex );
7
8      lightDir = vec3( gl_LightSource[0].position.xyz -
9                       vVertex );
10     eyeVec = -vVertex;
11
12     gl_Position = ftransform();
13 }
```

GLSL code 10.10: *Vertex shader of Phong reflection equation.*

```
varying vec3 normal, lightDir, eyeVec;

void main (void)
{
    vec4 final_color =
        (gl_FrontLightModelProduct.sceneColor *
            gl_FrontMaterial.ambient) +
        (gl_LightSource[0].ambient *
            gl_FrontMaterial.ambient);

    vec3 N = normalize( normal );
    vec3 L = normalize( lightDir );
    float lambertTerm = dot( N, L );
    if( lambertTerm > 0.0 )
    {
        final_color += gl_LightSource[0].diffuse *
                       gl_FrontMaterial.diffuse *
                       lambertTerm;

        vec3 E = normalize( eyeVec );
        vec3 R = reflect( -L, N );
        float specular = pow( max(dot(R, E), 0.0),
                         gl_FrontMaterial.shininess );
        final_color += gl_LightSource[0].specular *
                       gl_FrontMaterial.specular *
                       specular;
    }

    gl_FragColor = final_color;
}
```

GLSL code 10.11: *Fragment shader of Phong reflection equation.*

10.4 3D model arbitrary plane culling

10.4.1 Introduction to the arbitrary plane culling

On the previous section there have been detailed some processes for rendering three-dimensional models on the screen display of MeshInspeQTor, and how many ways of displaying are offered in this application. Now, it's time to present another visualization capability of this application: the **VirtualCutters**, a set of arbitrary user-defined **planes that dissect the model**.

The real effect is, by specifying some culling planes, the model will be only rendered if the occuped space is located at the so called **visible zone**, and it's defined by the **space delimited by all these culling planes —and the infinite if case—**.

10.4.2 The *VirtualCutting* process

10.4.2.1 Defining the planes

Basically, a VirtualCutter is a culling plane; so, a plane $\pi \subset \mathbb{R}^3$ can be defined in several ways:

- specifying three points p_1, p_2, p_3;
- a line $r : p + \lambda\vec{v}$ plus an external point q;
- two parallel lines: $r : p + \lambda\vec{v}$, $s : q + \mu\vec{v}$;
- two intersecting lines $r : p + \lambda\vec{v}$, $s : q + \mu\vec{w}$;
- a point p and two director vectors $\vec{v}$ and $\vec{w}$;

Additionally, with the last way of plane description, itc an be obtained another one: by **specifying the plane's normal vector associated with a displacement value**, like this[74]:

$$\pi : (x, y, z) = (p_x, p_y, p_z) + \lambda(v_x, v_y, v_z) + \mu(w_x, w_y, w_z);$$

now, by **resolving the zero-determinant equation of the two director vectors $\vec{v}, \vec{w}$, plus a generic point** —that is the difference between a generic point $X = (x, y, z)$ and the plane's given one p—, the resulting formula is the normal-based desired form:

$$\begin{vmatrix} X - p \\ \vec{v} \\ \vec{w} \end{vmatrix} = 0 \implies \begin{vmatrix} x - p_x & y - p_y & z - p_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{vmatrix} = 0$$

$$\Downarrow$$

$$\boxed{ax + by + cz + d = 0, \quad \vec{n} = (a, b, c)}$$

So, **every VirtualCutter will be defined by two variables: *(i)* its normal** $\vec{n} = (a, b, c) \in \mathbb{R}^3$ **and *(ii)* its displacement factor** $d \in \mathbb{R}$, assuming $d = 0$ as a plane that automatically passes thropugh the 3D scene center.

10.4.2.2 How to process the plane culling

▷ ***Plane culling pass test of an arbitrary point***

Once a VirtualCutter —or a set of them—, it's time to decide if a zone of a three-dimensional model must be rendered or not. So, **given the normal of a VirtualCutter, only the *positive space* w.r.t. this plane will be rendered**; of course, if a set of VirtualCutters have been specified, only the positive space that will be positive for all the VirtualCutters will be rendered.

Therefore, the **plane positive space** is defined as the 3D zone greater than the plane surface —understanding the plane normal always towards up—; so, given the reduced form of a plane, it's easy to know if a position $p \in \mathbb{R}^3$ is over, under, or within a plane $\pi\{\vec{n}, d\}$: simply calculating the value of p on the π equation and comparing its value:

- the plane equation is equal to zero, so if $\pi(p) = 0$, then it can be sensured that p is **contained** in the plane;
- if $\pi(p) = k \neq 0$, then **the sign k determines the space where this point is located**: if positive, it's in the positive space; in the negative one otherwise.

[74] Indeed, the form of a point plus two director vectors is the called plane's *vectorial equation*, while the other, based on the plane's normal vector, is the *reduced form*.

$$\pi(x,y,z) : ax + by + cz + d = 0 \ , \ \ P = (p_x, p_y, p_z) \in \mathbb{R}^3,$$

$$\Downarrow$$

$$\begin{cases} \pi(P) > 0 & \rightarrow \ \text{over the plane; } \textbf{positive space}. \\ \pi(P) = 0 & \rightarrow \ \text{within the plane; } \textbf{plane space}. \\ \pi(P) < 0 & \rightarrow \ \text{under the plane; } \textbf{negative space}. \end{cases}$$

Table 10.4: *Table of relative positions of a* 3D *point in relation to a plane.*

▷ ***The refinement step: a vertex projection to the culling plane***

However, **there's a step left** yet: calculating the **new position** q **for all** p **points located in negative space**; this q point will be the p **projection over the plane** π, so q will be contained in the plane. With this process, **ever negative space located** p **point will be replaced on the positive space** q **position**.

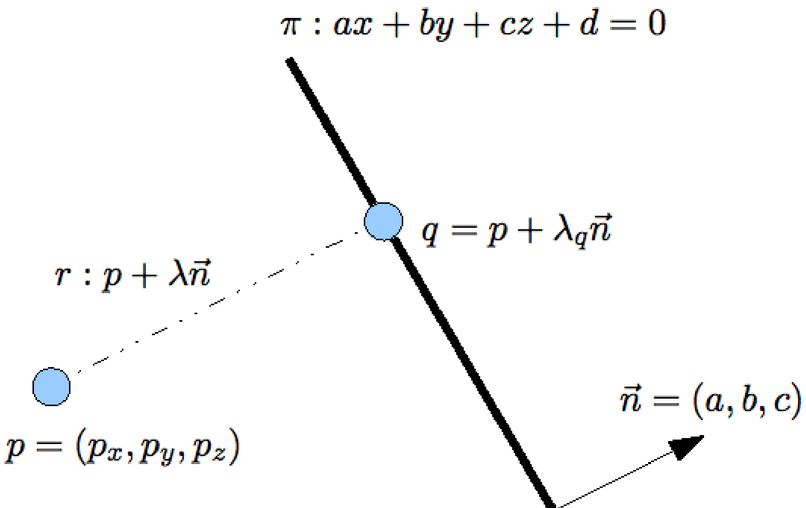


Figure 10.14: *Graphical scheme of a negative space located point* p*, and the* $q \equiv p'$ *point, being the* p *projection to the plane* π*.*

Obtaining all the equations and equivalences from the previous figure, the strategy of getting the q point for any p point is the following one: **we wish for a** q **point that is**

$$q \in \mathbb{R}^3 \ \equiv \ Proj_{\pi}(p) \ \rightarrow \begin{cases} q \ \in \pi \\ q \ \in r, \ \ \big(r \perp \pi, \ p \in r\big) \end{cases}$$

so, q **will be found by assuming it as a certain parametrization of the** r **line, imposed to be contained to** π:

$$r : p + \lambda \vec{n} \quad \Longrightarrow \quad q \ = \ p + \lambda_q \vec{n} \quad \Longrightarrow \quad \pi\big(p + \lambda_q \vec{n}\big) = 0$$

$$\Downarrow$$

$$a\big(p_x + a\lambda_q\big) \ + \ b\big(p_y + b\lambda_q\big) \ + \ c\big(p_z + c\lambda_q\big) \ + \ d \ = \ 0$$

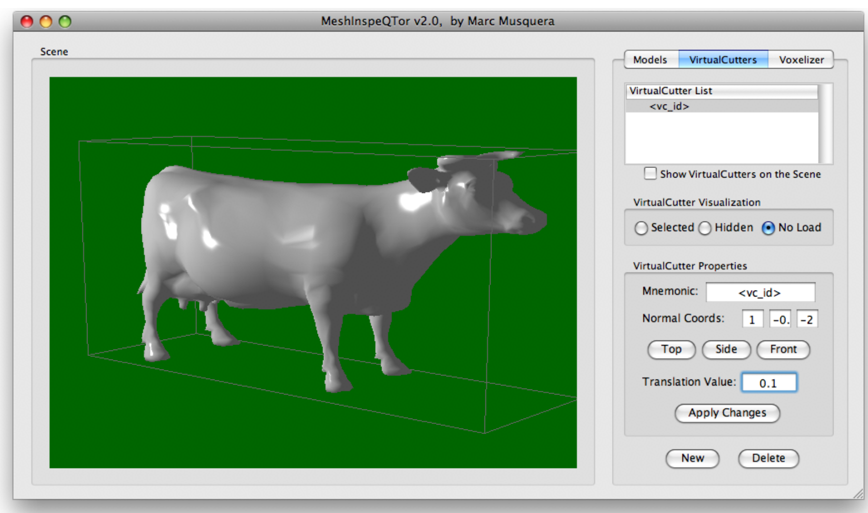
$$\lambda_q\Big(aa + bb + cc\Big) \;+\; \Big(ap_x + bp_y + cp_z\Big) \;=\; 0$$

$$\Downarrow$$

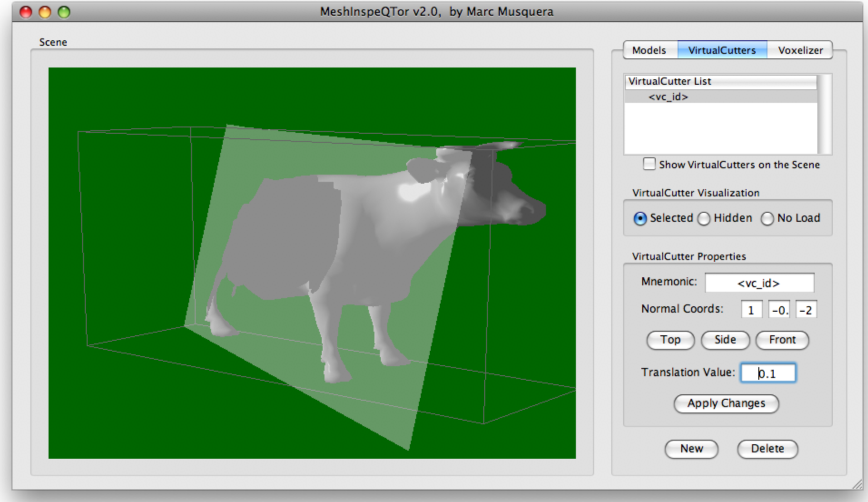
$$\boxed{\lambda_q \;=\; \frac{-\Big(ap_x + bp_y + cp_z\Big)}{a^2 + b^2 + c^2} \;=\; \frac{-(\vec{n}\dot{p} + d)}{\big(||\vec{n}||\big)^2}}$$

▷ ***Advantages of this preprocess***

This process will be this **useful for avoiding artifacts on the model rendering**, because a three-dimensional model is discretized onto polygonal faces —a priori triangles—, so a face is composed by a minimum of three vertices.



(a) *An entire cow.*



(b) *A sectioned cow.*

Figure 10.15: *Results example of applying a VirtualCutter to a generic three-dimensional model with MeshInspeTor.*

So, these are the **three possible solutions** for model arbitrary plane culling —the render process one -some VirtualCutters enabled, indeed—; obviously, it can be clearly seen that **executing this point projection to the plane is the best idea**:

- if **all** the vertices are located in the **positive space**, the face is **rendered**;
- if **no vertex** is located there, the face **won't be rendered**;
- but if **some vertices** are in positive space, there's a conflict:
  - if the face is **rendered**, there will be a **negative space invasion** rendering;
  - if the face is **not rendered, an artifact will appear**, provoquing a sharp shape similar to jaws in the model zone closer to the plane;
  - then, by **calculating the projection of all the *illegal* vertices onto the plane** and rendering the modified face, there's no negative space invasion nor artifacts.

In the previous page's figures there's an example of a dissected cow by a VirtualCutter with values $\vec{n} = (1, -0.5, -2)$ and $d = (0.1)$; and aditionally, the next figure shows a human arm dissected by two VirtualCutters —and here will be clearly visible the concept of positive space—.

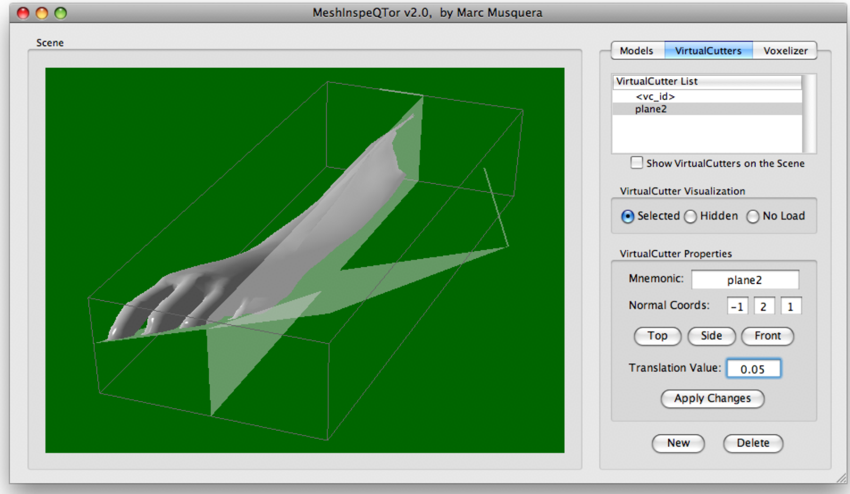


Figure 10.16: *An arm dissected by two VirtualCutters, featured on MeshInspeQTor. Please notice that the computer positive space is the intersection of all the positive spaces of all VirtualCutters.*

10.4.3 GPU dedicated algorithm for real-time VirtualCutter culling

This process, of testing the space location of every model vertex, and projecting to the VirtualCutters positive space, could be done as an offline process —for example, when a new VirtualCutter is defined—; however, MeshInspeQTor **executes this arbitrary plane culling in real-time** because of a modification of the programmable GPU pipeline that does this job **during rendering time**.

For this aim, there have been build **two shaders** —vertex and fragment—, communicated each other, **that process this culling method plus computes the previously mentioned Phong reflection equation for the rendered pixels**.

The strategy is the following one:

1. the **vertex shaders** tests the space location with respect all the VirtualCutters[75]. If the vertex doesn't pass the test, the plane projection of the vertex is executed and a flag is set and passed to the fragment shader.
2. the **fragment shader** only executes the Phong reflection equation; however, if the flag from the vertex shader is activated, the fragments derived from the discard vertex will be also discard and won't be rendered.

```
varying vec3 normal, lightDir, eyeVec;
varying float clipdistance;

uniform int vc_nofvc;
uniform mat4 vc_data1, vc_data2;

vec4 getCullingCoords( vec3 v, vec4 P )
{
    float num = -( dot(v,P.xyz) + P.w );
    float den = pow( length(P.xyz), 2.0 );
    float lambda = num/den;
    return( vec4(v + lambda*P.xyz, 1.0) );
}

void main()
{
    // vc culling process
    vec4 vertex = gl_Vertex;
    vec4 plane = vec4( 0.0, 0.0, 0.0, 1.0 );
    int idx = 0;
    clipdistance = 1.0;
    while( idx < vc_nofvc )
    {
        if( idx <= 4 )  plane = vc_data1[idx];
        else            plane = vc_data2[idx-4];
        if( dot(vertex,plane) < 0.0 )
        {
            clipdistance = dot(vertex,plane);
            gl_ClipVertex = gl_ModelViewMatrix * vertex;
            vertex = getCullingCoords( vertex.xyz, plane );
            idx = vc_nofvc;
        }
        idx++;
    }

    // phong's illumination per vertex model
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vVertex = vec3( gl_ModelViewMatrix * vertex );
    lightDir = vec3( gl_LightSource[0].position.xyz -
                    vVertex );
    eyeVec = -vVertex;

    gl_Position = gl_ModelViewProjectionMatrix * vertex;
}
```

GLSL **code 10.12:** *Vertex shader of real-time VirtualCutter activation.*

[75] Due to implementation restrictions, the maximum size of VirtualCutters is 8, because the four values of the i-th VirtualCutter are passed to the shader in the i-th column of a 4×4 matrix —there are two matrices—.

Please notice the **three uniform shader parameters**: `nofvcs` are indicating how many VirtualCutters have been defined, and the another two variables are the 4×4 matrices containing the culling planes definitions.

```
varying vec3 normal, lightDir, eyeVec;
varying float clipdistance;

uniform int vc_nofvc;
uniform mat4 vc_data1, vc_data2;

void main (void)
{
    if( clipdistance <= 0.0 )
        discard;
    else
    {
        vec4 final_color =
            (gl_FrontLightModelProduct.sceneColor *
                gl_FrontMaterial.ambient) +
            (gl_LightSource[0].ambient *
                gl_FrontMaterial.ambient);

        vec3 N = normalize( normal );
        vec3 L = normalize( lightDir );

        float lambertTerm = dot( N, L );
        if( lambertTerm > 0.0 )
        {
            final_color += gl_LightSource[0].diffuse *
                           gl_FrontMaterial.diffuse *
                           lambertTerm;

            vec3 E = normalize( eyeVec );
            vec3 R = reflect( -L, N );
            float specular = pow( max(dot(R, E), 0.0),
                             gl_FrontMaterial.shininess );
            final_color += gl_LightSource[0].specular *
                           gl_FrontMaterial.specular *
                           specular;
        }
        gl_FragColor = final_color;
    }
}
```

GLSL **code 10.13:** *Fragment shader of real-time VirtualCutter activation.*

10.5 Voxelization of polygonal meshes

10.5.1 Introduction of this functionality

And finally, the main contribution of MeshInspeQTor arrives: the **voxelization of three-dimensional models**; that means, the generation of volumetric representations of triangular meshes, perfectly modificable by using a **voxel navigator window provided by the application** GUI.

In the following sections will be explained the implementation details for all the functionalities supported by MeshInspeQTor and related to the model voxelization.

10.5.2 The VoxelMap data structure. Format and access.

Really, the voxelmap has been built as a **static array of boolean values** —`true` for valid voxels, `false` for the invalid ones—. Since the size is predetermined by the GUI[76], and **the voxelmap grid is assumed uniform and composed by a regular cube**, all the voxelmaps will have a size of

$$n \times n \times n \text{ voxels}, \quad n = \{1, 2, 4, 8, 16, 32, 64, 128, 256\};$$

so, the static bool array will be defined as the following code:

```
1        bool* voxelmap = new bool[n*n*n];
```

CPP code 10.14: *Internal declaration of a voxelmap.*

Additionally, the **access** to any cell is totally managed by an injective function for 3-dimensional arrays; given a voxelmap with a grid of $NX \times NY \times NZ$ voxels, the cell located at (i, j, k) is found by the following operation

`voxelmap.elementAt(i,j,k)` $\equiv$ `voxelmap[ ((i*NY) + j)*NX + k ]`

10.5.3 The octree as acceleration data structure for voxelization

In MeshInspeQTor every operation done over a voxelmap is done over the static array of booleans. This kind of structure allows *(i)* **fast trackings** over the voxelmap, *(ii)* inherent and easily **deducible neighborhood** by giving a voxel coordinate, and *(iii)* **instant access** to data.

However, as said in section 3.3, on page 28, voxelize a 3D model over a **regular grid** of voxels is **terribly inefficient, since to it's not an adaptive method**. Because of that, in that section some search trees are summarized, since they accelerate this methods.

On MeshInspeQTor the chosen search tree for voxelization process has been an **octree**, which is composed, internally, as a **dynamic tree structure** like the following:

```
1  struct octree{
2      bool value;              // indicates the volume value
3                               // (valid/invalid voxel).
4      bool truesons;           // true if all of its sons has
5                               // their 'value' at true.
6      float m_bboxmin[3];      // bbox (minimum coords).
7      float m_bboxmax[3];      // bbox (maximum coords).
8      octree* oct;             // the octree's eight sons.
9  };
10 octree* root;
```

CPP code 10.15: *Internal declaration of an octree.*

[76] Changing the voxelmap size implies a total reset of previously stored data.

And this is the code for **expanding a previously created octree node** —not valid for the root node— and creating its eight sons from its own data:

```
1   void expandOctreeNode( octree* oct )
2   {
3       float min[3], max[3], XXX[3], mid[3];
4       for( int k=0; k<3; k++ )
5       {
6           min[k] = oct->m_bboxmin[k];
7           max[k] = oct->m_bboxmax[k];
8           XXX[k] = (max[k] - min[k]) / 2.0;
9           mid[k] = min[k] + XXX[k];
10      }
11
12      oct->oct = new octree[8];
13      for( int k=0; k<8; k++ )
14      {
15          oct->oct[k].hadfaces = false;
16          oct->oct[k].value = false;
17          oct->oct[k].truesons = false;
18          oct->oct[k].oct    = NULL;
19
20          switch(k)
21          {
22              case 0:
23                  bboxmin[0] = min[0];
24                  bboxmin[1] = min[1];
25                  bboxmin[2] = min[2];
26                  break;
27              case 1:
28                  bboxmin[0] = min[0];
29                  bboxmin[1] = min[1];
30                  bboxmin[2] = mid[2];
31                  break;
32              [...]
33              case 7:
34                  bboxmin[0] = mid[0];
35                  bboxmin[1] = mid[1];
36                  bboxmin[2] = mid[2];
37          }
38
39          for( int z=0; z<3; z++ )
40          {
41              oct->oct[k].m_bboxmin[z] = bboxmin[z];
42              oct->oct[k].m_bboxmax[z] = bboxmin[x] +
43                                         XXX[x];
44          }
45      }
46  }
```

CPP code 10.16: *Expansion of an octree node.*

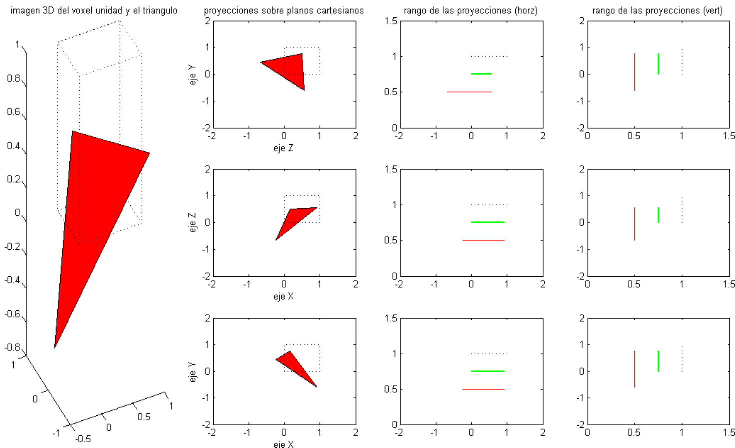
Once the octree is built with the voxelization of the model, the **transformation to the regular-grid voxelmap** is executed by an algorithm based on **knowing the space of each regular-grid voxel and finding the value of that space** —easily done by an octree depth tracking since each octree cell has its own bounding box—. This code will be ommitted here because the code is not complex and can be easily deduced.

10.5.4 The triangle-voxel intersection test

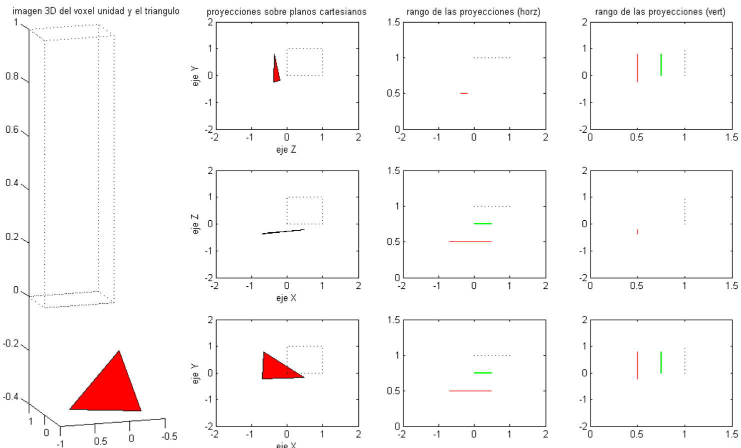
10.5.4.1 The 2D axis-aligned projection test

On MeshInspeQTor there's a **property method for voxelizing three-dimensional models**, as a variation of the Akenine-Möller proposal, described in 3.2.2.3, on page 24. In this own idea, **a triangle is approximated by a parallelogram**, similar to the generated one when a geometric vector addition between to vectors is done[77].

The parallelogram approximation isn't a real approximation but is automatically done when the **triangle is projected over the three axis $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, and after, compared to a voxel projection**[78]; then, **if there is a common domain between the cube and the triangle in one of the three axis projections, the triangle is considered intern**.



(a) *A test that results in intern triangle in relation to a voxel —cube—.*



(b) *A test that results in extern triangle*

Figure 10.17: *The 'triangle-voxel' testing implemented in MeshInspeQTor, based on 2D axis projections and implicit parallelogram approximation of the triangle.*

[77] Since a polygonal mesh will be composed by a high number of adjacent triangles, this approximation is considered valid because the parallelogram imposed zones wil surely overlap other adjacent triangles.

[78] Easily calculated because the regular grid of the voxelmap is build over the complete scene bounding box, and it's **axis aligned**.

```cpp
bool orthoProjFilter( octree* cell, float p1[3],
                      float p2[3], float p3[3] )
{
    // p1, p2, and p3 are the triangle vertices; so,
    // x_rang, y_rang and z_rang are the (x,y,z)
    // range of triangle (x,y,z) 2D projections.
    float x_rang[0] = min3( p1[0], p2[0], p3[0] );
    float x_rang[1] = max3( p1[0], p2[0], p3[0] );
    float y_rang[0] = min3( p1[1], p2[1], p3[1] );
    float y_rang[1] = max3( p1[1], p2[1], p3[1] );
    float z_rang[0] = min3( p1[2], p2[2], p3[2] );
    float z_rang[1] = max3( p1[2], p2[2], p3[2] );

    // x_bbox, y_bbox and z_bbox are the range
    // of cube (x,y,z) 2D projections.
    float x_bbox[2];
        x_bbox[0] = cell->m_bboxmin[0];
        x_bbox[1] = cell->m_bboxmax[0];
    float y_bbox[2];
        y_bbox[0] = cell->m_bboxmin[1];
        y_bbox[1] = cell->m_bboxmax[1];
    float z_bbox[2];
        z_bbox[0] = cell->m_bboxmin[2];
        z_bbox[1] = cell->m_bboxmax[2];

    float maxMin_X = max( x_rang[0], x_bbox[0] );
    float maxMin_Y = max( y_rang[0], y_bbox[0] );
    float maxMin_Z = max( z_rang[0], z_bbox[0] );
    float minMax_X = max( x_rang[1], x_bbox[1] );
    float minMax_Y = max( y_rang[1], y_bbox[1] );
    float minMax_Z = max( z_rang[1], z_bbox[1] );

    if( maxMin_X > minMax_X ) return false;
    if( maxMin_Y > minMax_Y ) return false;
    if( maxMin_Z > minMax_Z ) return false;

    return true;
}
```

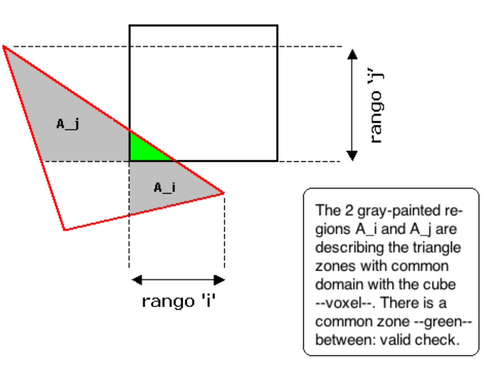
CPP code 10.17: *Axis-aligned projection triangle-cube testing.*

10.5.4.2 The cube space location in relation to the triangle

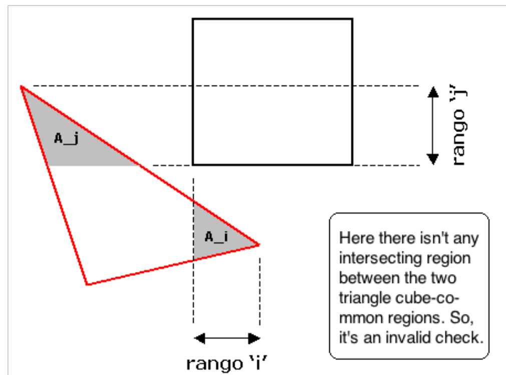
Nevertheless, **there is a problem with that method**, and is clearly noticeable with the following two schemes: **the common zone testing between the triangle and cube projections is a conservative method**. That means, if there is no common zone, the intersection won't exist sure, but is there is a common zone, the method cannot ensure there is really an intersection between the two geometric figures, as can be seen in figure 10.18, on the next page.

Hence, by solving this problem, MeshInspeQTor executes a **postfiltering test with every validated intersection**[79], consisting in a space location of the eight vertices of the cube in relation to the plane of the triangle —using the same formulation that shown in table 10.4, on page 130—:

[79] Since the algorithm is conservative, if the projection-based decides no intersection, it's not necessary.



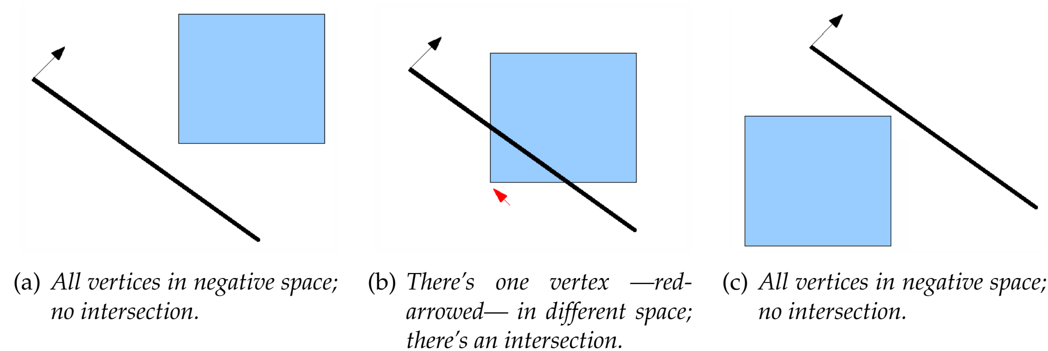
(a) *An intersecting triangle-cube situation; notice the common ranges.*



(b) *A non-intersecting triangle-cube situation, but with common ranges.*

Figure 10.18: *The 'triangle-voxel' testing conflict situations; notice the particular location and orientation of the triangle in relation to the cube. The test will validate as intern every external triangle with this particularity.*

- if **all the vertices are in the same space** —positive or negative—, there's no intersection with the triangle plane, so **there isn't intersection with the triangle**;
- if there are **at least two vertices in different spaces**, that means there's an intersection with the plane; thus, because the previous testing based on common ranges have found a valid common zone, **it's sure that the intersection exists**.



(a) *All vertices in negative space; no intersection.* (b) *There's one vertex —red-arrowed— in different space; there's an intersection.* (c) *All vertices in negative space; no intersection.*

Figure 10.19: 2D *projection of the triangle-voxel postfiltering test based on space location of the eight cube —a square here— vertices in relation to the triangle plane —the thick plane on the figures—.*

10.5.5 The voxel *vertizer* module

Moreover, **when a voxelization is executed over a preloaded three-dimensional model**, MeshInspeQTor stores within the voxelmap structure **some voxel associated data with every model vertex**, and the so called `VoxelVertizer` module is the responsible for that. Basically, for every vertex, the voxelmap stores:

- the **cell** —in (i, j, k) coordinates, relative to the $n \times n \times n$ regular grid of voxels— where the vertex is located;
- the **displacement from this voxel's center** to the cited vertex; thus, it's obtained a relative offset, with respect to the center of voxel where the vertex *is located*.

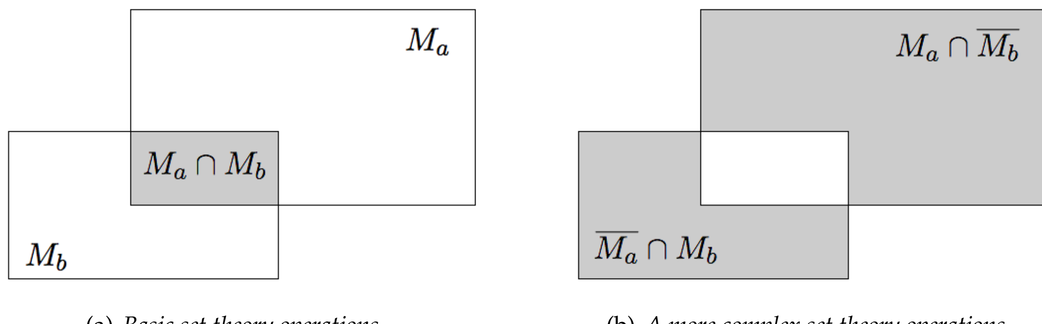
The respective **code for *vertizing* a vertex set —that is, obtaining this vertex-associated data for a regular grid of voxels—** is shown now. Of course, the word *vertizer* and the verb *to vertize* is an abuse of language, a completely invented aception, but it's for abbreviating this action of transformating the vertex model data to voxelmap data.

```
struct vertizer_struct {
    vector<unsigned char>   cellidx;
    vector<float>           offsets;
}

vertizer_struct vertizeVoxelMap( float minbbox[3],
                                 float maxbbox[3],
                                 int n,
                                 vector<float>* vertices )
{
    float voxel_length = abs( (maxbbox[0]-minbbox[0])/n );
    float lag = voxel_length/2.0f;

    float firstnode[3];
    firstnode[0] = minbbox[0] + lag;
    firstnode[1] = minbbox[1] + lag;
    firstnode[2] = minbbox[2] + lag;

    vertizer_struct ret;
    for( int x=0; x<(int)vertices->size()/3; x++ )
    {
        // cell indexes computation
        unsigned int indexes[3];
        for( int z=0; z<3; z++ )
        {
            indexes[z] = (vertices->at(3*x+z)-minbbox[z]) /
                             voxel_length;

            if( indexes[z] >= n )   indexes[z] = n-1;
            if( indexes[z] < 0 )    indexes[z] = 0;

            ret.cellidx.push_back( indexes[z] );
        }

        // vertex offset from cell center computation
        float centervox[3];
        for( int z=0; z<3; z++ )
        {
            centervox[z] = firstnode[z] +
                               (indexes[z]*voxel_length);

            ret.offsets.push_back( vertices[3*x+z] -
                                     centervox[z] );
        }
    }

    return ret;
}
```

CPP code 10.18: *Getting voxel cell coordinates of every model vertex, and its offset w.t.r. this voxel's center.*

10.5.6 Operations with mesh-generated voxelmaps based on the set theory

The voxelization process of a single three-dimensional model has been explained. However, MeshInspeQTor permits more; now, it's time to talk about the **set theory applied to complex sequential voxelizations**. The application permits to define a 3D model by its normal voxelmap or its inverted one —that means, *validated* voxels are considered *invalid* and viceversa—, for **combining them like sets of values by using *union* ∪ or *intersection* ∩ set theory operators**.



(a) *Basic set theory operations.* (b) *A more complex set theory operations.*

Figure 10.20: *A clear examples of applied set theory; by considering the squares M_a and M_b as three-dimensional volumetric representations, the application on mesh voxelization is noticeable.*

An example of set theory applied to voxelmaps would be a scene composed by two intersecting models M_a and M_b; so, if you want to obtain the voxelmap of M_a but without the intersecting part with M_b, you have to specify, as can be seen in figure 10.20(b), the following voxelization operation:

$$\texttt{voxelization}(M_a - M_b) \quad \equiv \quad M_a \cap \overline{M_b}$$

For achieving this purpose, MeshInspeQTor *(i)* executes a **voxelization per model** —per `OBJ` group, indeed—; then *(ii)* applies the **inverse** operation if needed, and finally *(iii)* combines the diverse voxelmaps per couples by applying the **indicated set operator**. It's important to notice that MeshInspeQTor **only permits to apply a global intersection or a global union with all the voxelmaps**; this is a truly limitation but for the most common cases it's not a real problem.

For some examples of set theory application to velizations, please see section B.2.3.2 on page 216. These are the appendixs, containing user manuals for the two application of the thesis developed *software suite*, so there are some snapshots of MeshInspeQTor after applying some set theory operators to a multi-mesh voxelization are featured.

10.5.7 The Voxel Carving

10.5.7.1 The standard *total refilling* strategy

As explained before on section 4.2.2, on page 33, the **objective of the voxel carving technique is to refill the empty inner zones of a convex mesh voxelization** —in the resulting voxelmap from a convex mesh voxelization, there won't be inner voxels; only outer voxels in the two disjoint subspaces split by the frontier voxels generated by voxelization—.

In the figure 4.6, on page 35, there is a **complete six-step voxel carving for refilling the voxelization of a three-dimensional torus**. So, the **code** for this six steps is done by the following commands:

1. generate an **auxiliar voxelmap, completely set to true**;
2. now is executed a per-boundingbox-$n \times n$-face voxels' **linear tracking in direction to its opposite cell** —*e.g.*, from the cell $[0, 0, 0]$ to the $[n, 0, 0]$ in direction $\mathcal{X}^+$, or from $[3, n, 2]$ to $[3, 0, 2]$ in direction $\mathcal{Y}^-$—. In total, $6 \times n \times n$ linear trackings.
3. the strategy for every linear tracking is the follow: **setting all the cells to false until the real voxelmap's corresponding cell is true**; then, the algorithm returns and begins another linear tracking from other cell.

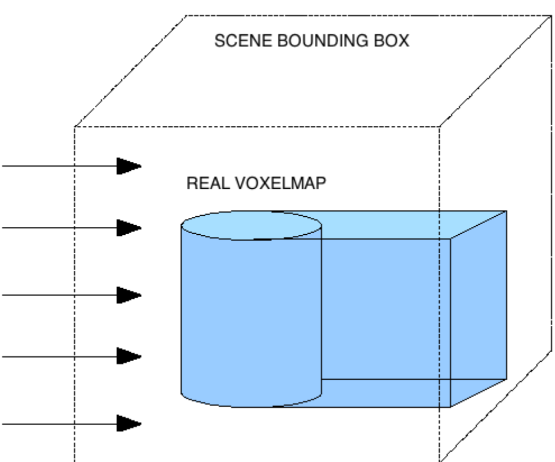


Figure 10.21: *A $n \times n$ linear tracking —the $\mathcal{X}^+$ one indeed— of a cylinder voxel carving.*

In this way, when the six $n \times n$ linear tracking steps are completed, **the auxiliar voxelmap has converged to the real voxelmap** —because the auxiliar cells have been rejected until the first really validated cell is found—, **with the inner parts refilled**[80].

10.5.7.2 The thesis' innovative *hole preserving* algorithm

The so called *total refilling* is the standard voxel carving method, but in MeshInspeQTor there has been another method, **experimental and, at the same time, ideated for this thesis**. Receives the name of *hole preserving*, and it's useful for execute a **voxel carving considering other inner objects as *holes* for the external ones**.

The main **differences between this new technique and the *total refilling* standard one** are basically:

- here the strategy is done with no optimization: **six intermediate steps and the resulting voxelmap is the intersection between them**;

[80] So, pay attention on that the algorithm does not execute six carving steps for building an intersection-with-all new voxelmap, as indicated in the previously mentioned figure 4.6. On the figure MeshInspeQTor was modified due to displaying the six steps, uniquely for documentation purposes.

- since **the voxelmap is layered** in the way that objects inside other objects are considered holes, and inner objects of this *holes* are considered again solid, and subsequently so on, **there are more voxel typologies than the simple *valid* and *invalid***, as shows the next table:

State	Possible future carving states	
EXTERNAL_VOID	nextcell: *invalid*	EXTERNAL_VOID
	nextcell: *valid*	OBJECT_BEGINNING
OBJECT_BEGINNING	nextcell: *invalid*	INTERNAL_VOID
	nextcell: *valid*	OBJECT_BEGINNING
INTERNAL_VOID	nextcell: *invalid*	INTERNAL_VOID
	nextcell: *valid*	OBJECT_ENDING
OBJECT_ENDING	nextcell: *invalid*	EXTERNAL_VOID
	nextcell: *valid*	OBJECT_ENDING

Table 10.5: *Table of supported tokens for the developed* OBJ *model loader.*

10.6 Developed File Formats

10.6.1 Scene *.SCN File Format

This own file specification permits store on hard-disk, for future loadings, the MeshInspeQTor **graphical scene parameters**, like the paths of the loaded OBJ models, as well as the already parametrized VirtualCutters.

```
1  @3DINSPEQTOR_SCENE:FILEFORMAT_v1.0
2  @SCENE_MODELS
3  <numberof-loaded-obj-files>
4  <objpath1>
5  <model1-id>
6  <visualize-mode> <visualize-style> <bbox-wanted>
7  <numberof-groups>
8      <group1-id>
9          <visualize-mode> <visualize-style> <bbox-wanted>
10     <group2-id>
11         <visualize-mode> <visualize-style> <bbox-wanted>
12     [...]
13 <objpath2>
14 <model2-id>
15 <visualize-mode> <visualize-style> <bbox-wanted>
16 <numberof-groups>
17     [...]
18 [...]
19 @SCENE_VIRTUALCUTTERS
20 <numberof-virtualcutters>
21 <vc1-id>
22     <normalcoords> <displacementvalue> <visualize-mode>
23 <vc2-id>
24     <normalcoords> <displacementvalue> <visualize-mode>
25 [...]
```

PSEUDO code 10.19: *MeshInspeQTor scene file format.*

10.6.2 VoxelMap *.vox File Format

This is the file format specification ready to store the **voxelmap information**, as well as the **vertized** associated data if case.

```
@MESHINSPEQTOR_VOXELMAP:FILEFORMAT_v2.0
@VOXELMAP_GRID
<voxelmap-grid-size>
<voxelmap-boundingbox>
@VOXELMAP_DATA
<voxelmap-bytedata-;-bits-as-voxelvalue-paritybit>
@VOXELMAP_FFDV
<sizeof-model-data>
    <vertized-model-vertices>
    <model-normals>
    <vertized-model-indexes>
<numberof-triangles>
    <model-triangles-vertexindexes>
```

PSEUDO code 10.20: *MeshInspeQTor voxelmap file format.*

CHAPTER 11

Thesis Software Suite: (2) ForceReaQTor

11.1 Application Overview

This is the second application of the *software suite* developed for this thesis project. ForceReaQTor is a **complete simulator of physically-based deformations** composed by a hybrid deformation system automatic generation module from three-dimensional volumetric representations models: *(i)* the inner layer is a 3D non-rigid skeleton based on a **mass-spring** system, while *(ii)* the surface layer is a *pseudo*-FFD-based **deformable triangular mesh**. However, there's no interactive feeling with the deformation model, but **a complete deformation specification panel set is featured** on the application's GUI for experimenting with the **generic 3D real-time deformations**.

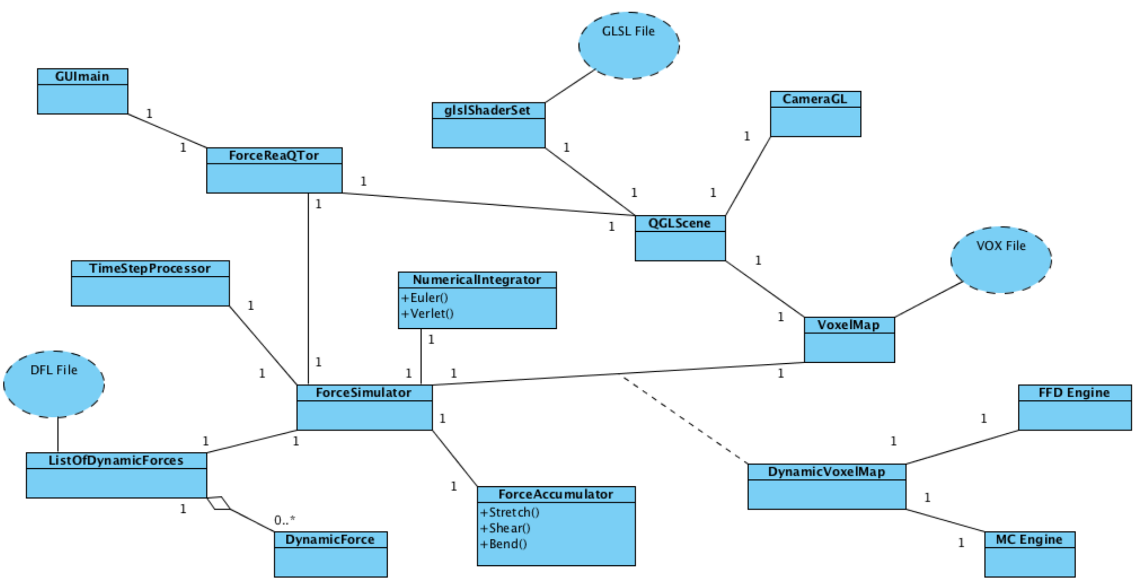


Figure 11.1: *Informal* UML *diagram of ForceReaQTor internal functionalities.*

In this chapter there will be described *(i)* all the **real-time deformation algorithms** together with *(ii)* all the **simulation parameters** and *(iii)* the different 3D **rendering** classes; everything including a detailed code publication and data structure used specifications.

11.2 From VoxelMap to 3D Mass-Spring System

11.2.1 Overview of the main ForceReaQTor preprocess

ForceReaQTor, as said in the previous section, applies deformation forces to non-rigid springs in a mass-spring system; and this **mass-spring system is provided by a convenient transformation of a generic voxelmap**, previously generated by using the previous application MeshInspeQTor —explained on the previous chapter—.

As mentioned before on section refsec:cloth, on page 65, a mass-spring model is composed by **mass points, connected together by a network of massless springs**. Therefore, ForceReaQTor transforms a voxelmap into a mass-spring system by:

- employing the **centers of the validated voxels as the mass points** of the system, by generating automatically their relaxing coordinates;
- the network of springs is automatically generated by linking each voxelmap node with —if it exists— its direct neighbour in the six face-cube- perpendicular directions; so: ***top*, *bottom*, *left*, *right*, *front* and *rear* voxels will be —if they are really a validated cell— the neighbour of every voxel**;

the following code is the **internal structure** adopted by the `DynamicVoxelMap` class, when ForceReaQTor imports a voxelmap by loading a `VOX` file and it adapts the map data to the deformable model; notice that the *vertized*[81] mesh-voxelmap association data is omitted due to clarity reasons:

```
1  typedef struct {
2      int* hash;               // acceleration hash-access
3                               // data for masspoints.
4      unsigned char* cuts;     // stores the possible
5                               // spring cuttings.
6      bool* fixed;             // indicates if a masspoint
7                               // is fixed.
8      float* coords;           // the masspoint set coords
9      unsigned int* nodes;     // renderization data block
10                              // for mass-points (OpenGL
11                              // dedicated).
12     unsigned int* springs;   // renderization data block
13                              // for springs (OpenGL
14                              // dedicated).
15     int n_nodes;             // number of mass-points
16     int n_springs;           // number of springs
17     int n_coords;            // number of node coords
18 } DynVoxelMapData_struct;
19 DynVoxelMapData_struct dynvoxelmap;
```

CPP code 11.1: *Internal data structure for mass-spring systems.*

[81] See section 10.5.5 on page 139 for more information.

The `hash` array block is used for **neighbour testing and accessing in a single operation**, a useful resource in a process where the computational time is a real handicap.

11.2.2 The conversion to a mass-spring system in detail

There's the **conversion code from voxelmap cell center to the relaxing mass-point coordinates**; notice that, in every following code, a variable with the 'm_' prefix will be, as the hungarian notation tells, module local variables.

```
// initialization of the acceleration hashing structure.
dynvoxelmap.hash = new int[ m_nvox*m_nvox*m_nvox ];
for( int x=0; x<m_nvox*m_nvox*m_nvox; x++ )
    dynvoxelmap.hash[x] = -1;

// m_bbox is a module variable containing the voxelmap
// bounding box ---elems [0,1,2] the minimum coordinates,
// and the [3,4,5] ones the maximum coords---; and m_int
// contains the regular-grid voxel edge length.
m_firstnode[0] = m_bbox[0] + abs(m_int/2);
m_firstnode[1] = m_bbox[1] + abs(m_int/2);
m_firstnode[2] = m_bbox[2] + abs(m_int/2);
vector<float> coords;
vector<unsigned int> nodes;

// automatic generation of the relaxing coordinates
// of the mass points, by assigning the voxel center
// position by knowing the number of voxels and the
// voxelmap bounding box.
float it = 0.0;
int X = m_nvox; int Y = m_nvox; int Z = m_nvox;
int counter = 0;
for( int i=0; i<m_nvox; i++ )
    for( int j=0; j<m_nvox; j++ )
        for( int k=0; k<m_nvox; k++ )
        {
            // 3D to 1D injective association
            int idx = ( (i*Y) + j )*X + k;
            if( voxelmap[idx] )
            {
                dynvoxelmap.hash[idx] = counter;
                nodes.push_back( counter );
                counter++;

                coords.push_back( firstnode[0] +
                                  abs(i*m_intx) );
                coords.push_back( firstnode[1] +
                                  abs(j*m_inty) );
                coords.push_back( firstnode[2] +
                                  abs(k*m_intz) );
            }
        }

dynvoxelmap.n_coords = coords.size()/3;
dynvoxelmap.coords = new float[ coords.size() ];
for( int x=0; x<(int)coords.size(); x++ )
    dynvoxelmap.coords[x] = coords[x];
```

CPP **code 11.2:** *Voxelmap conversion to a mass-spring system: mass point generation.*

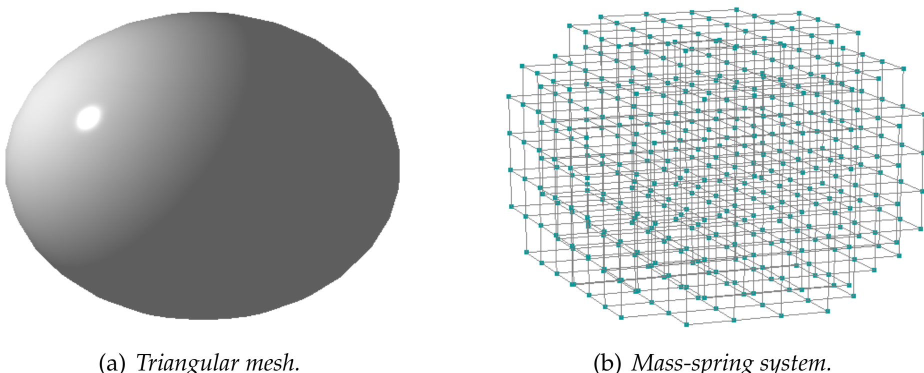
Let's see now the **conversion code relative to the spring generation**. Since the **springs are uniquely a graphical entity** —for an algorithm, the spring is stored as the neighbourhood between two mass points—, so this code is based on the **generation of the coherent data block for displaying springs with the OpenGL Vertex Arrays** —technique explained on section 10.3.1, on page 118—, as well for mass-point data block.

```
dynvoxelmap.n_nodes = nodes.size();
dynvoxelmap.cuts = new unsigned char[ nodes.size() ];
dynvoxelmap.nodes = new unsigned int[ nodes.size() ];
for( int x=0; x<(int)nodes.size(); x++ )
{
    dynvoxelmap.nodes[x] = nodes[x];
    dynvoxelmap.cuts[x] = CUT_NOCUTS;
}

vector<unsigned int> springs;
for( int facX=0; facX<m_nx; facX++ )
for( int facY=0; facY<m_ny; facY++ )
for( int facZ=0; facZ<m_nz; facZ++ )
{
    int i=facX; int j=facY; int k=facZ;
    int idx = ( (i*Y)+ j )*X + k;
    if( dynvoxelmap.hash[idx] == -1 ) continue;

    if( i+1 < m_nx )
    {
        int idx_x1 = ( ((i+1)*Y) + j   )*X + k;
        if( dynvoxelmap.hash[idx_x1] >= 0 )
        {
            springs.push_back( dynvoxelmap.hash[idx] );
            springs.push_back( dynvoxelmap.hash[idx_x1] );
        }
    }
    if( j+1 < m_ny )
    {
        int idx_y1 = ( (i*Y)      + j+1 )*X + k;
        if( dynvoxelmap.hash[idx_y1] >= 0 )
        {
            springs.push_back( dynvoxelmap.hash[idx] );
            springs.push_back( dynvoxelmap.hash[idx_y1] );
        }
    }
    if( k+1 < m_nz )
    {
        int idx_z1 = ( (i*Y)      + j   )*X + k+1;
        if( dynvoxelmap.hash[idx_z1] >= 0 )
        {
            springs.push_back( dynvoxelmap.hash[idx] );
            springs.push_back( dynvoxelmap.hash[idx_z1] );
        }
    }
}

dynvoxelmap.n_springs = springs.size();
dynvoxelmap.springs = new unsigned int[ springs.size() ];
for( int x=0; x<(int)springs.size(); x++ )
    dynvoxelmap.springs[x] = springs[x];
```

CPP code 11.3: *Mass-spring system renderization adequation: node-spring generation.*

11.2.3 Displaying the mass-spring system

Once the `dynvoxelmap.nodes` and `dynvoxelmap.springs` Vertex Array dedicated data blocks have been computed in the conversion preprocess, the displaying is able to be done. This is the result of **displaying a** $8 \times 8 \times 8$ **voxelmap of a sphere as a mass-spring system**.



(a) *Triangular mesh.* (b) *Mass-spring system.*

Figure 11.2: *The total conversion from a 3D model to its mass-spring system —the voxelmap is omitted but it can be deduced from the mass-spring system nodes.*

Notice that the voxelmap has been carved for a more exact non-rigid skeleton feeling of the mass-spring system; ForceReaQTor deforms better mass-spring systems derived from carved voxelmaps that from voxelmaps without carving process. This is due to the carving process adds internal validated nodes, so the **non-rigid skeleton will locate the masspoints in a more regular grid way**.

That's the **Vertex Array renderization code** by using the preprocessed data blocks.

```
1  glDisable( GL_LIGHTING );
2
3  // node rendering
4  glPointSize( m_voxelfactor );
5  glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
6  glColor4f( 0.0, 0.6, 0.6, 1.0 );
7  glEnableClientState( GL_VERTEX_ARRAY );
8      glVertexPointer( 3, GL_FLOAT, 0, dynvoxelmap.coords );
9      glDrawElements( GL_POINTS, dynvoxelmap.n_nodes,
10                     GL_UNSIGNED_INT, dynvoxelmap.nodes );
11 glDisableClientState( GL_VERTEX_ARRAY );
12 glPointSize( 1.0 );
13
14 // spring rendering
15 glLineWidth( m_springfactor );
16 glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
17 glColor4f( 0.5, 0.5, 0.5, 1.0 );
18 glEnableClientState( GL_VERTEX_ARRAY );
19     glVertexPointer( 3, GL_FLOAT, 0, dynvoxelmap.coords );
20     glDrawElements( GL_LINES, dynvoxelmap.n_springs,
21                     GL_UNSIGNED_INT, dynvoxelmap.springs );
22 glDisableClientState( GL_VERTEX_ARRAY );
23 glLineWidth( 1.0 );
```

CPP code 11.4: *Mass-spring system Vertex Array renderization code.*

11.3 The Dynamic Force Data Structure

11.3.1 Predefined deformations. The Timeline Simulation Specification

11.3.1.1 The GUI configuration panel

ForceReaQTor supports **dynamic deformations based on applying forces to the mass-spring nodes**, and then computing the visoelastic effects of the springs due to the node displacement. However, there is **no interactive method for three-dimensional model** deformation, such as a deformation occurs by a keyboard, mouse, or any action provided with other device.

Indeed, there's a **configuration panel** in the ForceReaQTor GUI, that allows the application user to define a **set of dynamical deformation tools**. Moreover, **every dynamic element has a lifetime**, specified by a starting and ending time instants, completely configurable from the GUI dedicated panel.

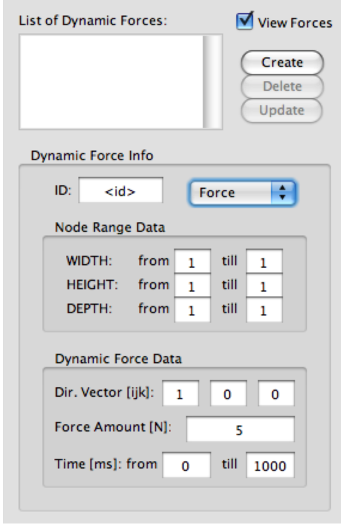


Figure 11.3: *The ForceReaQTor dynamic force configuration panel* —zoomed—.

In the above figure there's a **snapshot** of the previously mentioned **configuration panel** for dynamic force application. As can be seen, it's subdivided in **three zones** —apart from the list of stored ones, that permits creating, deleting or modifying dynamic applications—:

1. the **id and dynamic typification** zone; here there's a textbox for typing the unique **identificator** of the dynamic and the dynamic class can be chosen; there are three options, explained in the subsequent subsection: *(i)* **force** and *(ii)* **fixing node**, applied both two to a mass-point set, and *(iii)* **spring cutting**, applied to a spring set;

2. the **3D application range** zone, where the user can specify the node —or spring— range in the three axis directions;

3. the **dynamic force parameters** zone, which will manage the force amount (int newtons N) and direction —if case—, and the lifetime of the selected dynamical force, specified by starting and ending miliseconds;

11.3.1.2 Independence between lists of dynamic forces and voxelmaps

It's important to notice that **the node set range will be never dependant of the validated cells in the voxelmap**. Indeed, any **list of dynamic forces is independent of any voxelmap**, so the same list can be applied to different voxelmaps, and more than one list can be applied to the same voxelmap, because there are non-related data entities. Hence, ForceReaQTor will automatically filter the node set range: if the range achieve non-validated active voxelmap cells, that cells will be simply ignored.

11.3.1.3 Data structure for dynamic forces

ForceReaQTor is able to manage every dynamic force class described in this section, and for that, there are **two classes**, `DynamicForce` and `ListOfDynamicForces`, that supports all these features. At first, there's the **internal structure for a dynamic force**, with methods for discerning about the typification:

```
1                typedef struct {
2                    float x;
3                    float y;
4                    float z;
5                } triplet;
6
7                triplet direction;
8                vector<triplet> positions;
9                int miliseconds;
10               float newtons;
11
12               // distinction methods
13               bool isForce()
14                   { return (newtons > 0); }
15               bool isFixed()
16                   { return (newtons < 0); }
17               bool isCut()
18                   { return (newtons == 0); }
```

CPP code 11.5: *Dynamic Force internal data structure.*

Notice, however, that here the **dynamic force has no starting instant time nor ending one but lifetime in miliseconds**. There's no variable for storing the force identificator. So, the timeline location as well as the *id* management is done by the dynamic force container, the `ListOfDynamicForces` module:

```
1            typedef struct {
2                DynForce force;
3                int startms;
4            } simulationforce;
5
6            map<string,simulationforce> mapforces;
7            vector<string> lstforces;
8
9            map<string,simulationforce>::iterator itmap;
10           vector<string>::iterator itlst;
```

CPP code 11.6: *Dynamic Force Container internal data structure.*

11.3.2 Dynamic Factor Typologies

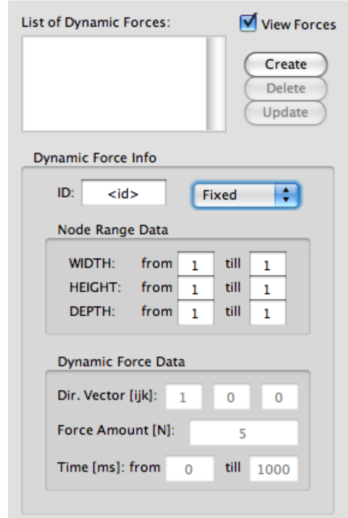
11.3.2.1 Dynamic Forces

That's the most common dynamical application offered by MeshInspeQTor. Indeed, the configuration panel has been built by thinking of this kind of success, because it permits to configure all the parameters. So, by specifying a nodeset range and force direction and amount with its own lifetime, it will be stored on the `ListOfDynamicForces` —sorted by starting instant time—:

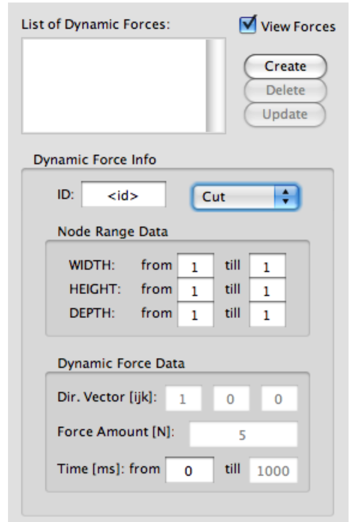
- the **`map`**`<`**`string`**`, simulationforce> mapforces` variable —line 6 of the previous code— stores the dynamic force data and its starting time —the ending time is computed by the starting time plus its duration— with **instant access** by using their identificator;
- the **`vector`**`<`**`string`**`> lstforces` —line 7— is always being resorted by any creation, delete or modification of dynamic forces, for achieving a **per-starting-time ordering of forces**; so, a sequential tracking of `lstforces` gives the dynamic forces in order of appearance.

11.3.2.2 Node Fixing

The node fixing is a kind of dynamical force that **avoids any position change for the mass-points included in the specified node set**. Because of this, the force amount and direction are not necessary parameters, as long as the lifetime specification, because in ForceReaQTor a mass-point is considered fixed **always** during a deformation.



(a) *Panel for 'fixing' forces; notice that only the node set range is available for being edited.*



(b) *Panel for 'cutting' forces; since a cutting is permanent once done, the starting lifetime is available.*

Figure 11.4: *The dynamic force configuration panels for 'fixing' and 'cutting' classes.*

11.3.2.3 Spring Cuttings

The last type of dynamical application is the **most complex** of the three: the spring cutting. It's really the most complex because the fixing avoid any modification, while the dynamic force changes the mass-point position as well as handles the system restrictions.

Nevertheless, the **spring cutting breaks the topology of the mass-spring system** because it cuts the neighbouring between two initially adjacent cells. Therefore, by **setting the spring —not node— range set, and specifying the starting instant time for the cutting action**, everything is done, the simulator will do the rest. There won't be any lifetime ending because in ForceReaQTor the cutting will be considered permanent.

11.4 The Deformation Simulator

11.4.1 A thread-powered module; the FPS imposed rating

ForceReaQTor executes the simulation of dynamic force application within a **parallel execution to the main application *thread*,** such that even the **mass-spring system deformation based on numerical integration is computed in parallel to the mass-spring's own renderization**.

```
void* run( void )
{
    renderscene = true;
    while( true )
    {
        if( !renderscene && timethrough_enabled )
        {
            clock_t clk = clock();
            ForceAccumulator();
            float lag = (clock()-clk) /
                        (float)CLOCKS_PER_SEC;
            NumericalIntegration( lag );

            counter += lag;
            timestep += lag;
        }
        else if( renderscene )
        {
            timestep = 0.0;
            renderscene = false;
        }
    }
}

// ForceReaQTor computational time execution
float getTimeCounter()  { return counter; }
void resetTimeStep()    { timestep = 0.0; }
float getTimeStep()     { return timestep; }
bool renderScene()      { if( renderscene ) return false;
                          renderscene = true;
                          NumericalIntegration_GL();
                          return true; }

// ForceSimulator execution control methods
void play( void )   { timethrough_enabled = true;  }
void pause( void )  { timethrough_enabled = false; }
void stop( void )   { timethrough_enabled = false;
                      counter = 0.0; timestep = 0.0; }
```

CPP code 11.7: *ForceSimulator main deformation thread execution.*

By using the `renderScene()` method —line 29—, ForceReaQTor permits a **controlled FPS renderization rate**; so, assuming, for example, a 20 FPS rate, it's known that the time period between two frames is $1000ms/20fps = 50ms/frame$. Then:

- thus, for each $50ms$ pass, a `renderScene()` calling has to occur from the main ForceReaQTor execution thread —that whose handles the GUI—; and, as can be seen in code's line 31, in this calling there's a mass-spring system integration rendering-dedicated (the `GL` postfix is describing that);
- if `renderScene()` returns true, at least one numerical integration step has been computed, so the new mass-spring system state matrix has to be renderized;
- so, if one calling returns *false* —see code for the possibility—, the simulation cannot be done in the specified framerate, and a message error must be thrown, and the simulation must be stopped.

all of that is resumed in the following code, **executed for each frame computational time**; there are some called methods not explained before, but the method name itself it's quite selfexplanatory, so it can be understood easily:

```
1  ForceSimulator* forcesimulator;
2  ListOfDynamicForces* listdynamicforces;
3  QGLScene* glscene; // opengl context for renderization
4
5  void newFrameRenderization()
6  {
7      float simulation_time =
8                      forcesimulator->getTimeCounter();
9      int ms_timelag = (int)(simulation_time*1000);
10     if( ms_timelag < simulation_timestep ) return;
11
12     float timelag = forcesimulator->getTimeStep();
13     vector<DynForce*> actualactiveforces =
14                 forcesimulator->getActiveDynForces();
15     vector<DynForce*> actualfixedforces =
16                 listdynamicforces.getFixedDynForces();
17     for( int k=0; k<(int)actualfixedforces.size(); k++ )
18         actualactiveforces.push_back( actualfixedforces[k] );
19     glscene->setListOfReactingDynForces( actualactiveforces );
20
21     vector<DynForce*> actualcutforces =
22                 simulator->getActiveCutForces();
23     glscene->activeDynamicCuts( actualcutforces );
24
25     if( !simulator->renderScene() )
26     {
27         forcesimulator->stop();
28         errorMessage();
29     }
30     else
31     {
32         glscene->updateDynamicSystemCoords();
33         glscene->renderScene();
34     }
35 }
```

CPP code 11.8: *ForceReaQTor per frame mass-spring renderization update.*

11.4.2 The detailed deformation algorithm

As can be seen in code 11.7, there are **two basic steps for computing physically-based deformations**:

- `ForceAccumulator` —line 9—, that computes the actual mass-spring system state matrix together with the current dynamical active forces —or cuttings— for **obtaining the restricting forces for the next integration step**.
- `NumericalIntegration` —line 12—, that gets the restricting forces computed by the previous function and executes the **numerical integration to a new mass-spring system state matrix**.

In the following pages, these two steps will be detailed in terms of algorithmic purposes. However, since `ForceAccumulator` is the most complex of the two steps, the explanation will begin with the other step, `NumericalIntegration`. But before focusing into the deformation, it will be useful to show the **internal structure that is handled by the `ForceSimulator` module**:

```
1   typedef struct {
2       int* hashing;            // same as DynamicVoxelMap
3       unsigned char* cuts;     //       "        "
4       bool* fixed;             //       "        "
5       int ncoords;             //       "        "
6       float* origin;           // the initial mass-point
7                                // set coordinates
8       float* cdprev;           // Verlet integrator's
9                                // previous coordinate set
10      float* vecvel;           // Euler integrator's
11                               // previous coordinate set
12      float* coords;           // actual mass-point
13                               // coordinate set
14      float* gl_coords;        // rendering mass-point
15                               // coordinate set
16      float* forces;           // actual retricting forces
17  } DynVoxelMapSpec_struct;
18  DynVoxelMapSpec_struct mapspec;
```

CPP code 11.9: *`ForceSimulator` internal handled data.*

11.4.3 The Numerical Integrators

11.4.3.1 Introduction to the integration algorithms

Roughly speaking, in ForceReaQTor's `ForceSimulator` module the numerical integrators are those whose modify the state matrix; and in `ForceSimulator`'s internal data —`DynVoxelMapSpec_struct`— it is:

- for the **Verlet integrator**: `coords` and `cdprev`;
- for the **Euler integrator**: `coords` and `vecvel`;

Additionally, there's a ***pseudo*-integrator for rendering purposes** thrown by calling `NumericalIntegration_GL()`, that uniquely transfers the `coords` data to `gl_coords` data —and `gl_coords` is the data block for the *next* rendered frame—.

Even, **the following algorithms for both Euler and Verlet numerical integrators supports a preprocessed acceleration** by a `validnodes` array, that includes, from all the possible nodes of the original voxelmap, not only the validated cells —that is, those that are mass-points too— but also those whose there aren't fixed.

11.4.3.2 Euler Integrator

```
void NumericalIntegration_Euler( float dt )
{
    for( int z=0; z<(int)validnodes.size(); z++ )
    {
        ValidNode nod = validnodes[z];
        for( int k=0; k<3; k++ )
        {
            mapspec.vecvel[nod.hash*3 + k] +=
                mapspec.forces[nod.hash*3 + k]*dt;
            mapspec.coords[nod.hash*3 + k] +=
                mapspec.vecvel[nod.hash*3 + k]*dt;
        }
    }
}
```

CPP code 11.10: *Euler numerical integration algorithm.*

11.4.3.3 Verlet Integrator

```
void NumericalIntegration_Verlet( float dt )
{
    for( int z=0; z<(int)validnodes.size(); z++ )
    {
        ValidNode nod = validnodes[z];
        float actual[3];
        for( int k=0; k<3; k++ )
        {
            actual[k] = mapspec.coords[nod.hash*3 + k];
            // the 1.89 and 0.89 floating values are
            // quasi-2 and quasi-1 respective values for
            // faster stability convergence.
            mapspec.coords[nod.hash*3 + k]  =
                1.89*mapspec.coords[nod.hash*3 + k] -
                0.89*mapspec.cdprev[nod.hash*3 + k] +
                mapspec.forces[nod.hash*3 + k]*dt*dt;
            mapspec.cdprev[nod.hash*3 + k]  = actual[k];
        }
    }
}
```

CPP code 11.11: *Verlet numerical integration algorithm.*

11.4.4 Force Accumulator

11.4.4.1 Introduction to Force Accumulation

The complex part of deformation algorithms will be now featured. The **force accumulators** are, in ForceReaQTor, the **total amount of forces applied to the mass-spring system, given an instant time**.

▷ ***Cloth 2D internal restrictive forces, applied to volumetric deformable systems***

The following featured algorithms will compute the **same restrictive deformation forces as applied for cloth simulation**, as explained in section 6.3.1.4, on page 65: *stretch*, *shear* and *bending*, but **on a 3D scheme instead of the basic 2D one**.

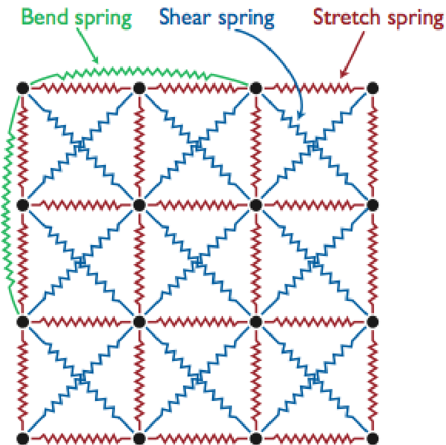


Figure 11.5: *The ForceReaQTor implemented restricting forces, in a 2D scheme.*

Besides, all the restricting force applications have been **optimized by using the preprocessed `validnodes`** data array, so therefore, *voxels* without their correspondent mass-point won't be accessed ever.

There are **another optimizations** available and ready to be supported, like the following two, closely related to the simetric restrictive force application to opposite and affected each together mass-points

- since the internal force between two nodes is computed with viscoelastic properties, and all the mass-spring system is mass-regular, it can be deduced that given two nodes n_a and n_b, and describing $\vec{F}_{ab}$ as the vectorial-component force between n_a and n_b but from the n_a point of view:

$$F_{ab} = F_{ba} \iff \vec{F}_{ab} = -\vec{F}_{ba}$$

  so calculating the force between two nodes once and giving to the other node the opposite value, the **number of processings is divided by two**;

- moreover, it's ensured a subsequent and logical force accumulator processing, so the same force —or its opposite— will **never** be computed more than once;

▷ ***The great computational handicap: the number of mass-points***

The two previously mentioned optimizations —no force calculus repetition and application of the simmetric forces— are a good chance for being implemented.

However, because of the **need of applying an important optimization about the number of mass-points** that are composing the deformable system, **no one of two can be applied** because a **force assimetry conflict** could surely be created with this new optimization.

The mass-point deformable system has a **terrible handicap** w.r.t. the computational cost of applying a force accumulator, and it's relatively independent from the dynamic effects processing: **the cardinality of those mass-points is the key for having a fluid animation**, or on the other side, falling into a chaotic system. Just as easy example, imagine a mass-point system that only occupes the $1/8$ of the entire voxelmap; **depending on the voxelmap grid size, the number of mass-points will increase brutally**:

VoxelMap Grid Size	Total Number of Voxels	$1/8$ of total
$16 \times 16 \times 16$	4.096	512
$32 \times 32 \times 32$	32.768	4.096
$64 \times 64 \times 64$	262.144	32.768
$128 \times 128 \times 128$	2.097.152	262.144

Table 11.1: *Exemplification of the brutal increasing of mass-point cardinality on a $1/8$-occuping deformable system by only changing the voxelmap grid size.*

So, it's easily deducible that **the more the voxel grid size increases, the more the number of force accumulator iterations will be dramatically reduced**, so the system will be able to become chaotic or unstable with facility. For solving that, there are to be explained some new concepts; the solution will be provided later, on future section 11.8.

▷ ***The algorithm for the Force Accumulator***

Keeping in mind this computational problem of the deformable system cardinality, let's go with the **algorithm for the Force Accumulator**, thatvcan be resumed on **two steps**:

1. setting the **time step local forces to null** —not the global mass-point set applied forces, only the time step relative—, and then **applying the external forces**, if case;
2. **processing the internal restrictive forces** between system adjacent mass-points for avoiding the deformation provoques by the —actual or not— external forces[82]

```
1  void ForceAccumulator()
2  {
3      setInitialForces();     // setting new mass-spring
4                              // system state matrix to an
5                              // initial zero-applied-force.
6      applyActualForces();    // external force application.
7
8      // restrictive internal force application; due to
9      // equivalent algorithmic behaviour, 'stretch' and
10     // 'bend' effects are supported by the same method.
11     if( stretch )   AccumStretchBending( ACCUM_STRETCH );
12     if( shear )     AccumShear();
13     if( bend )      AccumStretchBending( ACCUM_BENDING );
14 }
```

CPP **code 11.12:** *Force Accumulator sequential force application.*

[82] If a force in the past has modified the position of a mass-point, there's still a restrictive force that will try to return the cited mass-point to its original location. Until not achieving its relaxin position, there will be restrictive internal forces applied over it.

Additionally, here will be shown the code for calculating the **viscoelastic spring restrictive force** between two nodes —pointed by their respective *internal hashing* indexes—, as well as **the respective applying force** typification —*stretch*, *shear* or *bending*—.

By reminding the **spring coeficient restrictive force between two nodes** —featured on section 6.3.1.4, on page 65—, the acting force on a mass point i generated by a spring connection with another mass point j is

$$f_i = k_s\big(|x_{ij}| - l_{ij}\big) \cdot \frac{x_{ij}}{|x_{ij}|}, \quad f_j = -f_i$$

where l_{ij} is the reference —initial, relaxing— spring length. In ForceReaQTor, this l_{ij} parameter is called L; and this **fixed reference length is directly proportional to the applying force** —see figure 11.5 on the previous page— as can be seen in the following lines 17..22:

```
1  vector<float> getSpringCoefAccum( int hash_ini,
2                                    int hash_end,
3                                    int dynamicmode )
4  {
5      float Ks = m_param_elasticity;
6      float Kd = m_param_dampening;
7      float XX[3], VV[3];              // dir. & vel. vectors
8      float mX, L, coef, mXmL;         // spring eq. params
9      for( int k=0; k<3; k++ )
10     {
11         XX[k] = mapspec.coords[hash_ini*3+k] -
12                 mapspec.coords[hash_end*3+k];
13         VV[k] = mapspec.vecvel[hash_ini*3+k] -
14                 mapspec.vecvel[hash_end*3+k];
15     }
16
17     switch( dynamicmode )
18     {
19         case ACCUM_STRETCH: L = m_voxellength;   break;
20         case ACCUM_SHEAR:   L = m_diagMajor;     break;
21         case ACCUM_BENDING: L = 2*m_voxellength; break;
22     }
23
24     // offsetEqual returns true if mX is within
25     // [L - L*0.1, L + L*0.1] range. This is for
26     // converging to the relax avoiding numerical
27     // integration accumulation errors.
28     mX = modulevector( XX );
29     if( offsetEqual(mX, L, 0.1 )     mXmL = 0.0;
30     else                             mXmL = mX - L;
31
32     normalizevector( XX, XX );
33     coef = (Ks*mXmL) + (Kd*dotprod(VV,XX));
34
35     vector<float> ret;
36     ret.push_back( -coef*XX[0] );
37     ret.push_back( -coef*XX[1] );
38     ret.push_back( -coef*XX[2] );
39     return ret;
40 }
```

CPP code 11.13: *Per-spring Force Accumulator calculus.*

11.4.4.2 Stretching and Bending

As said previously, the *stretch* and *bend* dynamical effects are supported in ForceReaQTor by the same algorithm, because *stretch* is the **longitudinal force between neighbour nodes** while *bend* is the 2nd **order longitudinal force between two nodes** —that is, there is an existint, neighbour of both implied mass points, between them—.

The `processNeighbourForces` method is, then, the basis of *stretch* and *bend* effect application to this node; this method receives the reference mass point and the specified dynamical effect —*stretch* or *bend*— and **computes every longitudinal specified force between the reference mass-points and its existing** 1st **or** 2nd **order neighbours** —depending on the applied dynamical effect—.

```cpp
#define ACCUM_STRETCH   0
#define ACCUM_BENDING   1

vector<float> processNeighbourForces( int i, int j, int k,
                                      int dynamicmode )
{
    int step = 1+dynamicmode;
    vector<float> coef;
    for( int x=0; x<3; x++ ) coef.push_back( 0.0 );
    vector<float> tmp;
    int NX = m_nvoxels;
    int NY = m_nvoxels;
    int idx = ( (i*NY) + j )*NX + k;
    int hash_idx = mapspec.hashing[idx];

    // application of mutual restrictive neighbouring forces
    int neighbour[3];
    neighbour[0] = i+step<NX ? i+step : -1; // horizontal
    neighbour[1] = j+step<NX ? j+step : -1; // vertical
    neighbour[2] = k+step<NX ? k+step : -1; // frontal
    int neighidx[3];
    neighidx[0] = ( ((i+step)*NY) + j        )*NX + k;
    neighidx[1] = ( (i*NY)        + (j+step) )*NX + k;
    neighidx[2] = ( (i*NY)        + j        )*NX + (k+step);

    for( int x=0; x<3; x++ )
    {
        if( neighbour[x] == -1 ) continue;

        int hash_idx_NEIGH = mapspec.hashing[neighidx[x]];
        tmp = getSpringCoefAccum( hash_idx,
                                  hash_idx_NEIGH,
                                  dynamicmode );
        for( int z=0; z<3; z++ )
        {
            coef[z] += tmp[z];
            mapspec.forces[3*hash_idx_NEIGH+z] -= tmp[z];
        }
    }

    return coef;
}
```

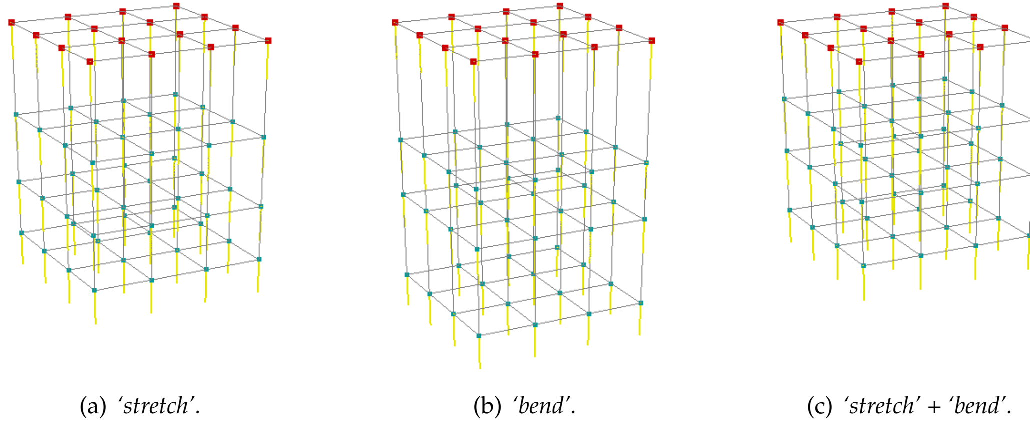
CPP code 11.14: *Longitudinal restrictive force application.*

Therefore, once the n-th order neighbour longitudinal restrictive force processing has been specified for a certain mass-point described by its (i, j, k) voxelmap indexes, the **basic scheme of this *stretch* and *bend* application is a simple tracking over the valid nodes**, as shown on the following code:

```
void AccumStretchBending( int accumulator_mode )
{
    vector<float> coef;
    for( int z=0; z<(int)validnodes.size();
                                    z+=1+accumulator_mode )
    {
        ValidNode nod = validnodes[z];
        coef = processNeighbourForces( nod.i, nod.j, nod.k,
                                       accumulator_mode );

        mapspec.forces[nod.hash*3+0] += coef[0];
        mapspec.forces[nod.hash*3+1] += coef[1];
        mapspec.forces[nod.hash*3+2] += coef[2];
    }
}
```

CPP code 11.15: *Stretch and Bend effect application algorithm.*

At the following figures there are featured **three mass-spring deformations of a $4 \times 4 \times 4$ cube by applying the recently explained deformation ForceReaQTor algorithms** when gravity deforms the cubic mass-spring system.



(a) *'stretch'*. (b) *'bend'*. (c) *'stretch' + 'bend'*.

Figure 11.6: *ForceReaQTor applying 'stretch' and 'bend' deformation effects to a $4 \times 4 \times 4$ cube after fixing the upper vertices and applying gravity force; notice that the combined deformation offers more equilibred results.*

11.4.4.3 Shearing

This is a more complex effect, because **it affects a non-spring direct deformation**: *shear* is a restrictive internal **force between a node and its cube-diagonal-opposites**.

Because of that, the previously detailed method `processNeighbourForces` cannot be used here; so, there's another method, called `processDiagonalForces`, that receives the mass-point reference in terms of its original location within the imported voxelmap.

However, the code of this function will not be shown because **the methodology is the same that for the *stretch* variant within `processNeighbourForces`, but linking every node with the eight cube-diagonal-opposite ones, not the six spring-connectec mass-points**.

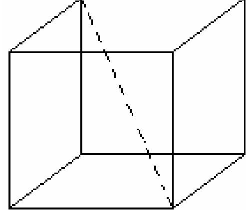


Figure 11.7: *One of the four diagonals of a single cube.*

As seen in the figure above, the only important thing to be careful with is that **now the reference distance won't be the voxel edge length but the major diagonal**[83] **of the cube**, as mentioned before. So, the code for the linear tracking of *shear* application will be shown for a better understanding of the *shear* linear tracking:

```
1  void AccumShear()
2  {
3      // the voxel's diagonal between the lower and greater
4      // vertices are computed here and stored on the local
5      // variable m_diagMajor, used later in the final
6      // getSpringCoefAccumulator(...) method.
7      float minordiag = (m_inty*m_inty) + (m_intz*m_intz);
8      m_diagMajor = sqrt( (m_intx*m_intx) + minordiag );
9
10     vector<float> coef;
11     for( int z=0; z<(int)validnodes.size(); z++ )
12     {
13         ValidNode nod = validnodes[z];
14         coef = processDiagonalForces( nod.i, nod.j, nod.k );
15
16         mapspec.forces[nod.hash*3+0] += coef[0];
17         mapspec.forces[nod.hash*3+1] += coef[1];
18         mapspec.forces[nod.hash*3+2] += coef[2];
19     }
20 }
```

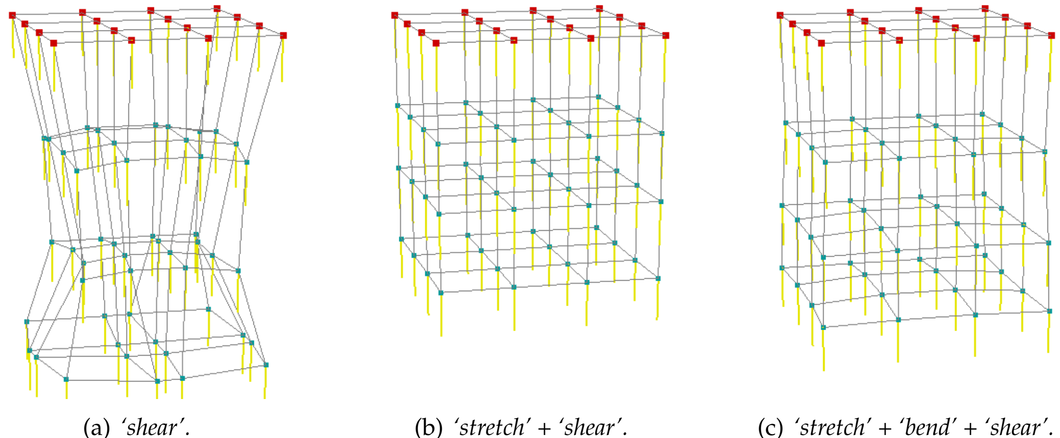
CPP code 11.16: *Shear effect application algorithm.*

The figures shown in the next page features the effect of *shear* applied to the same $4 \times 4 \times 4$ cube of the previous section, and also two more combinations: the middle one is a *shear* effect applied with *stretch*[84], and the right one is the mix of all the restrictive forces —*stretch*, *bend* and also the new one, *shear*—[85].

[83] The minor diagonal of a cube will be the diagonal of a cube's face, so the major diagonal covers the volume of the cube instead of the face area in the case of the minor one.

[84] Indeed, the *stretch* dynamical restriction force effect is the basic spring deformation application; with no *stretching*, it's near to impossible to obtain a realistic behavior of a deformation animation.

[85] The deformation conditions, of course, are also the same: the upper mass points of the cube has been fixed while the unique external force inciding the model is the gravity.



(a) *'shear'.* (b) *'stretch' + 'shear'.* (c) *'stretch' + 'bend' + 'shear'.*

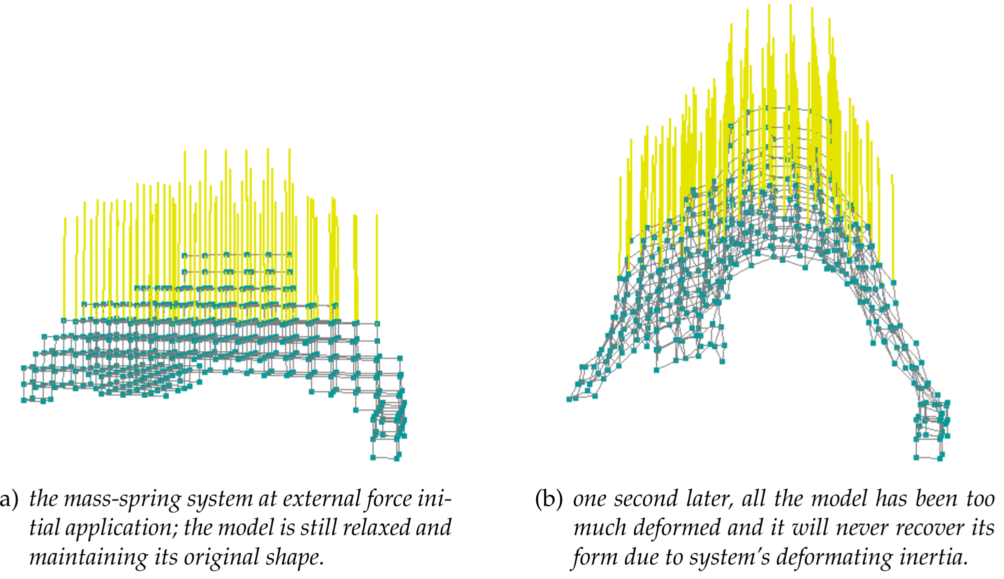
Figure 11.8: *A* $4 \times 4 \times 4$ *cube mass-spring system where it's applied 'shear' deformation forces together with some 'stretch' and 'bend' factors after applying gravity to the deformation model.*

11.5 Volume Preservation

11.5.1 The need for volume preservation

For achieving the cubical mass-spring system deformation in the figures 11.6 and 11.8, i has been said that, apart from applying gravity force, the upper vertices were fixed. The reason of doing this is that, **though any model is initially relaxed and maintaining its shape, when an external force is applied to at least one mass-point, a chain reaction begins and all the model loses its original form due to the imposed deformation**.

This problem can be easily seen in the following figure, featuring an initially undeformed mass-spring system of a dolphin voxelmap, and only 1 seconds after applying a external $10N$ elevating force to its dorsal fin.



(a) *the mass-spring system at external force initial application; the model is still relaxed and maintaining its original shape.*

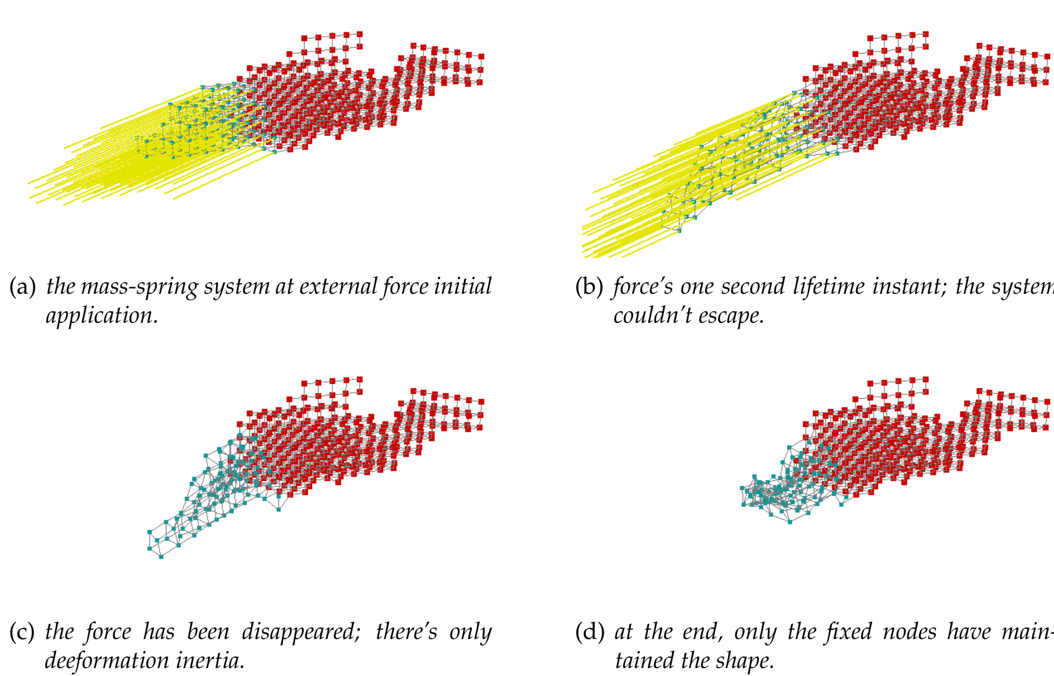
(b) *one second later, all the model has been too much deformed and it will never recover its form due to system's deformating inertia.*

Figure 11.9: *The shape/volume preserving example problem.*

11.5.2 Fixing nodes for volume preservation

Once exposed the problem, a **possible solution** would be to **specify some mass points to be fixed for ensuring that not all the system will be deformed**. However, it implies:

- a **post process**, non automatic —a priori—, that requires to **fix some system regular spaced mass-points**;
- the advantage of this technique is also its drawback: these mass points will never be able to be moved, so **the deformation can result in a strange behaviour** —or even provoquing system unstabilities—;
- the **deformation will be restricted in the volume between fixed mass-points, but also unhandled**, so the will be some free shapeless deformation zones while another zones will be untouched.



(a) *the mass-spring system at external force initial application.*

(b) *force's one second lifetime instant; the system couldn't escape.*

(c) *the force has been disappeared; there's only deeformation inertia.*

(d) *at the end, only the fixed nodes have maintained the shape.*

Figure 11.10: *Fixing nodes as volume preservation solution example.*

So much little amount of advantages in front of too much handicaps to be solved. At the figure above there's an example of why **the problem is not avoided by fixing nodes** —though, the fixing process is vast for growing up the drawback sensation— by applying an external force to another dolphin's mouth, by fixing its body. **Even, though achieving a total shapeless avoiding by fixing nodes, the deformation will never become realistic**.

11.5.3 ForceReaQTor's adopted solution: the *autostretching*

ForceReaQTor adopts a **fast, good, ensure and fiable solution for maintaining and preserving the original shape as long as also its volume**. The techniue consists basically in generating **invisible springs between a *phantom* model, undeformable and equivalent to the initially relaxing mass-spring system, and the corresponding points to the deformable model**.

This methodology, that in ForceReaQTor receives the name of ***autostretching***, and it offers the following **advantages**:

- an automatically handled deformation wit **no losing the shape even in the deformation itself**;
- **no other postprocesses** for avoiding undesired behaviours;
- all the system is deformed in a **self-managed way**;

however, there's some **limitations** that must be taken in account:

- the **deformation is restricted** in terms of *distance* with respect to the original mass point position;
- **not all kind of deformations are supported**; extreme bending deformations won't be possible due to the *autostretch* damping;
- the mass-spring system can't be displaced, even the displacement would be really desired, so **this preservation avoids controlled translations**.

```
1  vector<float> getSpringCoefAutoAccum( int hash_idx )
2  {
3      float Ks = m_param_elasticity;
4      float Kd = m_param_dampening;
5      float XX[3], VV[3];              // dir. & vel. vectors
6      float mX, L, coef, mXmL;         // spring eq. params
7
8      // now the XX value is the vector between the actual
9      // position and the phantom relaxing coordinates;
10     // also, the velocity is simply the actual, because
11     // the phantom model has no inherent velocity.
12     for( int k=0; k<3; k++ )
13     {
14         XX[k] = mapspec.coords[hash_idx*3+k] -
15                 mapspec.origin[hash_idx*3+k];
16         VV[k] = mapspec.vecvel[hash_idx*3+k];
17     }
18
19     // offsetEqual returns true if mX is within
20     // [L - L*0.1, L + L*0.1] range. This is for
21     // converging to the relax avoiding numerical
22     // integration accumulation errors.
23     mX = modulevector( XX );
24     if( offsetEqual(mX, L, 0.1 )     mXmL = 0.0;
25     else                             mXmL = mX - L;
26
27     normalizevector( XX, XX );
28     coef = (Ks*mXmL) + (Kd*dotprod(VV,XX));
29
30     vector<float> ret;
31     ret.push_back( -coef*XX[0] );
32     ret.push_back( -coef*XX[1] );
33     ret.push_back( -coef*XX[2] );
34     return ret;
35 }
```

CPP code 11.17: *Per-phantom-spring Force Accumulator calculus.*

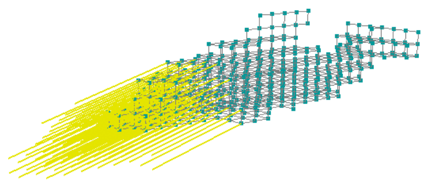
Notice that the main difference between this spring coeficient force autoaccumulator —between a mass-point and its phantom corresponding one— and the standard coeficient accumulator of the code 11.13 is that only one mass-point index is needed, and **the position as well as the velocity vectors are computed w.r.t. the mass-point itself**.

However, the linear application tracking is the same that for the standard *stretch* and *bend* effects, as can be seen in the next code —although, **pay attention on the heavyless constant `AUTOSTRETCH_DAMPING` value for no so restrictive forces as with the neighborhood longitudinal restrictions**—:

```
1  #define AUTOSTRETCH_DAMPING 0.25
2
3  void AccumAutoStretch()
4  {
5      vector<float> coef;
6      for( int z=0; z<(int)validnodes.size(); z++ )
7      {
8          ValidNode nod = validnodes[z];
9          coef = getSpringCoefAutoAccum( nod.hash );
10         for( int x=0; x<3; x++ )
11             mapspec.forces[nod.hash*3+x] +=
12                     coef[x] * AUTOSTRETCH_DAMPENING;
13     }
14 }
```

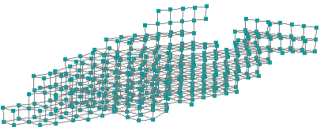
CPP code 11.18: *AutoStretching effect application algorithm.*

Here are the same frameset of the figure **??** but now applying *autostretching* and eliminating all the fixed mass-points. As it can be seen, the deformation is perfectly handled and no chaotic behaviour appears: **when the external force disappears, the mass-spring system returns perfectly to its relaxing shape**.

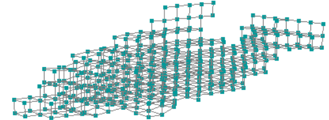


(a) *the mass-spring system at external force initial application.*

(b) *force's one second lifetime instant; the system deformation is more restricted but also more well-handled.*



(c) *the force has been disappeared; there's only deformation inertia but the* autostretching *yields the original system shape.*



(d) *at the end, the mass-spring system shape has been preserved.*

Figure 11.11: Autostretching *as volume preservation solution example.*

11.6 Collision handling

11.6.1 The need of collision handling during deformation

Till now, it has been explained that ForceReaQTor is able to manage several types of dynamic real-time deformation while the shape and volume of the mass-spring system is always preserved. However, there have been no words about **collision handling**. ForceReaQTor has adopted an **adaptive strategy that allows collision detection and arranging in run-time**.

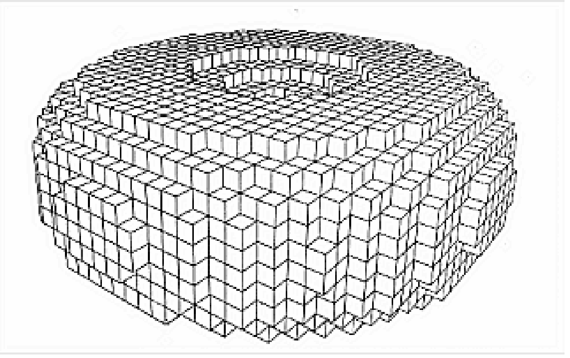
In a three-dimensional space where the physical laws have to lead the movements of solid viscoelastic models, the collision detection and management is **necessary for a correct behaviour of untrespassable elements**; so, collision handling is a must-have algorithm section **useful to avoid**:

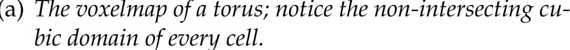
- **penetrations** between different mass-spring systems;
- system **self-crashings** by some mass-points going through the location of other mass-points;

11.6.2 The ForceReaQTor approach: spheres approximating cubes

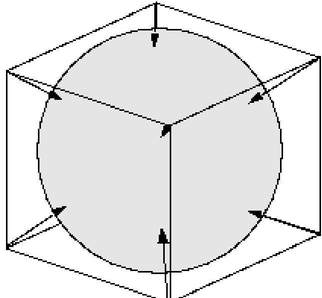
11.6.2.1 Introduction

ForceReaQTor imports voxelmaps and transforms them onto mass-spring systems. Every mass-spring —voxelmap cell, voxel—, then, has its own domain, non-intersecting with another mass-point domain: it has the shape of a cube. However, ForceReaQTor uses a **cube approxiomation in form of cube-concentric spheres** due to geometric reasons: a sphere is a unique geometrical object, while a cube is composed of six elements —six squares, or delimited planes—.





(a) *The voxelmap of a torus; notice the non-intersecting cubic domain of every cell.*



(b) *The sphere approximation of a cube for fast collision handling.*

Figure 11.12: *Scheme of the ForceReaQTor mass-point volumetric approximation buy using a sphere instead of a cube. Obviously, the sphere and the cube has the same center, and moreover, the sphere is tangent to all the cube faces.*

The **sphere diameter**, a priori, is the same as the *voxel length* —and since the cubes are regular-sized, it is **equivalent to the face edge length**—, so the sphere will be tangent to the cube's faces.

It's important to take in account the following indication: **the collision testing as well as collision processing —in case of a passing test— must be called both two within the numerical integrator, after the integration itself —for possible integration arrangements—**. If not, the collision handling won't be treate correctly.

11.6.2.2 Collision testing

Therefore, given two spheres S_1 and S_2, both two with the same diameter D, there is a collision between them if the euclidean distance between the two centers c_1 and c_2 is lower or equal than D, as is shown in the next figure:

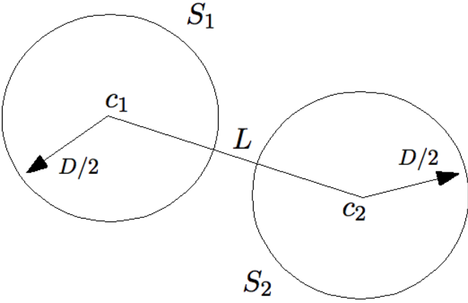


Figure 11.13: *Scheme of sphere intersection; if $||c_2 - c_1|| \equiv L \leq D$, it must be ensured that the two spheres S_1 and S_2 are intersecting.*

There's no matter with the unused cube space because although it has been said that the sphere diameter was the voxel edge length, **for avoiding continuous collision detections the desired sphere diameter will be** $L * 3/5$, where L is the voxel edge length. So, given a reference mass-point —`nod`— and a possible colliding mass-point, here's the code for their collision testing, returning also the distance between the two mass-points:

```
bool isMassPointColliding( ValidNode nod,
                           vector<int> voxeldest,
                           int hashdest,
                           float* distance )
{
    if( voxeldest[0] < 0 || voxeldest[0] >= m_nx ||
        voxeldest[1] < 0 || voxeldest[1] >= m_ny ||
        voxeldest[2] < 0 || voxeldest[2] >= m_nz )
        return false;

    // collision test
    float orig[3], dest[3], diff[3];
    for( int z=0; z<3; z++ )
    {
        orig[z] = mapspec.coords[nod.hash*3 + z];
        dest[z] = mapspec.coords[hashdest*3 + z];
        diff[z] = dest[z] - orig[z];
    }
    distance = MathUtils::modulevector( diff );
    if( length >= m_intx*3.0/5.0 ) return false;
    return true;
}
```

CPP code 11.19: *Collision testing between two mass-points.*

11.6.2.3 Collision processing

Within ForceReaQTor, it's called **collision processing** the **geometrical modification of a mass-point state vector due to a collision violation in terms of another mass-point(s)**. So, the collision processing will have to:

- change the newly integrated position;
- update the velocity vector component —only in Euler integration case—;

hence, since there are two different state matrices available in the deformation engine, it's deducible that there will be **two collision processing algorithms**: *(i)* one for the Euler integrator state matrix and *(ii)* another for the Verlet one.

The **state vector arrangements** processed by ForceReaQTor are purely geometric, so, attaching the position coordinates handling, and given the distance D between the two mass-points:

1. the reference mass-point will go back —understanding *back* as the opposite direction of the reference mass-point— $D * 2/5$ from its actual position;
2. the collision victim will go forward $D * 3/5$ from its position before colliding with the reference mass-point;

moreover, the rest of the **state vector parameters will must be modified for maintaining the new direction** from this moment.

▷ ***Verlet collision management***

```
void computeCollision_VERLET( ValidNode nod,
                              float hashdest,
                              float dt, float dist )
{
    float orig[3], dest[3], diff[3];
    for( int z=0; z<3; z++ )
    {
        orig[z] = mapspec.coords[nod.hash*3 + z];
        dest[z] = mapspec.coords[hashdest*3 + z];
        diff[z] = dest[z] - orig[z];
    }
    MathUtils::normalizevector( diff, diff );

    // Verlet state vector management
    for( int z=0; z<3; z++ )
    {
        mapspec.cdprev[nod.hash*3 + z] =
            mapspec.coords[nod.hash*3 + z];
        mapspec.cdprev[hashdest*3 + z] =
            mapspec.coords[hashdest*3 + z];

        mapspec.coords[nod.hash*3 + z] -=
            (dist*2.0/5.0)*diff[z];
        mapspec.coords[hashdest*3 + z] +=
            (dist*3.0/5.0)*diff[z];
    }
}
```

CPP code 11.20: *Euler collision handling function*

▷ ***Euler collision management***

```
1  void computeCollision_EULER( ValidNode nod,
2                               float hashdest,
3                               float dt, float dist )
4  {
5      float orig[3], dest[3], diff[3];
6      for( int z=0; z<3; z++ )
7      {
8          orig[z] = mapspec.coords[nod.hash*3 + z];
9          dest[z] = mapspec.coords[hashdest*3 + z];
10         diff[z] = dest[z] - orig[z];
11     }
12     MathUtils::normalizevector( diff, diff );
13
14     // Euler state vector management
15     for( int z=0; z<3; z++ )
16     {
17         mapspec.coords[nod.hash*3 + z] -=
18             (dist*2.0/5.0)*diff[z];
19         mapspec.coords[hashdest*3 + z] +=
20             (dist*3.0/5.0)*diff[z];
21
22         mapspec.vecvel[nod.hash*3 + z] =
23             -refdist*diff[z]/dt;
24         mapspec.vecvel[hashdest*3 + z] =
25              coldist*diff[z]/dt;
26     }
27 }
```

CPP **code 11.21:** *Euler collision handling function.*

11.6.2.4 Adaptive acceleration for collision handling

It can be deduced that, **for every numerical integration, the collision testing will be executed six times per mass-point system**; and although it's a fast testing, the required computational time is expensive, so **an adaptive testing has been developed** to avoid testing impossible collisions.

This adaptation is based on predicting, from every mass-point moving direction, which neighbour mass-point can be object of collision; in other words, **solving which mass-points have to be tested for a possible collision, by knowing the physical moving direction of the reference one**.

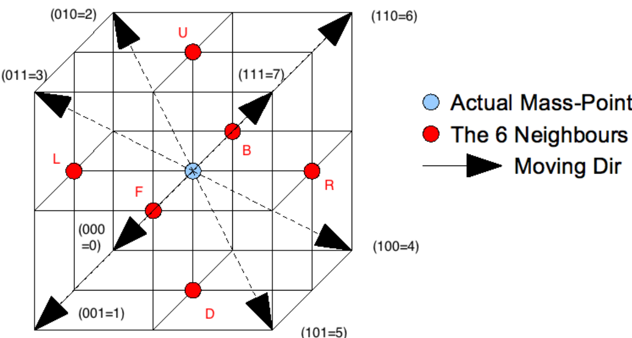


Figure 11.14: *Scheme of the per-mass-point adapting collision testing.*

Finally, once established the adaptive selection, let's go with the codes; pay attention on that the **moving direction is computed by different ways, depending on the selected integrator**:

- from velocity vector if there is an Euler integrator,
- from the vector difference between actual and previous position in the Verlet one.

so **there are two algorithms** for the adaptive selection, one per each integrator[86].

```
vector<int> getAdaptiveCollidingIdx_EULER( ValidNode nod )
{
    float signing[3];
    signing[0] = (  mapspec.vecvel[nod.hash*3+0] /
                    abs(mapspec.vecvel[nod.hash*3+0]) );
    signing[1] = (  mapspec.vecvel[nod.hash*3+1] /
                    abs(mapspec.vecvel[nod.hash*3+1]) );
    signing[2] = (  mapspec.vecvel[nod.hash*3+2] /
                    abs(mapspec.vecvel[nod.hash*3+2]) );

    vector<int> ret;
    ret.push_back( signing[0] );
    ret.push_back( signing[1] );
    ret.push_back( signing[2] );
    return ret;
}
```

CPP code 11.22: *Euler's adaptive neighbour selection for collision testing.*

Now take in acount the change in `signing` variable proceedings, since the Verlet integrator is different than the Euler one, so the moving direction is obtained also by a different way.

```
vector<int> getAdaptiveCollidingIdx_VERLET( ValidNode nod )
{
    float signing[3];
    signing[0] = (  mapspec.coords[nod.hash*3+0] -
                    mapspec.cdprev[nod.hash*3+0] );
    signing[1] = (  mapspec.coords[nod.hash*3+1] -
                    mapspec.cdprev[nod.hash*3+1] );
    signing[2] = (  mapspec.coords[nod.hash*3+2] -
                    mapspec.cdprev[nod.hash*3+2] );

    signing[0] /= abs(signing[0]);
    signing[1] /= abs(signing[1]);
    signing[2] /= abs(signing[2]);

    vector<int> ret;
    ret.push_back( signing[0] );
    ret.push_back( signing[1] );
    ret.push_back( signing[2] );
    return ret;
}
```

CPP code 11.23: *Verlet's adaptive neighbour selection for collision testing.*

[86] Due to computational indexing requirements, these codes returns the figure 11.14 binary codification of the direction identifier in a 3-element vector; however, instead of 0 and 1 values, the methods will return -1 and $+1$, since the neighbour indexing from the mass-point (i, j, k) is done by adding or substracting 1.

In the following table there's the selected neighbours to be tested, in relation to the reference mass-point moving direction; Just like this, the **number of testings is divided by 2**, because the three neighbours located *rear* the mass-point won't be tested:

Binary Code	Decimal Code	Selected Neighbours
000	0	L D B
001	1	L D F
010	2	L U B
011	3	L U F
100	4	R D B
101	5	R D F
110	6	R U B
111	7	R U F

Table 11.2: *Adaptive neighbour selection for collision testing w.r.t. a mass-point moving direction. The 'selected neighbours' idetificators are corresponding to the featured ones in the figure 11.14.*

11.7 Mass-Spring System Cuttings

11.7.1 Introduction to the spring cutting methodology

On the chapter 7, on page 75, the mesh deformation is focused in its **topolgy alteration**, and the three types of alteration have been described: *(i)* mesh cracking with connectivity preservation, *(ii)* mesh splitting and *(iii)* continuous remeshing. However, almost every technique for simulating a cutting onto a three-dimensional model **requires a lot of processing time**, because the major amount of these techniques focus their efforces on cutting the model's triangular mesh.

Nevertheless, ForceReaQTor implements a **cutting algorithm focused on the mass-spring system connection between the nodes**; in other words, when a cut is applied to the system, the springs are the objective. So, an **internal data structure is needed for cutting management**, and it has been described on the code 11.1:

```
1  #define CUT_NOCUTS          0x00
2  #define CUT_UP              0x01
3  #define CUT_DOWN            0x02
4  #define CUT_LEFT            0x04
5  #define CUT_RIGHT           0x08
6  #define CUT_FRONT           0x10
7  #define CUT_BACK            0x20
8
9  typedef struct {
10     int* hash;              // acceleration hash-access
11                             // data for masspoints.
12     unsigned char* cuts;    // stores the possible
13                             // spring cuttings.
14     [...]
15 } DynVoxelMapData_struct;
16 DynVoxelMapData_struct dynvoxelmap;
```

CPP code 11.24: *Internal data structure for spring cutting handling.*

This cuts array, indexed by the hash acceleration data block as usual, is **mass-point related**, so when a spring is cut, two cut state values have to be modified —the two spring mass-point extremes[87]—. Moreover, the value assignment is controlled by the **seven cutting definitions**, six for every possible spring cutting —a masspoint is able to have till six neighbour— and another macro for no-cutting starting state, comprising both them **bit values that can be combined for generating a cutting byte state**. So, the data setting and accessing methods are:

- **adding** spring *up* and *right* cuttings to a mass-point actual cutting state:

```
dynvoxelmap.cuts[hash_idx] |= CUT_UP | CUT_RIGHT;
```

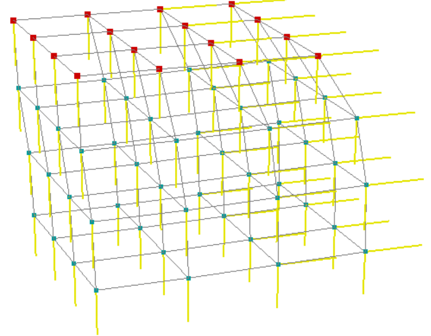
- **checking** if a certain mass-point has its *front* or *back* springs on cut state:

```
if( (mapspec.cuts[hash_idx] & DYNCUT_FRONT) ||
    (mapspec.cuts[hash_idx] & DYNCUT_BACK) )
{   [...]   }
```

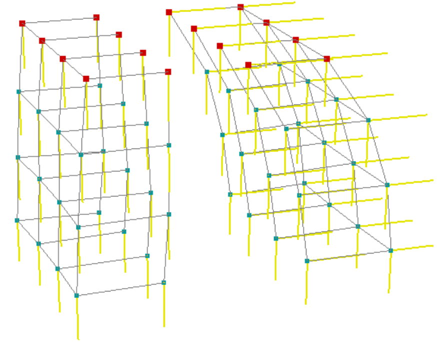
11.7.2 Repercussions on physically-based deformations

As can be noticed, **when a model is totally or partially cut**, not only its shape is modified; **its behaviour under deformation forces is also subject of changes**. ForceReaQTor supports real-time deformations over a model that it's being cut and dissected, by **truncating not only the spring connections between the isolated mass-points but also its restricting forces**.

So, **the application checks if there is a cut action between two mass-points before applying any of the dynamical deformation effects —*stretch*, *shear* or *bend*— where both two are involved**, by the checking command explained above. Let's see then the following snapshots of the example $4 \times 4 \times 4$ cube deformation when gravity together with a pulling external force is applied over it, and then the springs of the middle part of the cube disappear.



(a) *Deformation with the entire cube. The pulling force is done over the 'right' part of the cube but it affects the entire system.*



(b) *Deformation after cutting the middle 'column' springs; notice that the pulling force is only affecting the right piece of the cube.*

Figure 11.15: *Physically-based deformation with real-time cutting behaviour modification.*

[87] Two mass-points, m_L on the left side and m_R on the right one. If the spring that connect them is cut, m_L must active its CUT_RIGHT flag, and m_R its CUT_LEFT one.

11.8 Accelerated Force Accumulator looping

On the previous section 11.4.4.1, it has been said that there's a huge handicap within the force accumulator, dramatically associated to the number of mass-points that are comprising the deformable system. Now, after explaining the volume preservation —and the *autostretching* concept— and the collision handling, it's the moment to talk about the `SimAcc` data structure, acronym of *Simulation Accelerator*:

```
typedef struct {
    bool* affected;
    bool refval;
} SimAcc;
SimAcc simacc;

simacc.affected = new bool[nvoxels*nvoxels*nvoxels*];
simacc.refval = true;
```

CPP code 11.25: *The data structure capable of accelerating the force accumulator loopings.*

The idea behind this structure is so simple; having an *affected* value per each voxel, **this boolean array will be accessed by the already explained *hashing* value for the mass-spring system**. Depending on the the hashing *affected* value:

- if it's ***true***, the mass-point with that hashing code must be passed to Force Accumulator because there's at least one force —internal or external, it doesn't matter— affecting it, so it must be processed;
- if it's ***false***, the mass-point won't be affected by any system force, so computing a Force Accumulator over it is unuseful;

It's a so easy strategy, and **the obtained acceleration is superb** by following this methodology for *enabling* and *disabling* a mass-point's *affected* state:

```
{ at the simulation beginning, all 'affected' flags
  must be are set to 'false'. }
function ForceAccumulator( float simulation_time )
    { set to 'true' the 'affected' flag of those
      mass-points affected by the external forces. }
    setExternalForces( simulation_time );
    foreach MassPoint mp in DefSystem DS do
        if mp.itsAffectedFlagIsTrue() then
            { here, every mass-point involved with
              'mp' due to internal forces, must
              set to 'true' its 'affected' flag. }
            computeStretchBendShear( mp );
            { if the autostretching results in
              no motion because the mass-point is
              in its relaxing position, set its
              affected flag to 'false'. }
            computeAutoStretching( mp );
        endif;
    endforeach
endfunction
```

PSEUDO code 11.26: *Mass-point 'affected' flag configuration within Force Accumulator.*

11.9 Displaying the deformation system as a triangular mesh

11.9.1 The mass-spring system as a 3D non-rigid skeleton

It has been explained how the mass-spring system is deformed and cut in real-time with realistic behaviour and smooth animation by the framework developed for the simulator within ForceReaQTor. However, displaying the mass-spring system is not a good way of showing a 3D model.

Because of that, the mass-spring syste is used as a **three-dimensional non-rigid skeleton**, whose is deformed under a static triangular mesh. Then, **by applying a *vertized* triangular mesh** —see section 10.5.5 on page 139 for more information— **over the mass-spring system, a *pseudo*-FFD** —section 5.2.2, page 47— **deformation is applied**:

1. every **vertex** is described by its **displacement from its closest mass-point**;
2. any kinf of **physical deformation will displace mass-points to another positions**, and then the internal restrictive forces make their appearance;
3. so, rendering the mesh by **locating each vertex in the same displacement within the local coordinate system of its mass-point reference**, the mesh is rendered affected by mass-spring system deformation.

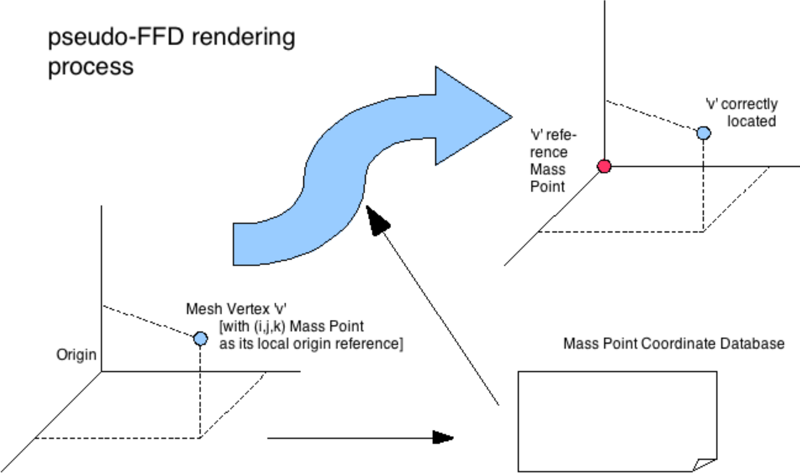


Figure 11.16: *pseudo-FFD rendering process for deforming a triangular mesh w.r.t. its inner non-rigid skeleton —the mass-spring system indeed—.*

11.9.2 A GPU-dedicated algorithm for the mesh rendering process

11.9.2.1 Overview of the shader program for FFD-renderization

This methodology explained above is totally executed over the GPU. Since the mass-point location is always changing due to the physically-based restrictive inner forces, this **vertex deformed position cannot be processed *offline*, so there must be executed in each rendering step**.

It's known that **the GPUs have an architecture focused on parallel processing of independent vertex operations** —see section 9.2, on page 98—, so it's the **best option**:

- this task will be **many executed times faster within the GPU** than from CPU because it is able to compute several vertexs at the same time;
- the the rendering step requires less time, and consequently, CPU has **more computational time for being dedicated to process real-time deformations**;

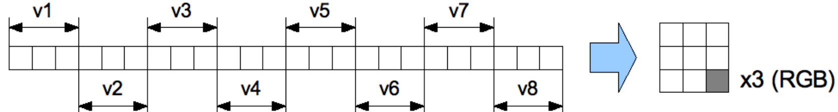
11.9.2.2 The handicap of passin the masspoints and hashing values to the GPU

The most complex problem of executing this task on a GPU is that **it's necessary to transfer the mass-point actual coordinates to the GPU memory in every rendering step**. Even, due to algorithm reasons, it's **also necessary the *hashing* data block**, because every vertex will have associated its normal but also its mass-point reference index in triplets (i, j, k)[88], passed as 3D *texture coordinates*.

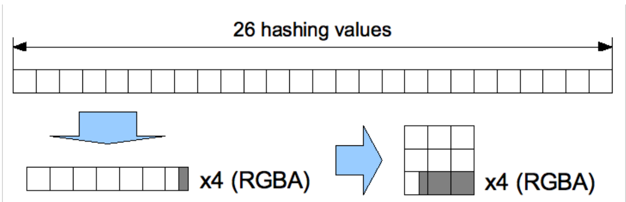
The **ideal transfer mode** of huge amount of data to the GPU is the **2D texture format**, but with some enhancements:

- using 1D textures is prohibitive because the actual graphics cards supports until 8192 ***texels —Texture Elements—* per texture dimension**;
- by using OPENGL `GL_ARB_texture_float` extension, there will be 32**-bit depth not clamped data per component**[89].
- the 2D textures in OPENGL must be POT —*power of two*—, so the predefined supported sizes are 256×256, 1024×1024 and similars. This limitation will surely provoque a huge amount of not-used texture space, but with an extension called `GL_ARB_texture_rectangle`, **2D textures can have any space**.

In the following figures there are the graphical schemes —with simple examples of easy understanding— for compressing the *mass-point* and *hashing factors* into 2D textures:



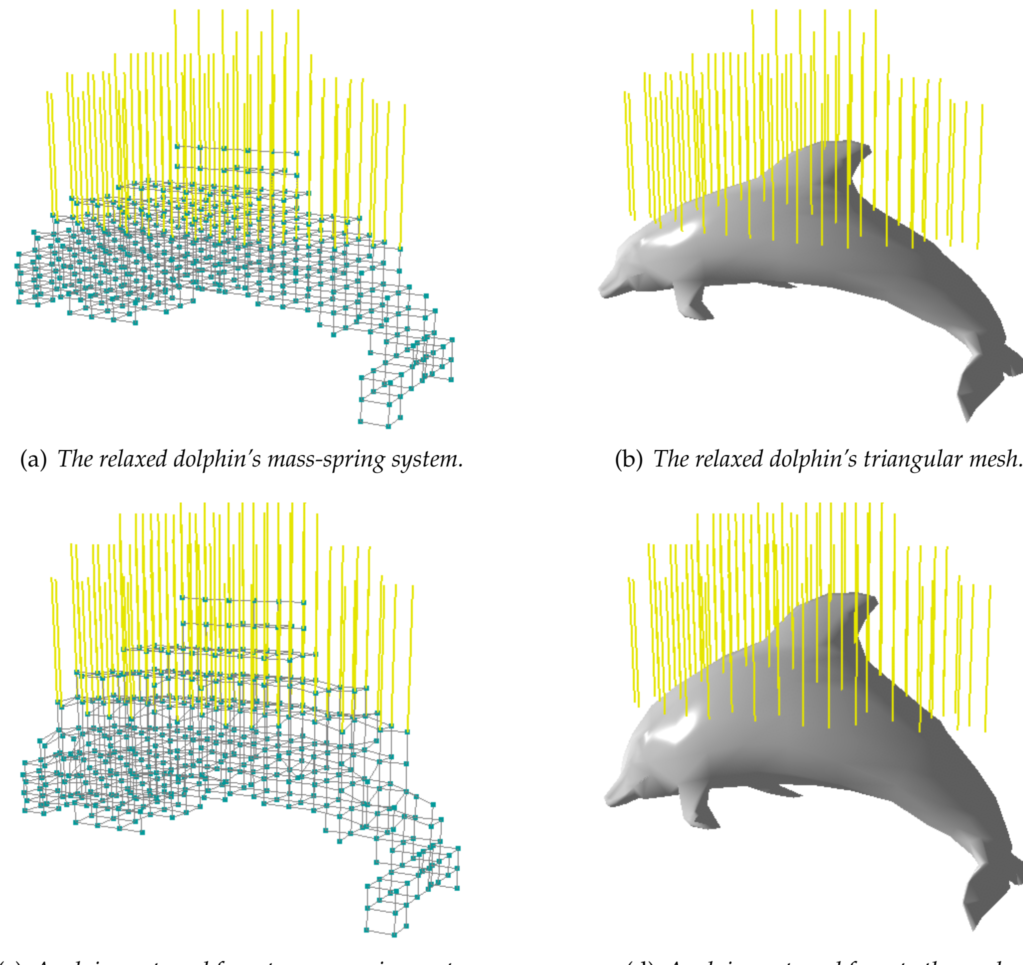
(a) *The mass-point 3D coordinate scheme; since they are described by 3D positions, every matrix cell will be a 32-bit floating RGB texture.*



(b) *The hashing value 'quadruplet-compression' scheme; since hashing factors are single and independent values, they will be also compressed in groups of 4 elements to minimizing the space with RGBA textures.*

Figure 11.17: *The texture compression schemes for the desired shader data: the gray cells of the final matrices indicate unused and value-emptied cells.*

[88] This is because the vertexs are *vertized* in relation to the original voxelmap.
[89] Usually, texture data contain color specifications, so the transferred data from CPU to texture is always clamped to $[0, 1]$ and there are 8 bits per *texel* component —enough for color codification–.



(a) *The relaxed dolphin's mass-spring system.*

(b) *The relaxed dolphin's triangular mesh.*

(c) *Applying external force to mass-spring system.*

(d) *Applying external force to the mesh.*

Figure 11.18: *ForceReaQTor deformable model rendering types: only the model skeleton —mass-spring system— or the model's skin —its triangular mesh, that lays over the mass-spring system—.*

11.9.2.3 The vertex shader algorithm

By enabling the `GL_ARB_texture_rectangle` extension also in the shader, the vertex shader executes, **for every entering vertex —entering it as displacement from a mass-point position—**:

- the **texture coordinates** are the 3D (i,j,k) triplet used **for finding the hashing value**, that are located in `ffd_vmhash` —as a $\lceil\sqrt{n}\rceil$-matrix instead of an n-array—;
- once the **hashing value** is obtained, it's used for accessing the other texture, `ffd_vxlcnt` —whose, in the same way as `ffd_vmhash`, in the form of a $\lceil\sqrt{k}\rceil$-matrix instead of an k-array—, **for getting the mass-point position**;
- then, once the the mass-point —the dynamic *voxel center* indeed— position is found, it only has to be **added to the entering vertex position** `gl_Vertex`, multiply it by the OPENGL visualization matrices, and send it to the fragment shader.
- apart, the **Phong illumination model is also computed**, and passed to the fragment shader, that is the same one as used for the standard Phong illumination —code 10.11, page 128—.

```
#extension GL_ARB_texture_rectangle : enable

// ffd-rendering dedicated external data
uniform sampler2DRect ffd_vxlcnt; // the masspoint coordset.
uniform sampler2DRect ffd_vmhash; // the hashing data block;
                                  // the texture coordinates
                                  // have the (i,j,k) values
                                  // for mass-point coord.
uniform float ffd_vmsize;         // the CPU's voxelmap size.
uniform float ffd_rootvx;         // the ffd_vxlcnt size.
uniform float ffd_rooths;         // the ffd_vmhash size.

// phong illumination model data
varying vec3 normal, lightDir, eyeVec;

// returns the mass-point 3D position by the hashing value.
vec3  getTextureData_VXLCNT( float vectoridx )
{
    float newidx = vectoridx;
    vec2 matidx = vec2( int(mod(newidx,ffd_rootvx)),
                        int(floor(newidx/ffd_rootvx)) );
    vec3 vxlcnt = texture2DRect( ffd_vxlcnt, matidx ).xyz;
    return vxlcnt;
}

// returns the (i,j,k) hashing value from the 3D_to_1D
// (i,j,k) coordinate conversion.
float getTextureData_VMHASH( float vectoridx )
{
    float newidx = floor(vectoridx/4.0);
    vec2 matidx = vec2( int(mod(newidx,ffd_rooths)),
                        int(floor(newidx/ffd_rooths)) );
    vec4 hash_4 = texture2DRect( ffd_vmhash, matidx );
    float coord = mod(vectoridx,4.0);
    return hash_4[int(coord)];
}

void main()
{
    // ffd_render
    vec3 ffd_index = vec3( gl_MultiTexCoord0.xyz );
    float ffd_hash = ((ffd_index.x*ffd_vmsize) +
                     ffd_index.y)*ffd_vmsize +
                     ffd_index.z;
    float ffd_cnt = getTextureData_VMHASH( ffd_hash );
    vec3 vxlcnt = getTextureData_VXLCNT( ffd_cnt );
    vec4 vertex = gl_Vertex + vec4(vxlcnt, 0.0);

    // phong precomputation
    normal = gl_NormalMatrix * gl_Normal;
    vec3 vVertex = vec3( gl_ModelViewMatrix * vertex );
    lightDir = vec3( gl_LightSource[0].position.xyz -
                     vVertex );
    eyeVec = -vVertex;

    gl_Position = gl_ModelViewProjectionMatrix * vertex;
}
```

GLSL code 11.27: *Vertex shader for rendering deformated triangular meshes.*

11.9.3 Autogenerated *skin* from the mass-spring *skeleton*

11.9.3.1 Introduction

In the previous chapter, on section 10.5 —page 134—, the voxelmap structure (and its vital importance on volumetric representations) has been explained. Its structure is a boolean array of $n_x \times n_y \times n_z$ elements —cubes, *voxels*—that contain a logical value, describing them as original mesh recipient or not.; that is, **a *voxel* is *true* if the referenced polygonal mesh is —totally or partially– contained within its volume range**.

Also, on section 4.1.2 on page 32 there's the definition of the **isosurfaces**, and some pages later, the **isosurface extraction algorithms** —esssentially focused on the **Marching Cubes** method— are schematized. Indeed, all the chapter 4 is about multiresolution meshes and **automatic mesh generation from isosurfaces**.

As a reminder, it's convenient to say that there's a **close relation between isosurfaces and voxelmaps**:

- an **isosurface** is a data structure representing a set of constant scalar values within a space volume;
- on the other side, a **voxelmap** is a volumetric space discretization with constant boolean values;

then, the isosurface extraction algorithms determine the so called **boundary voxels**, whose are between the empty space and the isosurface, and by setting constant values to the voxelmap cells, a polygonal mesh is automatically generated in relation to a constant **threshold value**, following a similar scheme to this 2D curve automatic generation[90].

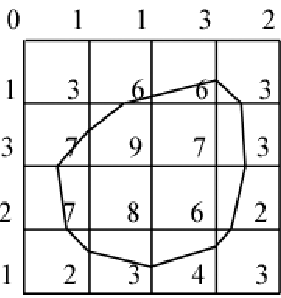


Figure 11.19: 2D *curve generation by the Marching Squares* —2D *Marching Cubes*— *technique. The example's threshold value for curve extraction is* 5.

11.9.3.2 ForceReaQTor and the deforming meshes

As said in previous pages, ForceReaQTor is able to **deform not only the mass-spring system structure but also a polygonal mesh**, by an internal process where the triangles are spread towards the deformable structure, like a *skin*, and by local displacements, the **three-dimensional surface model is able to follow in run-time the mass-spring deformation**, as it's shown on figure 11.18.

[90] The 2D version of Marching Cubes is called Marching Squares, and this methodology is identical to the 3D variant one.

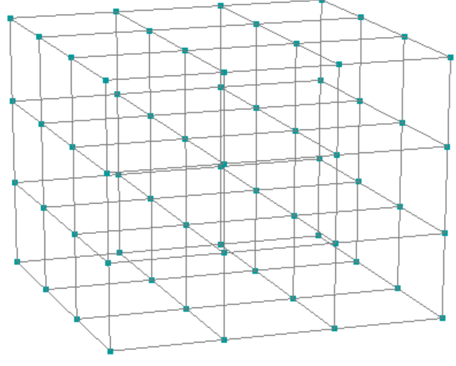
However, not always exists this polygonal mesh; it's true that MeshInspeQTor is capable of generating voxelmaps from three-dimensional models, but it's also true that a **voxelmap navigator** is available in this application.

This navigator permits the application user to modify any voxel boolean value, so **voxelmaps can be manually created by setting the boolean values with the navigation panel controls**[91] So, then there is no polygonal mesh associated to the voxelmap; this is the case, for example, of the $4 \times 4 \times 4$ cube used on almost every previous deformation snapshot.

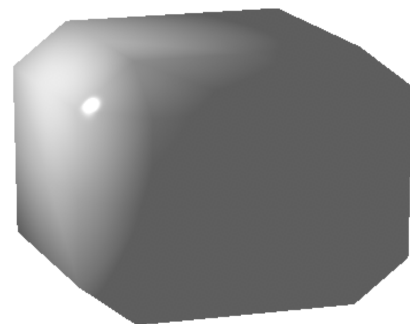
Because of that, ForceReaTor executes a **Marching Cubes instance when a voxelmap is imported**. Moreover, apart from being a **user-transparent action** made by the program, the elapsed time for evaluating the mesh autogeneration is unnoticeable, and it provides the voxelmap of:

- a **regular-sized triangular mesh** inspired by the voxelmap *shape* —that is, the mass-spring system structure indeed—.
- a **per-vertex normal postprocessing with no repeated vertex data**, so all the Marching Cubes data is filtered for minimizing the occuped space and redundant data[92].

Nevertheless, the **Marching Cubes autoextracted three-dimensional surfaces are inherently in a smoothed mood**, with no sharp zones, due to the boundary voxels treatment methodology. This phenomenon can be shown in the autogenerated *cube* from the $4 \times 4 \times 4$ cubical voxelmap:



(a) *Original Mass-Spring System.*



(b) *Marching Cubes autoextracted mesh.*

Figure 11.20: *Example of ForceReaQTor autogenerated mesh of the highly featured $4 \times 4 \times 4$ cube; the normals are per-face configured, but by applying the Phong illumination* GLSL *program —section 10.3.4—, the renderization is robust.*

[91] On appendix's section B.2.3.3, on page 217, there's a complete user tutorial and example snapshots of this voxelmap navigator.

[92] The Marching Cubes algorithm was thought for processing every vertex's triangles and immediately render them; so, since the method is robustly consistent, there are a lot of vertexs computed several times —once per associated face—. The postprocessing for per-vertex normal, by having every vertex once in memory and giving it the obtained normal by the mean of all its associated ones, is computatinally slower than the Marching Cubes method itself, but since it's an *offline* preprocess executed when the voxelmap is imported, the lack of time is mnimal.

The ForceReaQTor **isosurface generation** for a subsequent three-dimensional surface extraction is based on the boolean values. It's known that in Marching Cubes, the **boundary voxels are defined by their vertices' isovalues**, so from the combination of *internal* and *external* vertices —w.r.t. the threshold extraction value—, the triangulation is performed, as explained on figure 4.7 on page 36.

So, ForceReaQTor establishes a 28**-adjacency voxel connectivity, focused on the total number of adjacent voxels of each voxel vertex**, to specify if a certain voxel vertex is *connected —internal—* to any other voxel, or *unconnected —external—* to all of them; for example, given a certain voxel, the desired isovalue evaluation of its LUF[93] vertex is done by checking connectivity of the voxels at its:

1. of course, the **own** LUF direction;
2. the **individual** directions, L, U and F;
3. directions by **combination** of the three coordinates: LU, LF, FU;

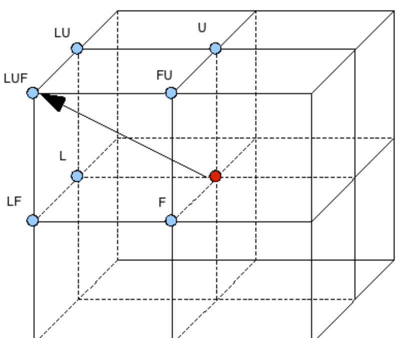


Figure 11.21: *Three-dimensional scheme of which voxels have to be checked for adjacency with a certain voxel, if it's desired to compute the isovalue of the* LUF *voxel vertex.*

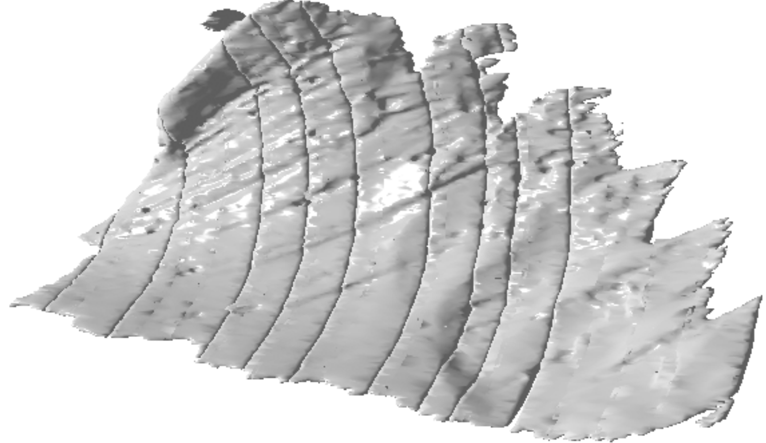
If there is at least adjacency within the per-vertex connectivity seven-voxel set, the vertex will be considered *internal* and marked with a 0.0 scalar; **the vertex will be considered *external* —scalar** 1.0**— to the model only when there is no adjacency in the checking test**. The isosurface extraction is done by a 0.5 **threshold value**.

11.9.3.3 The wrongly parametrized triangular meshes

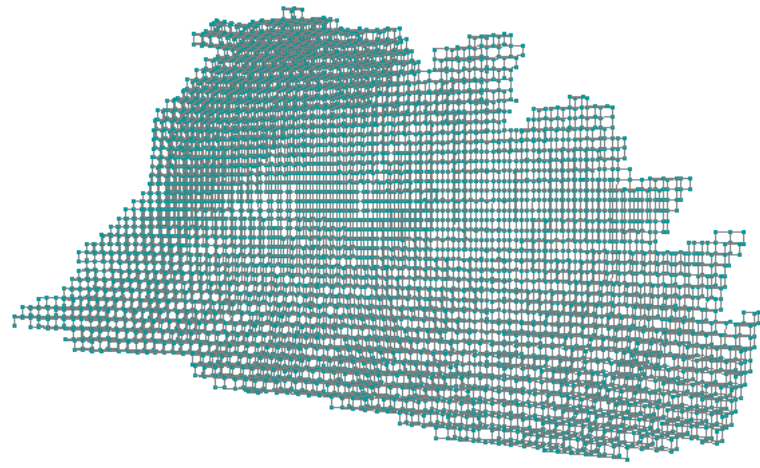
Additionaly, it has been said on the previous chapter's section 10.2.4 that the extracted isosurfaces from the *Visual Human* Project can be wrong in terms of *(i)* **artifacts** within the triangular mesh —called *dense rings* on figure reffig:p3d-bad— or *(ii)* **bad piecewise extractions** due to the lack in huge organ extraction capabilities.

Then, an **arrangement** of this kind of problematic meshes is possible by ForceReaQTor autogenerated mesh, since the **mass-spring system —the voxelmap— is density-dependant of original mesh's occuping volume, not of its geometrical definition**. It's the case, for example, of the *left external oblique inguinal human muscle*, object of this thesis' deformation studying.

[93] Acronym of *Left*, *Up*, *Front*.



(a) *Oblique muscle's original 'isosurface mesh', wrongly extracted mesh from Visual Human.*



(b) *Its associated mass-spring system, derived from a* $64 \times 64 \times 64$ *voxelmap.*

(c) *The ForceReaQTor's autogenerated triangular mesh from the mass-spring system volume with Marching Cubes.*

Figure 11.22: *Example of ForceReaQTor extracted mesh of the 'left external oblique human muscle', wrongly extracted from the Visual Human Project with the* `extractor` *software provided by the* IRI *at* UPC*; notice the simplicity of the extracted mesh by using Marching Cubes —w.r.t. the original mesh—, but also its consistency in relation of the original.*

11.9.3.4 Inherent *pseudo*-FFD rendering adequation

A few pages ago there has been detailed a programmable graphics algorithm for rendering deformable polygonal meshes; so, this **mass-spring system autoextracted mesh is also available for supporting this *pseudo*-FFD rendering methodology**, so the deformable interactions will be noticed in ths alternate polygonal *skin*.

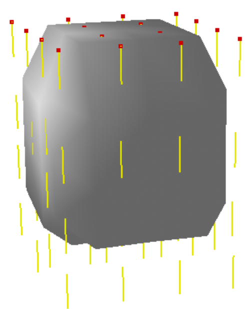
This is easily done because the *vertizer preprocess* is so simple, due to an **inherent property** to all the triangles generated by isosurface extraction by Marching Cubes: **the total surface of every generated triangle is contained within the same voxel**; it's deducible simply viewing the figure 4.7, on page 36, that features the possible triangulation configurations depending on the *internal-external* vertices.

The detailed algorithm of the implemented Marching Cubes won't be featured here because it is composed of a large amount of line codes focusing the different voxel configurations and vertex interpolations, but the **Marching Cubes main looping execution** is the following one:

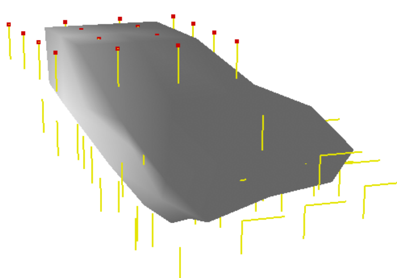
```
// Marching Cubes Isosurface Extraction Data
typedef struct {
    float values[8];     // 8 isovalues per voxel.
    float coords[24];    // 8 vertices * 3 coords.
    bool samevalue;      // 'true' if the 8 isovalues are
                         // the same -internal or external-,
                         // that means it isn't a boundary
                         // voxel and it can be avoided.
} MCData_struct;
MCData_struct* mcdata;
mcdata = new MCData_struct[nvoxels*nvoxels*nvoxels];

// Marching Cubes Rendering Data
float* mcVertexs;        // vertex displacement from
                         // container voxel center.
float* mcNormals;        // the normal data.
float* mcTriangs;        // the face index data block.
float* mcIndexes;        // the container voxel cell
                         // (i,j,k) indexes.

void doMarchingCubes()
{
    // this method configure the 'mcdata' array
    mcGenerateVoxelMapIsovalues();
    for( int i=0; i<nvoxels; i++ )
        for( int j=0; j<nvoxels; j++ )
            for( int k=0; k<nvoxels; k++ )
            {
                // per-voxel triangle extraction
                // within boundary voxels.
                int idx = ( (i*m_ny) + j )*m_nx + k;
                if( !mcdata[idx].samevalue )
                    mcPolygonizeVoxel( i, j, k, idx );
            }
    // finally, the rendering data is correctly set.
    perVertexNormalFiltering();
}
```

CPP code 11.28: *The rough ForceReaQTor Marching Cubes algorithm.*

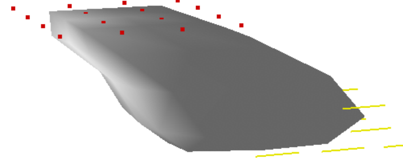
So, by applying the same **array-to-texture compression technique** detailed in the previous section 11.9.2.2, on page 176, and loading the same specified **FFD-rendering shader program**, the **real-time deformations for the autoextracted three-dimensional surface** is available within ForceReaQTor.



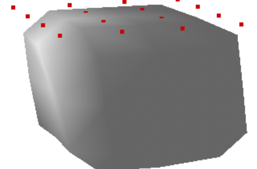
(a) *At first, the gravity force affects the cube; notice the fixed —red— upper mass points.*



(b) *Subsequently, a right external force is applied and the cube is pulled.*



(c) *By disabling gravity, the cube reacts more affected by the pulling force.*



(d) *Finally, disappearing the forces, the cube returns at relaxing state.*

Figure 11.23: *An example of deformation force set applied to the* $4 \times 4 \times 4$ *cube Marching Cubes autoextracted mesh in ForceReaQTor.*

11.9.3.5 Polygonal mesh cutting renderization capabilities

At the moment of writing these lines, it has not been possible to achieve the mesh cutting —or mesh splitting— based on the voxel broken connectivity, due to implementation problems within another parts of the project, in the force accumulator part —not problems of calculus but for optimizing the code—.

Indeed, it's the visualization part of the virtual scalpel implementation; the internal part is fully working as can be seen on section 11.7; however, on section 13.2.2, on page 197, there is a little explanation about the innovative proposal renderization method together with a scheme.

It **requires a marching cubes extraction triangular mesh, for taking advantage of the total containing of a triangle within a single voxel**, and is based on **vertex cloning**. Hence, voxel cloning or dynamic topology modification isn't necessary and only a **local marching cubes on the *new* empty zone will be necessary for simulating the property of non-emptyness of the model**.

11.10 Developed File Formats

11.10.1 Dynamic Force List *.DFL File Format

This own file specification permits store on hard-disk, for future loadings, the ForceReaQTor actual state of the **list of dynamic forces** with all their parameters and dynamic effect typifications —*force, node fixing* and *spring cutting*—.

```
1 @FORCEREAQTOR_DYNFORCELIST:FILEFORMAT_v1.0
2 <numberof-dynforces>
3 <type1> <noderange1-from(x,y,z)> <noderange1-tilll(x,y,z)>
4     <direction1-(i,j,k)> <force1-amount> [...]
5     <timeline1-start> <timeline1-duration>
6 <type2> <noderange2-from(x,y,z)> <noderange2-tilll(x,y,z)>
7     <direction2-(i,j,k)> <force2-amount> [...]
8     <timeline2-start> <timeline2-duration>
9 [...]
```

PSEUDO code 11.29: *ForceReaQTor dynamic force list file format.*

where type is *(i)* number 0 for dynamic forces, *(ii)* number 1 for node fixing and *(iii)* number 2 for spring cutting.

11.10.2 VoxelMap *.VOX File Format

This format is **imported, used and transformated for internal adequation and conveniences** by ForceReaQTor, but cannot be managed because, as it's said, it's not loaded and saved but imported. Really, this format is the same of the MeshInspeQTor voxelmap file —indeed, this is the ***bridge* between the two applications**—, explained before on section 10.6.2, on page 144.

CHAPTER 12

Tests and Results

12.1 Introduction to this chapter

12.1.1 A brief about the developed software and the offered results

In the last two chapters there has been presented an exhaustive and detailed journey across the *software suite* developed for this project:

- on the MeshInspeQTor part, the three-dimensional **visualizer** and oblique plane **cutting** tool —the VirtualCutter panel— has been described, as well as the soul of the application: the **mesh voxelizer**, that builds volumetric representations of three-dimensional objects.
- on the other hand, ForceReaQTor, the **physically-based deformation simulator** is presented in the subsequent chapter; the **underlying mass-spring deformable system**, dynamic force set, the two numerical integrators, the three internal restrictive behaviours when the system is being modified... and finally, the *skin* deformable layer, a polygonal mesh that supports the underlying mass-spring deformations and reacts in the same way.

Though there are already some snapshots in the both two application dedicated chapters, now it's time to **offer tangible and quantificable results, by animating —deforming— several models, some of them about medical focus**.

The global deformation software has been always though for **the aim of simulating the tactile sensations of deforming human muscles in a real-time computation but with losing so much realism**. Because of that, and although this software is capable to deform any kind of three dimensional models, the **deformable system that runs deeply in ForceReaQTor is provided by some important characteristics**:

1. the mass-spring system is **statically positioned**; that means, external forces can be applied over the mass-spring structure, and **it will surely be deformed, but it will never be displaced**;

2. moreover, the **deformations usually become viscoelastic** —with a little viscous damping factor—, for **simulating the lengthen and shorten forces in the same way that would affect onto flesh** —although the simulation parameters can exaggerate this effect, as it will be seen in this chapter, for a better visualization of the model deformations—;

3. **a specified deformation will never modify the original shape of the model**. In more technical words, vast bending deformations are not supported, and the external forces will apply to the model a strange *assimetrical scaling*; *ergo*, **the initial volume, position, orientation and shape of any deformable model will be always maintained**;

12.1.2 The medical-purpose three-dimensional models

As detailed on chapter 2, on page 7 —even in this thesis' document title is mentioned the focusing on it—, the **simulation of the human musculature** has been in mind all the time to provide the software of a real practical functionality, and my thesis director proposed **the human inguinal region as an interesting focus of human organ part**.

Also, the **extraction of the inguinal region three-dimensional models** —directly from the *Visual Human Project*, available from this university, UPC—, resulted in **too much noticeable model visualization drawbacks**:

- **wrong triangularization**, including the appearance of dense rings of triangles that only added a bad visualization of the model;

- **too much forced piecewise extractions** —due to remote machine RAM memory size problem— that really avoid a continuous mesh building and the resulting mesh after merging all the extracted portions wasn't satisfactory;

This two clear disadvantages provoqued the consideration of including a **remeshing technique** in the project objectives. And a **Marching Cubes pass over the 3D model-based generated voxelmap** was the final election, since it's a classical algorithm for iso-surface extraction and was also able to be used for remeshing —with the requirement of an intermediate volumetric representation of it—.

12.1.3 Test structure, and given results

Since the results of this project are not static —the soul of the project is the successful animation—, it's hard to present results in paper; however, there will be **two quantified processes**: *(i)* the voxelization and *(ii)* the marching cubes execution over a voxelmap.

For the animations, there will be featured some framesets with any **human inguinal muscle deformable system** —because the non-medical models have been featured during the rest of the document—. The tests will include

- some **deformation movements**, including so lenghten as well as shorten forces;

- **animation** with original and marching cubes' extracted **triangular meshes**;

- every **relevant data** will be given: FPS framerate, active restricting forces, and external force data;

12.1.4 Testing Machine Specification

All the tests, as well as the snapshot figures included from both MeshInspeTor and ForceReaQTor have been executed over the Apple MacBookPro specified in table 8.1, on page 85. The result table for the another used machine, an Apple iMac, also specified in the previous table, is not published because the both two machines are contemporary computers and there's no significant differences between the testings.

12.2 Introduction test over non medical models

12.2.1 Brief overview

Before beginning with the simulation of medical model deformation, it's time to **quantify the two generator methods** implemented for this project: *(i)* the three-dimensional **surface model voxelization** —with its correspondent voxel carving—, and *(ii)* the **marching cubes autoextracted** generation from the cited voxelmap.

For achieving these results, the polygonal mesh object is the `bunny.capped` one, composed by **69.664 faces**, a relatively high polygon cardinality —there's no a gigantic mesh but there's no a 300-face model too— and a **good computational time balance checkpoint**.

Figure 12.1: *The original* `bunny.capped` *polygonal mesh surface model.*

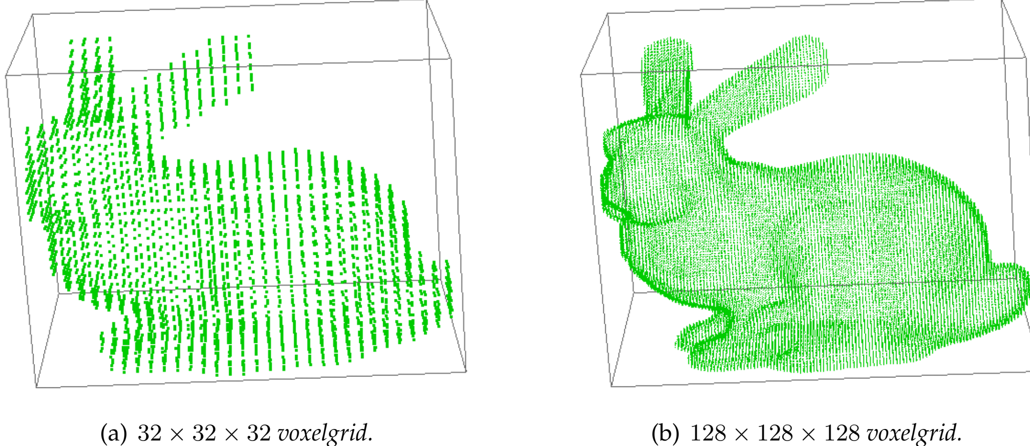
12.2.2 Mesh voxelization test results

The obtained timings for the `bunny.capped` model, by voxelizing it with MeshInspeQTor, are shown in the following table, specifying the **computational time needed for voxelizing as long as for executing a voxel carving over the resulting voxelmap**.

VoxelMap Grid Size	Voxelization Process	Voxel Carving Process
$32 \times 32 \times 32$	1.67425 seconds	0.010713 seconds
$128 \times 128 \times 128$	27.61050 seconds	0.697849 seconds

Table 12.1: *Computational times for the 'voxelization' and 'voxel carving' processings. There have been tested the timings for two different voxelmap sizes:* $32 \times 32 \times 32$ *and* $128 \times 128 \times 128$.

Finally, there are the **resulting voxelmaps** —before executing the voxel carving— below these lines. The $32 \times 32 \times 32$ voxelmap has more pointsize voxel marks for a better visualization.



(a) $32 \times 32 \times 32$ *voxelgrid.* (b) $128 \times 128 \times 128$ *voxelgrid.*

Figure 12.2: *The resulting voxelmaps from the* `bunny.capped` *voxelization processes.*

12.2.3 The Marching Cubes automatic mesh extraction

Now it's the turn to show the results of the inverse proceeding: instead of obtaining a volumetric representation of a 3D surface model, **marching cubes generates a polygonal —triangular— mesh from an arbitrary volumetric representation in a fast, robust and continuous way**.

So, as done before for the voxelization, here's the **timing table for the distinct marching cubes executed** over the previous voxelmaps.[94]. Since the data input for ForceReaQTor autoextracting mesh processing is really a voxelmap, the previously generated voxelmaps —figure above— have been used for it. Even better by using them, because in that way there exists a bridge connection between the two processes.

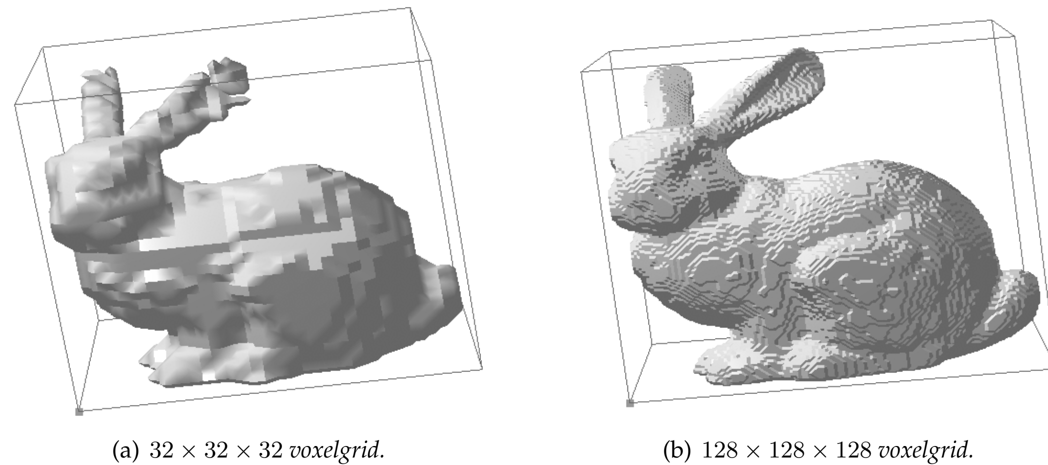
VoxelMap Grid Size	Per-face-normal MC	Per-vertex-normal MC
$32 \times 32 \times 32$	0.087254 seconds	13.3000 seconds
$128 \times 128 \times 128$	2.637540 seconds	N/A

Table 12.2: *Computational times for the 'per-face' and the 'per-vertex' marching cubes proceedings.*

And following this table, that proofs that Marching Cubes is **really a fast algorithm** —at least for immediate mesh renderization—, and the resulting meshes are:

- **simpler than the original**, because it's built from a discretized 3D space occuped by *boxes* —the voxels indeed—;
- **continuous and maintaining** the original *shape* of the bunny; even in the $32 \times 32 \times 32$ resulting one.

[94] As a reminder, it's important to know that in ForceReaQTor a marching cubes postprocessing is implemented for obtaining per-vertex normals instead of per-face normals and also for avoiding vertex data repetition.



(a) 32 × 32 × 32 *voxelgrid*.

(b) 128 × 128 × 128 *voxelgrid*.

Figure 12.3: *The resulting marching cubes' autoextracted meshes from* `bunny.capped` *voxelmaps.*

12.2.4 Lenghten and shorten forces over a model.

For demonstrating the real-time good capabilities of the implemented **collision handling** and the innovative **autostretching** technique for preserving the volume, the three dolphins of `dolphins` 3D model will be taken and different forces will be applied over them, in the two possible directions: **lengthen** and **shorten**, and the frameset will proof the good mass-spring adequation in all the situations.

The frameset for both two phenomenons is featured on the next page in a eight-figure set, where the left column is the frameset of a lenghten force set while the right one is for shorten forces deformation animations.

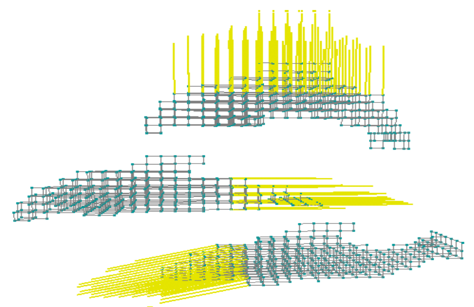
12.3 Test over medical models; the human inguinal muscle region

12.3.1 Brief overview

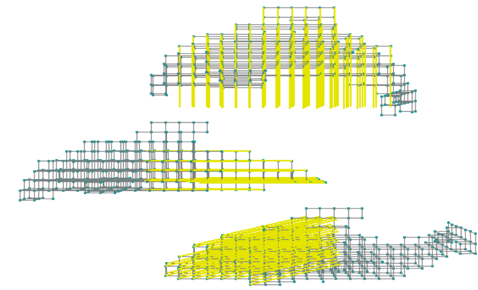
In here this part, the medical model deformation section, there will be presented the **three human inguinal region muscles**: *(i)* the oblique interna, *(ii)* the oblique external and *(iii)* the transversus, both three of the left body side —the right side ones are simetrical w.r.t. that three ones—. So now, in a different mood that for the previous section, there will be several *concrete* deformations because the mass-spring systems describing the human muscles are huge in mass-point cardinality, so the **deformations won't be as global as well as in the previous section**, there will be focused in certain organ regions.

Also, here will make their appearance two factors not involved before:

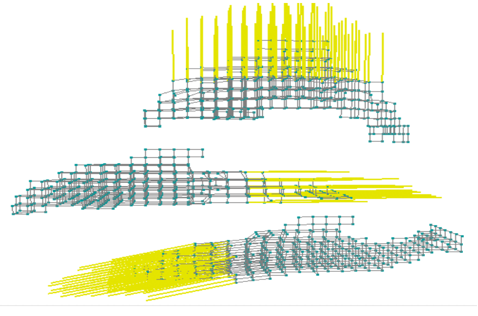
- the ***affected* internal acceleration flag data array**, previously explained on section 11.8, on page 174;
- the **Verlet integrator**, because the Euler one is so much unstable for complex-structure mass-springs ike the following ones; additionally, the mass-point set is larger than for the previous systems, so the **Force Accumulator is still sloweer than the required for the Euler integrator in most cases**;



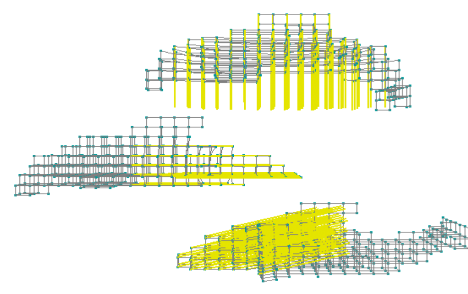
(a) *the initial instant, where the three 'lenghten' forces begin.*



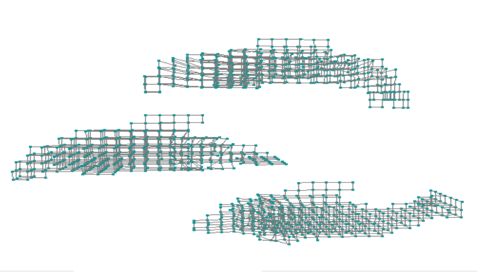
(b) *the initial instant, where the three 'shorten' forces begin.*



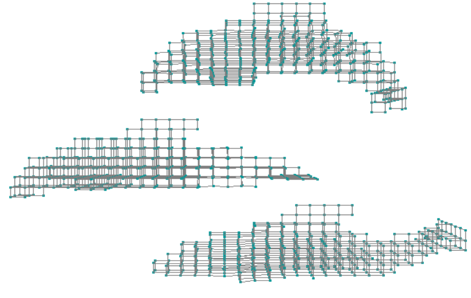
(c) *one second later, with the dolphins supporting their respective inciding 'pulling' forces.*



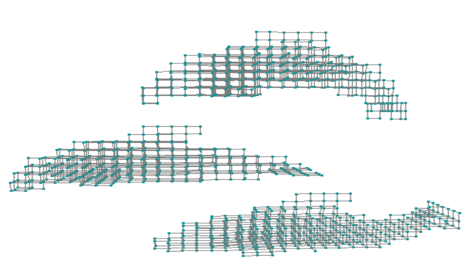
(d) *one second later, with the dolphins supporting their respective inciding 'pushing' forces.*



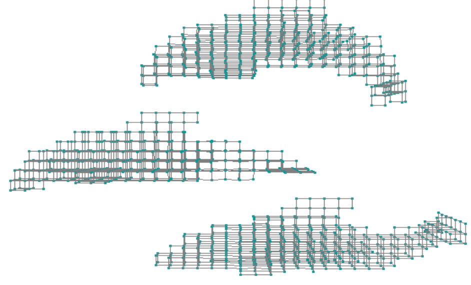
(e) *when the forces disappear, a 'shorten' movement is well-managed by the system.*



(f) *when the forces disappear, a 'lenghten' movement is well-managed by the system.*



(g) *finally, the three dolphins have come back to their relaxing positions.*



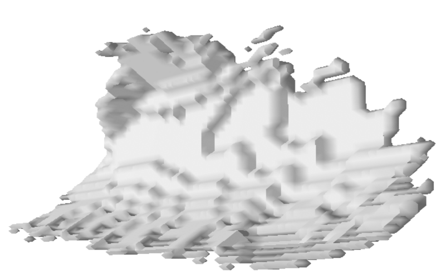
(h) *finally, the three dolphins have come back to their relaxing positions.*

Figure 12.4: *Lenghten (a,c,e,g) and shorten (b,d,f,h) force set deformation examples over the* `dolphins` *model. The rendering process has been at* 25 FPS, *with 'stretch' restrictive forces and Euler integrator. The forces are all* $20N$ *amount.*

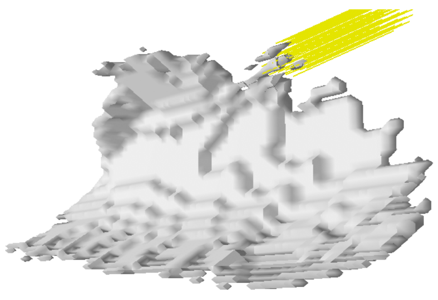
As a reminder: the Verlet integrator is not as exact as the Euler one but it's more stable and converges better to the solution because de Verlet approach is a linear approximation of the Euler numerical integration.

12.3.2 The *oblique* external muscle

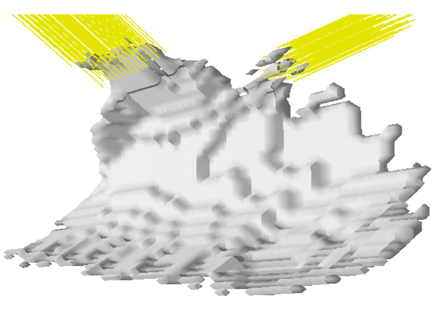
Over this a **huge model** —it is shown the marching cubes autoextracted model from a $64 \times 64 \times 64$ voxelmap— it has been decided to specify a 4-force set in different muscle places, for featuring the *affected* acceleration structure. **In every force accumulator represented in a snapshot, a new force appears so new *affected* mass-points are born**. However, the numerical integration doesn't require to visit every mass-point for each accumulatr, so the simulation is correctly adapted in size of managed data.



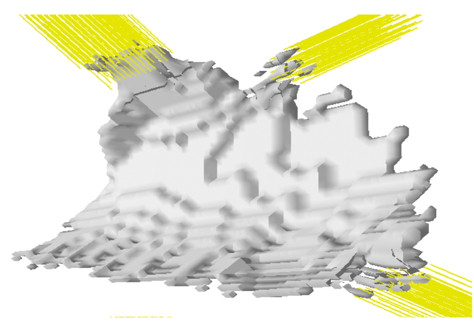
(a) *the initial instant, where no force is actuating yet.*



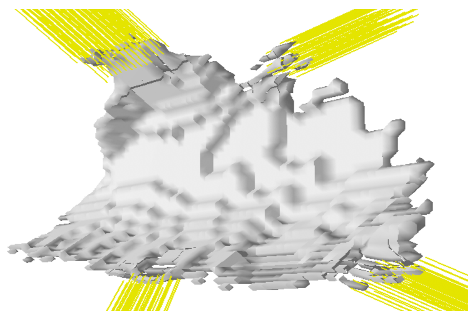
(b) *the first deformation force; notice the local deformation.*



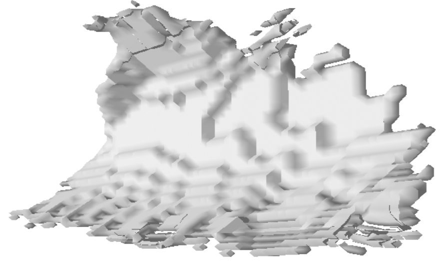
(c) *a second force appears, the fluid animation is not suffering.*



(d) *a third force involves the muscle, whose has three clearly independent deformation zones.*



(e) *the last force makes its act of presence.*

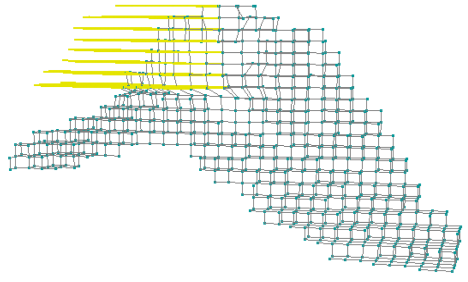


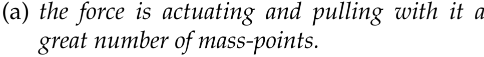
(f) *when the forces disappear, the 'shorten' reaction is well-managed by the system.*

Figure 12.5: *A deformation force set of the* `obliqext` *model, based on a dense* $64 \times 64 \times 64$ *voxelmap; here there is an abbuse of the* `simacc` *adaptative methodology for computing Force Accumulators. The integrator is the Verlet one using a* 30 FPS *animation, and the forces are* $20N$ *all*

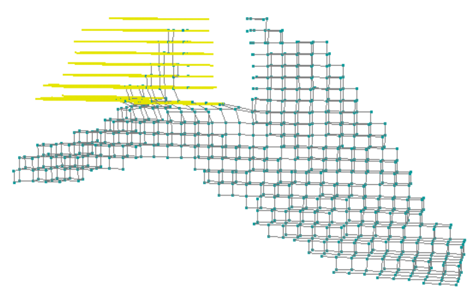
12.3.3 The *transversus* muscle

In this more curved muscle, it has been decided to proof the *affected* adaptative simulation handling with a **mass-spring system cutting while a force is inciding in the cut region.**

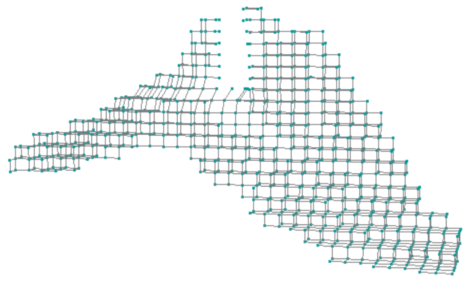




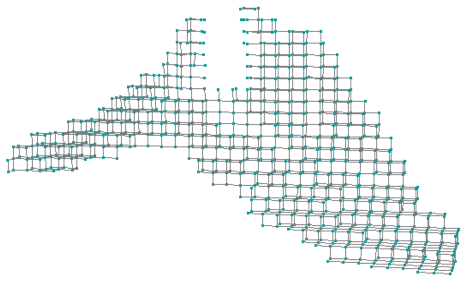
(a) *the force is actuating and pulling with it a great number of mass-points.*



(b) *when the cutting enters the scene, the isolated mass-points are shorten and come back to their relaxes position.*



(c) *when the force disappear, the rest of deformed mass-points begin their shorten movement.*



(d) *finally, the cut rests for ever but the mass-spring system is again relaxed.*

Figure 12.6: *A deformation force set for the* `transversus` *model, based on a* $32 \times 32 \times 32$ *voxelmap; here can be seen the influence of a cutting on a dynamically deformable model. Here there is* 30 FPS *animation with a Verlet integrator with 'stretching' and 'bending'; the force is* $30N$ *amount*

CHAPTER 13

Project Conclusions and Future Work

13.1 Contributions and Derived Conclusions

13.1.1 Final Conclusions

In this thesis, **a lot of computer graphics fields have been touched** for the aim of building a **quasi-entire interactive 3D model generic deformatior** —entire if the run-time interaction is not counted—:

1. 3D polygonal meshes format understanding;
2. voxelization of meshes, or automatic volumetric representation from 3D meshes;
3. space carving for voxelmap refilling;
4. isosurface extraction algorithms from voxelmaps;
5. physically-based animation algorithms, comprising force accumulators and position integrators based on mass-spring dynamic systems;
6. generic mesh deformation due to voxelmap's 3D non-rigid skeleton *underlapping* mapping;
7. GPU programming —shaders— for computing accelerations about applying deformations or computing realistic illumination methods;

Without entering in great deepness in any of these fields, the presented project is a conjunction of different disciplines —some of them with own contributions and algorithm variants—, **offering a global vision of physically realistic real-time computer graphics animation**, because of almost fields of animation are studied: from realistic illumination to realistic animation by deformation due to force application.

Due to this, a ***software suite*****, composed by two applications**, has been developed entirely for this thesis. The two applications are, of course, **fully-functional and end-user-ready**.

- MeshInspeQTor for mesh visualization and voxelmap generation,
- and ForceReaQTor as the physically-based deformation modeler simulator,

13.1.2 Research Contributions

13.1.2.1 Conservative mesh voxelization variant

Following the Akenine-Möller proposal for triangle-box intersection methods —reference [vox7] of the bibliography—, a conservative 2D based algorithm is proposed and implemented. A non-validated voxel it's sure that is correctly evaluated, but there are some exceptional cases where a validated voxel is really a non-valid one, due to an approximation: **a triangle is considered a rectangle**. However, for triangle meshes, where there are a lot of primitives, this is a final result **non-affecting approximation**.

The main idea behind this method is **to consider not a triangle but its three 2D projections onto the three bounding box planes** —six faces, parallels in sets of two—. In this project, since all the bounding boxes are *Axis Aligned* —therefore, of AABB type—, these planes are parallel of the axis planes

13.1.2.2 New approach for an hybrid deformation system

Similar to the geometric animation method called *Free Form Deformation* —FFD, see section 5.2.2 on page 47—, by Sederburg and Parray, here there is an approach of 3D object deformation is implemented by building a **non-regular grid of equidistant nodes** comprising a **non-rigid skeleton**, that reacts to external forces due to a **mass-spring phisically-based deformation model**.

Hence, there's a 2**-layer hybrid deformation system** composed by:

- an **underlying layer**, composed by a **physically-based deformation model** similar to 2D cloth deformation systems, mentioned on section 6.3.1.4 (page 65), capable of **reacting anisotropically** to:
  - external applied forces,
  - deformation-generated internal forces,
  - volumetric auto-collisions,
  - original relax shape returning memory,
  - splitting;
- an **overlying layer**, based on a **geometric deformation model similar to FFD** due to a local coordinate system based on the skeleton node influence zones;

13.2 Future Work

13.2.1 Persistent storage of the Marching Cubes extracted meshes

It would be a great idea to save the ForceReaQTor autoextracted polygonal meshes from the mass-spring system onto an OBJ file for future loadings and manipulations in MeshInspeQTor, and **establishing a closed circuit in the *software suite***. Also, it can be useful since one of the aims of this method is the mesh rearranging.

13.2.2 Innovative mesh splitting based on breaking voxel connectivity

Oppositely to all the mentioned methods for **virtual scalpel implementations**, deriving onto **mesh splittings** and **3D object dissects**, in this project a **new proposal was desired to be implemented, based on voxel connection splitting instead of cracking the polygonal meshes**; however, some complications on another research fields have impossibilited enough dedication; however, the internal cutting is done, this would be *only* a better visualization of it.

The basic idea would be conjunction between **two interconnected modules**:

- the fast and robust isosurface extraction method **marching cubes**, for an instant mesh **local** regeneration —focused, obviously, on the splitted object region—;
- a splitting of the springs of the **deformable inner 3D non-rigid skeleton**; by breaking these links, **no voxel cloning is needed**, and **only a vertex cloning** for the splitted voxel faces contained primitives will be necessary;
- since the splitting is done within the 3D non-rigid skeleton, this breaking will be **involving in run-time over the underlying deformable layer**, so the phisically-based system will **react immediately** to this system breaking;

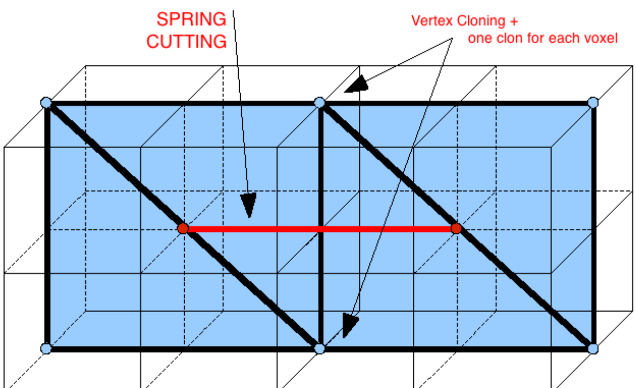


Figure 13.1: *Scheme of a possible splitting for a marching cubes' extracted mesh.*

13.2.3 More GPGPU-Focused Algorithms

Due to not enough time, a **GPGPU-oriented force accumulator** has **not** been able to be implemented; with an iterative accumulator like the ForceReaQTor one, executed in a gpu with its parallelism capabilities and advantages —with the indications of Dominik Göddeke at [gpu8] and [gpu9], but also here in section 9.3 on page 104—, a **less unstable system could be reached**, as long as **more FPS would be achieved by the simulation**.

However, the **briefly** scheme should be the following:

1. by using the FBO —*Frame Buffer Object*, see section 9.3.1.1— extension, the CPU-GPU looping is completed;
2. in each *iteration*, each independent 3D skeleton node is computed by applying their time-line pertinent deformation forces, passed in as a 1D texture; the 3D skeleton is passed as a 2D texture due to the *strech*, *shear* or *bending* restrictive conservation forces application;

3. each vertex writes on a previously specified *Frabe Buffer* pixel, and this buffer is returned as a 2D texture to the CPU, which at this moment has the per-node computed force accumulator;

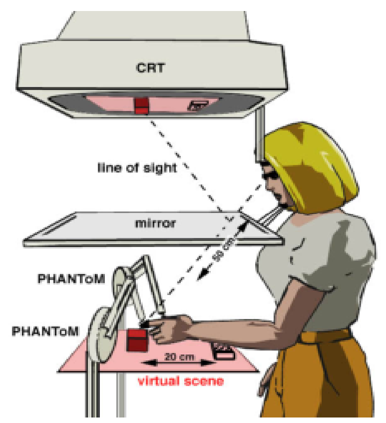
13.2.4 *Haptic*-Oriented Product

A **haptic device** is a sensor-positioned device able to interface the user via the sense of touch by applying forces, vibrations and/or motions to the user. In other words, a ***force-feedback* 3D device**.

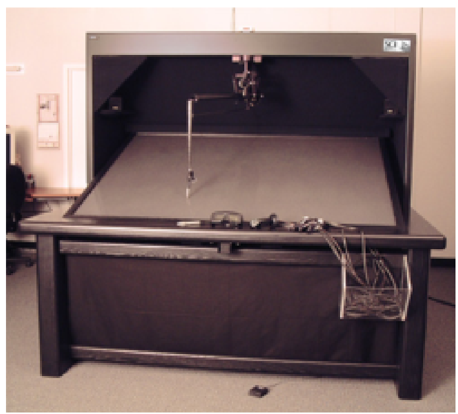
So, once the simulator has been completely developed, and all the desired deformation and cutting techniques have been tested and are satisfactory, the next step would be to **build an interactive system** based on a pen-shape *haptic* device. The shape of a pen should be ideal for simulating a **virtual scalpel** that would perform the deformations and cuttings.

13.2.4.1 The *workbench* prototype

The first visual support proposal associated to the *haptic* would be a ***workbench* displaying table**, like in the following images[95]:



(a) *A prototype graphical scheme.*



(b) *A prototype system.*

Figure 13.2: *Examples of 'workbench-haptic' virtual environment systems.*

By implementing for a system like this, the *haptic* device would be treated as the **virtual scalpel**, so the object splitting —or surgery incision— as long as the object external applied forces can be interactively done by the device, and an additional software layer should receive these actions and the physically-based model would react in **real time**.

Then, the experience will be more *real* due to an interactive device with touch-sensing and fully-functional as a 3D mouse, ideal to the 3D model interaction such as it was a real three-dimensional model.

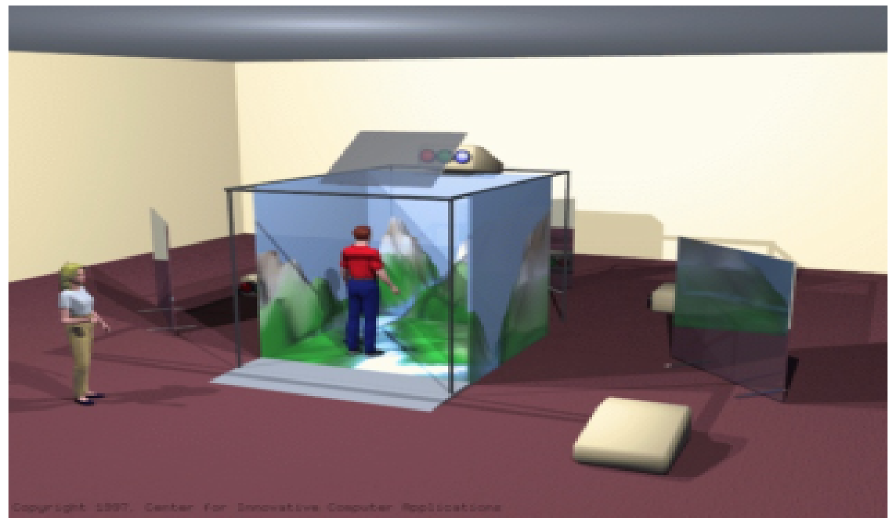
[95] The *workbench-haptic* figures are extracted from different webs; the graphical scheme in figure 13.2(a) is from `http://www.kyb.tuebingen.mpg.de/bu/projects/integration/semolab.html` and in 13.2(b), it's from `http://www.sci.utah.edu/stories/2001/fall_haptics_img-bench-aft.html` where the real prototype system is.

At CRV —*Centre de Realitat Virtual*—, at the UPC's *Edifici U*, a similar system is able to be used, so it should be a great idea to implement ForceReaQTor with *haptic*'s run-time feedback interaction.

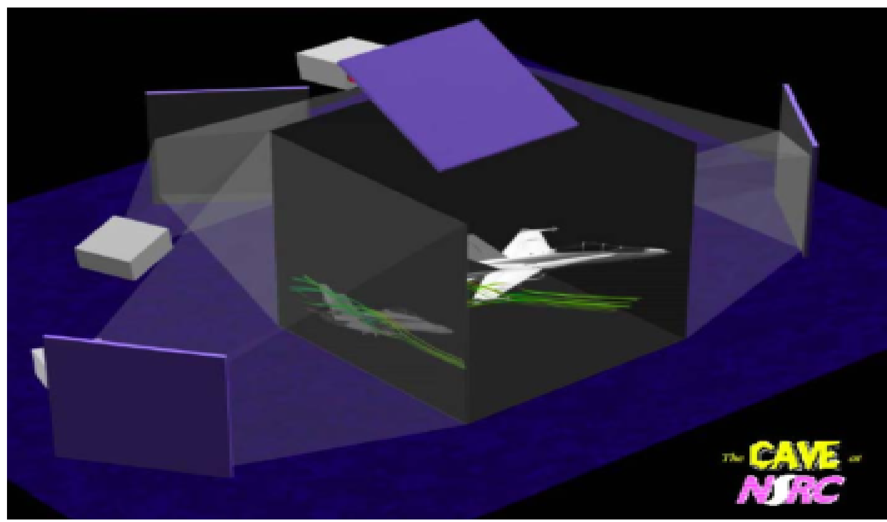
13.2.4.2 The CAVE-oriented environment

Another interesting proposal would be **changing the *workbench* with a CAVE virtual environment**, also available at CRV for virtual reality research purposes.

The CAVE is a $10 \times 10 \times 10$-foot structure placed in a $35 \times 25 \times 13$-foot darkened room, and consists in **rear-projected screen walls —and maybe ceil— with a front-projected floor** that are capable to show images on the floor, walls and ceil, that by using **stereoscopic 3D glasses**, immerses the user in a *real* 3D virtual world, since **images seem to float in space maintaining the user correct perspective visualisation** because of the glasses.



(a) *A 'really' virtual surrounding environment.*



(b) *The virtual 3D world generation by multi-camera interpolation.*

Figure 13.3: *The CAVE virtual environment scheme.*

By a control ***wanda*** **device**[96] **with associated *haptic* technology**, the interaction with the deformation system would be a great —and more real than with the workbench, due to the 'real' three-dimensionality— experience.

[96] A *wanda* is a classical-joystick similar control device that permits to capture 3D movements with position sensors; it has some action buttons and it's the basic tool for managing the virtual environment in a CAVE environment.

PART IV

Bibliography and Appendixes

APPENDIX

Bibliography

PROJECT'S FIRST CONTACT

[mot1] *Computer Animation*;
Jessica K. Hodgins, James F. O'Brien and Robert E. Bodenheimer Jr.
[www.cc.gatech.edu/gvu/animation/papers/ency.pdf]

[mot2] *Anatomy-Based Modeling of the Human Musculature*;
Ferdi Scheepers, Richard E. Parent, Wayne E.Carlson, Stephen F.May
SIGGRAPH 97 conference proceedings
[accad.osu.edu/~smay/Human/human.pdf]

[mot3] *Modeling and Deformation of the Human Body Using an Anatomically-Based Approach*;
Luciana Porcher Nedel, Daniel Thalmann
[ligwww.epfl.ch/~thalmann/papers.dir/CA98.muscles.pdf]

[mot4] *Wikipedia: Muscle*;
[http://en.wikipedia.org/wiki/Muscle]

[mot5] *Interactive Modeling of the Human Musculature*;
Amaury Aubel and Daniel Thalmann
[vrlab.epfl.ch/Publications/pdf/Aubel_Thalmann_CA_01.pdf]

[mot6] Creation and deformation of surfaces for the animation of human bodies
Nadia Magnenat-Thalmann, Daniel Thalmann
[ligwww.epfl.ch/~thalmann/papers.dir/deformations.MIT.pdf]

[mot7] *Anatomical Human Musculature Modeling for Real-time Deformation*;
ZUO Li, LI Jin-tao, WANG Zhao-qi
[wscg.zcu.cz/wscg2003/Papers_2003/C71.pdf]

[mot8] *Liver Surgery Planning Using Virtual Reality*;
Bernhard Reitinger, Alexander Bornik, Reinhard Beichel, and Dieter Schmalstieg
[www.computer.org/portal/cms_docs_cga/cga/content/Promo/promo3.pdf]

[mot9] *The Karlsruhe Endoscopic Surgery Trainer as an Example for Virtual Reality in Medical Education*;
U. Kühnapfel, Ch. Kuhn, M. Hübner, H.-G. Krumm, H. Maaß, B. Neisius
[www-kismet.iai.fzk.de/KISMET/docs/UKMITAT.html]

[motA] ***US Patent 7121832: Three-dimensional surgery simulation system***;
[www.freepatentsonline.com/7121832.html]

[cnt1] ***Stable Real-Time Deformations***;
Matthias Müuller, Julie Dorsey, Leonard McMillan, Robert Jagnow, Barbara Cutler
Proceedings of ACM SIGGRAPH Symposium on Computer Animation 2002
[graphics.cs.yale.edu/julie/pubs/Deform.pdf]

[cnt2] ***Surface Simplification Using Quadric Error Metrics***;
M. Garland and P. Heckbert
Proceedings of SIGGRAPH 97
[graphics.cs.uiuc.edu/~garland/papers/quadrics.pdf]

[cnt3] ***Progressive Meshes***;
Hughes Hoppe
[research.microsoft.com/~hoppe/pm.pdf]

[cnt4] ***Generalized Interactions Using Virtual Tools within the Spring Framework: Cutting***;
Cynthia D. Bruyns, Kevin Montgomery
[biocomp.stanford.edu/papers/mmvr/02/bruyns_mmvr_2002_cutting.pdf]

The State of the Art

[vox1] ***Wikipedia's Polygonal Mesh***;
[http://en.wikipedia.org/wiki/Polygon_mesh]

[vox2] ***Object Voxelization by Filtering***;
Milos Sramek
[http://www.viskom.oeaw.ac.at/~milos/page/vox/Voxelization.html]

[vox3] ***Volume Graphics***;
Arie E. Kaufman. et al
[www.cs.sunysb.edu/~vislab/projects/volume/Papers/Voxel/index.html]

[vox4] ***Cube Plane Intersection***;
Gernot Hoffmann
[www.fho-emden.de/~hoffmann/cubeplane12112006.pdf]

[vox5] ***Wikipedia's Sierpiński triangle***;
[http://en.wikipedia.org/wiki/Sierpinski_triangle]

[vox6] ***Sierpinski's Triangle Fractal***;
Leonard Belfroy
[http://www.machines-x.info/mathematics/TriangleFractalText.html]

[vox7] ***Fast 3D Triangle-Box Overlap Testing***;
Tomas Akenine-Möller
[www.cs.lth.se/home/Tomas_Akenine_Moller/pubs/tribox.pdf]

[vox8] ***Wikipedia's Binary Space Partitioning***;
[http://en.wikipedia.org/wiki/Binary_space_partitioning]

[vox9] ***Wikipedia's Kd-tree***;
[http://en.wikipedia.org/wiki/Kd-tree]

[voxA] ***Wikipedia's Octree***;
[http://en.wikipedia.org/wiki/Octree]

[voxB] ***Animació basada en física*** (in catalan);
Antonio Susin
[http://www.ma1.upc.edu/~susin/files/apuntsPart.pdf]

[mrs1] *Progressive Meshes*;
Hugh Hoppe
[research.microsoft.com/~hoppe/pm.pdf]

[mrs2] *Multiresolution 3D Rendering on Mobile Devices*;
Javier Lluch, Rafa Gaitán, Miguel Escrivá and Emilio Camahort
[www.gametools.org/archives/publications/UPV_jlluch_cggm2006.pdf]

[mrs3] *Wikipedia's Isosurface*;
[http://en.wikipedia.org/wiki/Isosurface]

[mrs4] *A Theory of Shape by Space Carving*;
Kiriakos N. Kutulakos
[www.cs.toronto.edu/~kyros/pubs/00.ijcv.carve.pdf]

[mrs5] *Hardware Accelerated Voxel Carving*;
Miguel Sainz, Nader Bagherzadeh and Antonio Susin
[www-ma1.upc.es/~susin/files/SIACG2002.pdf]

[mrs6] *Marching cubes: a high resolution 3D surface construction algorithm*;
W. Lorensen and H. Cline
ACM Computer Graphicsm 1987
[undergraduate.csse.uwa.edu.au/units/CITS4241/Project/references/Lorensen-Cline-brief.pdf]

[mrs7] *Algoritmo de construcción de superficies 3D de alta resolución: Marching Cubes* (in spanish);
Fernando Carmona, David Bitar, Miguel Cancelo, and Fernando Granado
[ma1.eii.us.es/miembros/rogodi/td0708/32/doc.pdf]

[mrs8] *The Marching Cubes Algorithm*;
Diane Lingrand et al
[http://www.polytech.unice.fr/~lingrand/MarchingCubes/algo.html]

[mrs9] *Topological considerations in isosurface generation*;
J. Wilhelms and A. Van Gelder
ACM Computer Graphics, 1990
[http://portal.acm.org/citation.cfm?id=902785]
Extended abstract at: [http://portal.acm.org/citation.cfm?id=99307.99325]

[mrsA] *Discretized Marching Cubes*;
C. Montani, R. Scateni, and R. Scopigno.
[kucg.korea.ac.kr/education/2003_2/vip618/paper/dmc.pdf]

[mrsB] *Polygonising a scalar field*;
Paul Bourke
[http://local.wasp.uwa.edu.au/~pbourke/geometry/polygonise/index.html]

[geo1] *Pointshop 3D: An Interactive System for Point-Based Surface Editing*;
Matthias Zwicker, Mark Pauly, Oliver Knoll, Markus Gross
SIGGRAPH 2002
[graphics.ethz.ch/Downloads/Publications/Papers/2002/Zwi02/Zwi02.pdf]

[geo2] *Shape Modeling with Point-sampled Geometry*;
M. Pauly, R. Keiser, L. Kobbelt, M. Gross,
SIGGRAPH 2003
[portal.acm.org/citation.cfm?id=882319]

[geo3] Contact *Handling for Deformable Point-Based Objects*;
Richard Keiser, Matthias Müller, Bruno Heidelberger, Matthias Teschner, Markus Gross
[www.beosil.com/download/ContactHandling_VMV04.pdf]

[geo4] *Point Based Animation of Elastic, Plastic and Melting Objects*;
M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross and M. Alexa
[graphics.ethz.ch/~mattmuel/publications/sca04.pdf]

[geo5] *Wikipedia's Moving Least Squares*;
[http://en.wikipedia.org/wiki/Moving_least_squares]

[geo6] ***Wikipedia's Jacobian Matrix***;
[http://en.wikipedia.org/wiki/Jacobian]

[geo7] ***Global and local deformations of solid primitives***;
Alan H. Barr
SIGGRAPH 1984
[http://portal.acm.org/citation.cfm?id=808573]

[geo8] ***Wikipedia's Tangent Space***;
[http://en.wikipedia.org/wiki/Tangent_space]

[geo9] ***Wikipedia's Manifold***;
[http://en.wikipedia.org/wiki/Manifold]

[geoA] ***Models Deformables***; (in catalan and english)
Antonio Susin
N/A link

[geoB] ***A Framework for Geometric Warps and Deformations***;
Tim Milliron, Robert J. Jensen, Ronen Barzel, Adam Finkelstein
[www.ronenbarzel.org/papers/warp.pdf]

[geoC] ***Free-Form Deformation of Solid Geometric Models***;
Thomas W. Sederberg, Scott R. Parry
SIGGRAPH 1986
[portal.acm.org/citation.cfm?id=15903]

[geoD] ***Interactive Axial Deformations***;
Francis Lazarus, Sabine Coquillart and Pierre Jancène
IFIP Working Conference on Geometric Modeling and Computer Graphics, 1993
[ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-1891.pdf]

[geoE] ***Wires: A Geometric Deformation Technique***;
Karan Singh, Eugene Fiume (Alias—Wavefront)
[www.dgp.toronto.edu/ karan/pdf/ksinghpaperwire.pdf]

[phy1] ***Advanced Character Physics***;
Thomas Jakobsen (IO Interactive)
[www.ma1.upc.edu/∼susin/files/AdvancedCharacterPhysics.pdf]

[phy2] ***Curs de Modelatge de Volums: Models Deformables***;
Antonio Susin
N/A link

[phy3] ***Models Deformables***; (in catalan and english)
Antonio Susin
N/A link

[phy4] ***Analysis of numerical methods for the simulation of deformable models***;
Michael Hauth, Olaf EtzmuSS, Wolfgang StraSSer
[springerlink.metapress.com/content/aaxt761nm1wvvbch/]

[phy5] ***Physically Based Deformable Models in Computer Graphics***;
Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman and Mark Carlson
[graphics.ethz.ch/∼mattmuel/publications/egstar2005.pdf]

[phy6] ***Decomposing Cloth***;
Eddy Boxerman and Uri Ascher
[www.cs.ubc.ca/spider/ascher/papers/ba.pdf]

[phy7] ***Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior***;
Xavier Provot
[graphics.stanford.edu/courses/cs468-02-winter/Papers/Rigidcloth.pdf]

[phy8] ***Cloth Animation***;
Christopher Twigg
[www.cs.cmu.edu/∼djames/pbmis/notes/pbmis_class14_cloth.pdf]

[phy9] *A Survey on Cloth Simulation*;
Yongjoon Lee
[tclab.kaist.ac.kr/~sungeui/SGA07/Students/YJL-mid.pdf]

[phyA] *Dynamic Real-Time Deformations using Space & Time Adaptive Sampling*;
Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, Alan H. Barr
[www-evasion.imag.fr/Publications/2001/DDCB01/DDCB01.pdf]

[phyB] *Desarrollo de un mètodo mixto malla-partícula para la animación de fluidos.*; (in spanish)
N. Suárez, A. Susin
[www-ma1.upc.es/~susin/files/CEIG05-Fluidos.pdf]

[phyC] *Fluid Simulation for Computer Animation*;
Greg Turk
[www-static.cc.gatech.edu/~turk/powerpoint_presentations/fluids.ppt]

[phyD] *Interactive Animation of Cloth-like Objects in Virtual Reality*;
Mark Meyer, Gilles Debunne, Mathieu Desbrun, Alan H. Barr
[http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.7036&rep=rep1&type=pdf]

[phyE] **emphPump it up: computer animation of a biomechanically based model of muscle using the finite element method**;
David T. Chen, David Zeltzer
SIGGRAPH 1992
[http://portal.acm.org/citation.cfm?id=134016]

[phyF] *A 3D Discrfete Model of the Diaphragm and Human Trunk* Emmanuel Promayon, Pierre Baconnier
[www.esaim-proc.org/articles/proc/pdf/2008/02/proc082305.pdf]

[cut1] *A Virtual Node Algorithm for Changing Mesh Topology During Simulation*;
Neil Molino, Zhaosheng Bao, Ron Fedkiw
[physbam.stanford.edu/~fedkiw/papers/stanford2004-01.pdf]

[cut2] *Hybrid Cutting of Deformable Solids*;
Denis Steinemann, Matthias Harders, Markus Gross, Gabor Szekely
[ieeexplore.ieee.org/iel5/11055/34910/01667624.pdf]

[cut3] *Generalized Interactions Using Virtual Tools within the Spring Framework: Cutting*;
Cynthia D. Bruyns, Kevin Montgomery
[biocomp.stanford.edu/papers/mmvr/02/bruyns_mmvr_2002_cutting.pdf]

[cut4] *Action Areas: On interacting in a virtual environment*;
Cynthia D. Bruyns, Steven Senger, Simon Wildermuth and Kevin Montgomery
[www.vis.uni-stuttgart.de/vmv01/dl/posters/9.pdf]

[cut5] *Real-time Incision Simulation Using Discontinuous Free Form Deformation*;
Guy Sela, Sagi Schein, and Gershon Elber
[www.cs.technion.ac.il/~schein/publications/Scalpel.pdf]

[cut7] *Combining FEM and cutting: a first approach*;
N/A —master thesis chapter—
[igitur-archive.library.uu.nl/dissertations/2003-0721-093328/c3.pdf]

[cut8] *A Delaunay approach to interactive cutting in triangulated surfaces*;
N/A —master thesis chapter—
[igitur-archive.library.uu.nl/dissertations/2003-0721-093328/c5.pdf]

Contributions and Results

[lib1] *Wikipedia's C++*;
[http://en.wikipedia.org/wiki/C%2B%2B]

[lib2] ***Wikipedia's Qt***;
[http://en.wikipedia.org/wiki/Qt]

[lib3] ***Wikipedia's Matlab***;
[http://en.wikipedia.org/wiki/Matlab]

[gpu1] ***Wikipedia's OpenGL***;
[http://en.wikipedia.org/wiki/OpenGL]

[gpu2] ***Programming Graphics Hardware***;
Randy Fernando, Cyril Zeller (nVidia Developer Technology Group), I3D
[http://s217587280.mialojamiento.es/resources/I3D_05_IntroductionToGPU.pdf]

[gpu3] ***Programming the GPU: High-Level Shading Languages***;
Randy Fernando (nVidia Developer Technology Group), I3D
[http://www.lsi.upc.edu/~ppau/filesWeb/I3D_05_ProgrammingLanguages.pdf]

[gpu4] ***Introduction, OpenGL basics, OpenGL buffers***;
Pere-Pau Vázquez
[http://s217587280.mialojamiento.es/resources/OpenGL1.pdf]

[gpu5] ***GPU Programming***;
Pere-Pau Vázquez
[http://www.lsi.upc.edu/~ppau/filesWeb/GPUProgramming1.pdf]

[gpu6] ***GPU Programming, GLSL Language Description***;
[http://www.lsi.upc.edu/~ppau/filesWeb/GPUProgramming3.pdf]

[gpu7] ***Wikipedia's Frame Buffer Object***;
[http://en.wikipedia.org/wiki/Framebuffer_Object]

[gpu8] ***Dominik Göddeke's GPGPU Tutorials***;
Dominik Göddeke
[http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/index.html]

[gpu9] ***Dominik Göddeke's GPGPU Basic Math Tutorial***;
Dominik Göddeke
[http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html]

[gpuA] ***Programming the GPU ... and a brief intro to OpenGL Shading Language***;
Marcelo Cohen
[www.comp.leeds.ac.uk/vvr/papers/gpu-vvr2.pdf]

[gpuB] ***nVidia's CUDA Home Page***;
[http://www.nvidia.com/object/cuda_get.html]

[gpuC] ***Wikipedia's CUDA***;
[http://en.wikipedia.org/wiki/CUDA]

[gpuD] ***SUPERCOMPUTING 2007 Tutorial: High Performance Computing with CUDA, 02. Programming CUDA***;
Massimiliano Fatica, David Luebke (nVIDIA Corporation)
[http://www.gpgpu.org/sc2007/]

[gpuE] ***Introduction to OpenGL***;
Michael Papasimeon
[www.cs.mu.oz.au/380/2007/slides/OpenGL_6up.pdf]

[gpuF] ***Computer Graphics: Introduction and Graphics Primitives***;
PM Burrage, GB Ericksson
[www.itee.uq.edu.au/~comp3201/OGL1_4.pdf]

[gpuG] ***GLSL Quick Reference Guide***;
[www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf]

[ami1] ***Wikipedia's UML***;
[http://en.wikipedia.org/wiki/Unified_Modeling_Language]

[ami2] ***Wikipedia's OBJ file format***;
[http://en.wikipedia.org/wiki/Obj]

[ami3] ***OBJ format***;
[www.eg-models.de/formats/Format_Obj.html]

[ami4] *OBJ file format complete specification v3.0*;
[www.martinreddy.net/gfx/3d/OBJ.spec]

[ami5] *Material definitions for OBJ files*;
[https://people.scs.fsu.edu/∼burkardt/data/mtl/mtl.html]

[ami6] *The NLM's Visual Human Project*;
[www.nlm.nih.gov/research/visible/visible_human.html]

[ami7] *OpenGL Vertex Arrays*;
[www.opengl.org/documentation/specs/version1.1/glspec1.1/node21.html]

[ami8] *Wikipedia's Phong Shading*;
[http://en.wikipedia.org/wiki/Phong_shading]

CONCLUSIONS AND FUTURE WORK

[] · · · ∅ · · ·

APPENDIX B

End-User Software Data Sheet: 'MeshInspeQTor'

B.1 Product Overview

B.1.1 Highlights

MeshInspeQTor is a graphical application that permits the user **open 3D models and visualizing them in a great variety of modes**. Additionally, there is a **cutting modality** that, specifying some cutting planes, the 3D model will be visible only in the upper plane subspaces; this is a useful tool for inspecting the model.

Besides, you can use MeshInspeQTor for **generating fast and robust *voxelmaps***; that is, volumetric representations of any 3D scene involving some 3D models. Even, you can refill them or editing these volumetric representations with a ***voxel navigator* tool**.

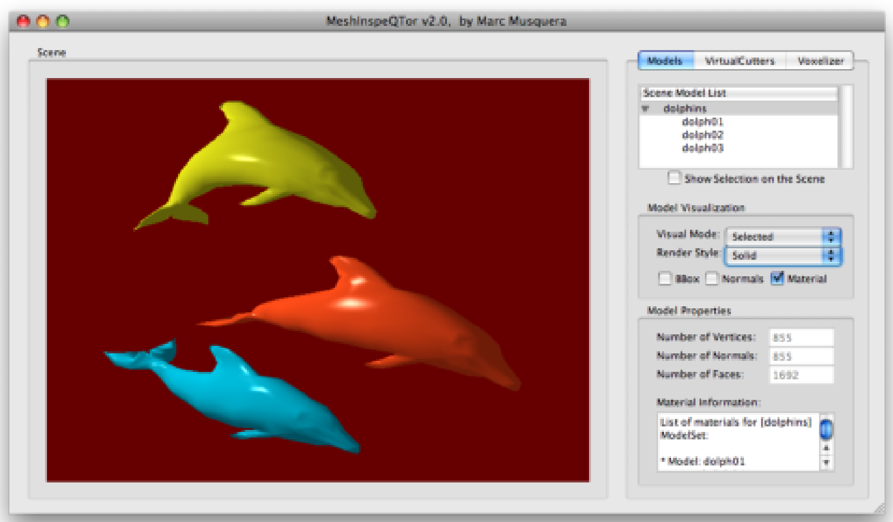


Figure B.1: *MeshInspeQTor rendering the* `dolphins` *model with materials and phong algorithm illumination. Notice the three application panel functionalities:* models, virtualcutters *and* voxelizer.

B.1.2 Key Features

- OBJ **model** format **importation**, with **material** specification included;
- mesh visualization editor, including **four kinds of mesh rendering**: *solid*, *wireframe*, *nodeset* and *draft*, a non-smoothed face remarking illumination method;
- capable to **dissect any 3D model** with the cutting planes;
- **property SCN file format**, comprised by the 3D models —plus their visual settings— and the previously set cutting planes;
- scene concept totally **editable**; the 3D models, as the cutting planes, can be **included or rejected from the scene, or hidden** —so, their occuped space matters—;
- **creation and edition of *voxelmaps*** automatically from the actual 3D scene, or manually with the *voxelmap navigator*, with a previously specified sizes from $< 1 \times 1 \times 1 >$ to $< 256 \times 256 \times 256 >$ regular *voxel* grid;
- besides, the **automatic *voxelmap* generation** from the 3D scene is **basic-set-theory capable**, so the generated *voxelmap* can be the result for intersection or union of models —or its inverted spaces—;
- the *voxelmap navigator* can manipulate automatically-generated *voxelmaps*;
- automatic *voxelmap* refilling with **two different carving methods**;
- the *voxelmaps* are totally able to be saved for future loadings with a **property VOX file format**; indeed, this file format specification is the input for the other software app in the package, ForceReaQTor —se appendix C for more information—;

B.2 Product Details

B.2.1 *Models* tab: 3D Model Visualization

B.2.1.1 Setting the 3D scene

This program works with the concept of **scenes**; that is, a set of 3D models within the same 3D space; besides, each OBJ file format can include more than one object, so **MeshInspeQTor is able to charge so much OBJ files to the same scene**.

So, once the models are available on the scene, they are shown on the **scene screen**, with the point-of-view centered on the *scene center*[97] and with al the models viewable —except for occlusions, of course—. They will be illuminated by a white light placed at north-west, behind the user position.

Then, each 3D object —from now, they will be called **entities**, or **group entities** if it's a group containing various entities— can be consulted, by clicking on the model selection panel, for **getting information about**:

[97] That is, the geometric center counting all the models. This center of scene point is not necessarily an existing point within one of the models.

1. number of **vertices**, **normals** and **faces**;

2. **material** info —*i.e.*, the ambient, diffuse and specular color emission—;

and also for selecting the **entity visualization properties**, that really modifies the visual style and scene involucration of the entity set:

- an **entity** can be, though charged on the scene, **hidden or rejected**;

- you can select four types of rendering: **solid**, **wireframe**, **nodeset**, and **draft**, a non-beuaty rendering mode with great usefulness for mesh composition purposes: it's a **mix of solid and wireframe modes, but with non-smoothing illumination and no material properties**;

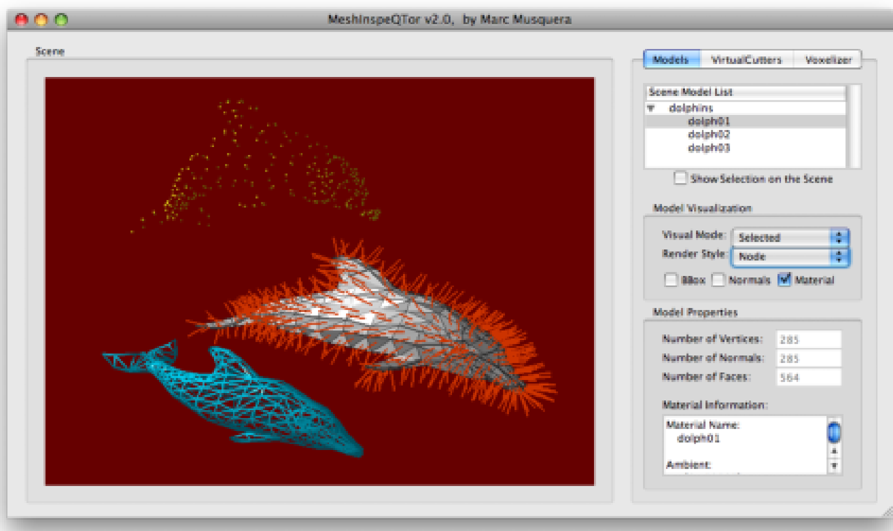


Figure B.2: *From up to down, wireframe, draft —with normal displaying; see below— and nodeset renderings of the* `dolphins` *model. Notice in the right panels the information about the* `dolphin1` *entity, already selected:* 285 *vertices composing* 564 *faces.*

Also, some **add-on visualizations** are available for every model:

1. the entity **material** (in case of its existence) can be disabled;

2. the **individual bounding box** for each entity or group entity is displayable;

3. there's the possibility of rendering all the entities' **face normals**;

B.2.1.2 Controls and management

Everything is saveable —and loadable— by the `File` menu, from where the user can **reset the scene, load a new one or save the actual one on a SCN file, or importing a new OBJ file** to the current scene. And additionally[98] from the `Camera` menu it's possible to center the camera again or changing the **camera point-of-view: top, side or front** —front camera is the default configuration—.

[98] And this involves the general application visualization, not only the 3D model —entity— displaying; that is, the following text is compatible with the three tabs of MeshInspeQTor state panel.

Besides, the **scene display screen is interactive** in terms of camera configuration by dragging the **mouse** while one of the three buttons are pressed[99]:

Mouse Button	Action while dragging
Left	Scene Rotation
Middle	Center-of-Scene Displacement
Right	Camera Zoom

Table B.1: *MeshInspeQTor mouse controls.*

B.2.2 *VirtualCutters* tab: The Mesh Dissection Process

B.2.2.1 Setting a Virtual Cutters

As said previously, a VirtualCutter is a plane that, once specified his **normal vector** (a, b, c) **and the displacement value** d, the plane equation

$$\pi_{vc} \ : \ ax \ + \ by \ + \ cz \ + \ d \ = \ 0$$

is assigned to it; those values can be **modified**, as long as can also be **deleted**. From this moment, the 3D scene will be visible only where it's **positive in relation to the VirtualCutter**; that is, following the **point-plane relative position cases**, being $p = (p_x, p_y, p_z)$:

$$\begin{cases} \pi_{vc}(p): & ap_x + bp_y + cp_z + d \ > \ 0 \implies \oplus \ \equiv \ p > \pi_{vc} \\ \pi_{vc}(p): & ap_x + bp_y + cp_z + d \ = \ 0 \implies \varnothing \ \equiv \ p \in \pi_{vc} \\ \pi_{vc}(p): & ap_x + bp_y + cp_z + d \ < \ 0 \implies \ominus \ \equiv \ p < \pi_{vc} \end{cases}$$

Also, as in the case of the 3d entities, **a VirtualCutter can be stored but can be selected —active and displayed—, hidden —active but invisible— and rejected —no loaded—**. In the following figure it's shown the dissection effect of the VirtualCutters:

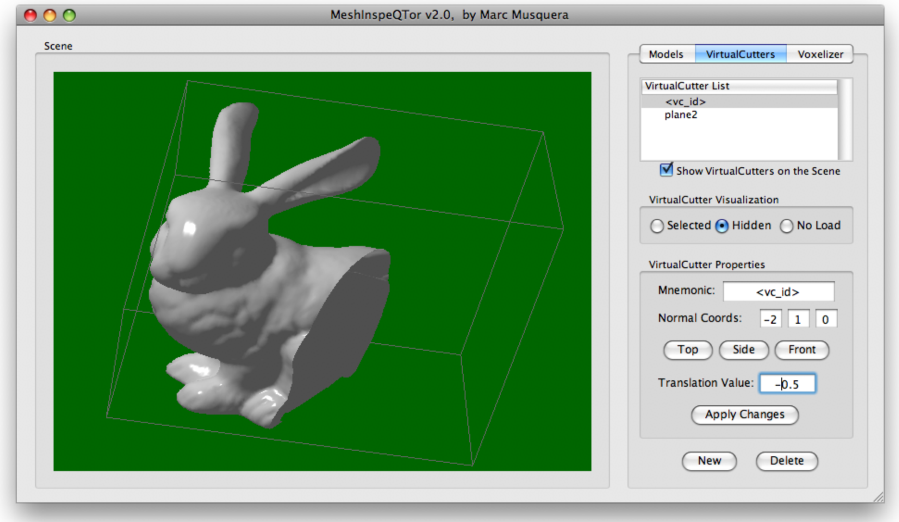


Figure B.3: *Here's the* `bunnycapped` *model in dissection by a VirtualCutter.*

[99] This is applicable to all the MeshInspeQTor panels and functionalities, not only the *models* one.

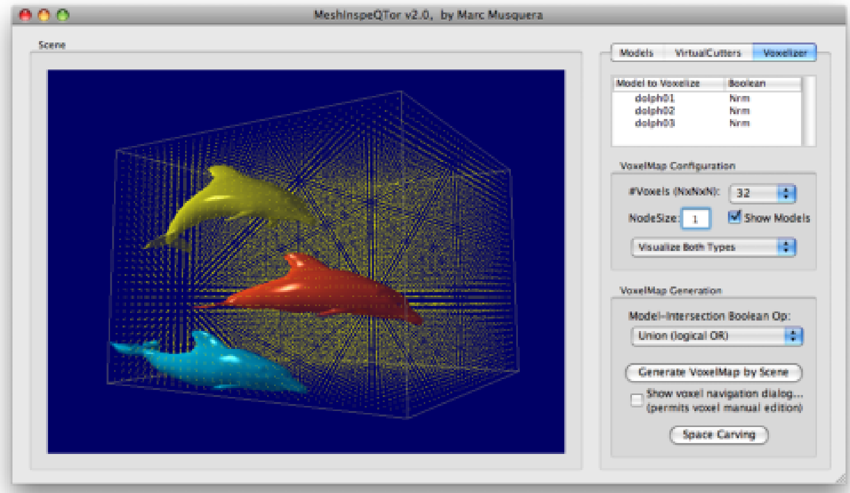
B.2.2.2 Management

The VirtualCutters are **implicitly** —if there is any VirtualCutter set— **saved in a SCN file**; so, if you want to save a set of cutting planes, you must do it by **saving a scene** —with or without entities, it doesn't matter—.

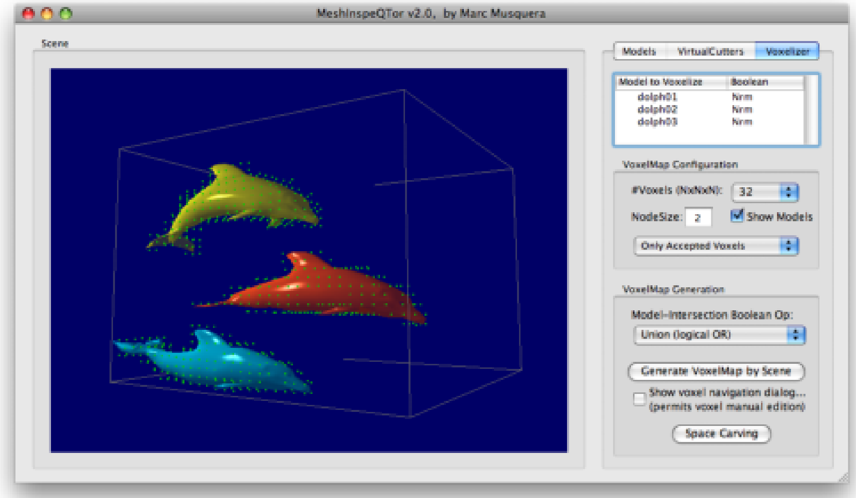
B.2.3 *Voxelizer* tab: The VoxelMaps, Volumetric Representations

B.2.3.1 Building a Voxelmap from a 3D scene

Once the 3D scene is set as the user wants, it's time for enter to the `Voxelizer` panel, selecting the desired VoxelMap size (it will be always a **regular grid**) and **click** on the `Generate VoxelMap by Scene` button. Then, the **scene voxelization** will begin.



(a) *The voxelization's previous step: scene bounding box plenty of null —yellow— voxels.*



(b) *The final result. The valid nodes —in green— now have size 2 for better visualization.*

Figure B.4: *The previous and final* `dolphins`*' $< 32 \times 32 \times 32 >$ voxelmap automatic generation steps.*

B.2.3.2 Basic Set Theory applied to Voxelization Method

This is the basic voxelization method: the **union of all entities's single voxelization**; however, MeshInspeQTor permits applying **basic set theory** to the voxelization method, and the **intersection** of voxelizations is also available. Besides, every entity can have assigned the `nrm` or the `inv` values; the first one is the standard entity but the second one will apply the **inverse of that entity voxelization, thus, its complement**.

In this way, using the `complementary` - unary operator with the binary operators `union` $\cup$ and `intersection` $\cap$[100], **some complex voxelizations are possible to be got**:

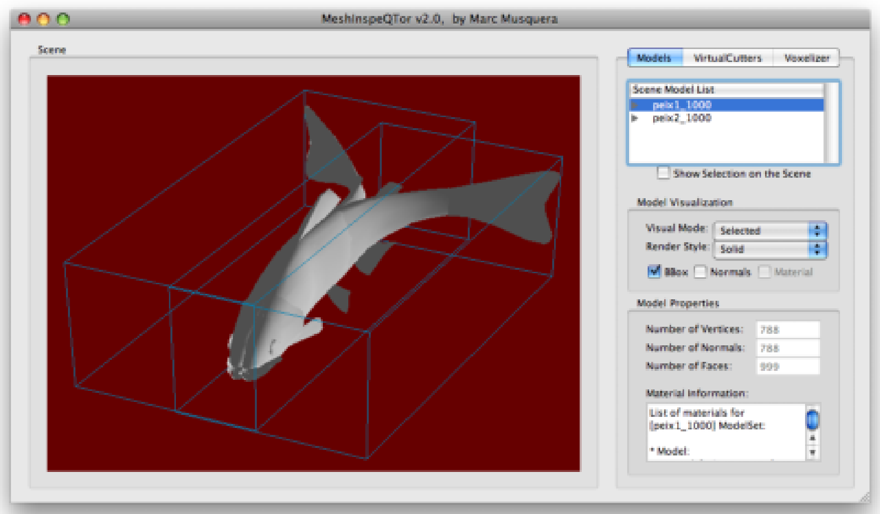


Figure B.5: *Two fish* 3D *models,* `fish1` *and* `fish2`*, will be used to obtain complex voxelizations by using basic set theory. Notice that the entity individual bounding boxes are really intersecting.*

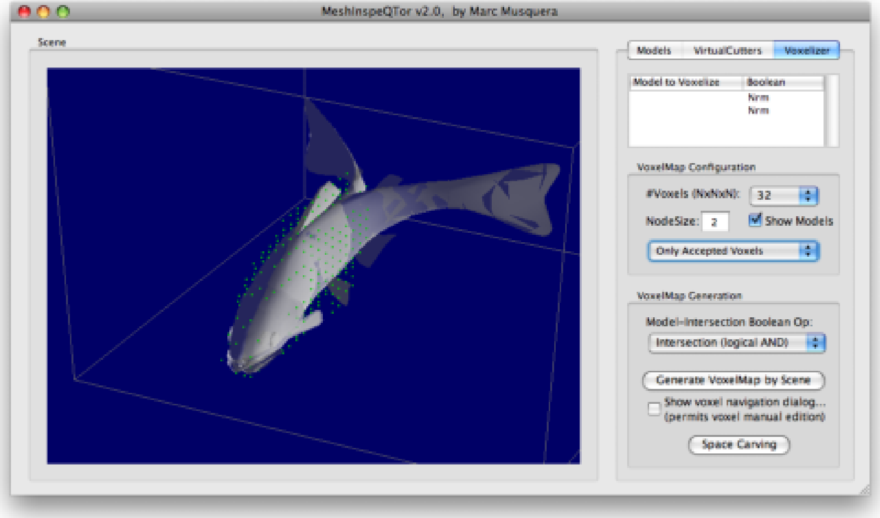


Figure B.6: *The* $< 32 \times 32 \times 32 >$ `fish1` $\cap$ `fish2` *automatic voxelization. Let's see that the valid nodes only those that were valid on the both two voxelizations.*

[100] However, `union` and `intersection` will be applied to all entities at the same time, so it's not possible to apply `intersection` and `union` at the same time.

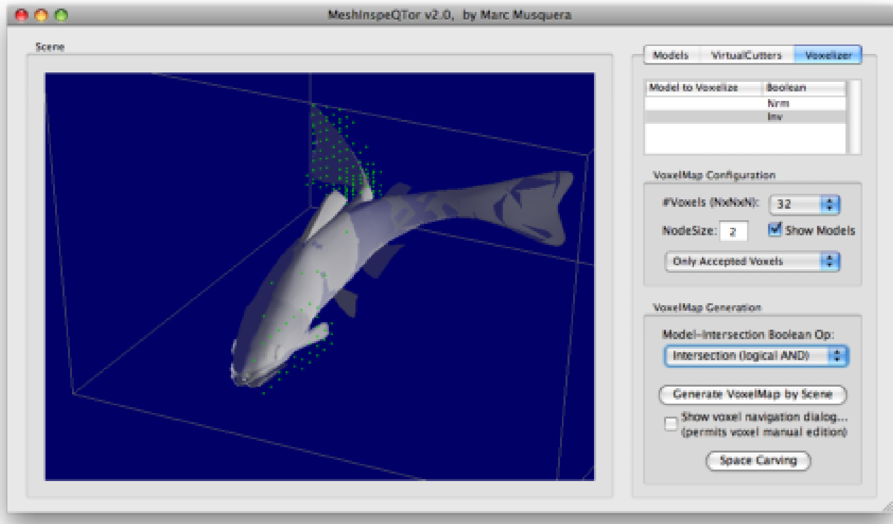


Figure B.7: *The* $< 32 \times 32 \times 32 >$ *VoxelMap equivalent to those voxels that are containing* `fish1` *but not* `fish2`*: so, the set theory operation is* $[\texttt{fish1} \cap \overline{\texttt{fish2}}]$.

B.2.3.3 Using the VoxelMap Navigator

MeshInspeQTor is capable, as it's been seen, for generating voxelizations from a 3D scene even applying basic set theory with the entities as sets. However, there's also the **possibility to modify a voxelmap manually**, with the **Voxelizer Navigator**, that allows the user:

- edit a cubic finite voxelmap region with the values valid, invalid or inverting the actual value; see figure B.8 for an example within the `torus` voxelmap;

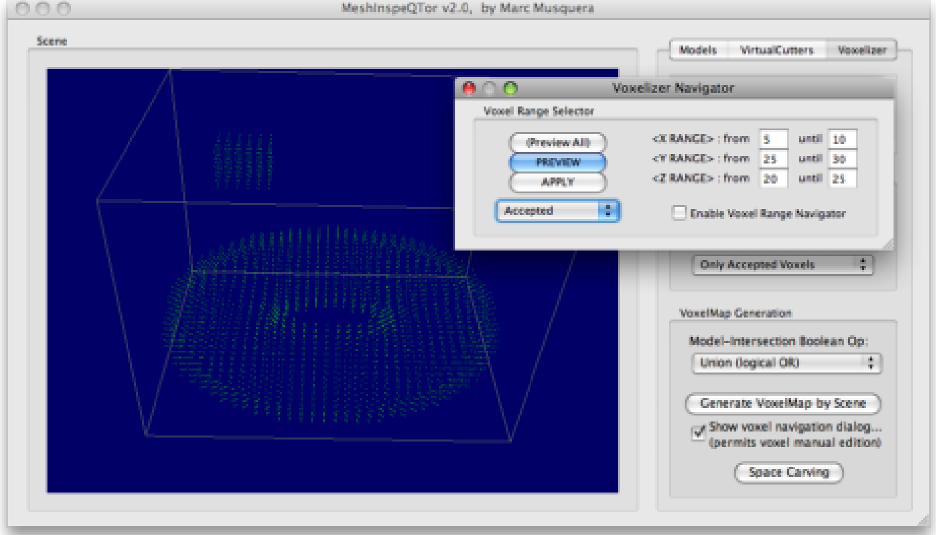


Figure B.8: *Voxelizer Navigator's manipulation example. Notice the added* cube *on the upper left screen part, within the* `torus` *voxelmap.*

- view a enclosed —cubic— range of the entire `torus` voxelmap, as it's shown on figure B.9 in the next page;

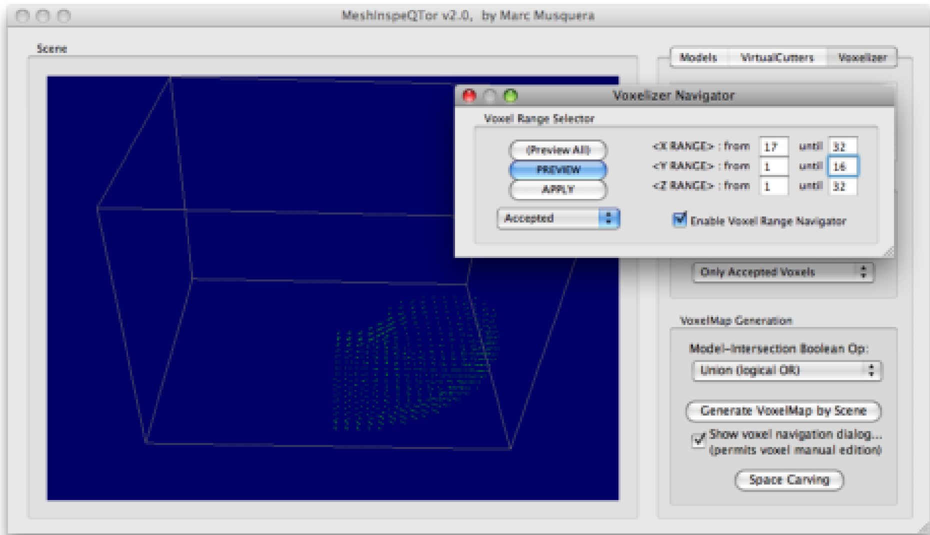


Figure B.9: *A voxelmap finite range visible with the Voxelizer Navigator's range visualization option.*

B.2.3.4 Carving a VoxelMap

When you request for a 3D scene voxelization, it's correctly and robustly well-done. However, a 3D object is only *simulating* a solid object, so the polygonal mesh is representing the object surface —that is, the **inner object space is really empty**—.

It's clearly viewable in the following figure, representing the voxelization of a **special scene composed specially for this thesis**, comprising a cylinder with two more non-intersecting objects within it: a cube and a sphere[101]; this 3D scene has been called `sphecylcub`, a obvious contraction of the scene composing geometric shapes.

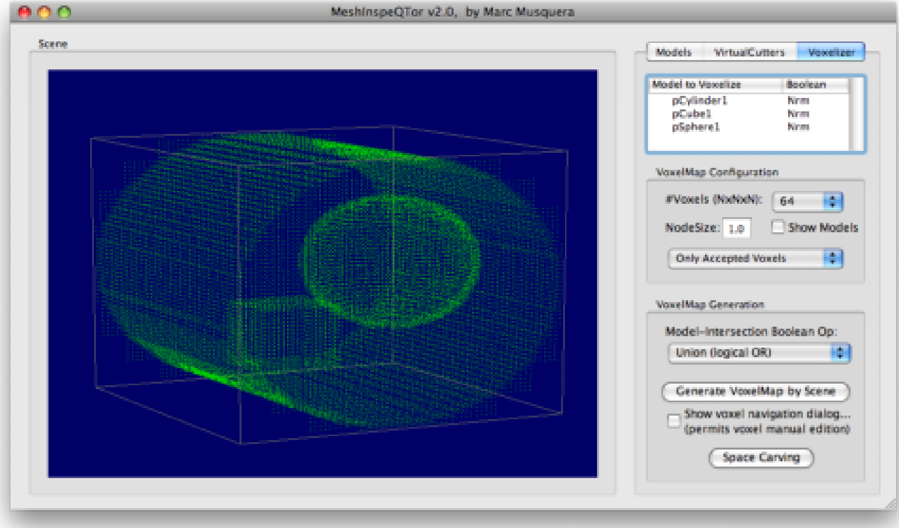


Figure B.10: *The thesis carving purpose scene* `sphecylcub` $< 64 \times 64 \times 64 >$ *VoxelMap. Notice the emptyness of the cylinder inner space, so the contained objects —cube and sphere— are also empty and clearly visualizable.*

[101] This voxelization is done with all the objects in `nrm` mode and the `union` set operation. See the section B.2.3.2 for more information about basic set theory on voxelmaps.

In the next figure a voxelization portion is visible, for more bright visualization of the respective emptyness of all object voxelizations.

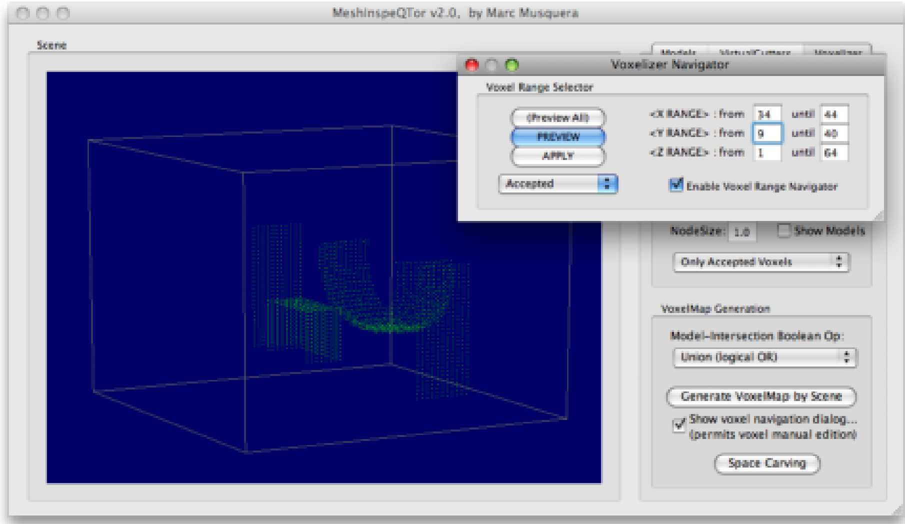


Figure B.11: *Portion of the* $< 64 \times 64 \times 64 >$ `sphecylcub` *voxelization that shows the absolute emptyness of the standard* 3D *scene objects voxelization.*

So, the **voxel space carving** is the **ideal tool for refill empty volumetric representations**. If you want to voxelize a 3D scene, and you want it **completely volumetric**, solid, simply click on `Space Carving` button after the voxelization is executed. Then, a **method selection dialog** like the featured in the next figure will appear **—two space carving methods are available—**:

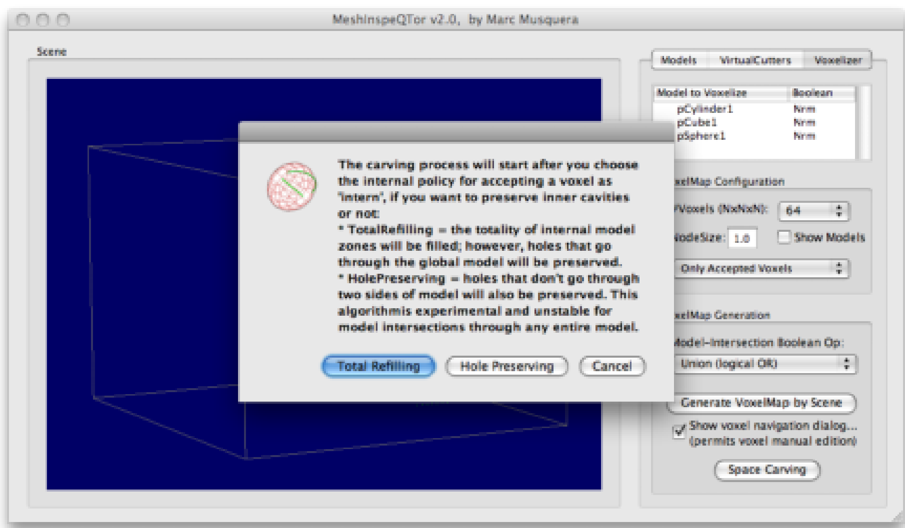


Figure B.12: *Space Carving policy selection dialog.*

- the **total refilling** algorithm is the most simple and the most obvious at the same time. It's performed as six cameras were on each scene bounding box face, focusing the scene itself, and the resulting refilling is the intersection of the six camera projections. The final result is **correct for convex meshes, but if the mesh is not convex, there can exist some incorrectly refilled zones**.

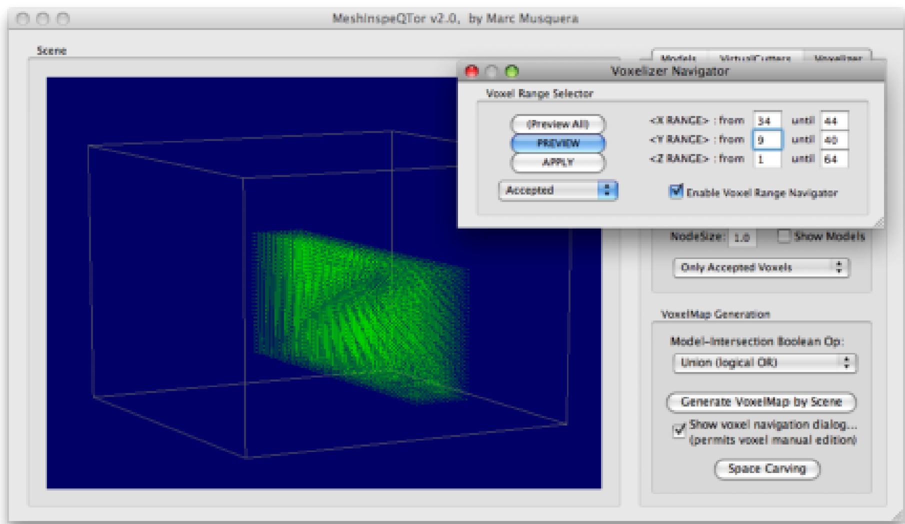


Figure B.13: *Space Carving done with 'Total Refilling' policy. Notice that all the inner zone corresponding to the more exterior voxelization is totally refilled.*

- the **hole preserving** algorithm is **ideal for non-convex objects**, or as in the case, objects containing another objects that are representing geometric holes. This method performs the same method as the previous carving policy but now **when another voxelization —inner to an already visited one— is encountered, it's considered a hole**. The final result can be seen in the following figure;

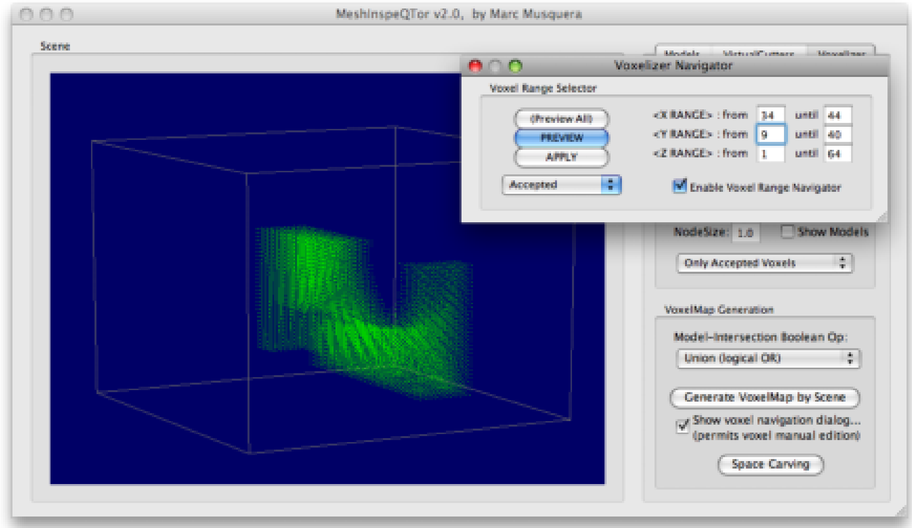


Figure B.14: *Space Carving done with 'Hole Preserving' policy. Notice that the more exterior voxelization is refilled, but the inner ones are considered holes, so they are still empty.*

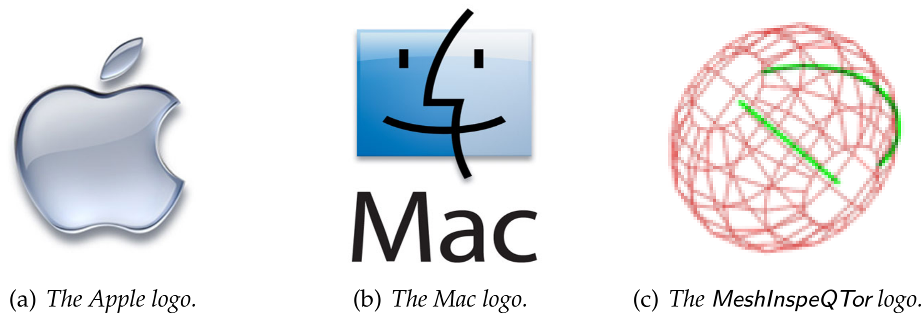
B.2.3.5 Management

The VoxelMaps, carved or not carved, manually or automatically generated, can be **saved to, or loaded from, the VOX file format**, that contains all the necessary information about the VoxelMap, independently from the 3D scene —in case it exists—. Besides, once a voxelmap is loaded, it can be modified as always as the user wants by the Voxelizer Navigator.

If you want to save additionally the scene parameters, you will have to save the scene on a separate SCN file. However, they won't be directly relationated, they will be two really different files with no connection.

B.3 Requirements and recommendations

MeshInspeQTor is a capable **3D model visualizer for OBJ file format objects**, with a information panel and a lot of visualization settings and modes. Also, it offers the possibility of **dissect any model** —or set of models, called scene— by the VirtualCutter panel, and aditionally, by using the Voxelizer panel a **scene complex volumetric representation can be generated and modified**.



(a) *The Apple logo.* (b) *The Mac logo.* (c) *The MeshInspeQTor logo.*

Figure B.15: *MeshInspeQTor software and hardware requirements logos.*

This application has been developed for Apple Mac Intel platforms using the QT4 framework for GUI design and OPENGL API for the graphics displaying. A *Shader Model 3.0* compatible GPU is **recommended** for the model visualization, and **required** for the virtualcutter dissection processes.

APPENDIX

End-User Software Data Sheet: 'ForceReaQTor'

C.1 Product Overview

C.1.1 Highlights

ForceReaQTor is a **visual dynamic deformable system simulator** able to generate, from any previously generated *voxelmap*, a **non-rigid 3D skeleton** that can be deformed by dynamic force set application in a enclosed cycle of time, following **real-time physically-based laws** with collision control.

Also, the simulator is able to perform some cuttings like a **virtual scalpel** usability, for virtual surgery purposes. A mesh will be capable to be **cut and subdivided so much times user wants**, until the 3D skeleton reaches his deformation limit.

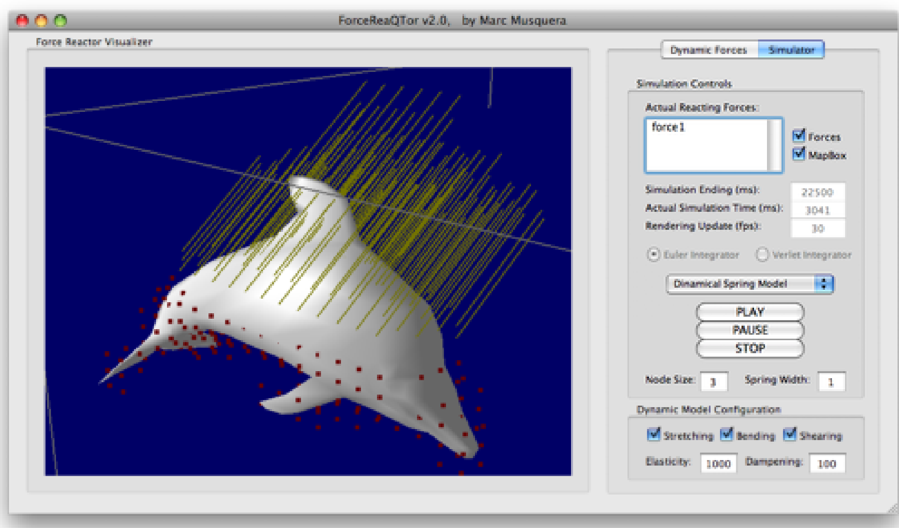


Figure C.1: *ForceReaQTor's simulation example by deformable forces to* `dolphin` *model.*

C.1.2 Key Features

- VOX **model** format **importation** —see appendix B for more information—;
- three kinds of **visualization**: textsc3d non-rigid skeleton, original mesh —if it exists—, and regenerated mesh (with polygon density in direct proportion to the 3D skeleton structure density);
- **all the visualizations can be deformed in the same way** by the same dynamic force configuration;
- **complete simulation settings tab**, with capabilities for *(i)* timeline controlling, *(ii)* dynamic forces configuration (with preview), *(iii)* active cuttings; *(iv)* besides, some 3D skeleton parts are capable to be fixed —rigid—;
- there's **no real-time interactive user-model interfacing** but, as said, a set of forces and cuttings through time; this set can be saveable and loadable by the **property DFL file format**;
- the **simulator performs the previously set dynamic force list**, and can be paused or stopped;
- the simulation is FPS-rate-controlling: **you can set the performing FPS-rate**, submitted to an error message if the desired rate is too much for the computer and/or algorithm capabilities;
- the physically-based deformable system has some **configurations able to be enabled and disabled in simulation run-time**: three kinds of real-time deformable **relaxing inner forces**: **stretch, shear** and **bending** and the value of two kinds of **relaxing parameters**: **elasticity** and **dampening**;

C.2 Product Details

C.2.1 *Dynamic Force* tab: The 3D Dynamic Deformation Model settings

C.2.1.1 A little introduction; the deformable skeletons and the dynamic force lists

As a **deformable model simulator with non-interactive interface but list of timeline-related deformable actions**, ForceReaQTor works with two basic data formats:

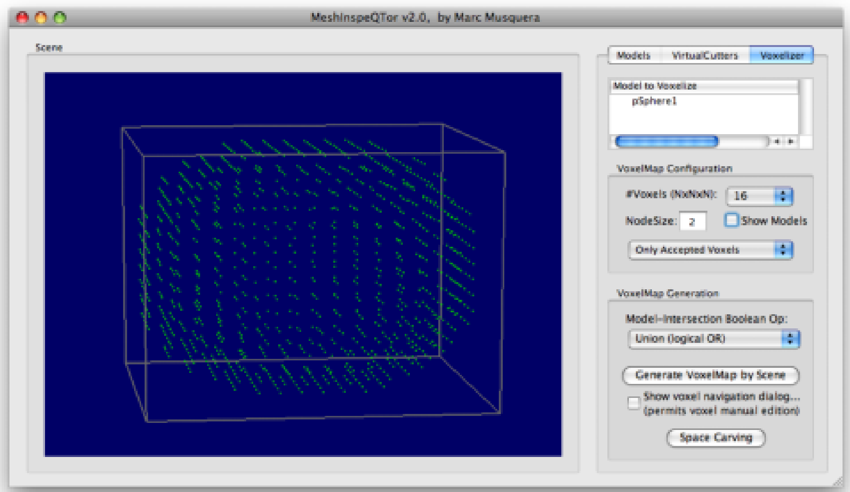
1. a 3D *non-rigid skeleton*,
2. and a *dynamic force list*, completely configurable, applied to this skeleton for deformating it following realistic physical laws;

However, it's important to notice that, though the force list will be applied to a deformable skeleton, these two data structures have **no implicit relation between them**, according to the ForceReaQTor specifications. That means, the same dynamic force list can be applied to different skeletons, or even, the same skeletons can be associated to more than one dynamic force list.

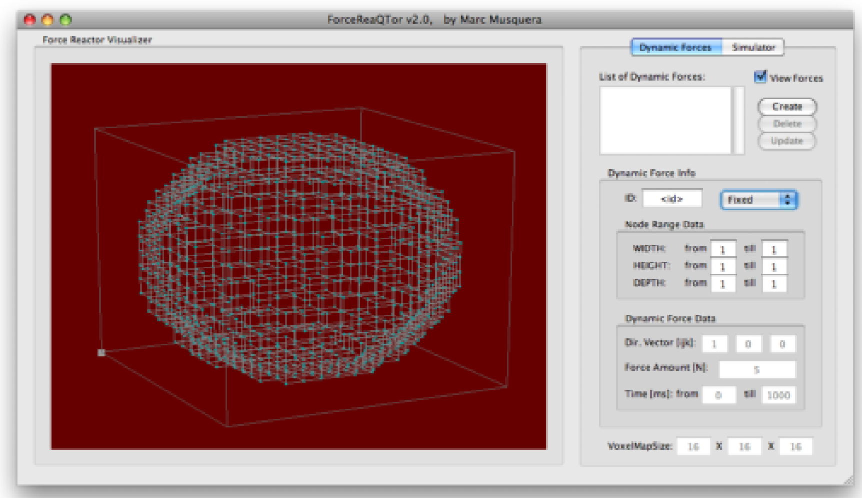
C.2.1.2 The VoxelMap and the 3D non-rigid skeleton

ForceReaQTor uses a **standard voxelmap** for generating a 3D non-rigid skeleton, by using the centers of validated voxels as **skeleton nodes**, and using a 6-connectivity policy[102], it creates a **non-rigid spring**. The final result is a kind of **complex net with axis parallel wires and nodes coinciding with the voxelmap valid voxel centers.**

Thus, as a **bridge** of the thesis' entire developed *software package*, **ForceReaQTor uses the MeshInspeQTor generable VoxelMaps**. Let's see this software relation in the two following figures: the first is from MeshInspeQTor after generating a voxelmap, and the second is ForceReaQTor generating the 3D non-rigid skeleton from this voxelmap.



(a) *The MeshInspeQTor's automatically generated voxelmap —see appendix B for more details—.*



(b) *The ForceReaQTor 3D skeleton processed from the voxelmap.*

Figure C.2: *An example of* $< 16 \times 16 \times 16 >$ *skeleton-node-modeling from* `sphere` *model voxelmap.*

[102] That is, two voxels are connected if they have a common face; *ergo*, a voxel will be full-connected if there are six adjacent voxels, each one with one of the six cube common faces.

C.2.1.3 Setting the Dynamic Force List

As said on the previous page, independelty of having a 3D skeleton charged in memory, ForreReaQTor offers a **complete configuration panel for creating, deleting and updating dynamic actions through the time-line**, for setting **three types of dynamic successes**, each one with some restricted parameters:

- **dynamic forces**; this item has **all the available paramaters**: *(i)* skeleton node range, *(ii)* force direction, *(iii)* force amount —in newtons— and *(iv)* time-line active action —that is, initial and final milisecond of force enabling—;

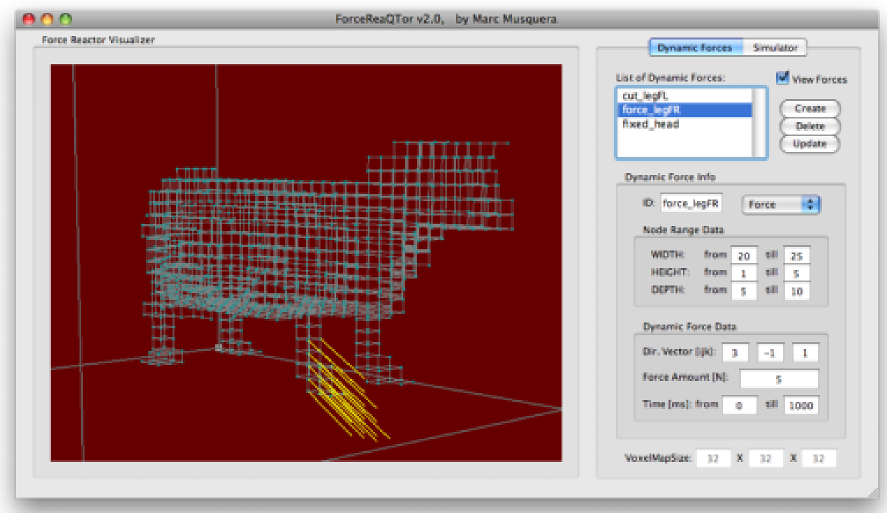


Figure C.3: *The preview of a previously specified dynamic force —yellow lines, maintaining the specified force vector direction, on the ‘front right leg’ zone—, applied to the* cow *skeleton.*

- **fixed skeleton nodes**; obviously, force direction and amount is unnecessary, and besides, ForceReaQTor will fix these nodes during all the simulation, so the time-line is also unnecessary. Then, **only the skeleton node range is available** here.

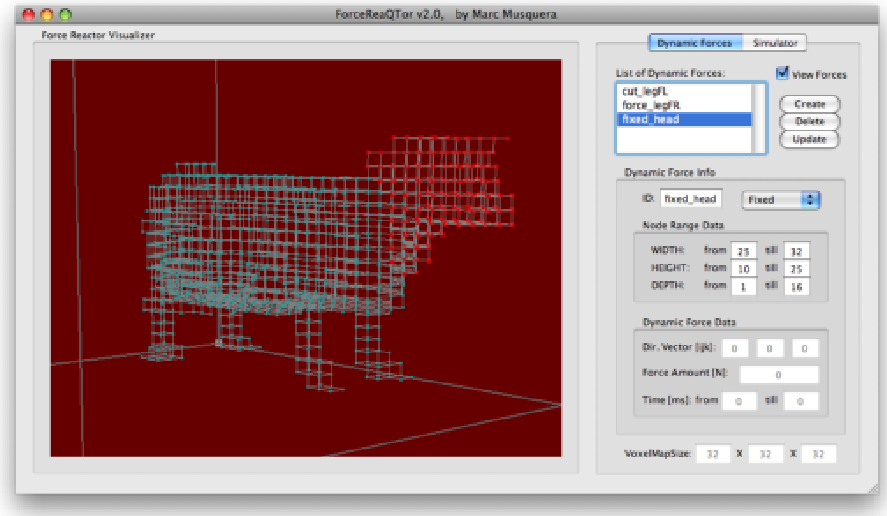


Figure C.4: *The preview of a fixed nodes application —red remarked nodes, on the ‘head’ zone—, also to the* cow *skeleton.*

- **skeleton spring cuttings**; as in the fixed nodes settings, force amount and direction will be disabled; nevertheless, a cut is permanent but only from the moment of cut, so **apart from the node range, the initial time-line milisecond are the available settings**.

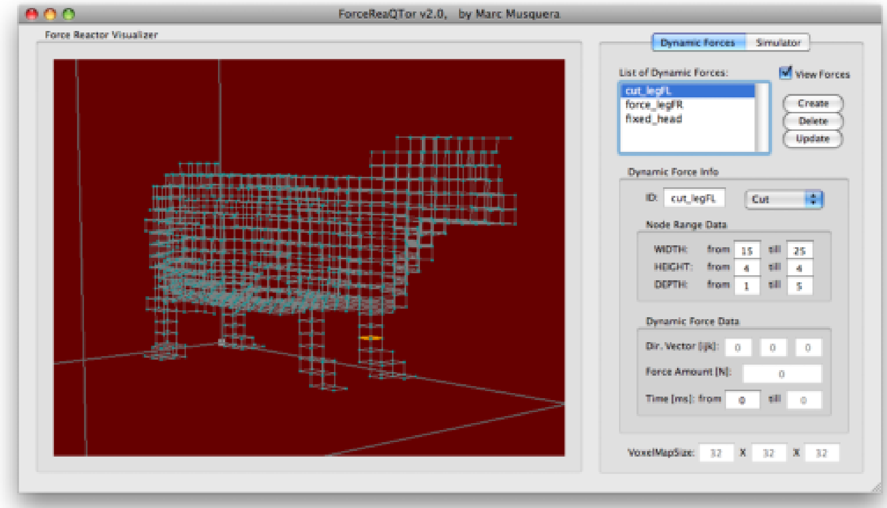


Figure C.5: *The preview of a specified cutting —orange remarked lines, on the 'front left knee' zone—, applied to the* `cow` *skeleton too.*

As it can be seen in the figures C.3, C.4 and C.5, **every dynamic action can be previewed on the previously charged 3D skeleton**. This process is **error-managed**; that means, if there is some dynamic action which node range is skeleton-out-of-range, ForceReaQTor does nothing, and the same with existing but not validated nodes.

C.2.1.4 Controls and management

Since **3D non-rigid skeletons** are generated automatically in the MeshInspeQTor VoxelMap file format importing action, the **VOX file format** will be the chosen for loading this kind of data. Additionally, the **dynamic force lists —including forces, node fixing and spring cutting—** are saveable and loadable by the property **DFL file format**.

Also, as for the dynamic action preview as long as for the simulation playing, there is some **visualization camera settings** by `Camera` menu, where camera can be **automatically skeleton-centered**, or its **position can be modified**: top-view, side-view and front-view —this last is the default one—.

Besides, there are itneractive camera controls with the **dragging mouse movement while any mouse button is pressed**; these are the mouse actions:

Mouse Button	Action while dragging
Left	Scene Rotation
Middle	Center-of-Scene Displacement
Right	Camera Zoom

Table C.1: *ForceReaQTor mouse controls.*

C.2.2 *Simulator* tab: The Real-Time Deformable Simulator

C.2.2.1 Deformable system visualization styles

ForceReaQTor offers **three types of deformable system rendering** for the visualization, perfectly modificable in simulation run-time:

1. **the first of them** has been seen in all the previous figures, it's the **3D skeleton model**;, comprising nodes and springs;

2. the second mode is the original polygonal mesh from where the voxelmap was generated —in case it exists—;

3. the another mode is the **Marching Cubes' extracted polygonal mesh**, an automatically generated mesh from the 3D skeleton shape by using the isosurface extraction algorithm *Marching Cubes*;

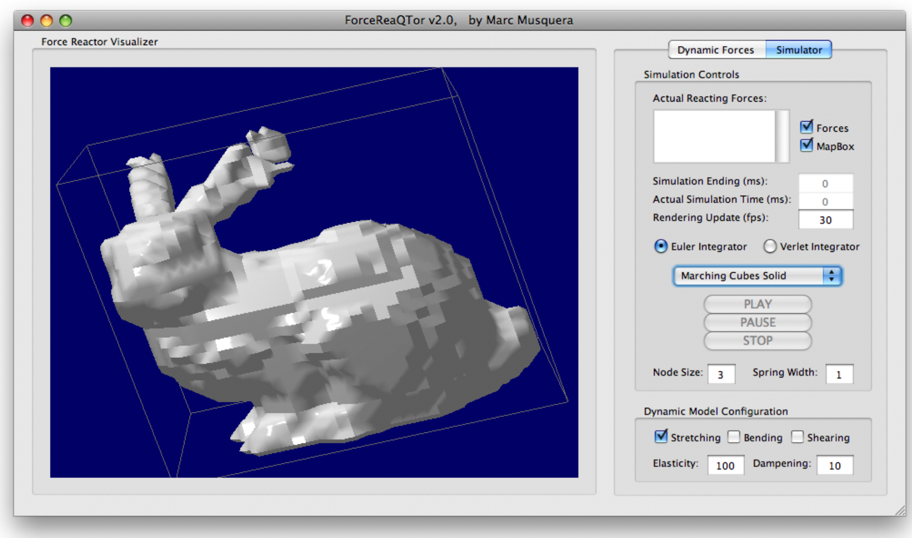


Figure C.6: *The ForceReaQTor's Marching Cubes extracted mesh rendering of a* `bunny`*.*

C.2.2.2 Playing the simulation

Once there is a 3D skeleton charged in memory, and a dinamyc force list is correctly configured —by a `DFL` file loading, or manually—, **the deformation set can be simulated in the `Simulation` tab**.

There are **three main buttons for simulation controlling**:

- **PLAY**, that reproduces the previously specified deformation set in a discrete smoothed time-line, applying subsequently all the set dynamic deformable actions;

- **PAUSE**, that pauses the simulation; if then the user clicks on textbfPLAY again, the simulation continues from the pause moment;

- **STOP**, that resets the simulation; that is, a pause but with a returning-to-zero timing —by clicking **PLAY** again, the simulation doesn't continue but begins again—;

additionally, it's important to notice that **in every simulation moment** —played, paused, stopped—, the scene visualization is interactively enabled by using the specified **mouse controls** on the table C.1.

C.2.2.3 The numerical integrators: Euler and Verlet

Before starting a simulation, the user must **choose a numerical integrator** for the 3D non-rigid skeleton; there are **two possible choices**:

1. the **Euler** numerical integrator, more **accurate** and realistic but also more **unstable** under time step closer to the limit;

2. the **Verlet** one, more stable for extremal conditions; however, it is also a **linear approximation of the Euler iterator**, and that implies two things: it's more **faster** but the simulation is more still **restricting** on deformations.

C.2.2.4 The frames per second —FPS— rate specification

In ForceReaQTor there is also a **pre-beginning simulation parameter**: the FPS-rate. The **simulation displaying and the deformation algorithm execution are computationally independent** —although they are really data-dependent—, so the user can **specify which framerate desires**.

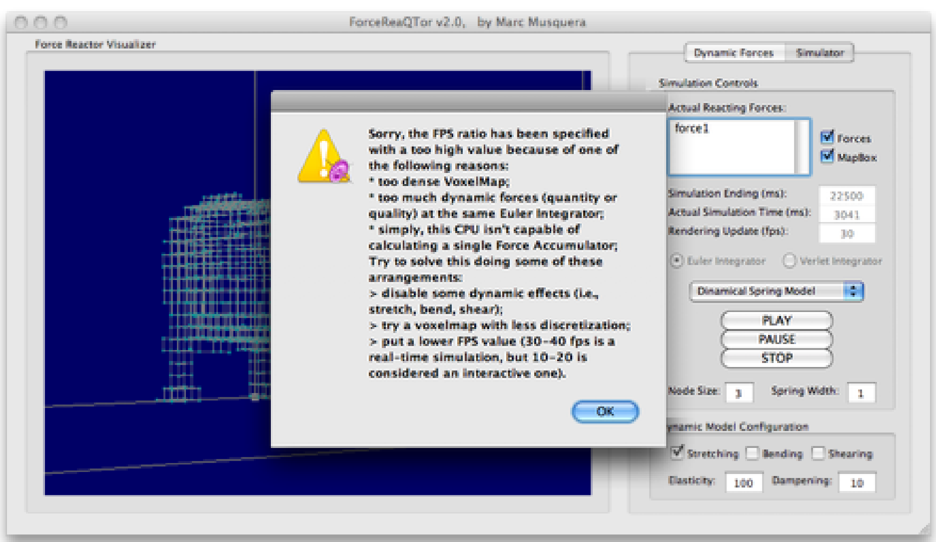


Figure C.7: *The* FPS *error message thrown when the framerate cannot be achieved.*

However, **if the specified FPS rate is too much high** an **error message** will be shown and the simulation will be stopped. There are —briefly— **two reasons** for that:

1. too much high *framerate* in FPS for so much complex deformation system —so, the computer is inherently not capable to display so much FPS—;

2. the specified *framerate* has a normal value but the deformable system has too much deformation restrictions —*e.g.* the three effects, *stretch*, *shear* and *bending*—; it will may execute only with *stretch* restriction;

Besides, if the numerical integrator selection has been the **Euler** one, it can be a good solution trying the **Verlet** integrator, due to its more efficiency and more stability —though caused by its more deformative restrictions—.

C.2.2.5 Run-Time simulation parameters

ForceReaQTor provides **real run-time simulation parameters** that modify the deformation characteristics; that is, parameters that can be enabled/disabled/modified **before as long as during simulation**. These run-time modifying parameters can:

- define the deformation nature by enabling or disabling some **restricting and relaxing forces** that avoid the total chaos of the skeleton deformations: *stretch*, *shear* and *bending*; those **restricting forces acts between skeleton nodes and springs**, and habiliting each one increases the computational time;
- set the numerical values of **elasticity and dampening** of the 3D skeleton node connection springs; these values can easily **destabilize** the system[103].

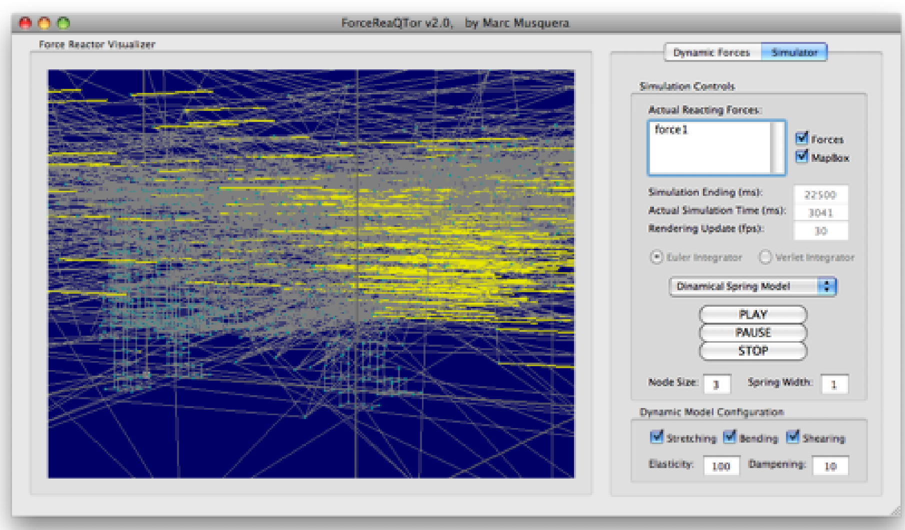


Figure C.8: *ForceReaQTor's fatal destiny: destabilized simulation.*

C.2.2.6 Controls and management

The file format management is the same that for the dynamic force list, due to **the skeleton is the deformable base and the list of forces is the deformation settings through the time-line**; so, the file formats `VOX` and `DFL` are required[104]. Additionaly, apart from the mouse controls of the table C.1, there are some **hot keys —actionable by pressing at the same time `CMD` + the specified hotkey— for controlling the simulation** in an easy and fast way, instead of the mouse clickings:

[103] Indeed, by enabling more than one restricting force can also destabilize the system, but due to computational reasons, not to numerical reasons.

[104] However, the dynamic force lists can be defined manually with the intervention of a `DFL` file loading.

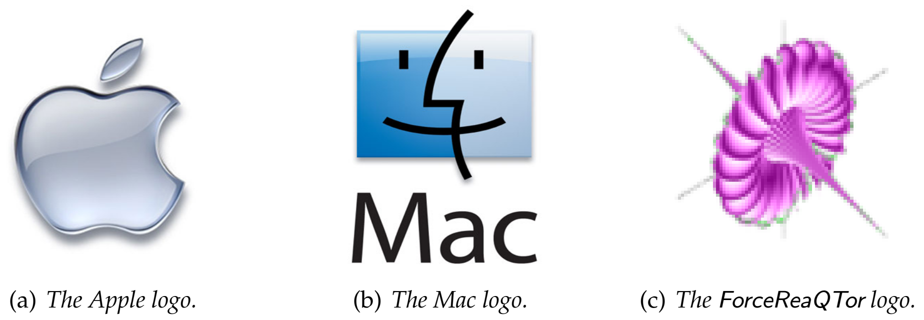
Hot Key	Simulation Action
CMD + 'I'	Play Simulation
CMD + 'J'	Pause Simulation
CMD + 'K'	Stop Simulation

Table C.2: *ForceReaQTor simulation hot key controls.*

C.3 Requirements and recommendations

ForceReaQTor is a **complete dynamic physically-based deformable model simulator**, that admits any volumetric representation —voxelmap— and the application transforms into a **3D non-rigid skeleton**. ForceReaQTor is capable to **perform any kind of force deformation** by a three —accumulative— relaxing types of forces as long as also **implements a virtual mesh cutting** that acts as a virtual scalpel.

However, theres no user-app interface[105] but a **dynamic time-line panel** that permits to set a dynamic force list —admitting cuttings too—, saveable and loadable as DFL files.



(a) *The Apple logo.* (b) *The Mac logo.* (c) *The ForceReaQTor logo.*

Figure C.9: *ForceReaQTor software and hardware requirements logos.*

This application has been developed for Apple Mac Intel platforms using the QT4 framework for GUI design and OPENGL API for the graphics displaying. A *Shader Model 3.0* compatible GPU with minimum of 128MB of own textscram is **required** for the deformable model visualization and computations.

[105] So, this is only a real-time simulator, not a complete interactive system.