UNIVERSITY OF NAVARRA

SCHOOL OF ENGINEERING

DONOSTIA-SAN SEBASTIÁN

Study of Parallel Techniques Applied to Surface Reconstruction from Unorganized and Unoriented Point Clouds

DISSERTATION
submitted for the
Degree of Doctor of Philosophy
of the University of Navarra by

Carlos Ignacio Buchart Izaguirre

December 13th, 2010

A mis abuelos:
Opa y Nanny
Pepe y Mima

Agradecimientos

El que da, no debe volver a acordarse;
pero el que recibe nunca debe olvidar
Proverbio hebreo

Los agradecimientos suelen ser la parte más difícil cuando se escribe la memoria de la tesis. Por ejemplo, uno no encuentra el orden adecuado, y no quiere dejar a nadie ni siquiera en segundo lugar; el cerebro se estruja a más no poder recordando a todo aquél que nos echó una mano en la tesis (*...el que recibe nunca debe olvidar*); ¡es que algunas veces uno desearía poder usar un comodín como los empleados en las expresiones regulares `[a-zA-Z]+`!

Primero que nada quisiera agradecer a mis directores de tesis, Diego y Aiert, no sólo por el tiempo y esfuerzo que le han dedicado a este trabajo (que también valoro enormemente), sino por el tiempo y esfuerzo que me han dedicado a mí, en mi formación como investigador y en aquellas horas más difíciles cuando las cosas parecían no salir y siempre venían con un buen consejo... o un Foster. ¡Gracias!

Mención honorífica se la llevan mis padres (Francisco e Inés) y mi hermana (Pul... ¡perdón! Titi), que, estando a miles de kilómetros de distancia, han sabido darme siempre una frase de aliento, una llamada en el momento oportuno (o no tan oportuno, pero el cariño es lo que cuenta). ¡Gracias!

A Jairo, porque entre los dos hemos sacado muchas partes de nuestras tesis y proyectos, por no decir la cantidad de chistes (¿malos?) que tanto me han hecho reír estos cuatro años. ¡Gracias!

También, y de una manera especial, a Alejo Avello, Jordi Viñolas y Luis

Matey. Muchas gracias por darme la oportunidad de desarrollar esta tesis en el CEIT, en el Departamento de Mecánica, en el Área de Simulación. A Ana Leiza y Begoña Bengoetxea, por su ayuda desde los primerísimos contactos con el CEIT y en todas las gestiones que han ido surgiendo en estos años. ¡Gracias!

A la Universidad de Navarra, por permitirme cursar mis estudios de doctorado; al TECNUN, por la formación profesional, académica y humana, y la ayuda de sus profesores y empleados en el desarrollo de esta tesis y en mis años como doctorando. ¡Gracias!

Agradecer también, de forma conmutativa, a mis compañeros y ex-compañeros de despacho, y del Departamento de Mecánica, todos (de una u otra forma) han contribuido en el desarrollo de esta tesis (mérito por soportarme en las mañanas incluido): Gaizka, Josune, Iker, Hugo, Maite, Ángel, Luis U., Jaba, Óskar, Jokin, Iñaki G., Javi, Iñaki D., Aitor C., Goretti, Imanol H., Emilio, Yaiza, Sergio, Alberto, Ibon, Álvaro, Denis, Mildred, Aitor R., Miguel, Dimas, Jorge Juan, Manolo, Imanol P., Jorge. ¡Gracias!

Estos años no hubiesen sido igual sin amigos de otros departamentos y de fuera, que le han impreso un carácter "multidisciplinar" a este doctorado: Nirko, Lorena, Jesús, Claudia, Wilfredo, Janelcy, José Manuel, Nacho, Elena, Alfredo, Paloma, Fernando, Ioseba, Raúl, Manu, Héctor, Tomás, Rocío, Musikalis. ¡Gracias!

Para cerrar, y sin que el hecho de estar en el último párrafo implique ningún tipo de orden cualitativo, muchas gracias a Patric, Jesús, José Luis, Rober, Karmele, Paqui, Patxi, Josemi, Franklin, Enrique, Álvaro, Dani, Héctor, Josetxo, Noelia, Marycarmen, por tantos pequeños (y grandes) favores. ¡Gracias!

¡Ah! Por si se me olvida alguien: ¡muchas gracias `[a-zA-Z]+`! `;)`

Abstract

Nowadays, digital representations of real objects are becoming bigger as scanning processes are more accurate, so the time required for the reconstruction of the scanned models is also increasing.

This thesis studies the application of parallel techniques in the surface reconstruction problem, in order to improve the processing time required to obtain the final mesh. It is shown how local interpolating triangulations are suitable for global reconstruction, at the time that it is possible to take advantage of the independent nature of these triangulations to design highly efficient parallel methods.

A parallel surface reconstruction method is presented, based on local Delaunay triangulations. The input points do not present any additional information, such as normals, nor any known structure. This method has been designed to be GPU friendly, and two implementations are presented.

To deal the inherent problems related to interpolating techniques (such as noise, outliers and non-uniform distribution of points), a consolidation process is studied and a parallel points projection operator is presented, as well as its implementation in the GPU. This operator is combined with the local triangulation method to obtain a better reconstruction.

This work also studies the possibility of using dynamic reconstruction techniques in a parallel fashion. The method proposed looks for a better interpretation and recovery of the shape and topology of the target model.

Resumen

Hoy en día, las representaciones digitales de objetos reales se van haciendo más grandes a medida que los procesos de escaneo son más precisos, por lo que el tiempo requerido para la reconstrucción de los modelos escaneados está también aumentando.

Esta tesis estudia la aplicación de técnicas paralelas al problema de reconstrucción de superficies, con el objetivo de mejorar los tiempos requeridos para obtener el mallado final. También se muestra cómo las triangulaciones locales interpolantes son útiles en reconstrucciones globales, y que es posible sacar partido de la naturaleza independiente de éstas para diseñar métodos paralelos altamente eficientes.

Se presenta un método paralelo de reconstrucción de superficies, basado en triangulaciones locales de Delaunay. Los puntos no están estructurados ni tienen información adicional, como normales. Este método ha sido diseñado teniendo en mente la GPU, y se presentan dos implementaciones.

Para contrarrestar los problemas inherentes a las técnicas interpolantes (ruido, *outliers* y distribuciones no uniformes), se ha estudiado un proceso de consolidación de puntos y se presenta un operador paralelo de proyección, así como su implementación en la GPU. Este operador se ha combinado con el método de triangulación local para obtener una mejor reconstrucción.

Este trabajo también estudia la posibilidad de usar técnicas dinámicas de una forma paralela. El método propuesto busca una mejor interpretación y captura de la forma y la topología del modelo.

Contents

I Introduction **1**

1 Introduction **3**

1.1 Applications 6

1.2 Data acquisition 7

1.3 Objectives 8

1.4 Dissertation organization 10

2 State of the art **11**

2.1 Interpolating methods 12

2.1.1 Delaunay triangulation 12

2.1.2 Local triangulations 14

2.2 Approximating methods 16

2.3 Parallel triangulations 18

2.3.1 Hardware accelerated algorithms 19

II Proposal **21**

3 GPU Local Triangulation **23**

3.1 Introduction 24

3.2 Sampling criteria 25

3.3 Description of the method 25

3.3.1 Preprocess phase – Computing the k-NN 25
3.3.1.1 k-NN based on clustering techniques 26
3.3.1.2 k-NN using kd-trees 27
3.3.1.3 Final comments 27
3.3.2 Parallel triangulation 27
3.3.3 Phase 1 – Normal estimation 28
3.3.3.1 Normals orientation 30
3.3.4 Phase 2 – Projection 31
3.3.5 Phase 3 – Angle computation 32
3.3.6 Phase 4 – Radial sorting 33
3.3.7 Phase 5 – Local triangulation 33
3.3.7.1 2D validity test 34
3.3.7.2 Proof 36
3.4 Implementation using shaders 37
3.4.1 Initial texture structures overview 38
3.4.2 Texture assembly 41
3.4.3 Phase 1 – Normal estimation 41
3.4.4 Phases 2 and 3 – Projection and angle computation 42
3.4.5 Phase 4 – Radial sorting 42
3.4.6 Phase 5 – Local triangulation 44
3.5 Implementation using CUDA 45
3.5.1 Data structures 46
3.5.2 Phase 4 – Radial sorting 48
3.5.3 Phase 5 – Local triangulation 48
3.6 Experiments and results 48
3.6.1 CPU vs Shaders vs CUDA 50
3.6.2 Reconstruction results 53
3.6.3 Big models 57
3.6.4 Comparison with an approximating method 58
3.6.5 Application in the medical field 61
3.6.6 Application in cultural heritage 61
3.7 Discussion 64

4 Parallel Weighted Locally Optimal Projection 65
4.1 Previous works . . . 65
4.1.1 Locally Optimal Projection Operator . . . 66
4.1.2 Weighted Locally Optimal Projection Operator . . . 67
4.2 Parallel WLOP . . . 68
4.2.1 Implementation details . . . 71
4.3 Experiments and results . . . 71
4.4 Discussion . . . 77

5 Hybrid surface reconstruction: PWLOP + GLT 79
5.1 Improving the input data set through points consolidation . 79
5.2 Results . . . 80
5.3 Discussion . . . 87

6 Study of multi-balloons reconstruction 89
6.1 Dynamic techniques . . . 89
6.1.1 Classic balloons . . . 91
6.2 Multi-balloons . . . 93
6.2.1 Scalar function fields . . . 95
6.2.2 Evolution process . . . 96
6.2.2.1 Global and local fronts . . . 96
6.2.2.2 Two-step evolution . . . 98
6.2.2.3 Gradient modulation term: κ_i . . . 99
6.2.2.4 Local adaptive remeshing . . . 100
6.2.3 Topology change . . . 104
6.2.3.1 Genus . . . 104
6.2.3.2 Holes . . . 106
6.3 Experiments and results . . . 106
6.4 Discussion . . . 109

III Conclusions **111**

7 Conclusions and future work **113**
7.1 Conclusions 113
7.2 Future research lines 115

IV Appendices **117**

A GPGPU computing **119**
A.1 Shaders 120
A.2 CUDA 122
A.2.1 CUDA Program Structure 123
A.2.2 Occupancy 124
A.2.3 CUDA Memory Model 124

B Generated Publications **127**

Index **129**

References **131**

List of Figures

1.1 Different point clouds 3
1.2 Noisy set of points 4
1.3 Ambiguities are commonly present in points from a scanned surface 4
1.4 Example of a points cloud extracted from a computer tomography 7
1.5 3dMface System, used in face scanning 8
1.6 Creaform REVscan laser scanner, used in reverse engineering 9

2.1 Voronoi diagrams and Delaunay triangulations 13
2.2 Comparison between different local triangulation methods . 15
2.3 Intersection configurations for Marching Cubes 18
2.4 Performance evolution comparison between CPUs and GPUs 19

3.1 Flow diagram of the proposed reconstruction algorithm . . 24
3.2 Normal estimation using PCA and wPCA 29
3.3 Minimal rotation 31
3.4 The *is_valid* function verifies if a point belongs to a partial Voronoi region. 35
3.5 The validity test is local and must be performed several times to obtain the local triangulation of a point 35
3.6 *is_valid* never invalidates Voronoi neighbors 36
3.7 Flow diagram of the shaders implementation 38
3.8 T_P and T_N – Points and normals textures 39

3.9 T_Q – Candidate points texture 40
3.10 T_I – Index table texture 41
3.11 T_α – Angles texture: similar to the T_Q structure but replacing the distance with the angle 42
3.12 $T_{Q'}$ – Projected candidate points texture 42
3.13 The shaders implementation of GLT uses a ring to check all the neighbors in parallel, discarding the invalid ones in each iteration 46
3.14 T_R – Delaunay ring texture 46
3.15 Flow diagram of the CUDA implementation 47
3.16 Time comparison of the computation of the neighborhoods 51
3.17 Time comparison of the normal estimation phase 51
3.18 Time comparison of the angle computation and projection phase 51
3.19 Time comparison of the sorting phase 52
3.20 Time comparison of the local Delaunay validation 52
3.21 Time comparison of the local Delaunay triangulation 52
3.22 Comparison between the different proposed reconstruction methods 53
3.23 Horse model rendered along with its wireframe 54
3.24 Neck and chest detail of the Horse model 54
3.25 A synthetic model 55
3.26 Running shoe along with its wireframe 55
3.27 Blade model 56
3.28 Top view of the Blade model 56
3.29 Overall view of time consumption in the reconstruction of the Asian Dragon model (3609K points) 57
3.30 Comparison with the Poisson reconstruction - Angel 59
3.31 Comparison with the Poisson reconstruction - Happy Buddha 59
3.32 Comparison with the Poisson reconstruction - Blade 60
3.33 Comparison with the Poisson reconstruction - Interior detail of the Blade model 60
3.34 Patients' Heads models 61

3.35 Reconstruction of a Trophy 62
3.36 Reconstruction of a Theatre 63
3.37 Reconstruction of a Tower 63

4.1 Local support neighborhood vs. the extended LSN 69
4.2 Speed comparison between WLOP, eWLOP and PWLOP with the Stanford Bunny 72
4.3 Projection of the Happy Buddha model using a random initial guess 73
4.4 Projection of the Happy Buddha model using an approximation from spatial subdivision techniques 74
4.5 Projection of different data sets using WLOP, eWLOP and PWLOP 75
4.6 The extended LSN computation times are common for both the eWLOP and PWLOP 75
4.7 The computation of the extended LSN is the same for the eWLOP and PWLOP 76
4.8 Projection of the Happy Buddha model resetting the initial data set 76

5.1 Stanford Dragon 81
5.2 Asian Dragon 82
5.3 Happy Buddha 82
5.4 Hand model reconstruction comparison 83
5.5 Hand model detail comparison 83
5.6 Noisy Foot with 20K points 84
5.7 Comparison of the hybrid PWLOP + GLT with the Poisson reconstruction 85
5.8 Results of the hybrid PWLOP + GLT reconstruction 86
5.9 Multi-resolution Happy Buddha using the hybrid approach 88

6.1 Competing fronts 91
6.2 Level sets 91
6.3 Flow diagram of the studied reconstruction using multiballoon 94

6.4 Multiple seeds placed in the Hand model 95
6.5 Multi-resolution grid used as satisfaction function 97
6.6 Slice of the Horse model's distance function 97
6.7 Uniform vs. Local adaptive remeshing 102
6.8 Detail of the effects of the local adaptive remeshing 103
6.9 Topology change 105
6.10 Rconstruction of a high-genus synthetic model 105
6.11 Multiple balloons evolving to reconstruct the Hand model . 107
6.12 Reconstruction of the Stanford Dragon model 107
6.13 Reconstruction stages for the Stanford Dragon model 108
6.14 Reconstruction of the Horse model 108

A.1 Thread hierarchy. 123
A.2 Memory hierarchy. 125

List of Tables

3.1 Total reconstruction time using three different implementations 50
3.2 Total reconstruction time of big models 57
3.3 Comparison with the Poisson reconstruction method 58

4.1 Speed comparison between WLOP, eWLOP and PWLOP with the Happy Buddha model 75

5.1 Comparison between the proposed hybrid approach and the Poisson reconstruction method 84

List of Algorithms

3.1 Local Delaunay triangulation in 2D 34
3.2 Adapted Bitonic Merge Sort for multiple lists sorting 44
3.3 Local Delaunay triangulation algorithm adapted for its implementation with shaders. 45
3.4 Local Delaunay Triangulation algorithm optimized for CUDA. 49
6.1 Overview of the algorithm proposed. 94
6.2 Local adaptive remeshing operator 102

Part I

Introduction

Chapter 1

Introduction

Science brings men nearer to God
Louis Pasteur

Surface reconstruction from a set of points is an amazing field of work due to the uncertainty nature of the problem. It is well known in computer graphics. It has been formally defined as *given a set of points P which are sampled from a surface S in $\mathbb{R}^3$, construct a surface $\hat{S}$ so that the points of P lie on $\hat{S}$* (Gopi et al., 2000), assuming there are neither outliers nor noisy points, i.e., it defines an interpolating reconstruction. The resulting surface $\hat{S}$ usually is a triangle mesh, a discrete representation of the real surface. If noise is present in the set of points (P is near S), an approximating reconstruction is needed in order to avoid interference caused by points out of the surface. Some examples of input points can be seen in Figure 1.1 while Figure 1.2 shows a noisy dataset.

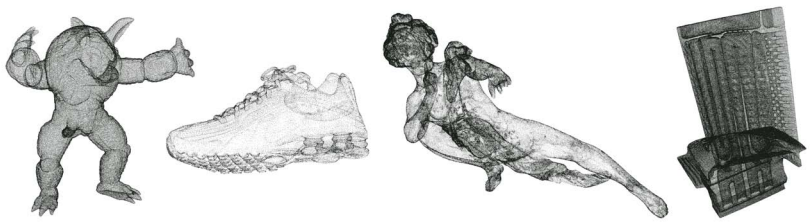


Figure 1.1: Different point clouds.

It is easy to see that surface reconstruction faces several problems: incomplete data, inconsistencies, noise, data size, ambiguities (see Figure 1.3). It is a vast researching area, and many methods have been already

Figure 1.2: Noisy set of points.

proposed. Despite the work that has been done in this field, the correct generation of a triangle mesh from a points cloud is still a study goal trying to improve drawbacks current techniques have. Also, it is still an open field because it is too complex, and probably impossible, to fully recovery an unknown surface without previously assuming some kind of information, such as normals' direction and orientation, presence of holes or sharp features, and the topology of the model. Almost every single existing method relies in at least one parameter (for example, the sampling rate), and is usually focused in a subset of problems, so it can better exploit the implicit model's characteristics in order to obtain a proper reconstruction.

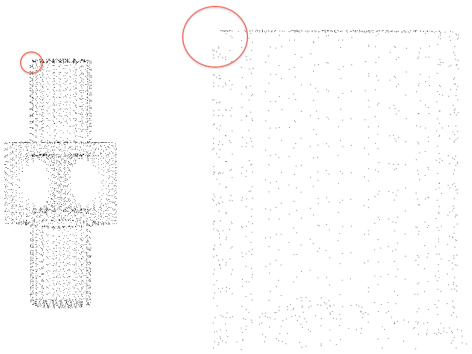


Figure 1.3: Ambiguities are commonly present in points from a scanned surface. For example, the top of this piece is a cylinder, but in some zones there where no points scanned, leading to several interpretations of the data.

Today's improvements in acquisition methods and computers, along with the demand for more detailed models, have led up to work, more

and more, with larger point clouds, increasing the reconstruction time and memory requirements. This have lead to develop more efficient algorithms in terms of memory and processing time.

Nowadays, the general evolution of the hardware and software is more focused on the distribution of the processing load than in the increment of the individual computing power. This tend can be found in many areas: search engines, numerical simulation, cryptography, movies rendering, etc., where parallel algorithms have been developed to take advantage of the modern hardware. For example, thanks to their high parallelization level, a single graphic card usually have more processing power than a top-line CPU. More on, since a few years ago it is common to find multi-core processors in personal computers, as well as powerful graphic cards, so it is no more a restricted field that requires high budgets. The parallel programming would offer a way to solve the dataset size issue, since the points processing can be split among different processors or computing nodes, also reducing the individual memory requirements of each computer.

Talking about the graphics hardware field, in recent years that technology has allowed developers to execute custom programs in the GPU (*Graphics Processing Unit*), taking advantage of the huge processing power it offers, in the so called GPGPU programming (*General Purpose GPU*)[1]. Although by the time this thesis is being presented another GPU surface reconstruction method has been proposed, the migration of any algorithm to the GPU is not trivial and usually requires a high number of modifications. At the same time, not every existing technique can be implemented in parallel, and even less in the GPU, being necessary in such cases, the creation of new processing paradigms. Finally, the GPU method mentioned above corresponds to an approximating reconstruction and, as it will be seen later, this work proposes an interpolating approach.

Another interesting and common fact about surface reconstruction is that there is not always a need for real-time reconstruction. Usually, the reconstruction is a preprocess for a later application, and then, does not necessarily have to be a fully automatic and real-time process. The user can guide the reconstruction or, as usually happens, can perform modifications to the resulting surface to correct any problem (such as holes filling) or include information the reconstruction process did not have access to (zones that could not be scanned, or solve ambiguities in the set of points). In any

[1]For more information about GPGPU programming, please refer to Appendix A.

case, although real-time is not a must in these cases, efficiency should be taken into account due to the size of data sets increases continually (most of the models have hundred of thousand points or even a few million).

The present work is mainly focused in parallelization. It studies the possibility of the design and implementation of parallel surface reconstruction techniques, and especially in their implementation in graphic hardware for a even higher speed boost. The vast processing power of current GPUs makes them highly desirable targets in terms of execution time of the reconstruction.

1.1 Applications

Surface reconstruction finds very useful applications in the industry in the fields of reverse engineering and CAD (*Computer Aided Design*). In this case, it is more common to use parametric surface patches instead of polygonal meshes to represent the reconstructed object.

Another important area of use is in movies and videogames production, where figures modeled in clay are digitalized to be later included into the media content.

Also, it is widely used in cultural heritage allowing the digitalization of sculptures, vases, historical sites, etc. for analysis, documentation or public exposition. For example, the engrave of an old ceil may be digitalized for its study in a more comfortable place and to avoid that its manipulation may deteriorate the engrave. Some examples of surface reconstruction in this field can be found in (Besora et al., 2008; Cano et al., 2008; Torres et al., 2009a).

In medicine, surface reconstruction provides a practical way to visualize certain kinds of tissues (Figure 1.4), as well as to extract an interaction surface to be used in simulators. It is also possible, for example, to supplant dental casts with 3D models of the teeth for dental prosthetics (bridges, crowns, etc).

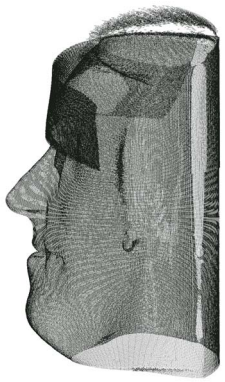


Figure 1.4: Example of a points cloud extracted from a computer tomography.

1.2 Data acquisition

The process to convert a real surface into a digital representation starts with the acquisition of a set of points that represents it. This process is called *surface scanning*, and the tools that acquires the points, *scanners*. Among the different kinds of scanners, some of the most common ones include:

- **Image-based scanners**: these scanners consist of at least a couple of calibrated cameras that capture the object from different points of view. These images are used to compute a view point dependent 3D set of points using a correspondence analysis, as described in (Jones, 1995). Figure 1.5 shows a face scanner.

- **Time-of-flight scanners**: by measuring the round-trip time of a pulse of light emitted to the object it is possible to compute the distance to it. Their main disadvantage is that they can detect only one point in their view direction, so either the laser must be moved, the object rotated or a mirror used to capture the back faces of an object.

- **Hand-held laser scanners**: these scanners project a laser line or dot onto the object, and the distance to it is measured by the scanner. An example of these scanners is shown in Figure 1.6.

- **Contact scanners**: this kind of scanners probe the object through physical touch and are very precise. Their main disadvantage is that the object must be touched, and in some cases it is not possible or it is not easy. Also, physical contact with the object may damage it.

Figure 1.5: 3dMface System, used in face scanning.

Finally, some other acquisition techniques include segmentation of medical images, data from simulations and video processing. The latter makes use of similar techniques to image based scanning, by replacing the multi-camera sources with a multi-view approach (epipolar geometry) and tracking 2D feature points along the video sequence taking advantage of the temporal coherence (for more information about this kind of techniques, please refer to (Sánchez et al., 2010)).

1.3 Objectives

As has been stated before, surface reconstruction is a vast research area that has been addressed from different points of view, but that remains open given the uncertain nature of the problem. Even more, some of the most reliable methods proposed depend on additional data such as normal orientation, that is not always available during the reconstruction process. Surface reconstruction methods should be able to take advantage of the additional information if it is present, but should not be dependent on it.

Additionally, the size of the point clouds is rapidly increasing, making

Figure 1.6: Creaform REVscan laser scanner, used in reverse engineering.

necessary the design of fast methods that can handle such amount of data. Parallel programming is presented as a very powerful solution. More on, GPGPU programming has helped in recent years to a dramatic speed increase in many fields. It is reasonable, then, to think that techniques developed should be implementable in the GPU.

For all this, the main objective of this thesis is *the study of parallel techniques applied to the design of surface reconstruction algorithms from unorganized and unoriented sets of points*, and additionally, *the optimization of existing reconstruction supporting methods* and *the implementation and acceleration of studied algorithms using commodity graphic hardware.*

The following specific objectives are intended to limit the scope of the thesis and focus the contributions of the work:

- **Input data**: although many surface scanners exist, not every system provides additional information from the digitalized object, so the method should be able to work without it. The input data, then, should consists in a set of points in 3D without any kind of structure or additional information. Also, the number of parameters should be as minimal as possible to make more flexible the use of the developed methods in automatic processes.

- **Noise tolerance**: the methods studied should be robust against noise. Given that some techniques are non-tolerant to noise by

their nature (e.g. interpolating techniques), it is convenient to study auxiliary techniques that may help to overcome this limitation. Even though, the goal is not to treat with high levels of noise.

- **Scalability**: as point clouds are becoming more and more large, the methods studied should be able to scale over the time in order to be useful even when the dimensionality of the datasets overpasses the size of current models.

Finally, the hardware employed in this project is restricted to commodity PCs and graphic cards of the same level. It is not pretended to be a study of surface reconstruction on PC clusters, large computing farms or specialized computers.

1.4 Dissertation organization

The rest of this memory is organized as follows. Chapter 2 reviews the most important reconstruction techniques. Chapter 3 presents the study of a surface reconstruction based on local Delaunay triangulations and its implementation in the GPU. Chapter 4 discusses a projection operator for points consolidation as well as a new parallel approach. This operator is combined with the surface reconstruction method proposed; this hybrid technique is described in Chaper 5. In Chapter 6 a different reconstruction approach, based in multiple active contours, is studied. Experimental results are shown at the end of each chapter. Finally, Chapter 7 shows the conclusions of this thesis, and it also comments possible future lines of work.

Chapter 2

State of the art

Science, my lad, is made up of mistakes,
but there are mistakes which it is useful to make,
because they lead little by little to the truth
Jules Verne

Surface reconstruction is an active research field, and many works have been presented, which can be classified using different approaches[1]. On one hand, if it is assumed that the points have no noise, i.e., they belong to the surface, then it is sufficient to find a relationship among the points (their connectivity) to reconstruct a discrete representation of the surface. These kinds of techniques are called *interpolating*. On the other hand, if noise or incomplete data is present, *approximating techniques* are required. The latter usually rely on implicit functions to define a similar surface while balancing outliers, noise and high details. Additionally, it is not uncommon for many methods, to filter the point set as a preprocess step, in order to improve it (removing outliers, for example). Also, some other techniques exist, called dynamic techniques, but they are discussed in Chapter 6.

The rest of this section will describe some of the most important works related to surface reconstruction, as well as a discussion on parallel techniques and GPU methods. The third classification mentioned above is commented separately in Section 6.1.

[1]For convenience, a classification similar to that employed in (Gopi et al., 2000) is used.

2.1 Interpolating methods

Interpolating methods rely on points that effectively are on the original surface. Although they are not robust against noise and outliers, the quality of the reconstructed mesh is usually better than those generated by approximating methods, given a noise free dataset. This quality represents both triangle proportions, and fidelity to the target surface.

The main weakness of interpolating methods is the presence of noise and outliers. It is interesting to note that isolate outliers are usually discarded without any additional work given proper neighborhood sizes. Sampling inaccuracies on the other hand, usually fall into the local support of interpolating methods and are included in the reconstruction. Under certain levels of noise, just a rough surface is obtained. In worse cases, however, the reconstruction method may fail to create a consistent surface.

2.1.1 Delaunay triangulation

Maybe the most known and common interpolating technique is the Delaunay triangulation, initially introduced by (Delaunay, 1934) (Figure 2.1). Given a set of points P in the d-dimensional Euclidean space, the Delaunay triangulation $DT(P)$ is such that no point $p \in P$ is inside the circum-hypersphere of any d-simplex[2] in P. It is important to comment that if the set of points P is in a 3-dimensional space, $DT(P)$ is not a surface but a tetrahedral mesh, so a post-process is required to extract the on-surface triangles from the full triangulation. Unless explicitly mentioned, the rest of this work assumes $d = 3$.

The $DT(P)$ is also the dual of the Voronoi diagram of P, where consecutive nodes of the Voronoi diagram are connected in the triangulation. The Voronoi region of a point $p \in P$ is given by all points that are nearer to p than to any other point of P (see Figure 2.1). This property is often used to construct either the Delaunay triangulation or the Voronoi diagram based on the other one.

Finally, the *local triangulation* of a point p_i is defined as the set of triangles that have p_i as a vertex. Given the *local Delaunay triangulations*

[2] A d-simplex is the minimum object in a E^d, the d-dimensional analogue of a triangle (which is the 2-simplex, for $d = 2$). A 3-simplex is, therefore, a tetrahedra.

(LDT) of every point in P, the global triangulation can be constructed as follows:

$$DT(P) = \bigcup_{p_i \in P} LDT(p_i) \tag{2.1}$$

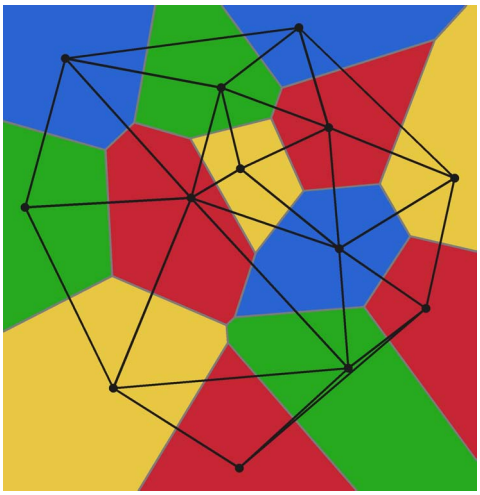


Figure 2.1: Voronoi diagrams and Delaunay triangulations are duals.

A good example of Delaunay triangulations in surface reconstruction is presented by (Attene and Spagnuolo, 2000). In this work, the Delaunay tetrahedralization of the input points is constructed and then, by the use of an Euclidean minimum spanning tree and a Gabriel graph[3], the boundary of the surface is obtained.

One of the most cited Delaunay based reconstructions is the Power Crust method proposed by (Amenta et al., 2001). It constructs the Voronoi diagram of the input points, and then extracts a polygonal mesh using the inverse of the medial axis transform obtained from the Voronoi diagram. The *medial axis* of an object is a tool for shape description and consists in the closure of the locus of the centers of all maximal inscribed discs in the target surface. The *medial axis transform* is the medial axis together

[3]A Gabriel graph is a subset of the Delaunay triangulation, where two points $p, q \in P$ are connected if the closed disc which diameter is the line segment pq contains no points from P.

with the radius of the maximal inscribed circle center in each point on the medial axis (Vermeer, 1994).

The Cocone algorithm (Amenta et al., 2000) also relies on Voronoi regions for the reconstruction. It extracts the reconstructed surfaced by filtering the triangulation as follows: for a sample point $p \in P$ and a Voronoi edge e in the Voronoi cell of p, if for the Voronoi cells adjacent to e have a point x such that the vector $\mathbf{px}$ makes a right angle with the normal of p, then the dual of the Voronoi cells is included in the reconstruction. Some other improvements to this method have been presented, concretely the Tight Cocone (Dey and Goswami, 2003), basically a hole-filling extension; and the Robust Cocone (Dey and Goswami, 2006), that incorporates tolerance against noisy inputs.

In (Allégre et al., 2007) a progressive method is shown, where the data set is simplified at the time it is selective reconstructed using point insertions and deletions in the Delaunay triangulation. At the end, the resolution of the resulting mesh is locally adapted to satisfy the feature sizes of the model.

2.1.2 Local triangulations

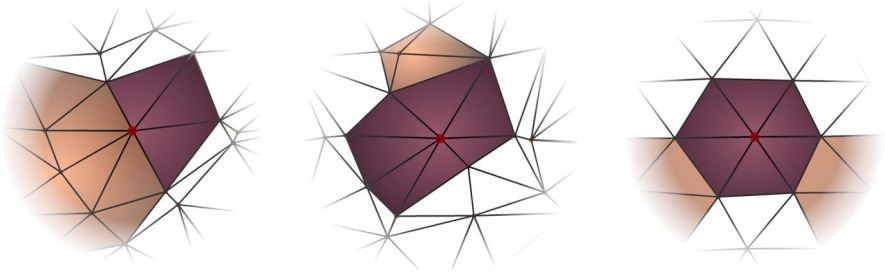
Previously mentioned techniques are based in global triangulations algorithms, but (Linsen and Prautzsch, 2001) show that local triangulations are also well suited for surface reconstruction and can lead to faster algorithms, even more if the main purpose of the reconstruction is the visualization of a polygonal mesh rather than its utilization in later processing tasks. Assuming a well sampled surface, (Linsen and Prautzsch, 2001) construct a triangles fan around each point using its 6-neighborhood.

For example, assuming a dense sampling of the original object, (Gopi et al., 2000) show that the neighborhood of p is the same as its projection onto the tangent plane of p, given that the distance to p is maintained. Based on this, a lower dimensional Delaunay triangulation is done in 2D, resulting in a faster reconstruction than in 3D (as was previously mentioned, the 3D Delaunay triangulation is composed of tetrahedra rather than triangles). The final mesh is obtained by merging all individual fans.

Another technique, very similar to local triangulations, are the use of *advancing frontal techniques* (AFT). An AFT starts with a minimal

subset of the final reconstruction and expands its boundaries iteratively covering the whole surface. Current methods based on advancing fronts perform such iterations one point at a time (here their similarity to local triangulations). For example, (Bernardini et al., 1999) start with a seed triangle and adds subsequent triangles using a ball pivoted around the edges of the boundaries of mesh, and (Crossno and Angel, 1999) use a localized triangulation approach, called Spiraling Edge, in which a point is not marked as finished until it is completely surrounded by triangles (exceptions are boundary points).

Comparing the fan creation of these reconstruction methods, while the three (Crossno and Angel, 1999), (Gopi et al., 2000) and (Linsen and Prautzsch, 2001) triangulate over a radial sorted set of neighbors, they differ in how this is done. The first two methods verify if each neighbor is valid in the fan: (Crossno and Angel, 1999) perform a whole empty circumsphere test (Figure 2.2(a)) and (Gopi et al., 2000) determine if they are valid Delaunay neighbors on the tangent plane of the central point (Figure 2.2(b)). On the other hand, (Linsen and Prautzsch, 2001) assume some uniformity on the points cloud and directly add a small number of neighbors as valid fan points (Figure 2.2(c)).



(a) (b) (c)

Figure 2.2: Comparison between three different local triangulation methods: (a) (Crossno and Angel, 1999), (b) (Gopi et al., 2000), (c) (Linsen and Prautzsch, 2001). New triangles are colored in violet while the triangles created in previous step of the correspondent method are shown in beige.

More recently, (Zhang et al., 2010) use a similar approach of lower dimensional reconstruction, this time over patches larger than the 1-neighborhood of previously discussed methods; those patches are defined

as regions where point's tangent planes have small variations.

2.2 Approximating methods

If either noise or outliers are present in the input data, interpolating techniques fail to generate a correct reconstruction, because all the points are taken into account for the triangulation, interpreting noise as fine details of the target surface.

Approximating techniques tend to solve this issue at the cost of losing detail. It is also true that many methods can achieve good quality reconstructions, but usually incurring in a high memory consumption. In this sense, local triangulations, such as the studied in the previous section, can reduce their memory footprint by subdividing the reconstruction into smaller blocks.

Among approximating reconstructions can be found many different approaches, but most of them are based on the construction of an implicit function $I(x)$. This function is commonly represented as a scalar field using spatial subdivision techniques, such as voxels or octrees. In this way, the surface reconstruction is transformed into an iso-surface extraction problem: most of the methods use functions such that $\hat{S}$ is defined by $I(x) = 0$ (for example, if $I(x)$ is a distance function). The robustness and quality of these algorithms are determined by the volume generation and the precision of the iso-surface when compared with the original surface, i.e., the resolution of the implicit function.

One of the first methods presented in this area was the work of (Hoppe et al., 1992). It computes a signed distance field from the points to their tangent planes. Such planes are estimated from the local neighborhood of each point using principal component analysis, and re-oriented using a connectivity graph to consistently propagate a common orientation.

(Bajaj et al., 1995) use α-shapes to determine the signed distance function. Given the Delaunay triangulation $DT(P)$ of P, an α-shape is, as a general idea, the subset of simplices of $DT(P)$ smaller than the selected value α. All the simplices that belong to the α-shape are marked as internal, while the others are marked as external. With this space classification the signed function is constructed.

(Ohtake et al., 2005) make use of weighted local shape functions in an adaptive fashion to create a global approximation, and (Nagai et al., 2009) follow a similar idea, introducing a Laplacian smoothing term based on the diffusion of the gradient field. Alternatively, (Samozino et al., 2006) center compactly supported radial basis functions in a filtered subset of the Voronoi vertices of the input points.

Using the Voronoi diagram of the input points, (Alliez et al., 2007) compute a tensor field and estimates from it the normals of the points. By solving a generalized eigenvalue problem, it computes an implicit function which gradients are most aligned with the principal axes of the tensor field.

Using a different approach, (Esteve et al., 2008) compute a piecewise tricubic Bspline surface from the discrete membrane (a collection of face-connected voxels that is derived from the points set). This algebraic surface is iteratively adjusted to the discrete membrane, adjusting the continuity between voxels and fitting the isosurface to the center of hard-tagged voxels (that contain input points).

Several approximating techniques have been based on physical phenomena. For example, (Jalba and Roerdink, 2006) aggregate points into a grid and convert the non-empty cells into source points for a physical simulation of heat flow, which is defined using the regularized-membrane equation. This creates the scalar field that is later used to reconstruct the target surface.

(Wang et al., 2005) propose the use of oriented charges to create the signed distance function. The space containing the points set is subdivided using an octree, and the charges are placed in the nodes of the neighborhood of the samples. Such charges are linear local distance fields; to obtain a global distance field, each charge is weighted by a Gaussian blending function.

Finally, the Poisson reconstruction (Kazhdan et al., 2006) is probably one of the best surface reconstruction methods widely available. It performs a global reconstruction by solving a Poisson system. An adaptive solver is used to adjust the precision of the solution near the surface. It also shares some local fitting characteristics since, in order to create the adaptive conditions, it defines locally supported functions on octree's nodes. One disadvantage of this method is that it relies on the correct orientation of the points' normals.

Among the iso-surface extraction methods, robust and parallel approaches have been created, such as Marching Cubes (Lorensen and Cline, 1987) and polygonizers (Bloomenthal, 1988; Gueziec and Hummel, 1995; Torres et al., 2009b). Marching Cubes determines the intersection of an iso-surface with each voxel matching its topology with a list of possible intersections (Figure 2.3), and uses it to generate a piecewise model voxel by voxel. Many other iso-surface extraction works are based on this approach, for example, (Raman and Wenger, 2008) developed an extension for the Marching Cubes to avoid the creation of skinny triangles. Other works proposed similar techniques that use alternative structures, such as tetrahedra (Bloomenthal, 1988; Gueziec and Hummel, 1995) and octahedra (Torres et al., 2009b).

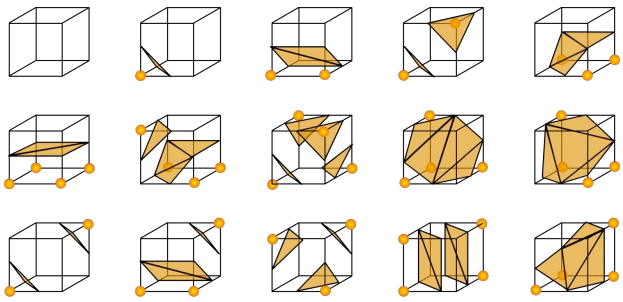


Figure 2.3: Intersection configurations for Marching Cubes (Image from (Favreau, 2010)).

2.3 Parallel triangulations

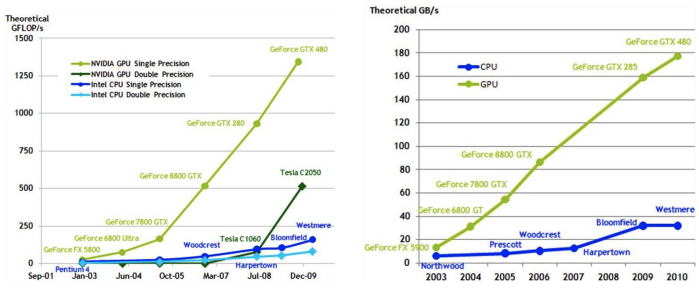
In general, not many parallel techniques have been developed, at least not directly (there are several works that implement equation solvers in parallel, for example).

One of the first parallel techniques presented is the DeWall algorithm (Cignoni et al., 1993), a recursive divide-and-conquer Delaunay triangulation in E^d where input points are spatially partitioned by an hyperplane and triangulated recursively. Final meshes are joined merging their boundaries using the partitioning simplex. Although it generates full Delaunay triangulations and it is not specific for surface reconstruction, it is a good example of a parallel approach.

Another work worth mentioning is presented in (Kohout and Kolingerová, 2003). It constructs a Delaunay triangulation in E^2 by randomized incremental insertion of points. Starting with a triangle containing all the input points, they are incrementally inserted in a parallel fashion, subdividing the triangle where the points fall, and updating the connectivity to preserve the circumcircle criterion. Each point insertion locks a part of the triangulation during the subdivision phase, avoiding inconsistencies due to concurrent access to a triangle that is being modified.

2.3.1 Hardware accelerated algorithms

In the last years, the processing power of graphic devices has been increasing drastically faster compared to the evolution of CPUs, as can be seen in Figure 2.4. This is mainly due to the high level of parallelism that a GPU has (usually, a GPU has at least several tens of streaming processors, while similar price-tagged CPUs only have two couples). Following this line, a concerted effort is being put in the development of new algorithms and the implementation of existing ones in graphic devices; this model is known as GPGPU.



(a) Floating point operations per second.

(b) Memory bandwith comparison.

Figure 2.4: Performance evolution comparison between CPUs and GPUs (image from (NVIDIA, 2010)).

Several hardware-accelerated iso-surface visualization methods have been proposed in the last years, but the hardware accelerated surface reconstruction is still an open line of research. (Klein et al., 2004) and (Kipfer and Westermann, 2005) implement the Marching Tetrahedra algorithm in the GPU, with the difference that in the first work a full

indexed tetrahedral representation is given while the second only stores the edges in order to decrease the amount of data sent to and processed by the GPU. (Jalba and Roerdink, 2006) mention a GPU implementation of the (Bloomenthal, 1988)'s implicit surface polygonizer. These works have shown the GPU's power to solve the surface reconstruction problem using spatial subdivision methods, but no implementations for interpolating approaches have been presented yet. One of the main drawbacks of traditional AFT is their low parallelization level, due to repeated iterations and the information needed from the current reconstructed mesh. On the other hand, local triangulations are more suited for parallel work if a proper data structure is given.

Finally, (Zhou et al., 2010) show a full GPU implementation of the previously described Poisson method that directly visualize the reconstructed surface, that achieves impressive reconstruction times of nearly six frames per second for half-million points sets.

Part II

Proposal

Chapter 3

GPU Local Triangulation

Sometimes when you innovate, you make mistakes.
It is best to admit them quickly,
and get on with improving your other innovations
Steve Jobs

A synthesis of this chapter has been published in:

Buchart, C., Borro, D., and Amundarain, A. "A GPU interpolating reconstruction from unorganized points". In Posters Proceedings of the SIGGRAPH 2007. San Diego, CA, USA. August 5-9, 2007.

Buchart, C., Borro, D., and Amundarain, A. "GPU Local Triangulation: an interpolating surface reconstruction algorithm". Computer Graphics Forum, Vol. 27, N. 3, pp. 807–814. May, 2008.

Buchart, C., Amundarain, A., and Borro, D. 3-D surface geometry and reconstruction: Developing concepts and applications, chapter Hybrid surface reconstruction through points consolidation. IGI Global. 2011. (Sent and under revision).

Examples of possible applications of this chapter in the medical field have been presented in:

San Vicente, G., Buchart, C., Borro, D., and Celigüeta, J. T. "Maxillofacial surgery simulation using a mass-spring model derived from continuum and the scaled displacement method." In Posters Proceedings of Annual Conference of the International Society for Computer Aided Surgery (ISCAS'08). Barcelona, Spain. June, 2008.

San Vicente, G., Buchart, C., Borro, D., and Celigüeta, J. T. "Maxillofacial surgery simulation using a mass-spring model derived from continuum and the scaled displacement method." International journal of computer assisted radiology and surgery, Vol. 4, N. 1, pp. 89–98. January, 2009.

Buchart, C., San Vicente, G., Amundarain, A., and Borro, D. "Hybrid visualization for maxillofacial surgery planning and simulation". In Proceedings of the Information Visualization 2009 (IV'09), pp. 266–273. Barcelona, Spain. July 14-17, 2009.

3.1 Introduction

Local triangulations, especially *local Delaunay triangulations* (LDT), are very suitable for parallelization since every point can be processed independently of others. By taking advantage of this property and the high parallelization level of modern graphic hardware, a new, highly efficient reconstruction method from unorganized points has been developed in this work, extending the Lower Dimensional Localized Delaunay Triangulation of (Gopi et al., 2000). A general scheme of the proposal is shown in Figure 3.1 and it will be described in the following pages. The method has been called *GPU Local Triangulation*, or GLT for short.

As an overview, given an unorganized data set P of n points, it involves, for each point $p \in P$, the computation of the k-nearest neighbors in a radius δ ($Nbhd(p)$), normal estimation, neighborhood projection onto the tangent plane, radial sorting of projected points around the normal and the local Delaunay triangulation itself. After all points have been processed, the final mesh is assembled from the local triangle fans and normals are uniformly oriented using a propagation based on the connectivity of the points.

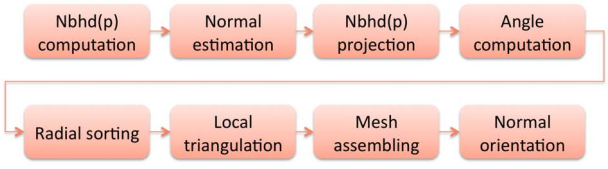


Figure 3.1: Flow diagram of the proposed reconstruction algorithm.

The only assumption made about the input points is that the data is well sampled, i.e., highly detailed surfaces can be reconstructed as long as the sampling rate conforms with the sampling criteria exposed in Section 3.2. No additional information as topology, geometry or normals is required, although this information can be incorporated in order to remove some steps (for example, normals estimation and their later propagation).

This method has been implemented using two different approaches: the first one makes use of shaders and the second one is implemented using CUDA. As each implementation has its own characteristics and issues to be solved, first the GLT will be described as a general algorithm (implementation unaware), and then each implementation will be discussed. Additionally, and mainly for comparison purposes, a CPU implementation has also been developed.

3.2 Sampling criteria

The sampling criteria is based on the Nyquist-Shannon theorem[1], such that the maximum distance δ between two adjacent points must be, at the most, half the distance between the two nearest folds of the surface. However, this general criteria could be applied locally in any subset of the whole surface, resulting in smooth transitions in the sampling rate. Anyway, if details (high frequency zones) have to be reconstructed too, the same criteria must be applied.

3.3 Description of the method

3.3.1 Preprocess phase – Computing the k-NN

In this stage, the k-nearest neighbors (k-NN) to each point $p \in P$ are obtained and they are represented as $Nbhd(p)$[2]. This problem is a particular case of the overall neighborhood problem. k-NN is a well known

[1] The Nyquist-Shannon sampling theorem states that a sampled analog signal can be reconstructed if the sampling rate exceeds $2B$ samples per second, where B is the highest frequency in the signal, expressed in hertz.

[2] As a notation commentary, when talking about the GLT, it will refer indistinctly to the $Nbhd(p)$ as the neighbors of p or the candidate points of p.

problem in computer graphics, statistics, robotics and networking, but as its computation is not the main objective of this work, just a brief explanation is given (two good reviews of the existing techniques can be found in (Friedman et al., 1977) and (Sankaranarayanan et al., 2006)).

In the present study, two different methods have been used to compute the k-NN of the input points set (either using clustering or kd-trees). k-NN is a quadratic problem, and the size of data could be very high, so the election of a good method is essential for the good performance of the overall algorithm.

The value of k depends mainly on how uniformly sampled the dataset is; the local sampling rate is only important if there are abrupt changes of density. Empirically, it was set $k = 32$ for almost all the tests. Although k can take any positive value, the GPU data structure for candidate points employed in the shaders implementation uses a power-of-two neighborhood size (see Section 3.4 for further explanation).

Finally, to avoid the selection of points in opposite sheets of the surface, the neighborhood is restricted to the δ-vicinity of p. However, if multiple sampling rates are present, the size must be set to the maximum sampling rate (the sampling criteria guarantees that there are enough points nearer to avoid a wrong triangulation). If the correct orientation of normals is given, it may be also included to remove opposite points.

3.3.1.1 k-NN based on clustering techniques

The first method implemented is based on the assumption of a regular sampling rate. It applies a clustering division to reduce the searching set. The bounding box space of the data set is divided in cells of dimension δ^3 to reduce the size of the problem. For each point p it performs a full circumsphere test against the points contained in its cell and in adjacent ones. All the points which fall into the δ-sphere are inserted in a reversed priority queue (where the first point in the queue is the farthest from p among the k nearest neighbors). This priority queue efficiently sorts new neighbors on-the-fly, avoiding the insertion into the queue of points that are not near enough. Additionally, the priority queue is fixed to a maximum of k elements.

3.3.1.2 k-NN using kd-trees[3]

The second method used is based on kd-trees (Friedman et al., 1977). The kd-tree is a generalization of the binary tree, where each node represents a subset of the points and a partitioning of it (the root is unpartitioned and contains all the points). This partitioning is done using orthogonal alternating planes to subdivide the space; all the points belonging to the same side of the dividing plane will be found in the same sub-tree. Leaves are not divided and contain a single point.

A full explanation of how to compute the nearest neighbors to a point using kd-trees falls beyond the scope of this work; for a proper discussion about this topic please refer to (Bentley, 1990). In this sense, a publicly available implementation of $\bar{k}$-NN using kd-trees, called ANN (*approximating nearest neighbors*), was used in this work (Mount and Arya, 2010).

3.3.1.3 Final comments

At the beginning of the development of the proposed GLT method, the computation of the k-NN of P used a priority queue driven search. Nevertheless, it was found that the utilization of the ANN library resulted in a performance boost of 75% of the execution time, on average. Given this huge improvement, priority queues were completely removed from the implementation, remaining only as an example of alternative k-NN methods. This difference in speed is mainly due to the brute force nature of the first approach; even when the problem's domain is reduced as much as possible, it is still necessary to compute the distance to many points, in comparison to a kd-tree search.

3.3.2 Parallel triangulation

As was previously mentioned in Section 2.1.2, local triangulations are independent point to point. Based on this property, the presented method performs parallel triangulations over each point. Following stages will be

[3]To avoid confusion, in this section the term k will be used to refer the dimensionality of the kd-tree (e.g., $k = 3$ for point sets in $\Re^3$) rather than the size of the search space, which will be called $\bar{k}$ only in this section.

referred to a single point $p \in P$, and its neighborhood $Q = \{q_j \in Nbhd\,(p)\}$, such that q_0 is the nearest neighbor of p. It is understood that these stages are applied to every single point of P to get the final reconstruction.

3.3.3 Phase 1 – Normal estimation

The input data is usually a set of unorganized points but additional information is assumed in many proposals. Thereby, (Crossno and Angel, 1999) make use of a wide set of information: normals, the correct neighborhood of each point and point kind, and (Bernardini et al., 1999) and (Kazhdan et al., 2006) assume the presence of an oriented set of points. Although this information is very useful, it is not generated by many acquisition techniques.

Several methods to estimate normals from point clouds have been proposed, but particularly useful in interpolating methods are those explained by (Hoppe et al., 1992) and (Gopi et al., 2000). As an overview, the first method uses *principal component analysis* (PCA) and the later sees the problem as a singular value decomposition problem. Although both are quite different approaches, in the end, they are equivalent (Gopi et al., 2000). Nevertheless it has been seen that the (Hoppe et al., 1992) proposal is easier to implement and solve, so it is the one used in this work.

Given a neighborhood Q, the tangent plane T of p (and therefore its normal $\mathbf{n}$) may be estimated as the least squares best fitting plane to Q, minimizing $-\mathbf{v}_j^\top \mathbf{n}$, where $\mathbf{v}_j = q_j - p$:

$$
\begin{aligned}
\min_{\mathbf{n}} \sum_{q_j \in Q} \left(-\mathbf{v}_j^\top \mathbf{n}\right)^2 &= \min_{\mathbf{n}} \sum_{q_j \in Q} \left(\left(-\mathbf{v}_j^\top \mathbf{n}\right)\left(-\mathbf{v}_j^\top \mathbf{n}\right)\right) \\
&= \min_{\mathbf{n}} \sum_{q_j \in Q} \left(\left(-\mathbf{n}^\top \mathbf{v}_j\right)\left(-\mathbf{v}_j^\top \mathbf{n}\right)\right) \\
&= \min_{\mathbf{n}} \sum_{q_j \in Q} \left(\mathbf{n}^\top \left(\mathbf{v}_j \mathbf{v}_j^\top\right) \mathbf{n}\right) \\
&= \min_{\mathbf{n}} \mathbf{n}^\top \left(\sum_{q_j \in Q} \left(\mathbf{v}_j \mathbf{v}_j^\top\right)\right) \mathbf{n} \\
&= \min_{\mathbf{n}} \mathbf{n}^\top C V \mathbf{n}
\end{aligned}
\tag{3.1}
$$

where CV is the following covariance matrix

$$CV = \sum_{q_j \in Q} \mathbf{v}_j \mathbf{v}_j^\top \tag{3.2}$$

As explained above, PCA is used to estimate $\mathbf{n}$. Let $\lambda^1 > \lambda^2 > \lambda^3$ be the eigenvalues of CV associated with eigenvectors $\mathbf{v}^1, \mathbf{v}^2, \mathbf{v}^3$, then $\mathbf{n}$ can be approximated by either $\mathbf{v}^3$ or $-\mathbf{v}^3$ (orientation is not a requirement in this stage of the reconstruction). To compute the eigenvectors of CV, the Deflation method (Press et al., 2007) was employed, given its implementation in the GPU is efficient.

In this work, a variation of the method of (Hoppe et al., 1992) was employed, using *weighted PCA* (wPCA) instead (Pauly et al., 2002). In this case, the only change is the computation of the covariance matrix:

$$CV = \sum_{q_j \in Q} \theta\left(\|\mathbf{v}_j\|\right) \left(\mathbf{v}_j \mathbf{v}_j^\top\right) \tag{3.3}$$

where $\theta(r)$ is the same weight function used in Section 4.1.2 (see Equation 4.4). Figure 3.2 illustrates the results of using PCA and wPCA.

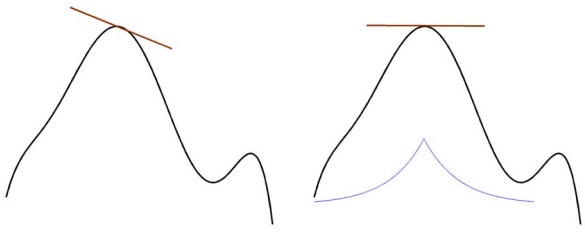


Figure 3.2: Illustration of normal estimation using PCA (left) and wPCA (right). The weighting function, showed in blue, regulates the influence of the farthest points.

The choice of this method over others is justified given it uses a 3×3 matrix, a native datatype on GPUs. Another option is the equivalent estimator of (Gopi et al., 2000), but it needs a $3 \times k$ matrix, so its implementation in the GPU would be harder and less efficient.

Normals are not only used in the visualization stage, as an output of the algorithm, but they are required to project and sort candidate

points for triangulation, then if the normal estimation is not precise enough later steps will produce a wrong triangulation, i.e., numerical errors could become a major problem in this process. During the normal estimation, the smallest eigenvalue is very susceptible to errors due to hardware precision and numerical errors. Given that the Deflation method must first compute the two higher eigenvalues and their eigenvectors to obtain the third one, rounding errors may become a problem (as also mentioned in (Press et al., 2007)). To solve this issue, the tangent plane T is estimated instead of the normal, using the eigenvectors associated with the two highest eigenvalues; in this way, the error propagation of the last eigenvector is reduced. The normal is obtained from this tangent plane using the cross product of the two eigenvectors.

3.3.3.1 Normals orientation

Normals estimated in this phase do not necessarily have a correct orientation (some of them may face the model inwards). Some methods rely on this orientation, usually because it is used to label zones as interior or exterior; just to mention a couple: the work of (Hoppe et al., 1992) and the Poisson method (Kazhdan et al., 2006).

To perform the normals unification it is common to take a normal as correct and then use some kind of propagation. For example, (Hoppe et al., 1992) treat the orientation problem as a spanning tree problem where the graph's vertices are the input points, and the connectivity is given by the triangulation. An arbitrary point is taken as to have the right normal and then it is propagated through the minimum spanning tree from vertex i to vertex j: if $\mathbf{n}_i^\top \mathbf{n}_j < 0$ then the normal $\mathbf{n}_j$ is reversed. If the whole orientation is wrong (normals are pointing inward at the model), then it is enough to reverse all the normals. Methods that rely on normal orientation usually propagate it before entering the reconstruction phase itself. The rest of the works may incorporate this task in any moment of the reconstruction pipeline, although it is common to leave it as a post-process.

In this work, however, normals orientation is not a problem during the computation of triangle fans; the local Delaunay triangulation is the same regardless of the normal orientation, except for points ordering. Depending on the uses of the reconstructed model, incorrectly oriented normals and triangles may have a negative repercussion. For example, if the goal of

the reconstruction is to create inner/outer zones based on the boundary representation of the model, the orientation of the normal is important. On the other hand, if the surface is only needed in the visualization mainstream, it may be possible to use a double-face illumination, making unnecessary a normal post-processing. Moreover, normal propagation is not a parallel task, so, in case it is needed, it is better to perform the orientation correction at the end of the reconstruction.

3.3.4 Phase 2 – Projection

Following (Gopi et al., 2000), GLT maps $Nbhd(p)$ onto the tangent plane T. The set of projected points $Q' = \{q'_j\}$ is given by the minimal rotation of each $\mathbf{v}_j$ onto the plane defined by $\mathbf{v}_j$ and $\mathbf{n}$ such that $\left(q'_j - p\right)^\top \mathbf{n} = 0$, i.e., q'_j relies on T (see Figure 3.3). In order to reduce the number of operations to be executed in the GPU this minimal rotation is computed as a projection onto T:

$$q'_j = p + \frac{\hat{q}_j - p}{\|\hat{q}_j - p\|} \|\mathbf{v}_j\| \tag{3.4}$$

where $\hat{Q} = \{\hat{q}_j\}$ is the orthogonal projection of q_j onto T:

$$\hat{q}_j = q_j - \left(\mathbf{n}^\top \mathbf{v}_j\right) \mathbf{n} \tag{3.5}$$

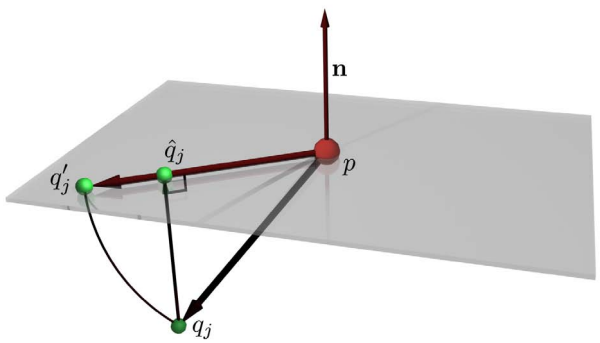


Figure 3.3: The minimal rotation can be expressed as a projection.

3.3.5 Phase 3 – Angle computation

In this section, the angle α_j between each projected neighbor q'_j and q'_0, with center in p is computed. The neighborhood of p must be sorted in a ring fashion in order to construct the Delaunay triangulation; this angle will be used later to sort the neighborhood. The computation of the angle is as follows:

$$\begin{aligned} \alpha_j &= \arccos\left({\mathbf{v}'_0}^{\top}\mathbf{v}'_j\right) \\ \mathbf{v}'_j &= q'_j - p \end{aligned} \tag{3.6}$$

To deal with the cosine's symmetry about π, it is enough to see in which side the point v'_j lies with respect to the plane formed by $\mathbf{v}_0$ and $\mathbf{n}$. So, let $\mathbf{m} = \mathbf{n} \times \mathbf{v}_0$; then, if the dot product $d = {\mathbf{v}'_j}^{\top}\mathbf{m}$ is negative set $\alpha_j = 2\pi - \alpha_j$ (Crossno and Angel, 1999), so $\alpha_j \in [0, 2\pi)$. As this angle only will be used later for sorting the neighborhood, the arccosine has been eliminated from Equation 3.6 to reduce the cost of this calculation. To avoid confusion in the notation, a new notation is used to represent this new *angular position*: $\bar{\alpha}_j = \cos(\alpha_j)$. The symmetry rotation when d is negative has also been modified to $\bar{\alpha}_j = -2 - \cos(\alpha_j)$, so $\bar{\alpha}_j \in (-3, 1]$, where 1 is equivalent to the smallest angle. To summarize, the computation of $\bar{\alpha}_j$ is finally

$$\bar{\alpha}_j = \left\{ \begin{array}{ll} {\mathbf{v}'_0}^{\top}\mathbf{v}'_j & m^{\top}\mathbf{v}_j \geqslant 0 \\ -2 - {\mathbf{v}'_0}^{\top}\mathbf{v}'_j & m^{\top}\mathbf{v}_j < 0 \end{array} \right\} \tag{3.7}$$

When implementing these two phases (projection and angle computation), it can be realized that combining them into a single phase leads to both mathematical and implementation simplifications. For example, $\mathbf{v}'_j$ can be expressed as

$$\mathbf{v}'_j = q'_j - p = \frac{\hat{q}_j - p}{\|\hat{q}_j - p\|} \|p - q_j\| \tag{3.8}$$

so Equation 3.4 can be simplified:

$$q'_j = p + \mathbf{v}'_j \tag{3.9}$$

3.3.6 Phase 4 – Radial sorting

The Delaunay local triangulation first requires a sorted set of points, more specifically *radial sorted*: starting at the nearest projected neighbor q'_0 of p, radial sorting creates a ring of consecutive points around p, with sorting key $\bar{\alpha}_j$. No further explanations will be given in this section, since the sorting method employed is highly dependent on the implementation.

3.3.7 Phase 5 – Local triangulation

The triangulation step is derived from the Lower Dimensional Localized Delaunay Triangulation algorithm presented by (Gopi et al., 2000). This verifies if each candidate point $q'_j \in Q'$ remains as a valid Delaunay neighbor when compared with q'_{j-1} and q'_{j+1}. If it does it continues with the next point, otherwise it is removed and backtracks to verify the validity of the triplet $\left\langle q'_{j-2}, q'_{j-1}, q'_{j+1} \right\rangle$. When no points are marked as invalid, the local Delaunay triangulation comprises all the remaining valid points. As candidate points are sorted by angle, and therefore in a cyclic way, any neighbor can be chosen as a starting point. As the nearest neighbor is always a Delaunay neighbor, the process starts with it (that is the reason why q'_0 was chosen to be the nearest point to p). In addition, it is used as a stop condition for the backtracking. It is important to remark that all these computations are done using the projected neighbors of p from Phase 2, thus, it is a 2D Delaunay triangulation; when projected points are replaced with the original ones, then it becomes the surface reconstruction in 3D.

Note that, even when the authors of this method do not comment the implementation, it is reasonable to assume that it is not a recursive method but an iterative backtracking one. In this sense, in this work an iterative version of the triangulation was developed (see Algorithm 3.1, the *is_valid* mentioned is the one explained in Section 3.3.7.1). In this design, an auxiliary ring structure $R = \{r_j\}, j \in [0, m], m = |Q'|$ is used. Each element of R corresponds with a projected neighbor of p (except for the last element, r_m that is a copy of the first r_0, closing the ring).

Algorithm 3.1 Local Delaunay triangulation in 2D

```
1: function LOCAL_2D_DELAUNAY(p, m, R)
2:     j ← 1
3:     while j < m do
4:         if is_valid(p, ⟨r_{j−1}, r_j, r_{j+1}⟩) = false then
5:             remove the j-th element from R
6:             m ← m − 1
7:             if j > 0 then
8:                 j ← j − 1
9:             end if
10:        else
11:            j ← j + 1
12:        end if
13:    end while
14: end function
```

3.3.7.1 2D validity test

The validity test actually determines if q'_i belongs to a partial Voronoi region of p. As was explained in Section 2.1.1, Delaunay triangulations and Voronoi diagrams are duals, i.e., either of them can be computed from the other one.

As a notation comment, in both this and the following section, as well as in the correspondent sections in the shader and CUDA implementations, the symbols used for projected neighbors will be changed as explained in the next paragraph, in order to make the text easier to read and to avoid confusion. This is because the validity test has a very local scope regarding the neighborhood of p (they only "see" points). In the rest of this manuscript the notation will be the same until now.

Given three consecutive projected neighbors $A = \{a_k\}, k \in \{1, 2, 3\}$, the function $is\_valid(p, A)$ verifies if a_2 remains in the Voronoi region of the points $\langle p, a_1, a_2, a_3 \rangle$. Define $b_k = a_k - p$ and $o_k = p + b_k/2$. Then, let $\bar{b}_k$ be the line perpendicular to b_k that passes through o_k, i.e., $\bar{b}_k$ is a Voronoi edge. Also, let s be the intersection point of the lines that go through o_1 and o_3, with directions $\bar{b}_1$ and $\bar{b}_3$, respectively. If the projection of s onto b_2 lies in the region between p and o_2, then s is a possible Voronoi vertex and a_2 is not a Delaunay neighbor of p (see Figure 3.4). It is important to note that this

test only determines if a point is not a Voronoi neighbor of p, but if the point passes the test, it is not guaranteed that it is connected to p in the Delaunay triangulation. It comes from the fact that three invalid consecutive points may result in a valid local region but may not be necessarily in the final triangulation (see Figure 3.5 for an example); this is solved by repeating the test several times only with the updated valid neighborhood, as proposed by (Gopi et al., 2000).

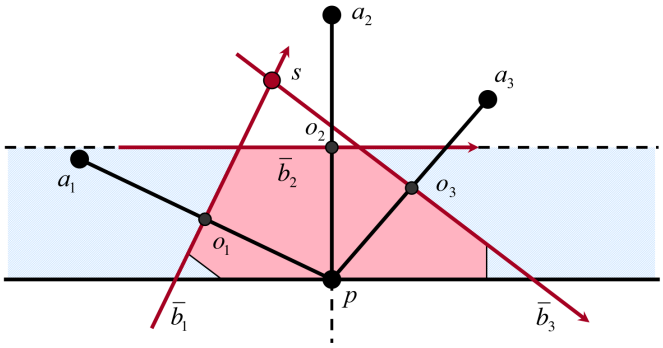


Figure 3.4: The *is_valid* function verifies if a point belongs to a partial Voronoi region.

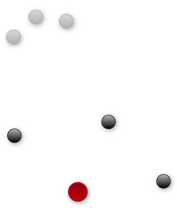


Figure 3.5: The validity test is local and must be performed several times to obtain the local triangulation of a point. In this figure, the three gray candidates will not fail the validity test of the central red point, but when checked with the dark neighbors they will be discarded.

In the next section, a mathematical proof about the validity of the method is provided. It also states that it does not reject valid points, which will allow the development of more GPU efficient versions of Algorithm 3.1.

3.3.7.2 Proof

In the local region defined by p and A, the vertices of the partial Voronoi diagram are defined by the intersections of the perpendicular bisectors of consecutive neighbors. If the function returns a `false` value, it means that a Voronoi edge nearer to p than a_2 exists, discarding a_2 as a Voronoi neighbor of p. It will now be proved that *is_valid* will never invalidate an actual Voronoi neighbor of p.

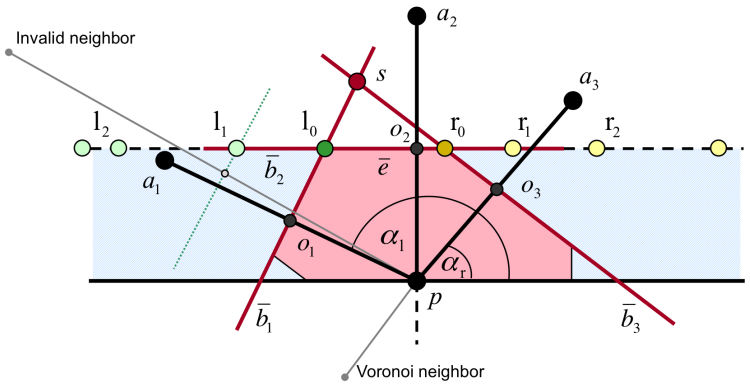


Figure 3.6: *is_valid* never invalidates Voronoi neighbors.

Let's change the local coordinate system as illustrated in Figure 3.6, so that b_2 lies on the positive Y-semiaxis, a_1 on its left and a_3 on its right (if both a_1 and a_3 lie on the same side of Y, a_2 is on a surface boundary and must be included in the triangulation). Now, if a_2 is a Voronoi neighbor of p, then a Voronoi edge $\bar{e}$ must exist and is limited by points l_0 and r_0, which are the intersections of the perpendicular bisectors of $\bar{b}_1$ and $\bar{b}_2$, and $\bar{b}_2$ and $\bar{b}_3$, respectively ($\bar{e}$ lies on $\bar{b}_2$). It is clear that l_0 is the farthest intersection to the right of perpendicular bisectors of the left side, and r_0 is the farthest intersection to the left of the right side, making $r_0^x > l_0^x$ for any right and left intersections. Let α_l and α_r be the angle of b_1 and b_3, respectively, with $\alpha_r \in (-\pi/2, \pi/2)$ and $\alpha_l \in (\pi/2, 3\pi/2)$. Three possible scenarios exist:

- b_1 and b_3 lie on the same line, i.e. $\alpha_l - \alpha_r = \pi$. In this case, $\bar{b}_1$ and $\bar{b}_3$ do not intersect and a_2 is not invalidated.
- $\alpha_l - \alpha_r < \pi$, then the intersection point s lies on the positive Y-semiaxis. As stated before, $r_0^x > l_0^x$ means that s is above $\bar{e}$, thus

a_2 is not invalidated.

- Finally, if $\alpha_l - \alpha_r > \pi$ it means that the intersection of $\bar{b}_1$ and $\bar{b}_3$ lies on the negative Y-semiaxis, following that a_2 is not invalidated either.

From this, the backtracking method proposed by (Gopi et al., 2000) (and therefore the proposed implementation described in Algorithm 3.1) can be substituted by algorithms that dot not rely on the validation order, allowing the parallelization of each validation, as well as GPU implementations as described in Sections 3.4.6 and 3.5.3.

As was mentioned before, the following sections will discuss two different GPU implementations of GLT. The first one uses shaders and it presents more challenges since it has more restrictions from hardware, but it was the only possible implementation available for several years. The second implementation is done using CUDA, a very recent architecture that is more flexible in terms of similarity to traditional programming languages.

3.4 Implementation using shaders

When using shaders, high amounts of data must be stored in textures in order to be accessible by the GPU. One problem here is that textures are not very flexible in how the data can be stored in them; they are grids with no more than four elements in each cell, and all the elements must be of the same datatype. Textures also present size restrictions (usually, the maximum size admitted is 4096×4096 texels). To solve this issue, a divide-and-conquer strategy is used, splitting the input points into K smaller and equal datasets called *passes*. The size of each pass is denoted by n'[4]. Passes are stored in textures and transferred to the graphic hardware for their processing using fragment shaders. Although it is a similar strategy to the one used by (Cignoni et al., 1993), GLT does not perform a spacial division of the points but may use any points ordering, for example, the storing order of the dataset, which also leads to a faster initialization. In addition, this scheme would allow an easy implementation of the method

[4]Actually, the last pass may be smaller due to internal fragmentation, but including this fact into the discussion and using a proper notation for it would only make it more difficult to read. As passes are independent it will be assumed that n' represents the size of the current pass.

on multi-GPU architectures because passes are completely independent of each other. Figure 3.7 shows the workflow for this implementation.

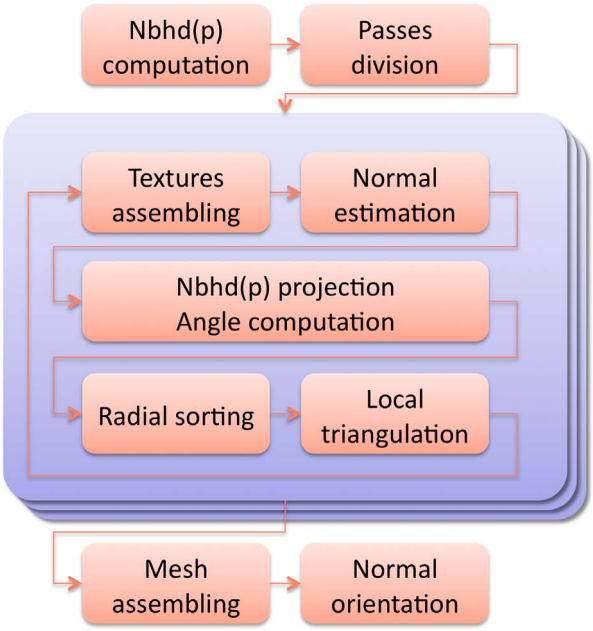


Figure 3.7: Flow diagram of the shaders implementation.

3.4.1 Initial texture structures overview

Before detailing the implementation of each stage, the base data structures used in all of them will be shortly described. Textures used only in a particular stage will be described in their respective section.

As mentioned in Section A.1, a fragment shader is not able to write to a random position, only in the same position of the fragment it is processing. This means that the data structures condition the way fragment shaders can process them. GLT performs two kinds of data manipulation: one is orientated to obtain points information such as the normal, the other one works with each candidate point individually. Therefore, data structures must be designed to fulfill these two methodologies: *per-point processing* (where a fragment represents information about a point) and *per-neighbor processing* (where a fragment represents information about a neighbor or

candidate point).

Although points are processed in several passes, information about many additional points may be required (e.g., points out of the current pass may be neighbors of others belonging to it). To simplify the design of the GLT, all the points are stored in a single texture called T_P. This texture uses a `RGB` format to store the `XYZ` coordinates of the points. Normals are stored in a similar texture (T_N), and it is updated pass by pass after each normal estimation step. In the current implementation, these two textures are global and shared among all the steps of the algorithm.

Access to the points texture is done by index, not by texture coordinate, since it is easier to store a single value (the index) than two (corresponding `U` and `V` coordinates). Equation 3.10 shows the conversion from index to texture coordinates

$$\begin{aligned} \texttt{U} &= i \ \textbf{mod} \ height \\ \texttt{V} &= \lfloor i/width \rfloor \end{aligned} \tag{3.10}$$

where *width* and *height* are the dimensions of the texture. To make the design of the conversion function easier, the texture employed is square. To minimize the memory required to store, the size of the texture is computed as shown in Equation 3.11.

$$size(T_P) = \left\lceil 2^{log_2(\sqrt{n})+1} \right\rceil \tag{3.11}$$

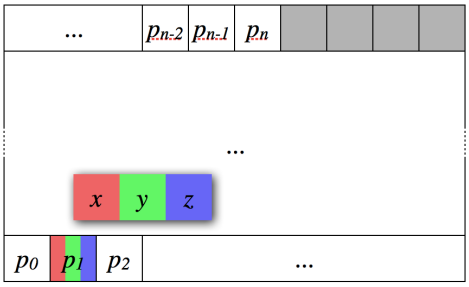


Figure 3.8: T_P – Points texture. The normals texture T_N has an identical structure.

Once the points are stored, the neighborhoods of points that are processed in a pass must also be uploaded to the graphic memory. To reduce

the structure complexity and satisfy the power-of-two size requirement of the Bitonic Merge Sort (Section 3.4.5), the size of each neighbor structure (k) is set to a power-of-two, although all the neighborhoods are bounded to $k-1$ points because the nearest point is inserted twice. The texture for the candidate points is called T_Q and presents a per-neighbor processing scheme. The whole neighborhood of a point is stored row-by-row, avoiding the data of a single neighborhood to be split in different `V` texture coordinates; the goal is to reduce shaders complexity using the index of a candidate point as an offset in the `U` coordinate. Each group is formed by the m neighbors of p sorted by its distance to p, plus the nearest point duplicated at the end to close the ring needed by the triangulation algorithm (see section 3.3.7); if $m < k-1$, the group is completed with invalid neighbors that will be discarded by the following steps (a special value is assigned to identify them). Each candidate point packs in the `R` the square distance to p. The other three `GBA` components are its index in T_P, its relative position in the neighborhood and m.

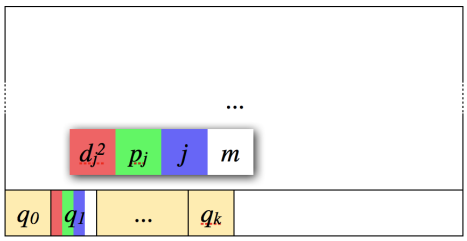


Figure 3.9: T_Q – Candidate points texture: squared distance to center, the neighbor, its index and the neighborhood size.

There is another texture, called T_I (Figure 3.10), that stores all the points of the pass and that is used as an index table to locate the neighbors of p in T_Q. It is employed as a common source by phases that require per-point processing. The structure consists of the point reference in T_P, the initial and final `U` coordinates in T_Q and its `V` position, packed in a `RGBA` format.

The size of T_Q is $m_Q \times m_Q$, where m_Q is a multiple of k (non-square textures can be used, as far as the horizontal dimension meets the previous statement). On the other hand, T_I's size is fixed to $n_P \times n_P$, where n_P^2 is the nearest integer to the number of points in the pass. As can be seen, a couple of fragmentation problems appear: the first can be found in either

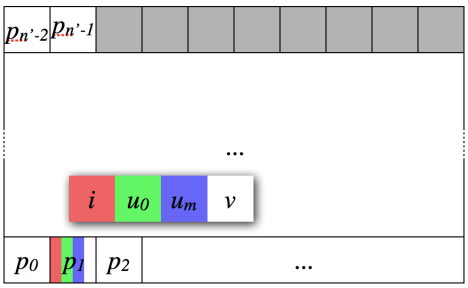


Figure 3.10: T_I – Index table texture: central point, the beginning and end of the candidate points block, and its row.

T_Q or T_I, where it was preferred to fill completely the candidate points since it has more size restrictions. The second is presented in the last pass where, based in the previous decision, T_Q would lose on average half of its space. A small m_Q minimizes this loss, but increases the number of passes and memory required by the method. After that, and due to the textures size restriction of 4096×4096, m_Q was fixed to 512 for the experiments, given that it balances all previous factors very well.

3.4.2 Texture assembly

In this step, the neighborhood of the points belonging to the current pass are computed, and the base data structures T_Q and T_I are assembled and transferred to the GPU. If a parallel system (multi-GPU architectures, clusters) is used, it would only be necessary to make copies of the input points and assign a set of passes to each node.

3.4.3 Phase 1 – Normal estimation

Given that the result is a per-input-point value, this shader makes use of the T_I texture as input fragments and reads $Nbhd(p)$ from T_Q. The output then, has the same size of T_I and it has a normal for each point of the pass, which is copied to the normal texture T_N before going to the following step.

As an interesting remark, the normal of a given point is only needed by such a point, so there is no need to compute all the normals before proceeding to the next stage; it can be done pass by pass.

3.4.4 Phases 2 and 3 – Projection and angle computation

In this stage, rather than perform multiple passes to process each neighbor using the pass points (as if using the read scheme of normal estimation), the candidate points texture T_Q is employed to perform this work in parallel. As a result, two additional textures are generated: T_α which has a similar structure of T_Q but with the angle stored in the place of the distance (Figure 3.11), and the other with the projected points ($T_{Q'}$, see Figure 3.12). As will be mentioned in section 3.3.7, the algorithm needs a candidate points ring for each local triangulation, so it makes a copy of the nearest point p_0 at the end of the candidate points list to close the ring, since it is always a neighbor of p in the final mesh.

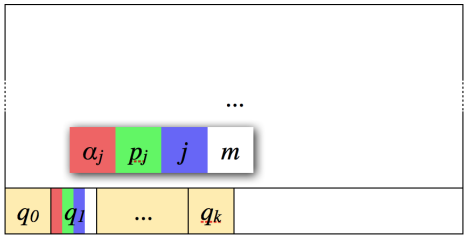


Figure 3.11: T_α – Angles texture: similar to the T_Q structure but replacing the distance with the angle.

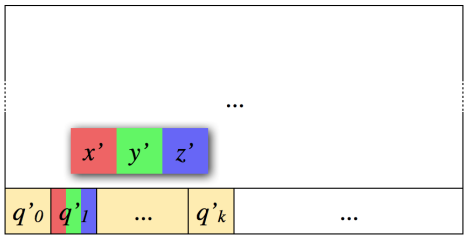


Figure 3.12: $T_{Q'}$ – Projected candidate points texture.

3.4.5 Phase 4 – Radial sorting

Neighbors are then sorted by angle in the graphic memory using a modified version of the GPU Bitonic Merge Sort algorithm (Batcher, 1968; Buck and

Purcell, 2004), where instead of a single big block of data, it sorts many small, fixed size blocks.

The Bitonic Merge algorithm sorts data sets of n elements, where n is a power-of-two (do not confuse this n with the cardinality of P). The data structure meets this criteria (k is a power-of-two) but not necessarily the number of neighbors of each point. As mentioned before in Section 3.4.1, invalid points are inserted to complete the required number of points. For a correct sorting, such false neighbors must remain at the end of the list, so a big angular position $\tilde{\alpha}$ (fixed to -1000.0 in this work) is used. An additional problem is that there could be cases where a point is very close in angle to the first or last point (which are the same), then, to avoid numerical errors and to guarantee a correct sorting, the angular position of the first and last (the same) neighbors are not computed, but are fixed to $\bar{\alpha}_0 = 2$ and $\bar{\alpha}_m = \tilde{\alpha}/2$ to ensure they remain at the beginning and the end of the valid candidate points, respectively. Any other point is inserted at texture assembly time, but special angles are assigned during the angle computing.

The original GPU implementation of the Bitonic Merge proposed by (Buck and Purcell, 2004) sorts a single array of data stored in a 2D texture. GLT must sort many lists in each pass, but creating copies of each neighborhood and sorting them one by one is very inefficient. To avoid a loss of the parallelization level, all the neighborhood must be sorted directly on their texture.

In order to solve this issue, GLT treats the multiple lists problem as if it were one. Taking advantage of the single row packing of T_Q texture, and the fact that the size of all neighborhoods is the same (counting the false neighbors), a new implementation that is halfway between the original Bitonic Merge Sort and the GPU version is possible. The rasterizer `X` position (equivalent to the `U` texture coordinate of T_Q) is used to offset the index of the current element. Additionally, this implementation avoids the use of the 1D-to-2D coordinate conversions present in the original GPU implementation. The final algorithm can be found in the Algorithm 3.2 and the proposed modification (simple but very efficient) has been remarked in red.

Please note that in the same process, $T_{Q'}$ is also sorted to keep coherent to the order of T_α (Line 12). In this way, two simultaneous sorts are done. Additionally, the order of some operations have been changed: even when the angles $\bar{\alpha}_j$ and $\bar{\alpha}_k$ are not needed until Lines 9 and 10, they

Algorithm 3.2 Adapted Bitonic Merge Sort for multiple lists sorting

```
1:  function ADAPTED_BITONIC_MERGE_SORT(j, stage, step, offset)
2:      Let pos_j be the current rasterizer position (x, y)
3:      ᾱ_j ← T_α[pos_j]
4:      sign ← (j mod stage) < offset ? 1 : −1
5:      k ← sign ∗ offset
6:      pos_k ← pos_j + (k, 0)
7:      ᾱ_k ← T_α[pos_k]
8:      dir ← 2 (1 − ((j/step) mod 2)) − 1
9:      pos_min ← ᾱ_j < ᾱ_k ? pos_j : pos_k
10:     pos_max ← ᾱ_j < ᾱ_k ? pos_k : pos_j
11:     T_α[pos_j] ← sign = dir ? T_α[pos_min] : T_α[pos_max]
12:     T_Q'[pos_j] ← sign = dir ? T_Q'[pos_min] : T_Q'[pos_max]
13: end function
```

are computed before in strategic places to reduce the latency of texture fetching ($\bar{\alpha}_j$ in Line 3, $\bar{\alpha}_k$ in Line 7). This is a common practice in shaders based programming and it is widely used in the implementation of other algorithms in this work as well.

3.4.6 Phase 5 – Local triangulation

In order to exploit the parallelism and to fulfill GPU restrictions when using shaders, in this implementation of GLT, Algorithm 3.1 was substituted by a series of simultaneous validations over the ring of candidate points (see Algorithm 3.3 and Figure 3.13). In each validation pass h, all candidate point are marked as valid or invalid, and in the next round all valid ones update their neighbors, in case any have been invalidated. When no changes occur, the local Delaunay triangulation of p finishes.

The R structure has been updated to support the new algorithm and it now contains references $r_j.prev$ and $r_j.next$ to the previous and next valid neighbors, as well as a validity flag $r_j.valid$. Initially, all the neighbors are considered valid and the ring is fully connected. For example, if $r_j = q'_j$, then $r_j.prev = q'_{j-1}$ and $r_j.next = q'_{j+1}$, and therefore, $r_{j-1}.next = r_{j+1}.prev = r_j$. Special cases are the first and last elements: $r_0.prev = r_{m-1}, r_m.next = r_1$ (remember that the last $r_0 = r_m$).

The supporting texture T_R that is used to store R can be seen in Figure

3.14. It includes the next and previous valid neighbor in the RG components, the index of the central point (p) and the validity flag in the BA texel components; as all the neighbors of p are stored under the same U coordinate (the same "row"), only the offset from the first neighbor is necessary.

Algorithm 3.3 Local Delaunay triangulation algorithm adapted for its implementation with shaders. $prev\_valid\_neighbor(j)$ and $next\_valid\_neighbor(j)$ are supporting functions.

1: $h \leftarrow 1$
2: Create $T_R^{(0)}$ from T_α
3: **function** LOCAL_2D_DELAUNAY($j, T_R^{(h-1)}$) $\Rightarrow T_R^{(h)}$
4: Let pos_j be the current rasterizer position (x,y)
5: $r_j \leftarrow T_R^{(h-1)}[pos_j]$
6: **if** $r_j.valid$ **then**
7: $r_j.prev \leftarrow prev\_valid\_neighbor(j)$
8: $r_j.next \leftarrow next\_valid\_neighbor(j)$
9: $r_j.valid \leftarrow is\_valid(p, q'_{r_j.prev}, q'_j, q'_{r_j.next})$
10: **end if**
11: **return** r_j
12: **end function**

Finally, this algorithm uses a ping-pong scheme: only two copies of T_R are actually used ($T_R^{(0)}$ and $T_R^{(1)}$) and in each pass they alternate the reading and writing roles. This is due to a hardware limitation: texture scattering is not possible to textures that are being used for fetching.

3.5 Implementation using CUDA

In the previous section numerous adaptations required to implement GLT using shaders were discussed. Although with the arrival of CUDA many of these adaptions became unnecessary, not every implementable solution is efficient (many factors like processors occupancy and accesses to global memory have a strong influence in the performance of the CUDA kernel). In this section, the main challenges when creating the CUDA implementation will be discussed. The correspondent algorithm workflow for the CUDA implementation can be seen in Figure 3.15.

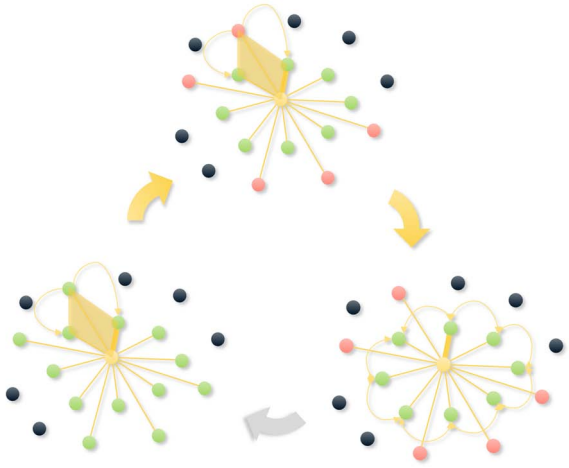


Figure 3.13: The shaders implementation of GLT uses a ring to check all the neighbors in parallel, discarding the invalid ones in each iteration.

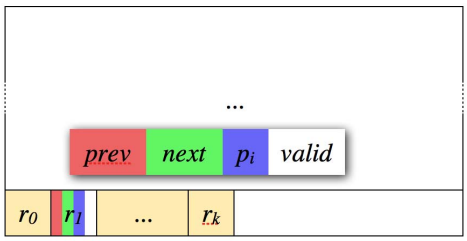


Figure 3.14: T_R – Delaunay ring texture: previous and next neighbor, the central point and validity flag.

3.5.1 Data structures

In contrast to shaders, CUDA allows the use of more complex datatypes such as structures, simplifying the design and increasing the data that a single point may directly store. Even with this advantage, it is convenient to inspect the different algorithms and see what data is actually needed by each of them. If all the data is stored into a single package, it will increase unnecessarily the bandwidth required to upload them, as well as reducing the efficiency of GPU's cache.

In this sense, instead of dedicating individual sections to each data

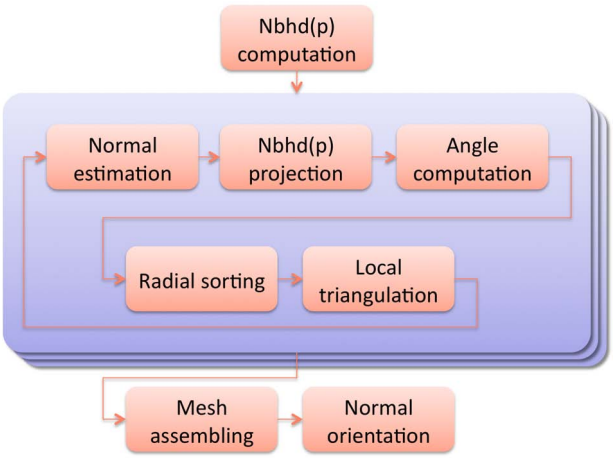


Figure 3.15: Flow diagram of the CUDA implementation.

structure (as happened with the more complicated ones used in the shaders development), they will be enumerated and, when possible, the equivalent or more similar texture structures from the shader implementation will be mentioned:

- C_P, C_N: store the input points and their normals. They are equivalent to T_P and T_N, respectively.

- C_R: contains the neighborhood information, angles and distances. As CUDA kernels are able to write information in the same memory space of the input parameters, this structure is reused in the triangulation phase (as a low-level implementation detail, it makes use of the C `union` declaration to preserve the readability of the code while reusing the fields). This structure is then a combination of the T_Q, T_α and T_R.

- $C_{Q'}$: this structure contains the projected neighborhoods, in the same way $T_{Q'}$ does.

- C_m: stores the size of each neighborhood. It has no direct equivalent in the shader implementation, where this data is stored directly in the neighborhood textures for performance issues.

3.5.2 Phase 4 – Radial sorting

In the shader implementation of GLT, the radial sorting is performed using a modified version of the Bitonic Merge Sort, optimized to work with several small lists at a time. For the CUDA implementation, three different sorting algorithms were tested: the modified Bitonic Merge Sort (see Section 3.4.5), Insertion Sort (Knuth, 1998) and the widely known Bubble Sort. Comparing the times of each one, it could be seen that, even when it has a quadratic complexity, the Bubble Sort produced the best results. It is due to the fact that the lists are small, and that the other algorithms use more registers, reducing the occupancy of the GPU.

3.5.3 Phase 5 – Local triangulation

The ring-based algorithm exposed in Section 3.4.6 has been proved to be both valid (from proof in Section 3.3.7.2) and efficient when implemented with shaders. But the ping-pong scheme used is no longer needed given that CUDA allows the threads to randomly write in any global memory position. However, initial tests showed that the GPU occupancy was very low using it, decreasing the performance in CUDA. On the other hand, a direct implementation of Algorithm 3.1 is not possible since it depends on a dynamic structure to delete the invalid neighbors. Instead, a mixed version has been created using the extended ring structure of the shaders implementation and the basis of the iterative backtracking, while introducing a new $r_j.valid$ field that replaces the erasing of invalid neighbors.

3.6 Experiments and results

Three versions of the Local Delaunay Triangulation method were tested: the proposed CPU implementation of (Gopi et al., 2000), and the two discussed variations of GLT: using shaders and using CUDA. Additionally, tests using another existing method, the Poisson reconstruction (Kazhdan et al., 2006) are shown.

All tests in this work were performed on several known and public datasets, as well as in a few private models. The public models used in this

Algorithm 3.4 Local Delaunay Triangulation algorithm optimized for CUDA.[5]

```
1: function LOCAL_2D_DELAUNAY(p, m, R)
2:     min ← 1
3:     j ← 1
4:     while j < m do
5:         if r_j.valid and is_valid(p, ⟨r_j.prev, r_j, r_j.next⟩) = false then
6:             r_j.valid ← false
7:             r_j.prev.next ← r_j.next
8:             r_j.next.prev ← r_j.prev
9:             if j > min then
10:                j ← r_j.prev
11:            else
12:                min ← r_j.next
13:                j ← min
14:            end if
15:        else
16:            j ← j + 1
17:        end if
18:    end while
19: end function
```

work were obtained from the Stanford 3D Scanning Repository (Stanford Computer Graphics Laboratory), the Computer Graphics Group (Czech Technical University in Prague) and the Institute of Information Theory and Automation (Academy of Sciences of the Czech Republic). Private sets were designed in 3Ds Max, scanned in the Industrial Design Laboratory (TECNUN, University of Navarra). Patient's heads were obtained from the test dataset of the MAXIPLAN Project, developed by CEIT. For tests using the Poisson reconstruction, the MeshLab[6] package was employed.

The testing hardware consisted of an Intel Core2Duo of 3.00GHz, with 3GB of RAM and an NVIDIA 9850 with 1GB of VRAM.

[5]It can be seen that r_j, $r_j.prev$ and $r_j.next$ have been indistinctly used as points and indices. It has been done for simplicity. Actually, both $r_j.prev$ and $r_j.next$ are indices and the r_j neighbor is accessed from Q' using the j index.

[6]Mesh processing tool developed with the support of the 3D-CoForm project. `http://meshlab.sourceforge.net`

3.6.1 CPU vs Shaders vs CUDA

Experiments consisted in a comparison of the different GLT implementations running in the hardware specified above. The setup of the method was 32 points for the neighborhood size and a support radius similar to the sampling rate of each model (some of them were obtained empirically since it was not provided).

For each model, ten executions were done and times were averaged. Execution times are shown in Table 3.1 and in the sequence from Figure 3.16 to Figure 3.21 a time comparison of each phase using the three GLT implementations while Figure 3.22 shows total reconstruction times.

Model	Size	CPU	Shaders	CUDA
Noisy Foot	20K	1.55s	0.82s	0.44s
Stanford Bunny	35K	2.81s	1.47s	0.77s
Horse	48K	3.65s	1.84s	0.92s
Running Shoe	51K	4.07s	2.06s	1.06s
Collar	60K	4.71s	2.41s	1.28s
Patient's Head 1	148K	11.18s	5.82s	3.05s
Armadillo	172K	9.59s	6.95s	3.66s
Angel	237K	18.62s	10.60s	5.59s
EG'07 Dragon	240K	18.45s	9.68s	5.00s
Patient's Head 2	325K	24.46s	13.50s	6.38s
Hand	327K	24.29s	12.73s	6.39s
Stanford Dragon	437K	33.64s	17.37s	8.90s
Happy Buddha	543K	41.47s	21.46s	11.08s
Patient's Head 3	611K	46.19s	24.68s	13.25s
Ramorig	622K	46.80s	24.42s	12.54s
Blade	882K	69.30s	35.57s	18.37s

Table 3.1: Total reconstruction time using three different implementations. In general, Shaders implementation beats up CPU by a factor of 2 while CUDA offers the same boost with respect to the Shaders. These times are also shown in Figure 3.22.

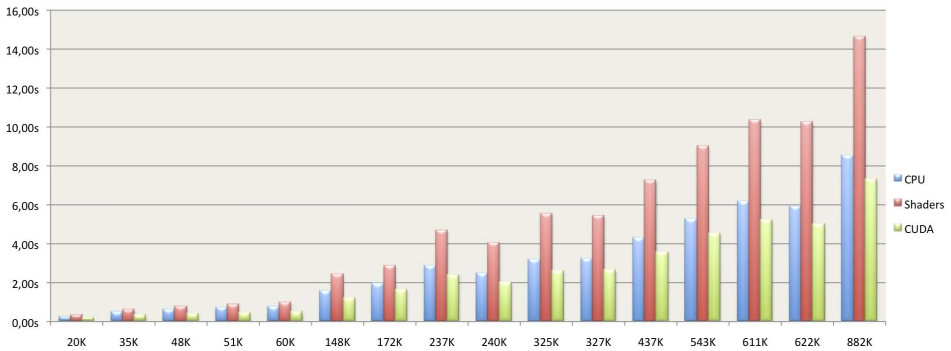


Figure 3.16: Time comparison of the computation of the neighborhoods.

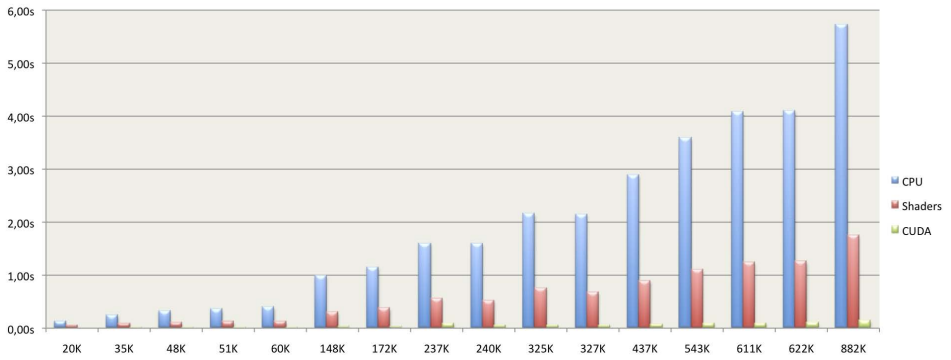


Figure 3.17: Time comparison of the normal estimation phase.

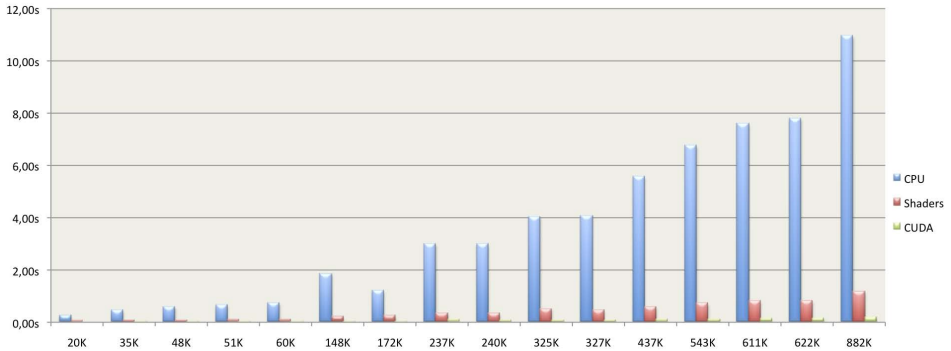


Figure 3.18: Time comparison of the angle computation and projection phase.

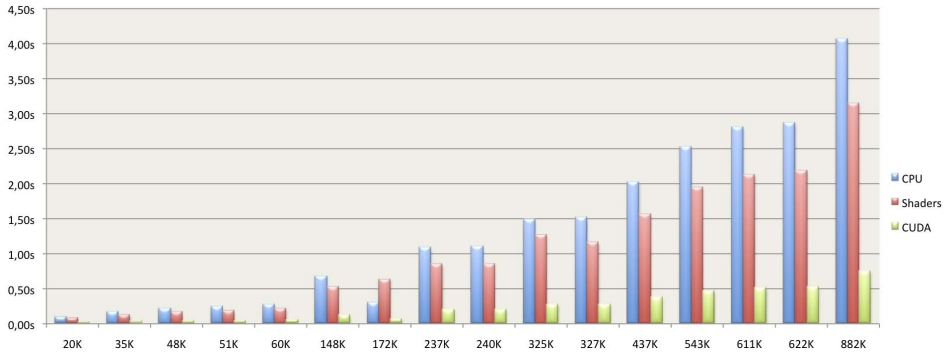


Figure 3.19: Time comparison of the sorting phase.

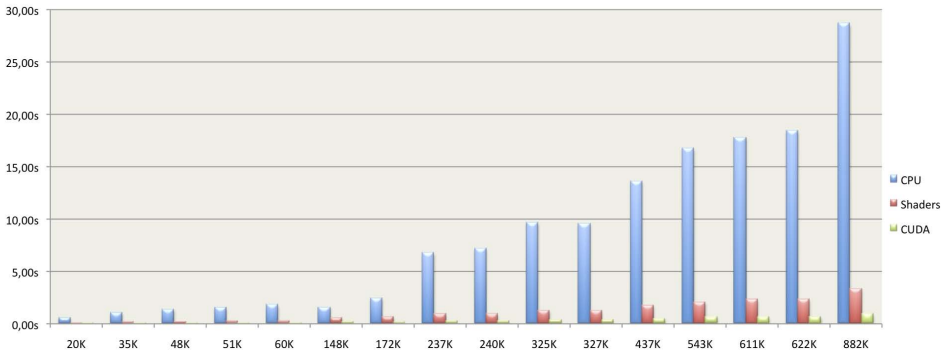


Figure 3.20: Time comparison of the local Delaunay validation.

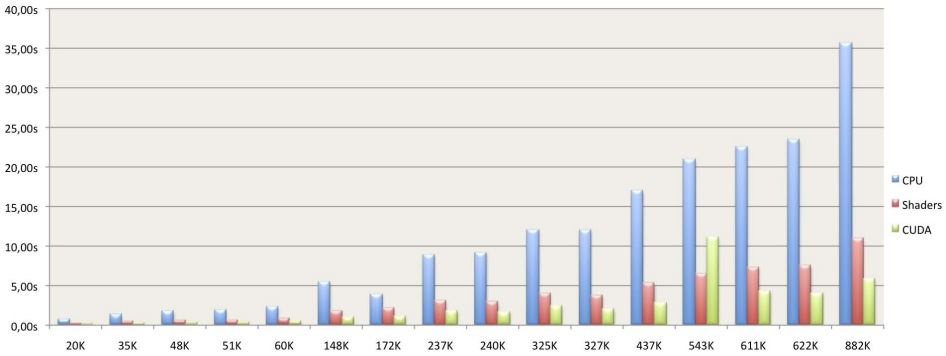


Figure 3.21: Time comparison of the local Delaunay triangulation (validation plus mesh creation).

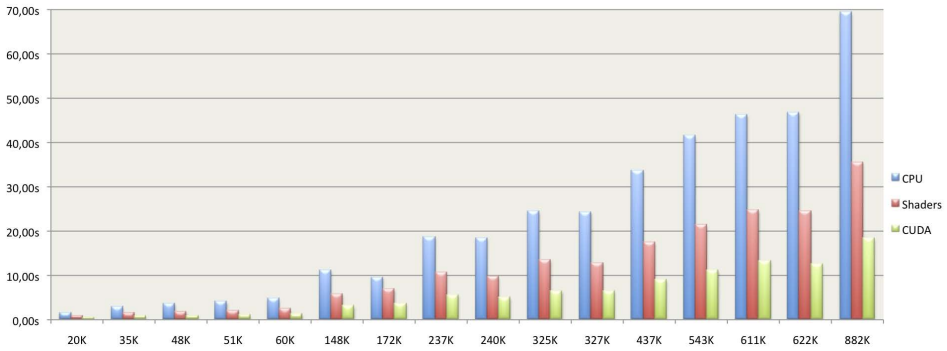


Figure 3.22: Comparison between the different proposed reconstruction methods.

3.6.2 Reconstruction results

In this section a few reconstructed models from Table 3.1 are shown. For a more complete list of images, please refer to Section 5.2, where final results of this work are exposed. Please, note that all the images in this work were rendered with flat shading to emphasize the actual mesh quality.

Figures 3.23 and 3.24 show the behavior of the GLT under well sampled meshes (in this case, the model is obtained from a previously reconstructed dataset). Regarding tests of GLT from another sources, Figure 3.25 is the reconstruction of a synthetic model and Figure 3.26 is from a rough scanning of a shoe.

This last figure shows a hole in the top zone of the model; holes and overlapping triangles may be generated if four or more points lie on the same circumference. In such case, the Delaunay triangulation is not unique.

Finally, in Figure 3.27 and Figure 3.28, the Blade model is shown. This is the biggest model shown in Table 3.1, with 882954 points. Although a bigger model (the Asian Dragon) was reconstructed in these experiments, it was removed from the graphs due to the difference in the order of magnitude of the times (its times are shown in Figure 3.29).

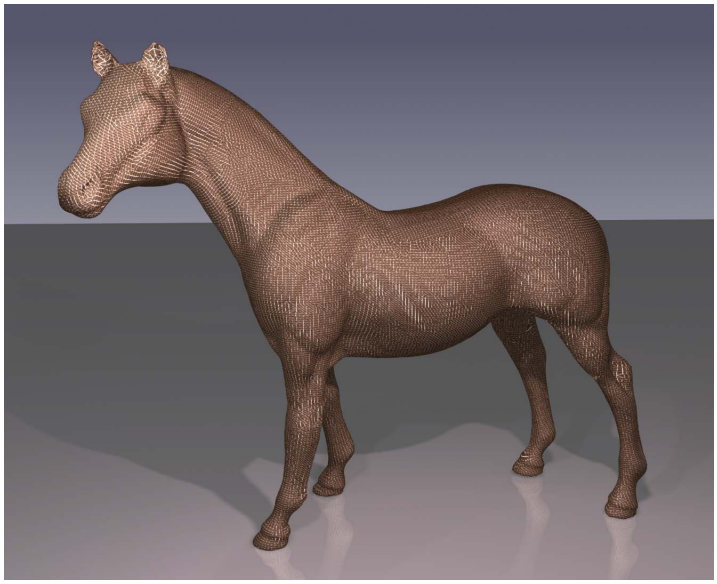


Figure 3.23: Horse model rendered along with its wireframe.

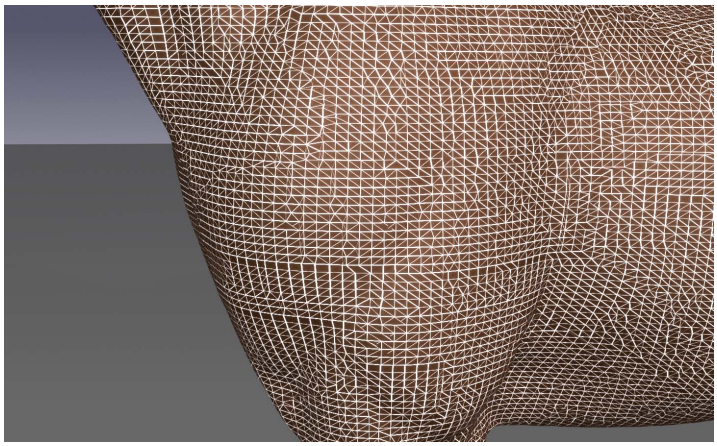


Figure 3.24: Neck and chest detail of the Horse model.

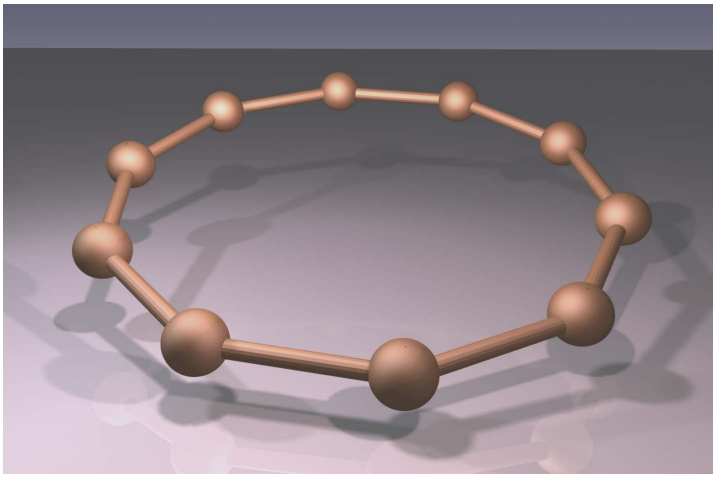


Figure 3.25: A synthetic model.

Figure 3.26: Running shoe along with its wireframe. Note that, even when the mesh quality is pretty good, some holes may be created due to bad sampling as the one visible at the top of the shoe.

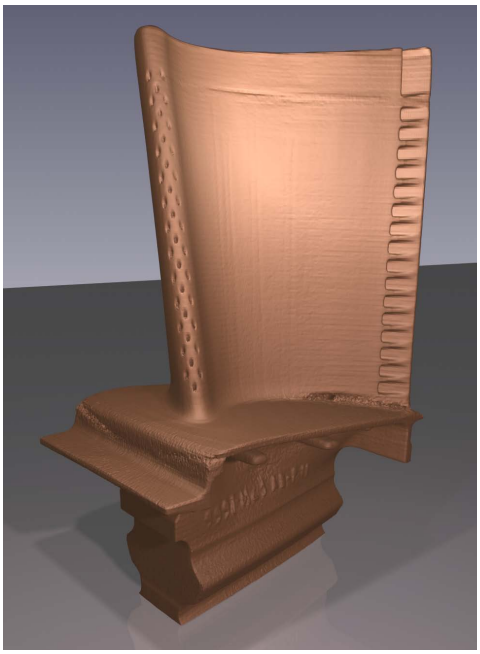


Figure 3.27: Blade model.

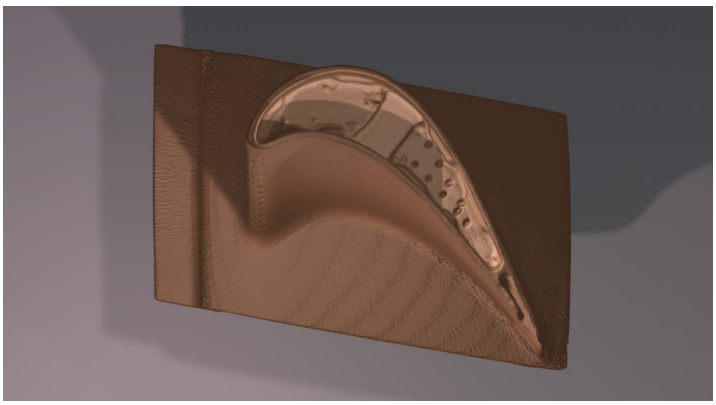


Figure 3.28: Top view of the Blade model. Small structures, as holes, were satisfactorily reconstructed.

3.6.3 Big models

In this section, reconstruction results for three models with more than a million points are presented. Figure 3.29 details the time consumption for each phase during the reconstruction of the Asian Dragon model. Table 3.2 summarizes the total reconstruction time of each model using the CUDA implementation. It is remarkable to say that, even when the Welsh Dragon model has more than a million points, it is not quite representative given that its points are distributed in a highly uniform way.

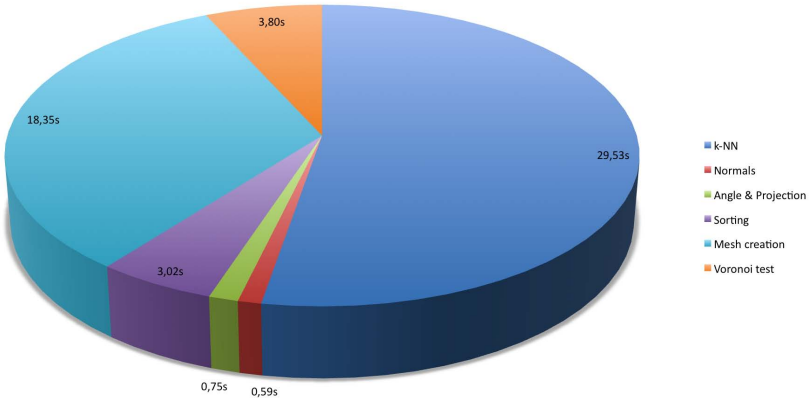


Figure 3.29: Overall view of time consumption in the reconstruction of the Asian Dragon model (3609K points). It can be seen how more than 75% of the time is spent in CPU phases (neighborhoods computation and mesh assembling). Normal propagation is not shown in this chart. The total reconstruction time for this model, including the propagation, is 73.25 seconds.

Model	Size	CUDA	Normal propagation
Welsh Dragon	1105K	19.48s	4.92s
Asian Dragon	3609K	56.04s	16.15s
Thai Statue	4999K	78.16s	—

Table 3.2: Total reconstruction time of three big models using the CUDA implementation. Due to memory limitations, the normal propagation of the Thai Statue model was not possible.

Studying the normal propagation in big models, it has been found that

the memory consumption is quite significative given the data structures required for it. The Thai Statue model is a good example of it, as it can be seen in Table 3.2, where the memory employed exceeds the 2GB, the limit for a single process in the operating system used (Windows XP SP3 32bits).

3.6.4 Comparison with an approximating method

In addition to the comparison of the presented method, GLT, with the original Lower Dimensional Localized Delaunay triangulation of (Gopi et al., 2000), the Poisson reconstruction (Kazhdan et al., 2006) has been employed to reconstruct some of the models previously listed. Table 3.3 summarizes these results, and a graphical comparison between GLT and Poisson is shown from Figure 3.30 to Figure 3.33.

Model	Size	Poisson	CUDA GLT
Noisy Foot	20K	4.84s	0.44s
Angel	237K	32.38s	5.59s
Happy Buddha	543K	58.74s	11.08s
Blade	882K	135.66s	18.37s
Asian Dragon	3609K	181.82s	73.25s

Table 3.3: Total reconstruction time using the Poisson reconstruction tool from MeshLab. The Noisy Foot, the Blade and the Asian Dragon employed an octree of depth 10, while the Angel and the Happy Buddha a depth of 9.

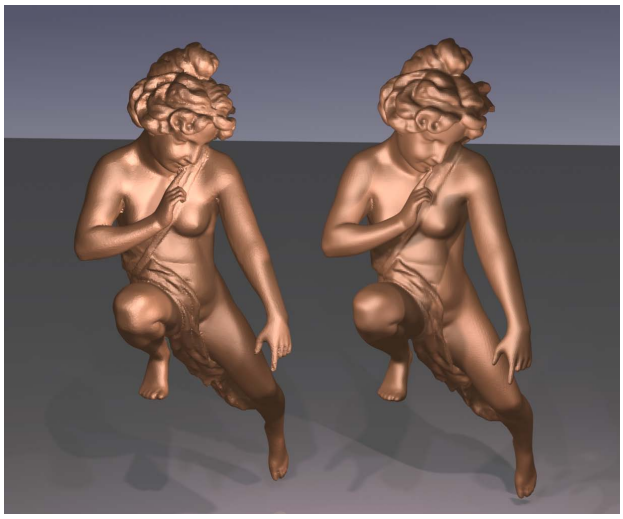


Figure 3.30: Comparison with the Poisson reconstruction - Angel. The small details, like foot toes, are better recovered by an interpolating technique like GLT (left) than by an approximating one, as the Poisson reconstruction (right).

Figure 3.31: Comparison with the Poisson reconstruction - Happy Buddha. The left image corresponds with the reconstruction using the GLT method, and the right image used the Poisson method.

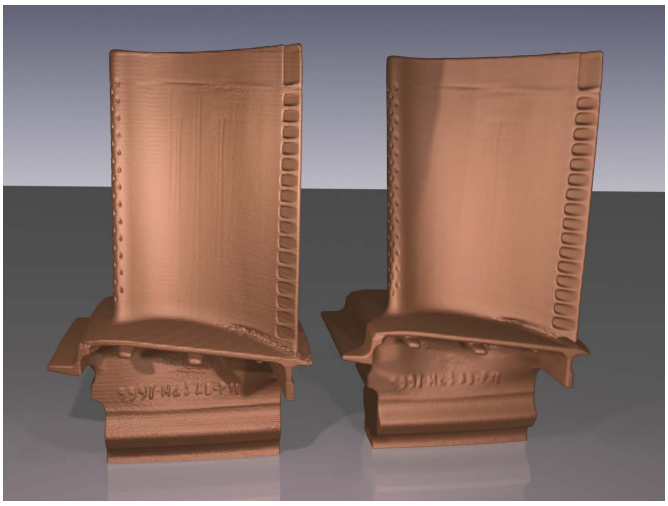


Figure 3.32: Comparison with the Poisson reconstruction - Blade. The serial number is more readable in the surface generated with the GLT (left), but the Poisson method (right) is less prone to topological issues like the union of the blade with the base, where sampling is ambiguous.

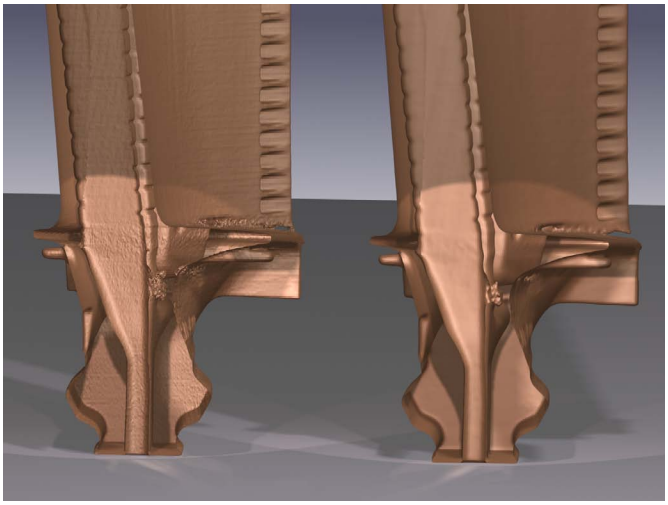


Figure 3.33: Comparison with the Poisson reconstruction - Interior detail of the Blade model. The left image was reconstructed using the GLT method, and the right image using the Poisson reconstruction.

3.6.5 Application in the medical field

Figure 3.34 shows the results of the proposed GLT method in the medical application MAXIPLAN (a project developed by CEIT). GLT is used to reconstruct the skin of patients' heads. The points are extracted from the segmentation of CT images (*computerized tomographies*). In this case, the classical Marching Cubes method is not enough since the mesh should be deformed using a mass-spring model from the segmented points (San Vicente et al., 2009). Using an approximating technique, such as direct iso-surface extraction, may create non box-aligned points, a requirement of the simulator. Creating a morpher that links the mass-spring model with the extracted iso-surface would be unnecessarily expensive compared with a direct interpolating reconstruction.

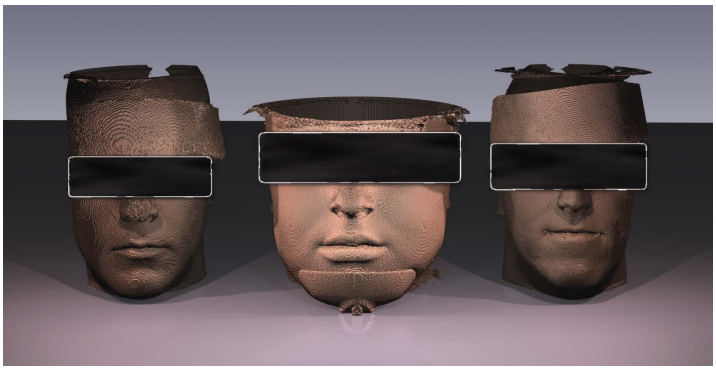


Figure 3.34: From left to right: Patient's Head 1, Patient's Head 2 and Patient's Head 3 datasets. As points are extracted from a volumetric source (CT images), the reconstruction presents this stair-like appearance.

3.6.6 Application in cultural heritage

The models studied in previous sections could be classified into an "objects" category, i.e., models where the scanner goes around the target and it is usually near the scanned object. This is a valid scenario if the target object is an sculpture, a vase, an anatomical region, and similars; but if the desired surface comes from an open area such as a room or a garden, or where the scanner is far from the target, like a building, then the reconstruction

becomes different since the points cloud tends to be more sparser and it is far from being uniformly distributed.

This differentiation is necessary before exposing results of the reconstruction method applied to cultural heritage, since it presents data sets of both kinds. This section presents results of applying the proposed method to three different models from cultural heritage projects[7]. Figure 3.35, for example, comes from a scanned Trophy, belonging to the "object" type, while the Figure 3.36 is a Roman Theatre, it is, an open space. It can be seen that GLT fails to properly reconstruct the Theatre model, given the points, due to the nature of the scanning process used, have a very uneven distribution, as well as too much missing areas. This sparse and non-uniform structure makes impossible to determine a satisfactory sampling distance: near the scanner points are very dense while it becomes sparser as it goes away the centre. Finally, the Tower shown in Figure 3.37 could be properly reconstructed although it comes from a similar scanning process, because it provided four different views of the tower that mutually compensate the lack of information (these view were aligned as a preprocessing step). Also, this model is more simple than the Theatre, in terms of the sampling distance, so it is possible to find a maximum value that do not generate false positives (points not belonging to the real neighborhood of a point).

Figure 3.35: Reconstruction of a Trophy.

[7]Models courtesy of Prof. Juan Carlos Torres, University of Granada.

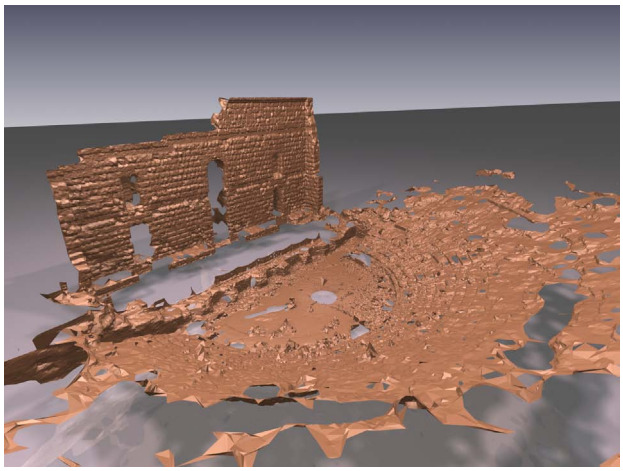


Figure 3.36: This Theatre has not been succesfully reconstructed due to the sparseness of the points of the auditorium zone. Only the wall could be extracted correctly.

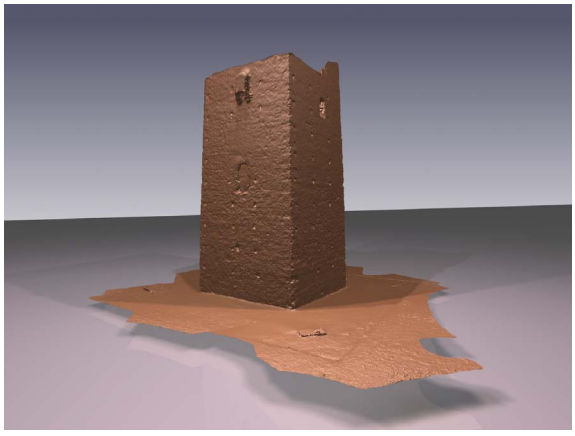


Figure 3.37: Reconstruction of a Tower. The points of the different views of the scanner were aligned before the reconstruction.

3.7 Discussion

In this chapter, a local Delaunay triangulation for surface reconstruction has been presented, proving (both mathematically and empirically) that local triangulations are good candidates for parallelization and that they are suitable for GPU implementation. It has been shown how the proposed GLT method can efficiently reconstruct a surface from a set of points, showing an average improvement of two or three times over the state-of-the-art methods.

Other main contributions of this chapter include changes to the normal estimation phase using Weighted PCA, the improvement of the angle computation phase using a modification of the technique proposed by (Crossno and Angel, 1999) and the development of a multi-list sorting algorithm based on the Bitonic Merge Sort.

Regarding the surface generated, as GLT interpolates the input points it depends on their quality. Once the surface has been generated, a post-processing operator such as mesh smoothers can be applied to obtain a softer surface. Also, a preprocessing step can be applied to improve the points before the reconstruction.

The withdraws of the method proposed are its dependence to some uniformity in points distribution as well as a proper points density along zones with higher detail. For example, holes may appear if there are not enough points, or crossing triangles may be created if the density is not enough for the level of detail. Also, it was shown that, although the method itself can handle large datasets, the normal propagation stage still represents an issue to be solved.

Among the application of studied techniques in other fields, the divide-and-conquer nature local triangulations makes them appropriate for mesh streaming. For example, a model is partitioned using a uniform grid, and each cell or block is sent individually; each block that already has its neighbors received can be safely reconstructed.

Chapter 4

Parallel Weighted Locally Optimal Projection

The simplification of anything
is always sensational
Gilbert Chesterton

A synthesis of this chapter has been published in:

Buchart, C., Amundarain, A., and Borro, D. 3-D surface geometry and reconstruction: Developing concepts and applications, chapter Hybrid surface reconstruction through points consolidation. IGI Global. 2011. (Sent and under revision).

4.1 Previous works

There are some interesting works regarding a preprocess step before reconstruction. As mentioned before, one of the major problems concerning all the methods is the quality of the input data. What preprocessing steps try to do is to improve such quality, removing noise and generating a more uniform set of points. It is especially convenient for interpolating methods, due to their dependence on the input data.

(Alexa et al., 2003) study the direct visualization of point set surfaces, defining a projection operator that projects the points near the data set onto the surface, and then constructing a moving least-square surface (Levin, 2003).

In more recent years, (Lipman et al., 2007b) present a parameterization-free projection operator called LOP (Locally Optimal Projection) to deal with outliers, and (Huang et al., 2009) developed a new WLOP (Weighted LOP), improving the previous operator with a more uniform distribution of points and a clearer convergence. This operator is also used to boost up normal estimation and propagation. One of the main problems of these operators is that they are very expensive in terms of execution time when compared to the time required for the most common reconstruction algorithms.

Finally, (Zhang et al., 2010) propose a meshless parameterization and de-noising of a set of points by rigid alignment of locally flattered proximity graphs.

In this chapter, the works of (Lipman et al., 2007b) and (Huang et al., 2009) are studied and a parallel implementation is presented in order to increase the performance of the operator.

4.1.1 Locally Optimal Projection Operator

Given a set of points $P = \{p_j\}, j \in J = \{1, 2, \ldots, |P|\}$, an initial guess $X^0 = \{x_i\}, i \in I = \left\{1, 2, \ldots, \left|X^0\right|\right\}$, a repulsion parameter $\mu \in [0, 1/2)$ and support radius h, the *Locally Optimal Projection* (LOP) algorithm presented by (Lipman et al., 2007a) computes the projection of X onto P in an iterative way[1]. For each iteration $k \in 1, 2, 3 \ldots$ it defines the projection for the i-th point of X:

$$x_i^{(k)} = \frac{\sum_{j \in J} p_j \alpha_{ij}^{(k-1)}}{\sum_{j \in J} \alpha_{ij}^{(k-1)}} + \mu \frac{\sum_{i' \in I \setminus \{i\}} \left(x_i^{(k-1)} - x_{i'}^{(k-1)} \right) \beta_{ii'}^{(k-1)}}{\sum_{i' \in I \setminus \{i\}} \beta_{ii'}^{(k-1)}}, i \in I \tag{4.1}$$

where

$$\alpha_{ij}^{(k)} = \frac{\theta\left(\left\|x_i^{(k)} - p_j\right\|\right)}{\left\|x_i^{(k)} - p_j\right\|} \tag{4.2}$$

$$\beta_{ii'}^{(k)} = \frac{\theta\left(\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|\right)}{\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|} \left|\frac{\partial\eta}{\partial r}\left(\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|\right)\right|, i' \in I \backslash \{i\} \tag{4.3}$$

$\theta(r)$ is a weight function with compact support radius h, that defines the area of influence of the operator; it can be seen as a Gaussian filter applied to the distance function from $x_i^{(k)}$. $\eta(r)$ is a repulsion function that penalizes points going too close to each other. The support radius h is usually set as $h = 4\sqrt{d_{BB}/|P|}$, where d_{BB} is the diagonal length of the bounding box of P (Huang et al., 2009). In the work of (Lipman et al., 2007a), such functions are defined as

$$\theta(r) = e^{-r^2/(h/4)^2} \tag{4.4}$$

$$\eta(r) = \frac{1}{3r^3} \tag{4.5}$$

4.1.2 Weighted Locally Optimal Projection Operator

In their extension of LOP, (Huang et al., 2009) comment that the repulsion force may decrease too fast in many situations, contributing to a non-uniform distribution of points in X. They then propose a new repulsion term "which decreases more gently and penalizes more at larger r, yielding both a better convergence and a more locally regular point distribution" (Huang et al., 2009, p. 3). They also set, empirically, $\mu = 0.45$. The new repulsion function is

$$\eta(r) = -r \tag{4.6}$$

which also leads to a simplified β term

$$\beta_{ii'}^{(k)} = \frac{\theta\left(\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|\right)}{\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|}, i' \in I \backslash \{i\} \tag{4.7}$$

[1] In the whole of this work, $\|\cdot\|$ represents the Euclidean norm of a vector

To improve even more the distribution of points, it incorporates *locally adaptive density weights* v_j for each $p_j \in P$, and $w_i^{(k)}$ for each $x_i^{(k)} \in X^{(k)}$:

$$v_j = 1 + \sum_{j' \in J \setminus \{\mathrm{j}\}} \theta\left(\left\|p_j - p_{j'}\right\|\right) \tag{4.8}$$

$$w_i^{(k)} = 1 + \sum_{i' \in I \setminus \{\mathrm{i}\}} \theta\left(\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|\right) \tag{4.9}$$

In this way, the *weighted LOP* (WLOP) operator (Huang et al., 2009) is defined as

$$x_i^{(k)} = \frac{\sum_{j \in J} p_j \hat{\alpha}_{ij}^{(k-1)}}{\sum_{j \in J} \hat{\alpha}_{ij}^{(k-1)}} + \mu \frac{\sum_{i' \in I \setminus \{i\}} \left(x_i^{(k-1)} - x_{i'}^{(k-1)}\right) \hat{\beta}_{ii'}^{(k-1)}}{\sum_{i' \in I \setminus \{i\}} \hat{\beta}_{ii'}^{(k-1)}}, i \in I \tag{4.10}$$

where

$$\hat{\alpha}_{ij}^{(k)} = \frac{\frac{\theta\left(\left\|x_i^{(k)} - p_j\right\|\right)}{\left\|x_i^{(k)} - p_j\right\|}}{v_j} \tag{4.11}$$

$$\hat{\beta}_{ii'}^{(k)} = \frac{\theta\left(\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|\right)}{\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|} w_i^{(k)}, i' \in I \setminus \{i\} \tag{4.12}$$

4.2 Parallel WLOP

Once that the bases of the LOP and WLOP operators have been exposed, this section will discuss their performance as well as describe a new operator proposed to dramatically reduce the time needed for the projection.

The main issue of both LOP and WLOP is their execution time. Some straightforward optimizations (probably present in the original

implementations, but no details are given in the publications) may include computing squared distances instead of the actual distance; and avoiding evaluating the weight function $\theta(r)$ on the full domain, but only where its weight does not vanish, i.e., evaluate the function in the h-vicinity of each point (the main issue with it is the computation of such vicinity).

Even with these optimizations, WLOP is especially slow because it must evaluate the local density weights in each iteration, which means having to recompute the h-vicinity of each point several times. In this work, it has seen that if a good initial guess X^0 is provided (see below for further explanation), the neighborhoods variations are small since the points tend to move just a little (weights do change between iterations). The points that fall into the support of the local density weights of a point have been called the *local support neighborhood* (LSN) of such a point. In this thesis it is proposed to preload an *extended LSN* that includes all the points that may fall in the support radius h during the iterations of the operator (see Figure 4.1). Even when some neighbors in the extended LSN may fall outside the support radius (thus computing unnecessarily the weight function), it is worth the time gained by avoiding the costly neighborhood computation. In tests performed, an extended radius of $2h$ works well in most of the cases. Also, bounding the maximum number of neighbors is important given the fact that the CUDA matrices are static; in the tests, 64 points of vicinity were enough to correctly project most of the models, while 128 were used in noisy models.

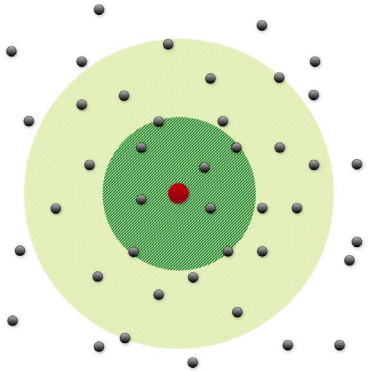


Figure 4.1: Local support neighborhood (dark area) vs. the extended LSN (lighter area).

The new local weight densities may be expressed in function of their

local support neighborhoods as

$$\bar{v}_j^{(k)} = 1 + \sum_{j' \in \bar{J}_j} \theta\left(\left\|p_j - p_{j'}\right\|\right) \tag{4.13}$$

$$\bar{w}_i^{(k)} = 1 + \sum_{i' \in \bar{I}_i} \theta\left(\left\|x_i^{(k)} - x_{i'}^{(k)}\right\|\right) \tag{4.14}$$

where $\bar{I}_i$ defines the precomputed neighborhood of x_i, and $\bar{J}_j$ is the precomputed neighborhood of p_j. To simplify the notation, $\bar{\alpha}_{ij}^{(k)}$ and $\bar{\beta}_{ii'}^{(k)}$ will denote the $\hat{\alpha}_{ij}^{(k)}$ and $\hat{\beta}_{ii'}^{(k)}$ terms with precomputed neighborhoods. In the same way, neighborhoods in Equation 4.10 are replaced by their precomputed counterparts. Experiments reveal that simply avoiding the expensive neighborhood computation with the previously exposed technique, it is possible to obtain speed up ranges around 50% in the execution time in the CPU. This variation of the WLOP operator has been called *extended WLOP*, or eWLOP for short.

The only problem so far is the initial guess X^0. However, a good and easy-to-compute initial set can be obtained by the spatial subdivision of P, taking X^0 as the centroid of the points inside each cell. This also follows the common sense that a uniform data set may be crudely represented by a grid-based subsampling of the point set.

At this point, it raises the question about parallelization. The less parallel part of the algorithm is neighborhoods computation, but since all these calculations have been moved to a preprocessing stage, it must be done just once; the proposed implementation has used the ANN library of (Mount and Arya, 2010) to perform the k-nearest neighbors (k-NN) computation[2]. The rest of the algorithm is data independent in the same iteration k, i.e., a projected point $x_i^{(k)}$ only depends on previously projected points $x_{i'}^{(k-1)}$ and the original points P.

Based on the preceding analysis, a *Parallel WLOP* (PWLOP) that performs on average 25 times faster than the original WLOP is proposed. PWLOP uses graphical hardware acceleration by implementing it on the CUDA architecture. It is worth mentioning that, although the algorithm

[2]When talking about neighborhoods, k is the number of neighbors, not the current iteration, which is represented between parenthesis: (k).

is dependent between consecutive iterations, there is no need to do CPU processing after an iteration has finished, so the operator can execute completely in the GPU before downloads the projected points into the main memory. Additionally, and for testing purposes, the original WLOP has been modified introducing the extended LSN, but without the parallelization plus of PWLOP (the eWLOP).

4.2.1 Implementation details

The PWLOP operator is susceptible to many optimizations that help to speed up the method and to reduce the number of GPU registers used, thereby incrementing the GPU occupancy and increasing performance. Some of these optimizations include:

1. Terms like $\hat{\beta}_{ii'}^{(k)}$, where a norm appears several times; this norm may be computed only once.
2. Expressions such as $\dfrac{\sum_{j \in J} p_j \hat{\alpha}_{ij}^{(k-1)}}{\sum_{j \in J} \hat{\alpha}_{ij}^{(k-1)}}$ can be computed once.
3. The weight function $\theta(r) = e^{-r^2/(h/4)^2}$ may be expressed as $\theta(r) = e^{-\hat{h}r^2}$, where $\hat{h} = (4/h)^2$ is precomputed. Moreover, the distance r may be used in its squared form, avoiding the costly squared root (in this sense, for example, many k-NN algorithms return the squared distances).
4. Finally, there are several terms that may produce a division-by-zero when the distance between two points vanishes (more likely between X and P). To solve this issue, and to avoid inconvenient branches, since the divisor is always positive (distances) it is common to add a small value to it.

4.3 Experiments and results

Figure 4.2 shows a comparison of the execution time of the WLOP, the extended WLOP and PWLOP for a simple model.

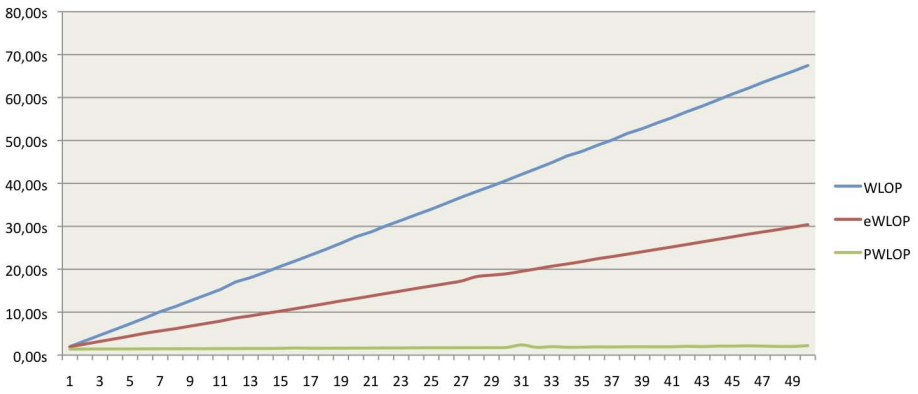
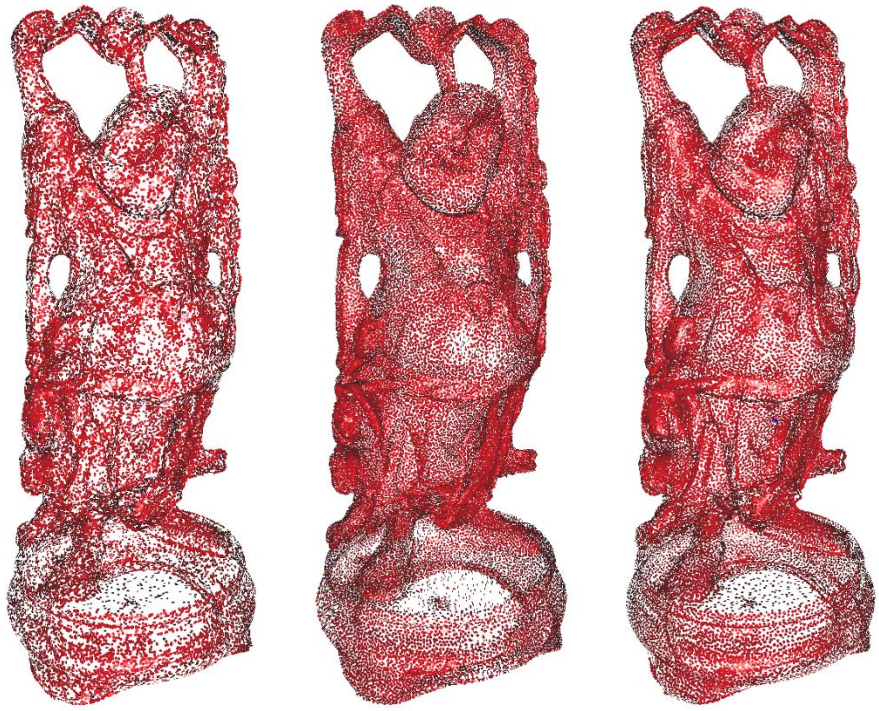


Figure 4.2: Speed comparison between WLOP, eWLOP and PWLOP in the projection of 22K points from the Stanford Bunny (35K points), using a different number of iterations.

In these experiments, the local support neighborhood of the eWLOP and PWLOP was set to 32 points. For WLOP tests and rendering of results, the Points Consolidation API was employed (it is publically available in (Huang et al., 2010)).

Figure 4.3 and Figure 4.4 show projection results for the Happy Buddha model. In Figure 4.3, the initial guess is a random subsampling of the model and the number of iterations is set to 50 for both methods. As the initial guess is crude, the PWLOP operator fails to converge since the local weight densities were calculated over wrong neighborhoods. On the other hand, Figure 4.4 shows a more accurate initial guess, extracted from the grid subdivision described above. As the initial guess is near a solution, the number of iterations was reduced to only 10. It can be seen how the results from WLOP and PWLOP are almost identical, while PWLOP executes four times faster than WLOP.

As an additional test, the speed comparison seen in Figure 4.4 was extended to include the eWLOP and increasing the number of iterations. Results can be seen in Table 4.1. It shows how PWLOP can execute 50 times more iterations than the original WLOP in the same period of time under equal circumstances (the initial guess set). Also, the average speed up offered by PWLOP against the extended WLOP is of 25 times, ignoring the common neighborhood computation time (6.55s).



(a) Random $|X^0|$=54K (b) WLOP (91.75s) (c) PWLOP (8.9s)

Figure 4.3: Projection of the Happy Buddha (543K) with $h = 0.2$ and 50 iterations.

A full speed test over several models is shown in Figures 4.5 and 4.6. There it can be seen how PWLOP is an order of magnitude faster in the worst case. It was found that more than a half of the execution time of PWLOP is spent in the computation of the extended LSN (see Figure 4.7), very reasonable since it is a heavy task and it is done in the CPU instead of the GPU as with the rest of the algorithms.

Concerning the projection of noisy data sets, such analysis is deferred to the next section, since it will be easy to see the changes between original and projected datasets when they are reconstructed.

Finally, as described along this chapter, the validity of the PWLOP operator comes from the use of an eLSN for each point. Figure 4.3 illustrated what happens if this premise is not valid. In the following experiment, a

(a) Grid $|X^0|$=55K (b) WLOP (24.63s) (c) PWLOP (6.87s)

Figure 4.4: Projection of the Happy Buddha (543K) with $h = 0.2$ and 10 iterations.

random set of points is used as the initial set and, instead of using the eLSN and computing it only in the first iteration, the LSN is used for several iterations and it is marked to be recomputed each 10 iterations. This leads to a midpoint approach between the original WLOP, that computes the LSN in each iteration, and the PWLOP, that uses only the first eLSN. Results of this experiment are shown in Figure 4.8. As can be seen, this approach leads to a slightly better projection than WLOP. On the other hand, and as expected, the required time is increased considerably compared with PWLOP, although it remains inferior to the one employed by WLOP.

Iterations	WLOP	eWLOP	PWLOP
10	24.63s	14.59s	6.87s
100	101.79s	87.53s	9.58s
500	963.11s	410.36s	22.69s

Table 4.1: Speed comparison between WLOP, eWLOP and PWLOP when projecting the Happy Buddha model (543K), $\left|X^0\right|$=55K, $h = 0.2$.

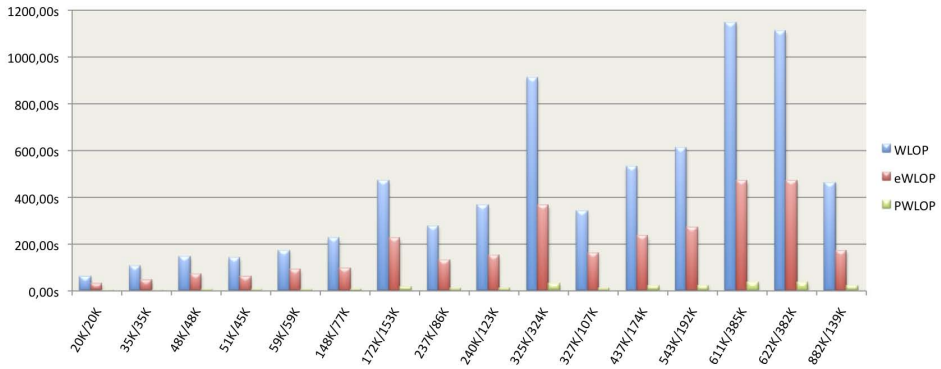


Figure 4.5: Projection of different data sets using WLOP, the eWLOP and the proposed PWLOP operators. For all the data sets, 50 iterations were used as well as LSN = 64.

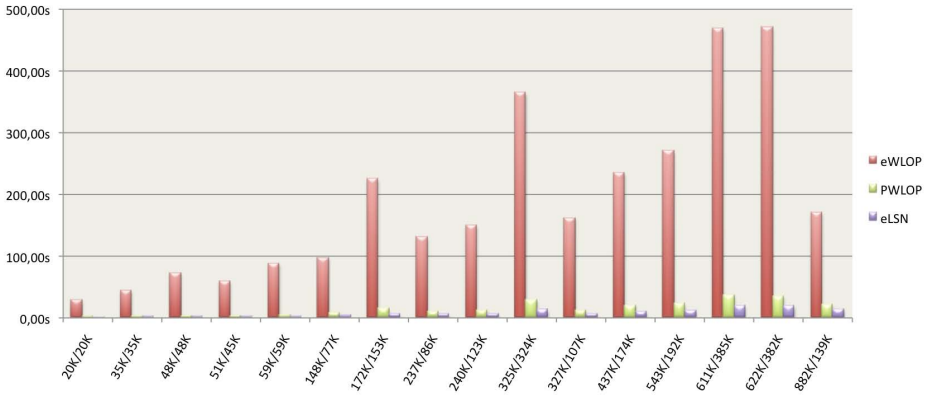


Figure 4.6: The Extended LSN computation times are common for both the eWLOP and PWLOP. The test configuration is the same of Figure 4.5.

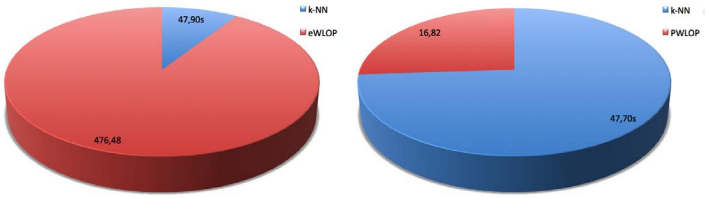


Figure 4.7: The computation of the extended local support neighborhood is the same for the eWLOP (left) and PWLOP (right). The speed boost obtained by the parallelization can be seen in this figure. The model is the Asian Dragon (3609K) projected onto 325K points. Parameters are set to 50 iterations, LSN = 64 and $h = 0.02$.

(a) WLOP (91.75s) (b) PWLOP (8.9s) (c) PWLOP (41.82s)

Figure 4.8: Projection of the Happy Buddha (543K) with $h = 0.2$ and 50 iterations, resetting the initial data set each 10 iterations.

4.4 Discussion

In this chapter, a projection operator for points preprocessing, WLOP, that includes noise removal and redistribution of input points, is studied, as well as a new operator, PWLOP, with parallel capabilities is presented. This new operator has been designed to be implemented in the GPU using CUDA, because WLOP's limitations made it unsuitable for parallel architectures. PWLOP is an order of magnitude faster than the previous WLOP operator, while keeping the accuracy of the original operator.

Chapter 5

Hybrid surface reconstruction: PWLOP + GLT

–[*Luke*] *I can't believe it*
–[*Yoda*] *That is why you fail*

A synthesis of this chapter has been published in

Buchart, C., Amundarain, A., and Borro, D. 3-D surface geometry and reconstruction: Developing concepts and applications, chapter Hybrid surface reconstruction through points consolidation. IGI Global. 2011. (Sent and under revision).

5.1 Improving the input data set through points consolidation

Although the local nature of the method presented makes it very favorable to parallelization, it shares one of the biggest disadvantages of many reconstructions based on Delaunay triangulations: their sensibility to noise and outliers. In this sense, the points projection operators presented in Section 4.2 help to solve this issue by creating a more uniform set of points, improving the quality of the reconstruction.

Moreover, the operator PWLOP developed in this work has execution times that fall in the same order of magnitude of GLT for even medium models, making it a reasonable choice for a preprocessing stage, without considerably increasing the total time required to extract the surface.

This hybrid reconstruction scheme allows the GLT to acquire some of the characteristics of approximating methods: more tolerance to noise and outliers, as well as being more robust against frequency changes.

5.2 Results

To test the integration of the PWLOP operator with GLT, the following methodology was followed: for each model, the initial guess of the projection contains about a third of the number of input points. All the models were then projected using the default support radius h, 50 iterations and 64 points for the local support neighborhood (except for the Noisy Foot model, where a 128 points neighborhood and 100 iterations were used). Regarding GLT, the reconstruction radius was fixed to $h/2$.

Several examples of the results of the hybrid reconstruction method proposed are shown in Figures 5.1 to Figure 5.4, while Figure 5.8 shows a full comparison of all the experiments done.

Figure 5.5 compares with more detail the quality of the mesh between the original Hand model and the projected one. In this case, the projection operator produces the same effect of a points collapsing strategy, improving the quality of final triangles.

The effects of the PWLOP operator in noisy data sets can be seen in Figure 5.6. The projection improves the quality of the mesh, although not as much as an approximating reconstruction can, as shown in Figure 5.7.

In this line, comparing the reconstruction time employed by the hybrid reconstruction and an approximating method (Poisson reconstruction), Table 5.1 lists the results of the experiments presented in Section 3.6.4. The total time of the combined PWLOP + GLT approach is lesser than the need by the Poisson method. An interesting case is the Blade model, where the presence of many fine details makes necessary a higher octree depth in the configuration of the Poisson method. In this sense, GLT is invariant to these issues, given a proper sampling rate is set. In the original

work of (Huang et al., 2009), Poisson reconstruction is also improved using a consolidation of points based on the WLOP operator; so, regarding the reconstruction time, it is also increased in a similar proportion, but, in anyway, the PWLOP operator presented in this work has demonstrated to be much faster than WLOP, so it can be also used for those cases where GLT fails to create a proper mesh (for example, very highly clouds of points, or if a watertight surface is needed).

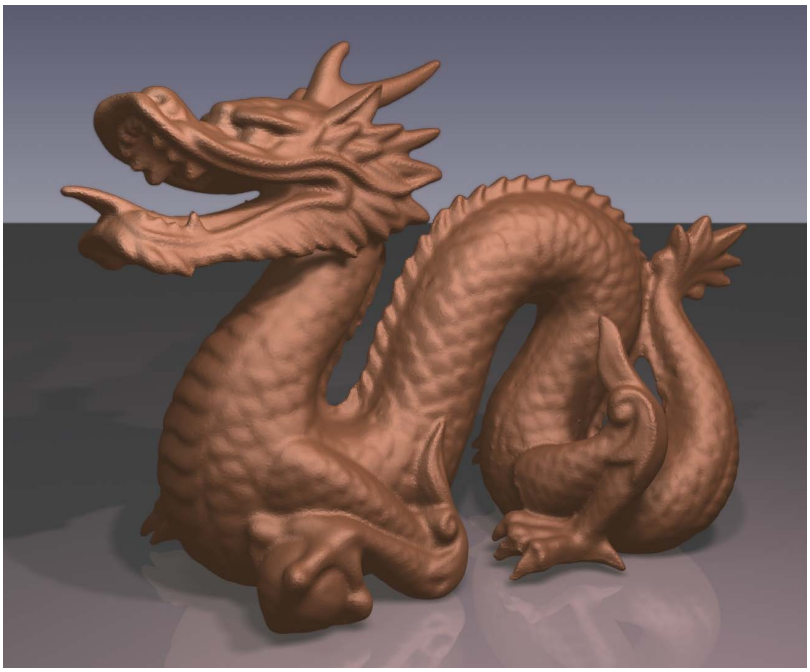


Figure 5.1: Stanford Dragon.

Figure 5.2: Asian Dragon.

Figure 5.3: Happy Buddha.

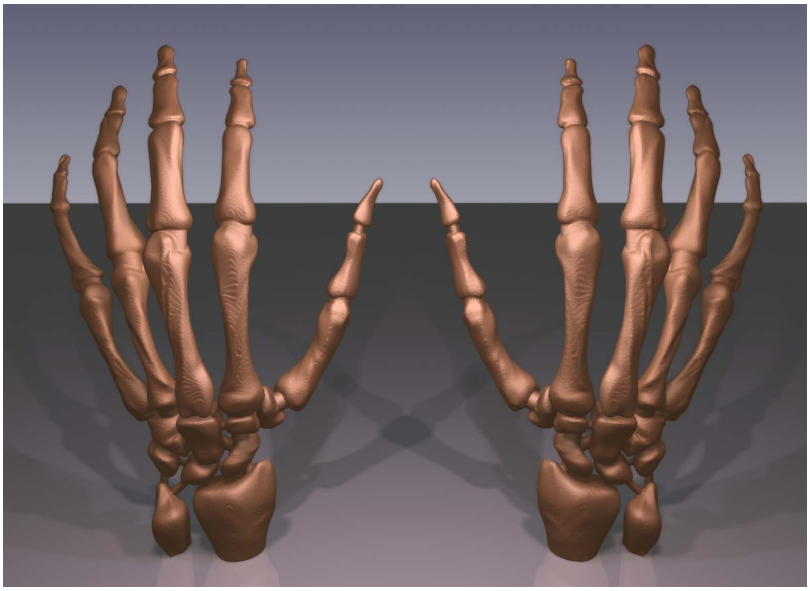


Figure 5.4: Hand model reconstruction comparison. (left) Original model with 327K points reconstructed using only the GLT method. (right) Reconstruction of a 107K projected points using the hybrid method.

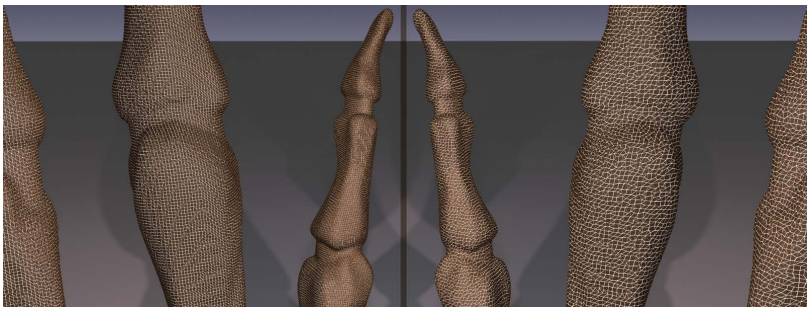


Figure 5.5: Hand model detail comparison. (left) Original data set with 327K points. (right) Projected set with 107K points.

Figure 5.6: Noisy Foot with 20K points. From left to right: original data set; self-projected using PWLOP with LSN=64; self-projected using PWLOP with LSN=128. Bigger neighborhoods are necessary because in noisy data sets the points move more between iterations. It can be seen how the nails are still recognizable. For all the images the number of PWLOP iterations was set to 100.

Model	Size	Projected Size (Hybrid)	Poisson	Hybrid
Angel	237K	86K	32.38s	11.73s
Happy Buddha	543K	192K	58.74s	25.63s
Blade	882K	139K	135.66s	22.97s
Asian Dragon	3609K	325K	181.82s	75.89s

Table 5.1: Comparison of reconstruction times between the Poisson method and the proposed hybrid approach. The introduction of the PWLOP operator improves the final mesh quality, and also reduces the number of points employed in the reconstruction. The size of the projected sets has been chosen to try to match the resolution of Poisson's results.

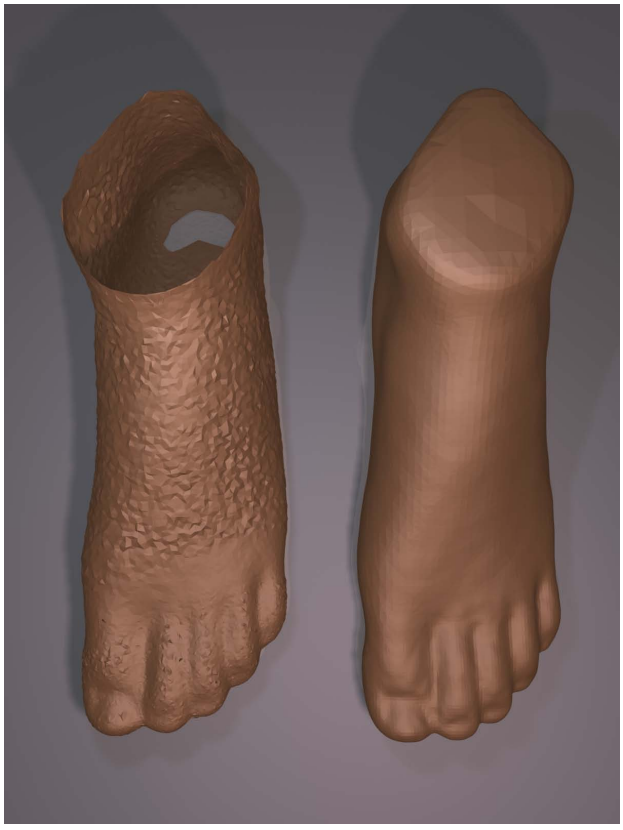


Figure 5.7: Noisy Foot with 20K points: (left) using PWLOP + GLT, 1.52s, (right) using the Poisson reconstruction, 4.84s. It can be seen how, even when the projection operator improves the quality of the mesh, the continuum origin of approximating approaches results in softer meshes.

Figure 5.8: Results of the hybrid PWLOP + GLT reconstruction.

5.3 Discussion

Points set preprocessing is a useful tool if an interpolating reconstruction technique is going to be used, in order to reduce the noise of the data, to increase the quality of the reconstructed mesh and, in some cases, to reduce the size of the input set.

The projection operator PWLOP studied in the previous chapter creates an uniform set of points that approximates the input data. In this chapter, the operator is used as a preprocessing phase to clean up the points set and to increase its uniformity. Experiments showed that the reconstructed mesh from this hybrid approach presents a higher triangle quality as well as that the reconstruction time is reduced.

A direct application of the hybrid PWLOP + GLT method is in the creation of multi-resolution meshes; several meshes can be reconstructed by varying the size of the initial points set of PWLOP. A quick example of this application can be seen in Figure 5.9.

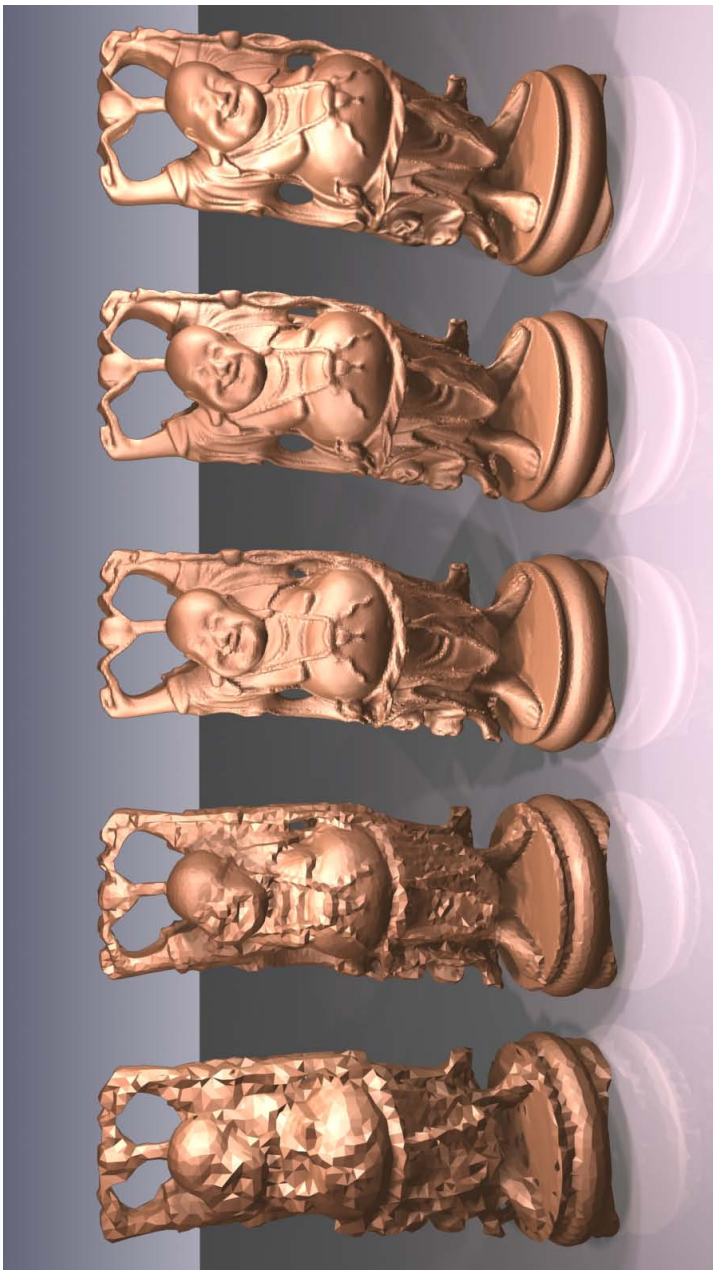


Figure 5.9: Reconstruction of the Happy Buddha in multiple resolution using the hybrid PWLOP + GLT approach. Note that the first model is too coarse that it no longer satisfies the sampling condition and points near corners are misinterpreted. The approximate resolutions of each model are, from left to right, 10K, 26K, 118K, 198K and 396K triangles.

Chapter 6

Study of multi-balloons reconstruction

If debugging is the process of removing bugs,
then programming must be the process of putting them in
Edsger Dijsktra

This chapter studies the possibility of using dynamic techniques for surface reconstruction, designing a parallel paradigm in order to improve the recovery of global properties of the model.

6.1 Dynamic techniques

Techniques studied in previous chapters are also known as *static techniques* because they directly reconstruct the target surface. There is another group, known as *dynamic techniques*, which consist of deformable models that adjust the destination surface by minimizing an energy function associated with the model (such minimization is usually performed in an iterative way, given the non-linear nature of most of these functions)[1].

Dynamic methods have been widely used in the segmentation of medical images. For example, (Kass et al., 1987) formulate the *snakes* or deformable contours, one of the most commonly employed techniques. Snakes are parametric curves that move under the influence of internal forces (defined

[1]Observe that dynamic techniques may be classified also within the approximating category (Section 2.2), if the construction of the surface is not taken into account.

by the curve) and external forces (derived from the image data). Several works proposed different formulations for the internal and external forces, but one of the most relevant is the Gradient Vector Flow (Xu and Prince, 1998), which is a mixed balance force that points toward the object boundary near the boundary, but that varies smoothly in homogeneous regions, allowing a better shape recovering as well as forcing the snake to enter into concave zones. Also worth mentioning is the T-Snake (McInerney and Terzopoulos, 1999), a topology aware snake that is defined in terms of affine cell image decomposition[2].

Following the idea of snakes, *balloons* reconstruct a surface represented by a distance function (Miller et al., 1991; Duan and Qin, 2001). In general, balloons consist in subdivision surfaces that evolve to extract the shape of the 0-valued distance, i.e., they can be seen as the 3D extension of the snakes. Balloons can also reconstruct the topology of the model in a natural way; for example, when two evolving components of the balloon are in collision it is possible to merge the surfaces, increasing the genus of the model. In this line, the work of (Sharf et al., 2006) is especially good, presenting a competing scheme where active parts of the balloon, called *fronts*, are inflated by priority according to their relative size. In this way, a more intuitive interpretation of data is obtained (Figure 6.1). Along with the definition of balloons, (Duan, 2003) mention the possibility of using multiple seeds in parallel to improve the reconstruction time. Also, another 3D reconstruction scheme from point clouds is the use of discrete membranes (a connected region of voxels) to recover the shape and topology of the voxelized point set (Esteve et al., 2005). In a more specific scenario, (Li et al., 2010) present a reconstruction method for arterial objects based on that those models are basically 1D; from this fact, the method evolves tubular snakes along the skeleton of the scanned models.

Finally, the use of the *level set method* (Osher and Sethian, 1988) is widely used in the field of surface reconstruction. Contrary to active contour approaches like snakes and balloons, the level set method manipulates the surface indirectly. The contour (a curve or surface, depending on the dimension) is embedded as the zero level set of a higher dimensional function, which is evolved under the control of a differential equation

[2] *Affine cell image decomposition* is a partitioning scheme, very similar to a triangulation, where the space is divided in cells defined by simplices, and cell vertices are classified as interior, exterior or boundary, depending on their relation with an automatic structure (the snake in this context).

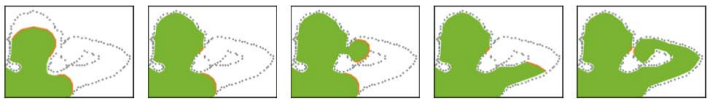


Figure 6.1: Competing fronts reconstruct in a more intuitive way the shape and topology of the data (image from (Sharf et al., 2006)).

(Ibañez et al., 2005). The evolution of the contour will vary depending on the formulation of the level-set function. Its main advantages are that complex shapes can be arbitrarily modeled and topology is easily handled. As an example, (Zhao et al., 2001) formulate the level-set function as a surface energy equivalent to the surface area weighted by some power of the distance function. For a full review of the level set method, please refer to (Sethian, 1999).

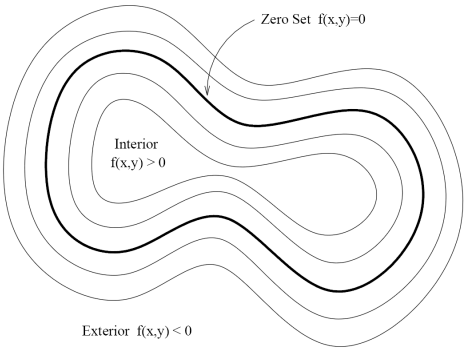


Figure 6.2: Illustration of level sets (image from (Ibañez et al., 2005)).

6.1.1 Classic balloons

In this section, the definition of Competing Fronts (Sharf et al., 2006) and the Intelligent Balloon (Duan and Qin, 2001) are presented. In a general way, a balloon is a subdivision surface $\hat{S}$ that evolves from an arbitrary seed (usually a genus-0 triangle mesh, e.g., a sphere) to minimize an energy function over the surface domain Γ like:

$$E(\mathbf{x}) = \sum_k \omega_{(k)} \int_\Gamma E_{(k)}(\mathbf{x}) \tag{6.1}$$

where $E_{(k)}$ is a set of forces that regulate the evolution of the balloon, and $\omega_{(k)}$ is the correspondent weights. Such terms are usually classified into external forces (e.g., attraction to the target surface) and internal forces (such as curvature or tension constrains).

When the subdivision surface is a polygonal mesh, the minimization of the function showed in Equation 6.1 can be expressed as:

$$\min E(\mathbf{x}) = \min \sum_i \left\| \sum_k \omega_{(k)} E^i_{(k)} \right\|^2 \tag{6.2}$$

where $E^i_{(k)}$ is the k-th force term of the i-th vertex. The forces used by the Intelligent Balloon include a deformation potential, a boundary constraint, a curvature constraint and an angular constraint. In the case of the Competing Fronts, these terms are an attraction force (similar to the deformation potential) and a tension term. The forces used in this work are better explained in a subsequent section.

Both Intelligent Balloon and Competing Fronts work similarly. Given a target surface S, they evolve a triangle mesh $\hat{S}$, minimizing the distance between S and $\hat{S}$. This evolution is done by moving the vertices of $\hat{S}$ in the outward normal direction toward S. When a vertex reaches a local minimum of the distance function to S it stops moving. This eventually creates unconnected mesh components of $\hat{S}$ (*fronts*). When the deformable model completely stops, the method checks if there are remaining parts of S that have not been satisfied yet. In that case, $\hat{S}$ is locally refined to allow the vertices to move towards the target surface. The quality of the mesh is guaranteed by interleaving deformation and mesh optimization operators. Finally, topology can be changed by merging colliding fronts, which increases the genus of the model by one; the nature of balloons makes hole filling an implicit feature of these methods, so hole recovering must be done in a post-processing task. Finally, if unconnected structures have to be reconstructed, additional balloons must be included.

In the case of the Competing Fronts, each front is assigned a tension factor that controls the evolution of the front. This control creates a competition among fronts, allowing bigger fronts, i.e., those with more

tension, to evolve faster than small ones. In this way, it recovers the shape of the model in a coarse-to-fine manner.

6.2 Multi-balloons

One of the main advantages of dynamic techniques is that they provide tools to better control the reconstruction of the dataset. For example, in the reconstruction of a soft tissue from medical images, specific conditions may be easily included into an active contours formulation to respond to additional data from MRI (*magnetic resonance imaging*) for better contrast. Another example is the topology control: when two zones of a balloon collide, they are usually stopped to avoid a self-crossing of the surface. These regions can be merged to increase the genus of the model by one.

In general, and summarizing the previous exposition, balloons perform the reconstruction based on a single deformable surface that is inflated and subdivided as needed, merging confronting components if required by the topology of the model.

There are several shape configurations that deserve attention, for example, long tubular models and objects with several wide zones connected by small tunnels as well as the previously mentioned models with several unconnected regions. In these cases, although active contours can correctly reconstruct the model, they may incur in unnecessary refinements to satisfy frequent changes in the level of detail.

Inspired in the competing fronts reconstruction (Sharf et al., 2006) and the multiple seed extension mentioned by (Duan, 2003), a multi-balloon scheme is studied in this section. The idea is to place several independent balloons in each "cave" and let them evolve in parallel. Using multiple balloons not only may improve the performance of the reconstruction, but it concurrently creates a more natural and intuitive reconstruction workflow, where coarser areas are reconstructed first, refining only those parts of the model that have not been fulfilled yet. As mentioned before, this coarse-to-fine mechanism brings to mind that used by (Sharf et al., 2006), although in this case it would be possible to actually complete the coarser parts of the model before intruding in narrow zones, thereby recovering in a more satisfactory way global properties from the data set and also

reducing the number of times that the distance function has to be refined. An overview of the designed algorithm is shown in Algorithm 6.1, while Figure 6.3 shows a graphical representation of its workflow.

Algorithm 6.1 Overview of the algorithm proposed.

```
1: Compute distance (F) and satisfaction (G) fields
2: while not reconstructed do
3:     while level is not finished do
4:         for each active balloon B_i do
5:             Inflate(B_i)
6:         end for
7:         Detect and solve collisions
8:         Increase genus if desired
9:     end while
10:    Refine distance function
11: end while
```

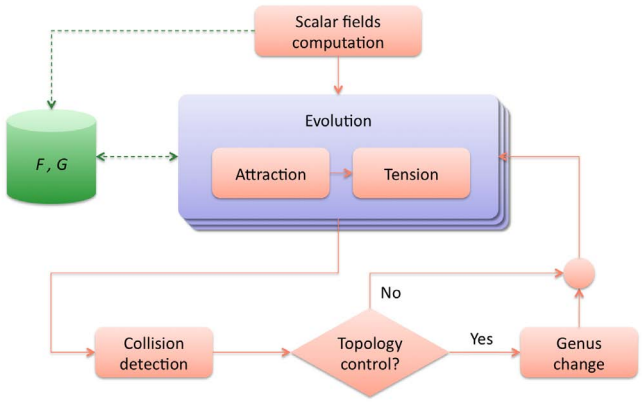


Figure 6.3: Flow diagram of the studied reconstruction using multiballoon.

In general, the proposal starts with the computation of an unsigned distance field to the input points that will guide the reconstruction. Multiple seeds are placed inside the model, that will reconstruct coarser regions of the model first, with an independent balloon evolving and adjusting inside

of each one. When all these regions are satisfied, finer details and unfinished regions are recovered by refining the distance field, and therefore increasing the resolution of balloons. In the meantime, collisions are detected to avoid balloon intrusions, and topology changes are tracked. This process continues until all the data set is reconstructed.

Although the method works positioning the seed of each balloon in any place, it is more efficient to use one balloon for each region of the model; of course, it is a must to place seeds in each unconnected region. Those zones may be defined in several ways, for example: not connected cells of a discrete distance field, a very long tube-like model divided so several balloons start at different points as proposed by (Li et al., 2010), etc. For an illustrative example of seed positioning, please see Figure 6.4.

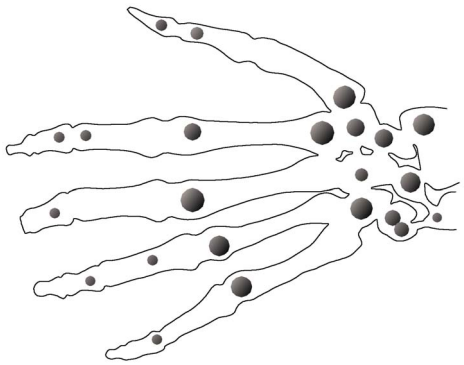


Figure 6.4: Illustrative representation of multiple seeds for the Hand model.

The following sections are organized as follows: Section 6.2.1 explains the unsigned distance function and the spatial subdivision structure used as a helper, and Section 6.2.2 exposes the evolution process, its concepts and different phases. Section 6.2.3 studies the topology control scheme used in this work. Finally, Section 6.3 shows some experiments and results of this proposal.

6.2.1 Scalar function fields

Although not fully introduced yet, this section explains the use and computation of the *distance function field* (F) and the *satisfaction function*

field (G) that guides the evolution of the balloons. F serves as an attraction force for balloons as mentioned before. G, on the other hand, is more widely used: for tracking the position of inactive regions in collision detection, to determine which parts of the model have not been reconstructed yet, and to set the mesh resolution. For the definition of F, a kd-tree is used to compute the nearest point to a vertex, while for G, a multi-resolution 3D grid is employed where its voxels are tagged to reflect the position of the balloons.

Concerning performance, it has been seen that for data sets inferior to a million points, kd-tree queries are nearly as fast as a grid, but considerably more accurate and the data structure requires less memory[3]. But even using a kd-tree for the distance function itself, an underlining grid structure is still necessary to store information about satisfied regions and to guide the resolution of the mesh.

For most of the models, a few levels of refinement are needed for a proper reconstruction; so a multi-resolution grid, similar to a n-tree[4], was used instead of the more classical octree, preferring fewer levels of depth in the tree rather than a high partitioning of the space. This structure is very similar to a scalar field with fixed resolution, but each cell of the grid can be refined to better adapt the level-set zero of the distance function (see Figure 6.5). Although multiple resolutions are possible in each cell and in any level, experiments showed that it is simpler to have a few levels of refinement (2 or 3). The resolution of the first level is imposed by the model (so the coarser regions are recovered easily), while following levels partition the previous one in 2^3 (like an octree), 3^3 or 4^3 sub-levels. As an example of the distance function, please see Figure 6.6.

6.2.2 Evolution process

6.2.2.1 Global and local fronts

Both Competing Fronts and Intelligent Balloon make use local fronts to define active areas of the balloon, i.e., zones that are not near enough to

[3] Approximating Nearest Neighbors, (Mount and Arya, 2010), has been used to speed up the queries, setting up an error of a fraction of the cell's size.

[4] n-trees are the generalization of binary trees, quadtrees and octrees. Each node of the tree may either have no children, or have n subnodes.

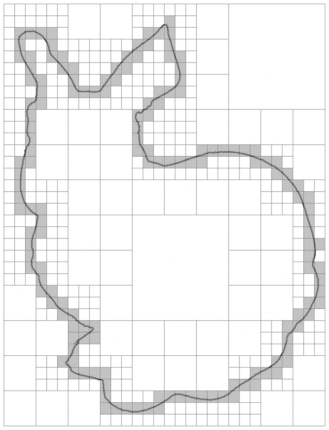


Figure 6.5: Multi-resolution grid used as satisfaction function.

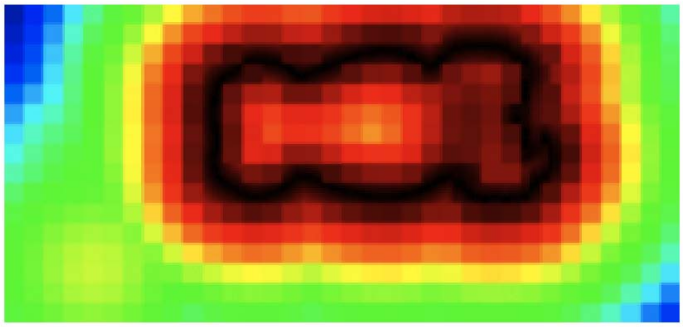


Figure 6.6: Slice of the Horse model's distance function.

the target surface. These active fronts are those that are evolved, reducing the size of the minimization problem.

The creation of the fronts may face several questions: is any active region of the balloon considered a front? is there a minimum front size? It is important because very small fronts (with respect to the precision of the distance function) may create an ill-posed system, or produce pledges.

Another issue is the tracking of vertices along successive remeshing operators (refinements, smoothing, etc.). Such operations destroy and create vertices, and may change the index scheme of the vertices as well as new fronts, or may join existing ones. Tracking the vertices of a front is

costly in terms of memory and speed. On the other hand, creating fronts is a fast operation in comparison to the complexity introduced by vertices tracking structures, so in this work fronts are computed on-the-fly when required. This fronts are used only to speed up collision detection and to determine regions to be joined in genus changes. As fronts are not used in the evolution process, then they must be, at least, as large as the size of tunnels and connecting regions in the target model.

In contrast to previous works, the balloons used in this work employ a global front structure, i.e., the whole balloon is taken into account in the evolution. This approach improves the overall quality of the mesh although the evolution speed is reduced. The next section describes a solution to this problem.

6.2.2.2 Two-step evolution

In previous works, only active fronts ($\hat{S}$ components that are not near enough to S) are included in the evolution of the balloon. In this new formulation, a different approach is studied, evolving the whole balloon at each step, but weighing the vertices movement depending on their distance to S (nearer vertices vary less than further parts). In this way, as the whole balloon is used, it better recovers the open region where it is evolving, and it implicitly generates a coarse-to-fine shape recovering.

In order to reduce the computational requirements of the evolution of the balloon, the minimization system has been replaced by a two-step inflation process, where external and internal forces are independently applied.

In the first step, described in Equation 6.3, vertices are moved in the normal direction toward the target surface, guided by the distance field[5].

$$\begin{aligned} x_i^{(k+1)} &= x_i^{(k)} + r^i \mathbf{n_i^{(k)}} \\ r_i &= \kappa_i F(x_i^{(k)}) \end{aligned} \tag{6.3}$$

The κ_i is a new modulation term that regulates the attraction force; it will be explained in Section 6.2.2.3. Using a different formulation from

[5]From now on, k does not refer to a force but to an iteration of the evolution process.

the one proposed in previous works, the system is expressed in terms of the relative displacement r_i from $x_i^{(k)}$, rather than the position of the vertex itself. This point of view comes from a first implementation of the evolution based on a minimization system. Using r_i provides a twofold advantage: the number of variables that govern the method is reduced, and r_i is scale independent, which means that minimization methods based on gradient descent do not have to deal with the absolute size of the model. When the method was changed to avoid the minimization system, the use of a relative displacement remained useful for a faster and easier computation.

In the second step (Equation 6.4), internal forces smooth the surface and reduce the tension on it.

$$\begin{aligned} E_t^i &= \frac{\sum_{v_j \in N\left(x_i^{(k+1)}\right)} c_{ij}\left(x_i^{(k+1)} - v_j\right)}{\sum_{v_j \in N\left(x_i^{(k+1)}\right)} c_{ij}} \\ c_{ij} &= cot(\alpha_{ij}) + cot(\beta_{ij}) \end{aligned} \tag{6.4}$$

The tension term E_t^i, is the Laplace-Beltrami operator over the neighborhood of each vertex x_i of the front (Meyer et al., 2002). This is basically a smoothing operator that is applied to the balloon after each iteration.

In order to guarantee that the balloons do not penetrate each other, the method detects collisions among the *destinationvertices* (the vertices of the next $k+1$ iteration) before the evolution phase. As is usual in these problems, the intersection between the bounding boxes of the destination fronts is checked to speed up the detection. The vertices of the intersecting triangles are removed from the evolution by not allowing them to move, i.e., $x_i^{(k+1)} = x_i^{(k)}$.

6.2.2.3 Gradient modulation term: κ_i

To prevent the balloon from being excessively attracted if it is filling a hole, and therefore going away the target surface, a new gradient modulation term (κ_i) is introduced into the evolution process. This term is proportional

to the directional derivate at $x_i^{(k)}$ (see Equation 6.5). In case a vertex is about to escape through a hole, the gradient modulation term is simply reversed, keeping the evolving vertex inside the model. This modulation term is kept proportional to the cell size of the distance function to better control the inflation process.

To distinguish between a hole and a cave (which gradients are very similar from the vertex point of view), the nearer voxel of the distance functions (different from the one that contains the vertex) that contains target points is selected. If this point is in front of the vertex of the balloon, then it is assumed that the vertex is about to enter a tunnel so it should be allowed to advance, otherwise the vertex is in a hole and it must be retracted to avoid the balloon escapes through it. In order to speed up the process, only voxels within a certain distance from the vertex are taken into account; in the tests, the search radius has been fixed to three times the cell size.

$$\kappa_i = \delta \left\{ \begin{array}{ll} 1.0 & \text{in a tunnel} \\ \\ \dfrac{1 - 2\mathbf{n_i}^\top \nabla_i}{3} & \text{in a hole} \end{array} \right\} \tag{6.5}$$

where δ is the size of the distance field cell that contains the vertex.

6.2.2.4 Local adaptive remeshing

To allow the balloon to evolve in open areas and reconstruct finer details, regions of the balloon that are not near enough the model are subdivided. Each of these "fronts" is marked for subdivision if its area is much greater than expected for the level of refinement of the distance field, in proportion to the number of triangles of the front.

The mesh quality is guaranteed by remeshing fronts as they evolve, combining the mesh smoothing operator described in Equation 6.4 with a connectivity optimizer similar to the presented by (Vorsatz et al., 2003) and (Botsch and Kobbelt, 2004).

The connectivity optimizers previously mentioned work in a very similar way, by using a combination of edge collapses, splits and flips operations. The overall process is as follows

- An edge is split if it is larger than $\epsilon_{max}\mu$.
- An edge is collapsed if it is shorter than $\epsilon_{min}\mu$.
- An edge is flipped if it reduces the valence excess of the neighboring triangles.

where μ is the objective edge length, satisfying that $\epsilon_{max} > 2\epsilon_{min}$ (Vorsatz et al., 2003). Concerning the values of these terms, (Botsch and Kobbelt, 2004) deduced that they should be set to $\epsilon_{max} = \frac{4}{3}$ and $\epsilon_{min} = \frac{4}{5}$, although it does not satisfy the (Vorsatz et al., 2003)'s proportion. In this work, and in order to fulfill additional requirements, different values were used, as it will be explained bellow.

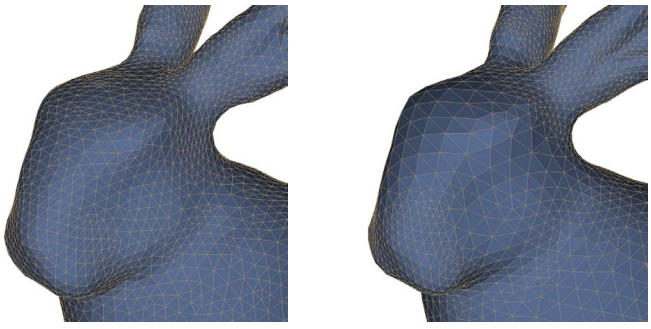
Regarding the edge splits and collapses, when the original remeshing operator is applied, the mesh resolution changes imposed by the hierarchical distance field are lost, since the operator tends to create an uniform mesh. In this work, a reduction of the mesh resolution (edge collapse) avoids recovering fine details while augmenting the resolution means larger systems to be solved as well as to allow the balloon to grow through holes. On the other hand, an increment of the resolution (by edge splitting) in very wide areas reduces the stability of the system.

In order to overpass these limitations, the following modification is proposed: instead of being ϵ_{min} and ϵ_{max} the minimum and maximum edge lengths, they have slightly different meanings: they represent the minimum and maximum scaling factors for the average edge length of the neighborhood of each edge. For stability purposes, the objective edge length μ has been set to the minimum cell's size of the distance function, so $\mu = \delta_{min}$; this guarantees that almost degenerated triangles do not affect the overall mesh resolution . The value of both ϵ_{min} and ϵ_{max} were also relaxed in order to allow a softer resolution change in the mesh after several consecutive iterations of the method. It was found that $\epsilon_{min} = \frac{3}{5}$ and $\epsilon_{max} = \frac{11}{6}$ works well for most of the models. The proposed local adaptive remeshing operator is listed in Algorithm 6.2 and a comparison between it and the uniform remeshing (Vorsatz et al., 2003; Botsch and Kobbelt, 2004) is shown in Figure 6.7, while Figure 6.8 shows another example of the effects of the new operator.

The optimization of the valence of each vertex makes that most of the vertices have a valence of six. For two neighboring triangles $\triangle(a, b, c)$ and

Algorithm 6.2 Local adaptive remeshing operator

1: **for** iter times **do**
2: Let σ_i be the average length of the $Nbhd(edge_i)$
3: Collapse all edges shorter than $\epsilon_{min} \max(\sigma_i, \mu)$
4: Split all edges larger than $\epsilon_{max} \max(\sigma_i, \mu)$
5: Flip edges if the total valence excess is reduced
6: **end for**



(a) Uniform remeshing (b) Local adaptive remeshing

Figure 6.7: Comparison of the uniform remeshing operator (a) and the proposed local adaptive remeshing operator (b) on the reconstruction of the Stanford Bunny model. Note that the proposed operator preserves the different local resolutions, at the time that unifies the triangle size of similar zones.

$\triangle(a, b, d)$, the operator flips the edge from a to b if this operation reduces the total valence excess[6]:

$$\sum_{p \in \{a,b,c,d\}} (v_p - 6)^2 \tag{6.6}$$

In order to reduce the computation of Equation 6.6 before and after edge flipping, in this work a simpler expression was deduced. Edge flipping reduces the valence of vertices a and b, while increasing the valence of c and d, so an edge must be flipped if

[6] As a notation comment, in this section v_x will denote the valence of vertex x.

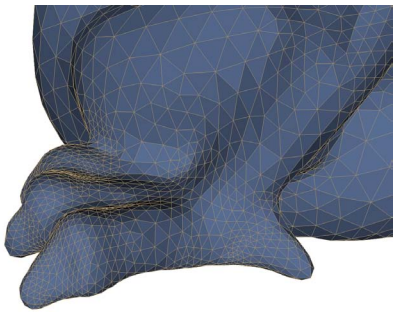


Figure 6.8: Detail of the effects of the local adaptive remeshing operator proposed in this work. Note that the resolution changes smoothly through the mesh thanks to the new average objective edge length.

$$\sum_{p\in\{a,b,c,d\}} (v_p - 6)^2 \quad > \quad (v_a - 1 - 6)^2 + (v_b - 1 - 6)^2 + (v_c + 1 - 6)^2 + (v_d + 1 - 6)^2 \tag{6.7}$$

Expanding out each expression of Equation 6.7, rearranging the terms and simplifying leads to the simpler condition

$$v_a + v_b > 2 + v_c + v_d \tag{6.8}$$

At the end of the evolution process, fine details may have not been reconstructed yet due to the resolution of the mesh near the features. The general process to extract these details consists on increasing the mesh resolution and let the model evolve some few additional times. Previous issues regarding instabilities in the evolution process (due to the disproportion between the mesh resolution and the distance field resolution) are not present in this moment, since the balloons are sufficiently closed to the target surface. Regarding the resolution increment, instead of refining again the distance function, it was found that is easier and more memory efficient to simply subdivide large triangles. The uniform remeshing operator previously tested was used for this task, since it automatically subdivide edges larger than a threshold. The objective edge length is set to be the one of the smallest triangles currently present in the model.

6.2.3 Topology change

Topology recovery is a desirable feature of a surface reconstruction algorithm. For example, the Intelligent Balloon (Duan and Qin, 2001) automatically increases the genus of the model each time two regions collide. To gain a higher control over the genus of the model, in (Sharf et al., 2006) fronts are stopped and then merged at the end of the reconstruction under user indication. As mentioned before, the concept of fronts as not adjusted regions of the balloon, is only used in this phase, during collision detection, holes detection and genus change.

6.2.3.1 Genus

The proposed multi-balloons method takes a step forward and provides an heuristic to determine if two colliding mesh components should be merged or not. Given that balloons are placed inside independent regions of the distance field, it is reasonable to use the size of the front as a parameter for topology change. If two fronts of the same balloon are colliding, or if several balloons were placed in a very long, tube like region, the area of the fronts will not only be similar but also much larger than the cell size of the distance field. On the other hand, fronts too small are much less probable to be reconstructing the same region, thus, they do not need a genus change.

The first step for topology control is the detection of the regions in collision. A full description of collision detection is out of the scope of this thesis, but a review of the most common methods can be found in (Borro, 2003). Basically, collision detection is performed at the end of each evolution cycle; colliding components are removed from the evolution and they are retracted to their initial position. Given the fact each balloon must be tested against itself and the rest, it may be a very slow process. To speed up this test, only active fronts are included in the detection. Then, the intersection of the bounding boxes of such regions is tested and only regions with intersecting bounding boxes are actually tested, a common practice in this kind of problem.

When either an automatic genus change is detected or the user chooses to increase the genus of the model, the nearest pair of vertices from the two fronts to be merged is found. The topology change process is the same presented by (Duan and Qin, 2001; Duan, 2003): the neighborhoods

of these two vertices are aligned to face toward each other, and their one-neighborhoods is put into correspondence. If needed, a neighborhood is refined to match the number of vertices of the larger one. Then, the two vertices are removed and their neighborhoods are reconnected as shown in Figure 6.9. Finally, the new connecting region is relaxed using the smoothing operator described in Section 6.2.2.2.

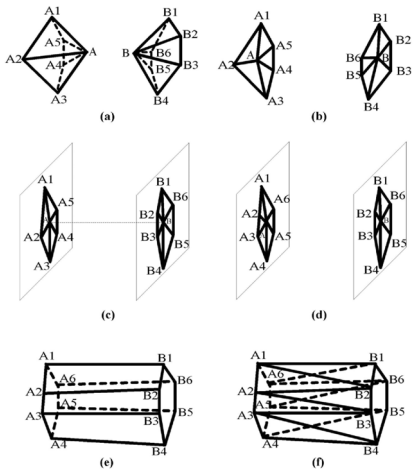


Figure 6.9: Topology change. Triangle fans (a) are first projected (b), oriented (c) and the number of triangles is equaled (d). Genus change is done by removing the neighboring triangles and reconnecting the two neighborhoods (e, f). (Image from (Duan, 2003)).

Illustrating the previous exposition, and as an overview of most of the concepts previously exposed, Figure 6.10 shows the reconstruction of a high-genus synthetic model, starting with four seeds.

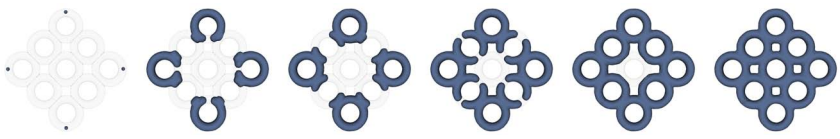


Figure 6.10: Reconstruction of a high-genus synthetic model. Colliding fronts are merged to increase the genus of the deformable model.

6.2.3.2 Holes

By its definition, a balloon is a watertight surface, so no holes are present. The approach used to choose between intruding into a tunnel or not can be also used to detect holes in the mesh as a post-processing task. Those mesh zones of the reconstructed model that are not near enough to any input point should be considered as to be filling a hole. Once those zones have been detected, the user may choose to retain them or to recover the hole. For this last operation, points around the hole are projected onto the mesh, equaling the resolution of the mesh to the points'. After it, the triangles inside can be removed to recovered the hole.

6.3 Experiments and results

Figure 6.11 shows the evolution of several balloons inside the Hand model. As balloons collide they are merged, eventually increasing the genus of the model. It can be seen that, for this model, the current evolution process is not strong enough to fully recover the wrist region.

Note that times shown are particularly high for several reasons. First, this implementation is not parallel yet because the mesh library used (OpenMesh (Botsch et al., 2002)) is not thread-safe. Also, the method is not fully automatic in its current state, so some manual operations must be done, incrementing the time needed for the reconstruction. For the same reason, the visualization of the inflation process is required, introducing the rendering into the timing. However, as it was mentioned in the introduction, this chapter describes the proposal of a new method more focused on the extraction of global properties and the topology of the model.

Figure 6.12 shows the final reconstruction of the Stanford Dragon model. Two independent balloons were inflated inside the points cloud, and the distance field reached two refinement levels. The total number of triangles of the reconstructed model is 90730, with 45367 vertices, about a 10% the size of the input points set. As an illustration, Figure 6.13 shows several stages of the reconstruction of this model.

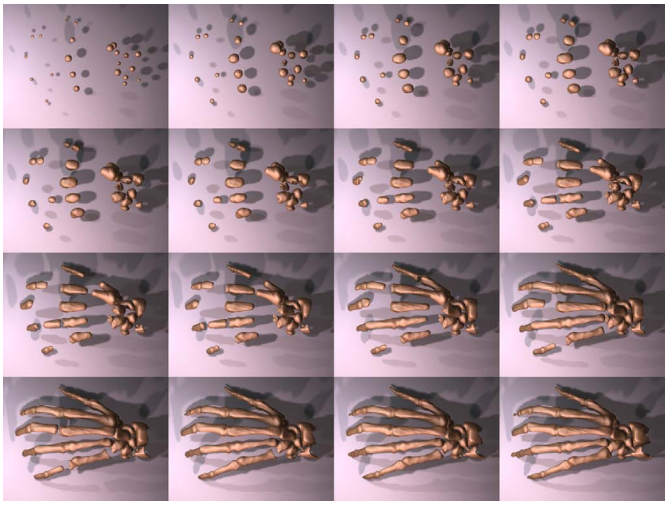


Figure 6.11: Multiple balloons evolving to reconstruct the Hand model (sequence is from left to right, and top to bottom). The total reconstruction time is about 7 minutes.

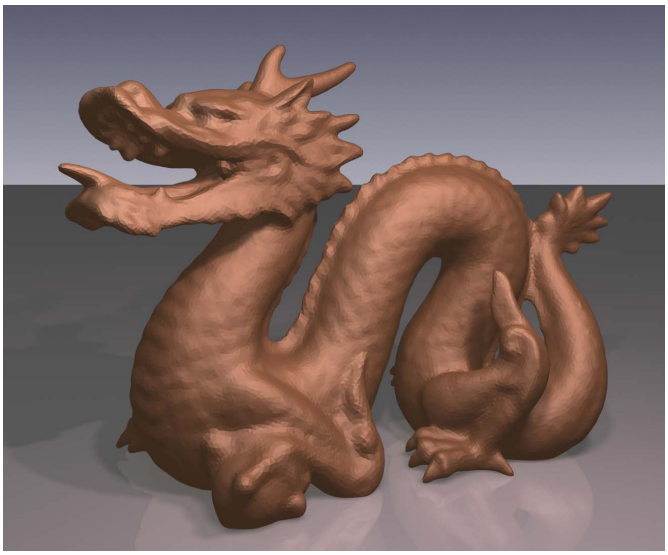


Figure 6.12: Reconstruction of the Stanford Dragon model using two initial balloons. Approximated reconstruction time: 5 minutes.

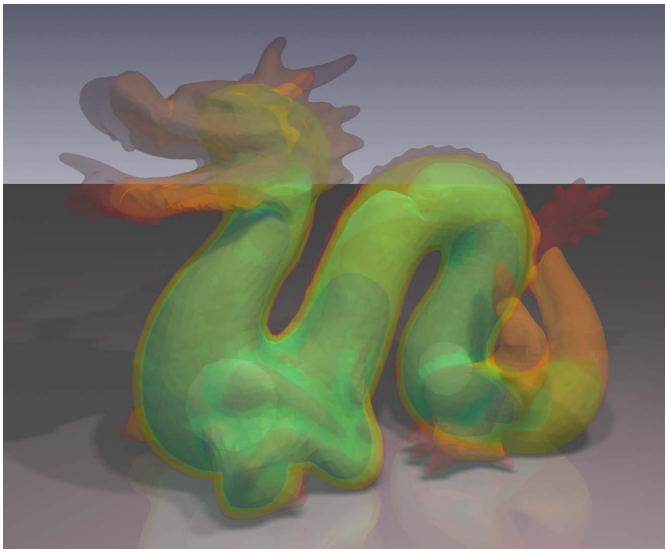


Figure 6.13: Overlapped reconstruction stages (shown with different colors) for the Stanford Dragon model using two balloons. The seeds evolve recovering the coarser sections of the model, then merge and continue evolving and refining until the Dragon is completely reconstructed.

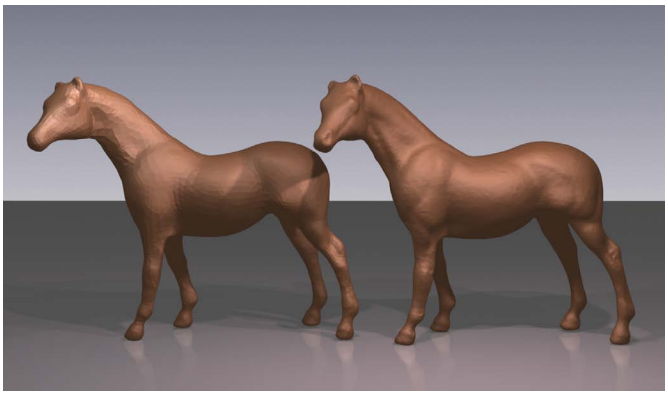


Figure 6.14: Reconstruction of the Horse model: (left) extracted mesh without the final fitting, (right) the mesh resolution is increased and a final fitting finishes the reconstruction.

6.4 Discussion

The present study is an example of the use of dynamic techniques in surface reconstruction. The main goal of this section has been the study of a multi-balloons technique that can be used in a parallel fashion. This approach is particularly useful in complex objects, that present a high genus, or that are composed by several regions (connected or not). By placing multiple seeds along the model, it is possible to better recover the global properties and the topology of the target object, before extracting finer or less relevant details.

Tests performed provide a clear example that the approach is valid and may help in the reconstruction of more complex models. In those experiments, the execution time was not a priority, although it is clear that it must be improved for a real world use. Also, the parallel implementation of the method was not possible given that the mesh library used (OpenMesh) is not currently thread-safe.

Regarding the evolution process of the balloons, it was design having in mind the previous consideration: coarser regions must be recovered before finer ones. In this sense, a new global evolution was designed so it takes into account the global properties of the deformable model. This separates the commonly used attraction and tension terms, leading to a local interpretation of the distance to the target surface (the attraction term) and a global deformation control of the balloon (the tension term). This approach, combined with the gradient modulation factor and the hole/tunnel identification, becomes a convenient evolution function. By the time this work is being written, the main drawback of this approach is its execution time: the global tension term is currently very time consuming in relation with the rest of the process.

Additionally, a local adaptive remeshing operator has been presented. The goal of this operator is to guarantee the mesh quality at the time that it keeps resolution changes across the mesh, originated by the refinement of the distance field function. The use of this local adaptive remeshing avoids the mesh to be refined unnecessarily, creating a larger and unstable system. Some other post-processing tools have been proposed (although not implemented yet), including hole identification and automatic genus increment.

Finally, several aspects of this chapter require a deeper study in the future, such as the fully automatization of the process and an overall optimization of the reconstruction process, including the use of a thread-safe library that allows a parallel implementation of the method.

Part III

Conclusions

Chapter 7

Conclusions and future work

By three methods we may learn wisdom:
first, by reflection, which is noblest;
second, by imitation, which is easiest;
and third by experience, which is the bitterest
Confucius

7.1 Conclusions

This work studies the application of parallelization techniques to the surface reconstruction problem. The main contributions of this work are focused on the design and development of parallel methods that take advantage of the multi-core architecture of CPUs, but mainly of the high processing power of the modern graphic hardware, as well as the optimization of all the phases of such methods for each particular implementation.

It has been shown that local triangulations are very suitable for parallelization, given its independent nature. In this line, an existing interpolating reconstruction method has been studied in this work, given it is fully local and a simple method.

Taking this study as a start point, a new parallel surface reconstruction method, called *GPU Local Triangulation* (GLT), has been developed and three different implementations are presented: one in CPU for reference, and two for the graphic hardware, employing shaders and CUDA. The different phases of GLT were optimized for each implementation and some of them improved in terms of quality with respect to the original work.

The central phase of GLT, the neighborhood validation, tests if a neighbor is in the Voronoi region of a point or not. In this work, a full explanation of this function is given, and it has been mathematically proved that, regardless of the validation order, the test will not discard any valid Delaunay neighbor, leading to a parallel design of the function, which is particularly useful in the shaders implementation.

By definition, interpolating techniques, such as GLT, are not very robust against noise and outliers. Points preprocessing is a common way to solve these issues. In this work, the previous points consolidation operators LOP and WLOP have been studied. By taking advantage of the *extended local support neighborhoods* (eLSN), a new operator, called *Parallel WLOP* (PWLOP), has been designed, reducing the processing time needed by the operator in its CPU implementation. Additionally, and thanks to this new method, a CUDA implementation of the operator is now possible, dramatically improving the performance with respect to the original CPU implementations.

Using PWLOP as a preprocessing phase, the GLT receives a cleaner points cloud as input, avoiding the implicit problem that interpolating methods have with noisy point clouds. In this way, not only the quality of the computed surface is improved, but since the consolidated points are a subset of the original one, the processing time of the reconstruction may also be reduced.

Surface reconstruction based on dynamic techniques were also studied. In this case, it has been seen that the use of multiple evolving balloons recovers the shape and topology of a model in a more intuitive and natural way thanks that it is an actual coarse-to-fine reconstruction. This approach would also lead to a parallel implementation where each balloon freely evolves, independently from other ones. As additional characteristic of this method, it guarantees a watertight surface.

Finally, several additional tools for mesh processing has been presented along with the multiballoons definition: a remeshing operator for non-uniform resolutions, a hole detection tool and a metric for genus change of the model.

7.2 Future research lines

Several research lines are open to continue with this work:

- The GLT algorithm may produce some wrong triangulations. If four points lie on the same circumference, the Delaunay triangulation is not unique and therefore a hole may be created or two overlapping triangles generated. As a solution, several approaches may be taken, for example, a postprocessing mesh repairing operator or the extension of the validity function in order to identify problematic regions (where the four points passed as arguments belong to the same circumference) and mark such points for later repairing.
- Recently, NVIDIA has announced their intentions to port CUDA to the x86 platform. Also, OpenCL implementations in all the platforms are becoming more stable and efficient. These advances may be used to execute in the CPU the less parallel phases, such as neighborhood sorting, taking advantage of a more general purpose processor, at the time the graphic card is freed to receive more points, allowing the phases to be overlapped for more efficiency.
- Currently, GLT uses a single sampling distance when computing the neighborhoods of each point, which is also set to a fixed size. It would be useful to study adaptive parameters for both the distance and the neighborhood size, in order to improve the computation of the candidate points in non-uniform areas.
- Although the GLT reconstruction core can be used in distributed environments, the normal propagation stage is not suitable for such configurations. This propagation may become very demanding in terms of memory given the amount of information about mesh connectivity that must be stored. The study of alternative methods to overcome this issue is then necessary.
- Another stage that is executed in the CPU is the computation of point neighborhoods and, at it has been seen in this work, it is a time consuming task. A GPU implementation of a k-NN algorithm would improve the time required for this step.
- Level-of-detail and multi-resolution meshes fall out of the scope of this thesis, but it is interesting to study the advantages and disadvantages

of the application of hybrid reconstruction methods, such as the proposed PWLOP + GLT, in those fields.

- The formulation of balloons should be improved to recover even better the shape of the model, and it may also be interesting to find a physical meaning to the two-step evolution process.

- The tension term of the evolution process is currently applied in a global way. It has been seen that already adjusted regions of the model tend to stabilize over the time, so it would be interesting to reintroduce the idea of active fronts in order to reduce the time required by the smoothing operators, combining it with the global evolution to preserve the overall mesh quality and adjustment.

- Reconstruction using multiple evolving balloons has to be fully parallelized. The mesh processing library used in the implementation of balloons is not thread-safe, so the parallel implementation is not possible yet.

  As an example of another possible extensions, a recent work of (Vasilakis and Fudos, 2009) takes the GLT method as a base for the triangulation of an unorganized set of points. They extends the GLT reconstruction by introducing a k-NN method in the vertex shaders, as well as replacing the mesh storage by a direct visualization using geometric shaders.

Part IV

Appendices

Appendix A

GPGPU computing

Insanity: doing the same thing over and over again and expecting different results

Albert Einstein

With the arrival of programmable GPUs, many people have been using them to perform more than special effects. GPGPU computing is a recent field of work in computer science. The goal of this area is to take advantage of the high computational power of modern graphic processors (which currently is an order of magnitude higher than CPUs) and their specific operation sets for computer graphics. Among the most common techniques to perform GPGPU computing can be found the use of shaders and, more recently, CUDA. Although other CUDA-like technologies exist, such as Direct Computing (Microsoft) and OpenCL, only CUDA will be treated in this section, since it was the one employed in the work, and the three are very similar and most of the concepts can be shared.

GPGPU computing is mainly based in the *SIMD* (Single Instruction, Multiple Data) programming model. In this model, multiple processing units execute the same instruction over a data set under the supervision of a common control unit, i.e., data is processed in parallel but synchronously.

A.1 Shaders

Shaders are special programs used to modified the fixed rendering pipeline, employed in conjunction with graphic APIs[1] such as OpenGL or Direct3D. Depending on the graphic API used, the shader programming language to be employed may vary. The most important shading languages are Cg© (NVIDIA®), HLSL (Microsoft®) and GLSL (Khronos Group). Cg and HLSL are very similar, and the three are C-like languages.

Different shader types exist to manipulate data in different stages of the pipeline:

- **Vertex shaders**: transform 3D vertices (once per run) to the 2D coordinate system of the viewport. Vertex shaders can modify the position, color and texture coordinates of the vertices, but cannot create new vertices
- **Geometric shaders**: assembles the geometric primitive that will be sent to the rasterizer. These shaders can create or destroy vertices and are usually employed for tessellation of parametric or implicit surfaces.
- **Fragment shaders**: also known as pixel shaders, compute the color of the individual pixels that come from the rasterizer. Fragment shaders are very flexible and are commonly used in objects lighting and texturing, special effects and even in non-polygonal based visualization, such as volume rendering. Their main disadvantage is that they cannot write data to a different pixel coordinate than the one assigned by the rasterizer.

Given natural analogies with the SIMD model, shaders based GPGPU computing usually makes use of fragment shaders to work. Although less formal, it is easier to see how it works in a scheme:

- First, data is stored in textures, as if they were arrays. The only limitations here are those self imposed by the texture structure: all the elements must have the same structure and each of them can stored up to four values of the same basic data type (floats, integers),

[1] API stands for *Application Programming Interface*.

corresponding to each of the four color components: red, green, blue or alpha channel.

- The viewport is configured appropriately to draw data. For example, if the computation is one output per each input, the viewport must be setup to have the same size as the texture. The rendering output is set to be another texture, so results can be written back to memory.
- The fragment shader is enabled. Textures and any other individual parameters are loaded.
- A textured rectangle is drawn to fulfill the viewport. In this way, each rasterized pixel will correspond to a texel.

In this way, the fragment shader is executed for each element of the data and the results are written to the specified texture that can be later read or used as the input of another shader (thus avoiding the costly transfer to and from the main memory).

Some common GPGPU applications are iterative processes of the form of $x_{i+1} = f(x_i)$. In this case, a technique called *ping–pong rendering* is commonly used. It consists in the use of two interchangeable textures of the same size and structure, one for reading the data and one for writing; after each iteration, their roles are simply swapped.

Fragment shaders impose some restrictions in the programming model that must be taken into account:

- No random position scattering (writing). A fragment shader can only write in the position specified by the rasterizer. For example, it is not possible for a shader to store its results in different cells of a grid; in this case, the value of each cell must be computed by an individual shader and each shader must have the corresponding rasterizer position. This follows that the viewport determines the structure of the output.
- Only modern GPUs allow branching (execution bifurcations produced by conditional statements and loops), but its use must be reduced as much as possible in order to avoid high speed penalties. If two threads of the same shader enter different regions of a branch, each set is executed by the two threads but only the corresponding memory states are kept for each one.

- Transfers between main memory and graphic memory must be carefully scheduled to reduce bandwidth overhead.

- Current graphic hardware imposes different restrictions with respect to the texture size. Initially these constrains included not only the maximum size, but also that the size must be a power-of-two. Nowadays this limitation has disappeared and the maximum size is often 4096×4096.

A.2 CUDA

CUDA™ (*Computing Unified Device Architecture*) is a C extension developed by NVIDIA®, which allows a higher level of abstraction than those obtained with shaders. CUDA capable devices can accelerate the execution of computationally intensive programs exploiting the data level parallelism of the executed algorithms. These devices can work together in order to solve large problems and they always work within a host (a PC). This technology was introduced in desktop computers with the G80 GPU in late 2006 that was included in the GeForce® 8800 graphics card family. At the same time, NVIDIA launched its Tesla® dedicated GPGPU device. Basically, the only difference between a Tesla device and a normal GPU is that the first lacks a display output. Recent Tesla devices based on the Fermi™ architecture also have four times more arithmetic precision than its graphics device equivalent.

CUDA has solved some of the main disadvantages of GPGPU programming through shaders, e.g. the fixed-position scattering limitation. This has allowed numerous algorithms to became easier to implement in the GPU. Regardless of this ease, it is still necessary to design the algorithms to use efficiently the resources of the CUDA device.

Additionally to CUDA, other similar technologies exist, such as Direct Computing (Microsoft®) and OpenCL™ (Khronos Group). The following sections will introduce some basic concepts on CUDA, however, most of them are applicable to other computing technologies.

A.2.1 CUDA Program Structure

A CUDA program is built using both regular functions, that are executed in the host, and CUDA functions, called *kernels*, that are executed in the CUDA device. These functions are separately compiled using the standard C++ compiler for the CPU code and the NVIDIA `nvcc` compiler for the CUDA kernels.

CUDA kernels, when called, are executed many times in parallel using the threading capabilities of the device. As shown in Figure A.1, these threads are grouped in blocks, that at the same time are grouped in a grid. Grids and blocks are one, two or three dimensional arrays and their size is only limited by the CUDA device.

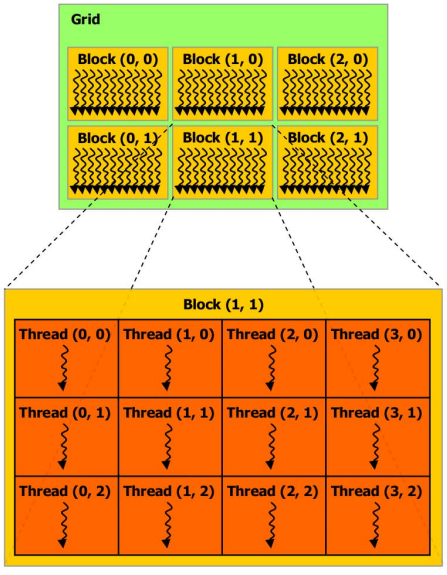


Figure A.1: Thread hierarchy (image from (NVIDIA, 2010)).

The size of the grid and the blocks can be set by the user in every kernel call. Each thread executes the same kernel code and has a unique ID that can be accessed from the kernel giving a natural way to do computations across the elements of a matrix.

A.2.2 Occupancy

When a group of threads is received to be executed, the multiprocessor device splits them into warps that are individually scheduled. A *warp* is a group of threads (32 by the time this memory was written) that starts together at the same program address but that are free to branch independently. A warp executes a command at a time, so full efficiency is realized when all the threads of the warp follow the same instruction path.

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. It is an important metric in determining how effectively the hardware is used: a higher occupancy eases the device to hide memory latency and therefore helps to improve performance. For more information about occupancy and CUDA programs optimization, please refer to (NVIDIA, 2009).

A.2.3 CUDA Memory Model

CUDA devices have their own memory space and threads can not access directly to the host memory. Instead, the data must to be transferred from the host to the device in order to make the computations, and from the device to the host to get the results.

Figure A.2 shows an overview of the device memory model. Each thread has a private local memory where local variables are stored. At block level, there is a shared memory visible to all threads of the block which size can be set dynamically before the kernel invocation. Finally, there is a global memory that can be randomly accessed by all threads and it is persistent across kernel executions. There are also two read-only memories, i.e constant memory and texture memory. Constant memory is a small space that can be accessed randomly very fast. In contrast, texture memory, which inherits from graphics applications, is a large memory that can be organized in up to three dimensions. It is locally cached and can be accessed through a hardware interpolator.

Contrary to traditional shaders, CUDA threads can randomly read and write in any position of the global memory. However, this memory has a big access latency and should be used with care. Although having many threads can hide the access latency, the global memory has a limited amount of bandwidth and can be easily collapsed making computing units go idle.

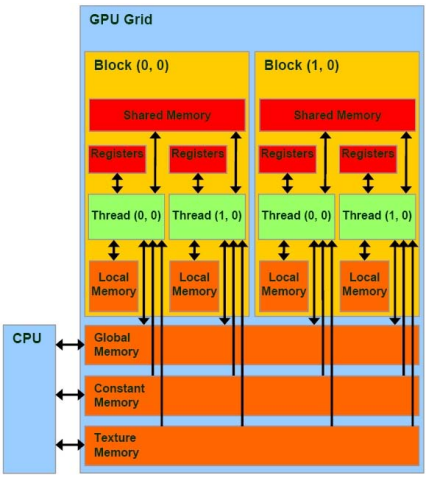


Figure A.2: Memory hierarchy.[2]

This problem can be solved moving the data from global memory to shared memory, which has much smaller latency, and using it across the threads of the same block.

In conclusion, in order to use efficiently the CUDA device, special care should be taken in the data level parallelism of the algorithm and in the memory access patterns, trying to minimize the memory transfers between the host and the device. For these reasons, many existing algorithms can not be directly implemented in CUDA, needing new approaches that exploit the benefits of this architecture.

[2]Courtesy: NVIDIA

Appendix B

Generated Publications

Many of life's failures are people who did not realize how close they were to succes when they gave up
Thomas Edison

Journals

Buchart, C., Borro, D., and Amundarain, A. "GPU Local Triangulation: an interpolating surface reconstruction algorithm". *Computer Graphics Forum*, Vol. 27, N. 3, pp. 807–814. May, 2008.

San Vicente, G., Buchart, C., Borro, D., and Celigüeta, J. T. "Maxillofacial surgery simulation using a mass-spring model derived from continuum and the scaled displacement method." *International journal of computer assisted radiology and surgery*, Vol. 4, N. 1, pp. 89–98. January, 2009.

Book chapters

Buchart, C., Amundarain, A., and Borro, D. *3-D surface geometry and reconstruction: Developing concepts and applications*, chapter Hybrid surface reconstruction through points consolidation. IGI Global. 2011. (Sent and under revision).

Conferences

Buchart, C., Borro, D., and Amundarain, A. "A GPU interpolating reconstruction from unorganized points". In *Posters Proceedings of the SIGGRAPH 2007*. San Diego, CA, USA. August 5-9, 2007.

San Vicente, G., Buchart, C., Borro, D., and Celigüeta, J. T. "Maxillofacial surgery simulation using a mass-spring model derived from continuum and the scaled displacement method." In *Posters Proceedings of Annual Conference of the International Society for Computer Aided Surgery (ISCAS'08)*. Barcelona, Spain. June, 2008.

Buchart, C., Borro, D., and Amundarain, A. "GPU Local Triangulation: an interpolating surface reconstruction algorithm". In *Proceedings of the EuroVis 2008*, volume 27, pp. 807–814. Eindhoven, The Netherlands. May 26-28, 2008.

Buchart, C., San Vicente, G., Amundarain, A., and Borro, D. "Hybrid visualization for maxillofacial surgery planning and simulation". In *Proceedings of the Information Visualization 2009 (IV'09)*, pp. 266–273. Barcelona, Spain. July 14-17, 2009.

Index

α-shapes, 16

Advancing frontal techniques, 14
Angle computation, 32
    Cosine symmetry, 32
Approximating techniques, 11, 16

Ball Pivoting, 15
Balloons, 90, 91
    Competing fronts, 90, 91
    Fronts, 90, 92
    Intelligent balloon, 91
Bitonic Merge sort, 43
Bubble sort, 48

Cocone algorithm, 14
    Robust Cocone, 14
    Tight Cocone, 14
CUDA, 122
    Kernels, 123
    Warp, 124

d-simplex, 12
Data structures
    Angles texture T_P, 42
    Candidate points array C_R, 47
    Candidate points ring texture T_R, 44
    Candidate points texture T_Q, 40
    Neighborhood size array C_m, 47
    Normals array C_N, 47
    Normals texture T_N, 39
    Points array C_P, 47
    Points indexing texture T_I, 40
    Points texture T_P, 39
    Projected points array $C_{Q'}$, 47
    Projected points texture$T_{Q'}$, 42
    Ring of candidate points T_R, 33
Delaunay triangulation, 12, 18
    2D validity test, 34
    2D validity test proof, 36
    Lower dimensional method, 14, 33
DeWall algorithm, 18
Discrete membranes, 17
Distance function field, 95
Dynamic techniques, 89

Gabriel graph, 13
Global triangulation, 13
GPGPU, 5
GPU, 5
Gradient Vector Flow, 90

Implicit function, 16
Interpolating techniques, 11, 12
Iso-surface, 16

k-NN, 25
    ANN, 27
    Clustering, 26
    kd-trees, 27

Level set method, 90
Local support neighborhood, 69
    Extended LSN, 69
Local triangulation, 12
    Delaunay, 12, *see also* Delaunay triangulation
LOP, 66
    Extended WLOP, 70
    Parallel WLOP, 70
    Weighted LOP, 68

Marching Cubes, 18
    Marching tetrahedra, 19
Medial axis, 13
    Medial axis transform, 13
Multiballoons, 93

Normal
    Estimation, 28
    Orientation, 30

Parallelization, 24
Poisson reconstruction, 17, 58
    GPU implementation, 20
Power Crust, 13

Sampling criteria, 25
    Nyquist-Shannon sampling theorem, 25
Satisfaction function field, 95
Shaders, 120
    Fragment, 120
    Geometric, 120
    Occupancy, 124
    Ping pong, 45
    Vertex, 120
Spiraling Edge, 15
Surface reconstruction, 3

T-Snakes, 90
Voronoi diagram, 12

References

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and T. Silva, C. "Computing and Rendering Point Set Surfaces". *IEEE Transactions on Visualization and Computer Graphics*, Vol. 9, N. 1, pp. 3–15. 2003.

Allégre, R., Chaine, R., and Akkouche, S. "A flexible framework for surface reconstruction from large point sets". *Computers and Graphics*, Vol. 31, N. 2, pp. 190–204. 2007.

Alliez, P., Cohen-Steiner, D., Tong, Y., and Desbrun, M. "Voronoi-based variational reconstruction of unoriented point sets". In *Proceedings of the Eurographics Symposium on Geometry Processing (SGP'07)*, pp. 39–48. Barcelona, Spain. July 4-6, 2007.

Amenta, N., Choi, S., Dey, T. K., and Leekha, N. "A simple algorithm for homeomorphic surface reconstruction". In *Proceedings of the Annual Symposium on Computational Geometry*, pp. 213–222. Hong Kong, China. June 11-14, 2000.

Amenta, N., Choi, S., and Kolluri, R. "The Power Crust". *Sixth ACM Symposium on Solid Modeling and Applications*, pp. 249–260. 2001.

Attene, M. and Spagnuolo, M. "Automatic Surface Reconstruction from Point Sets in Space". *Computer Graphics Forum*, Vol. 19, N. 3, pp. 457–465. 2000.

Bajaj, C. L., Bernardini, F., and Xu, G. "Automatic Reconstruction of Surfaces and Scalar Fields from 3D Scans". In *Proceedings of the SIGGRAPH 1995*, pp. 109–118. Los Angeles, CA, USA. August 6-11, 1995.

Batcher, K. E. "Sorting networks and their applications". In *Proceedings of the Spring Joint Computer Conference*, pp. 307–314. April 30–May 2, 1968.

Bentley, J. L. "K-d trees for semidynamic point sets". In *Proceedings of the Annual Symposium on Computational Geometry*, pp. 187–197. Berkley, CA, USA. June 6-8, 1990.

Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C. T., and Taubin, G. "The Ball-Pivoting Algorithm for Surface Reconstruction". *IEEE Transactions on Visualization and Computer Graphics*, Vol. 5, N. 4, pp. 349–359. 1999.

Besora, I., Brunet, P., Callieri, M., Chica, A., Corsini, M., Dellepiane, M., Morales, D., Moyés, J., Ranzuglia, G., and Scopigno, R. "Portalada: A virtual reconstruction of the entrance of the ripoll monastery". In *Proceedings of the International Symposium on 3D Data Processing, Visualization and Transmission*. Atlanta, GA, USA. June 18-20, 2008.

Bloomenthal, J. "Polygonization of implicit surfaces". *Computer Aided Geometric Design*, Vol. 5, N. 4, pp. 341–355. October, 1988.

Borro, D. *Colisiones en estudios de mantenibilidad con restitución de esfuerzos sobre maquetas digitales masivas y compactas*. PhD thesis, Escuela Superior de Ingenieros, Universidad de Navarra. 2003.

Botsch, M. and Kobbelt, L. P. "A remeshing approach to multiresolution modeling". In *Proceedings of the Eurographics Symposium on Geometry processing (SGP '04)*, pp. 185–192. Nice, France. July 8-10, 2004.

Botsch, M., Steinberg, S., Bischoff, S., and Kobbelt, L. P. "Openmesh-a generic and efficient polygon mesh data structure". In *Proceedings of the OpenSG Symposium*. January, 2002.

Buchart, C., Amundarain, A., and Borro, D. *3-D surface geometry and reconstruction: Developing concepts and applications*, chapter Hybrid surface reconstruction through points consolidation. IGI Global. 2011. (Sent and under revision).

Buchart, C., Borro, D., and Amundarain, A. "A GPU interpolating reconstruction from unorganized points". In *Posters Proceedings of the SIGGRAPH 2007*. San Diego, CA, USA. August 5-9, 2007.

Buchart, C., Borro, D., and Amundarain, A. "GPU Local Triangulation: an interpolating surface reconstruction algorithm". *Computer Graphics Forum*, Vol. 27, N. 3, pp. 807–814. May, 2008.

Buchart, C., Borro, D., and Amundarain, A. "GPU Local Triangulation: an interpolating surface reconstruction algorithm". In *Proceedings of the EuroVis 2008*, volume 27, pp. 807–814. Eindhoven, The Netherlands. May 26-28, 2008.

Buchart, C., San Vicente, G., Amundarain, A., and Borro, D. "Hybrid visualization for maxillofacial surgery planning and simulation". In *Proceedings of the Information Visualization 2009 (IV'09)*, pp. 266–273. Barcelona, Spain. July 14-17, 2009.

Buck, I. and Purcell, T. J. *A Toolkit for Computation on GPUs (GPU Gems)*, chapter 37, pp. 621–636. Addison-Wesley. 2004.

Cano, P., Torres, J. C., Melero, F. J., Martín, D., Moreno, J., and España, M. "Generación automatizada mediante escáner láser de documentación de patrimonio histórico". In *Proceedings of the Congreso Internacional de Rehabilitación del Patrimonio Arquitectónico y Edificación*. 2008.

Cignoni, P., Montani, C., Perego, R., and Scopigno, R. "Parallel 3D Delaunay Triangulation". *Computer Graphics Forum*, Vol. 12, N. 3, pp. 129–142. August, 1993.

Crossno, P. and Angel, E. "Spiraling edge: fast surface reconstruction from partially organized sample points". In *Proceedings of the Conference on Visualization (VIS'99)*, pp. 317–324. Los Alamitos, CA, USA. October, 1999.

Delaunay, B. "Sur la sphere vide. A la mémoire de Georges Voronoi". *Bulletin of Academy of Sciences of the USSR*, Vol. 7, pp. 793–800. 1934.

Dey, T. K. and Goswami, S. "Tight cocone: A water-tight surface reconstructor". *Journal of Computing and Information Science in Engineering*, pp. 127–134. 2003.

Dey, T. K. and Goswami, S. "Provable surface reconstruction from noisy samples". *Computational Geometry: Theory and Applications*, Vol. 35, N. 1, pp. 124–141. 2006.

Duan, Y. *Topology adaptive deformable models for visual computing*. PhD thesis, State University of New York. 2003.

Duan, Y. and Qin, H. "Intelligent balloon: a subdivision-based deformable model for surface reconstruction of arbitrary topology". In *Proceedings of the ACM symposium on Solid modeling and applications (SMA'01)*, pp. 47–58. New York, NY, USA. 2001.

Esteve, J., Brunet, P., and Vinacua, A. "Approximation of a variable density cloud of points by shrinking a discrete membrane". *Computer Graphics Forum*, Vol. 24, N. 4, pp. 791–808. 2005.

Esteve, J., Vinacua, A., and Brunet, P. "Piecewise algebraic surface computation and smoothing from a discrete model". *Computer Aided Geometric Design*, Vol. 24, N. 6, pp. 357–372. 2008.

Favreau, J.-M. "Marching cube cases". 2010.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. "An algorithm for finding best matches in logarithmic expected time". *ACM Transactions on Mathematical Software*, Vol. 3, N. 3, pp. 209–226. 1977.

Gopi, M., Krishnan, S., and Silva, C. T. "Surface Reconstruction based on Lower Dimensional Localized Delaunay Triangulation". *Computer Graphics Forum*, Vol. 19, N. 3, pp. 467–478. September, 2000.

Gueziec, A. and Hummel, R. "Exploiting triangulated surface extraction using tetrahedral decomposition". *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, N. 4, pp. 328–342. 1995.

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. "Surface Reconstruction from Unorganized Points". In *Proceedings of the SIGGRAPH 1992*, pp. 71–78. 1992.

Huang, H., Li, D., Zhang, H., Ascher, U., and Cohen-Or, D. "Consolidation of Unorganized Point Clouds for Surface Reconstruction". *ACM Transactions on Graphics*, Vol. 28, N. 5, pp. 176–182. 2009.

Huang, H., Li, D., Zhang, H., Ascher, U., and Cohen-Or, D. "Points consolidation API". 2010.

Ibañez, L., Schroeder, W., Ng, L., and Cates, J. *The ITK software guide*. Insight Software Consortium, second edition. 2005.

Jalba, A. C. and Roerdink, J. B. T. M. "Efficient surface reconstruction from noisy data using regularized membrane potentials." *IEEE Transactions on Image Processing*, Vol. 18, N. 5wan, pp. 1119–34. May, 2006.

Jones, M. "3D distance from a point to a triangle". 1995.

Kass, M., Witkin, A., and Terzopoulos, D. "Snakes: Active contour models". *International Journal of Computer Vision*, Vol. 1, N. 4, pp. 321–331. 1987.

Kazhdan, M., Bolitho, M., and Hoppe, H. "Poisson surface reconstruction". In *Proceedings of the Eurographics Symposium on Geometry Processing (SGP'06)*, pp. 61–70. Cagliari, Sardinia, Italy. June 26-28, 2006.

Kipfer, P. and Westermann, R. "GPU Construction and Transparent Rendering of Iso-Surfaces". In Greiner, G., Hornegger, J., Niemann, H., and Stamminger, M., editors, *Proceedings of the Vision, Modeling and Visualization 2005*, pp. 241–248. 2005.

Klein, T., Stegmaier, S., and Ertl, T. "Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids". In *Proceedings of the Pacific Conference on Computer Graphics and Applications (PG'04)*, pp. 186–195. 2004.

Kohout, J. and Kolingerová, I. "Parallel Delaunay triangulation based on circum-circle criterion". *Proceedings of the 18th spring conference on Computer graphics - SCCG '03*, Vol. 1, N. 212, p. 73. 2003.

Levin, D. "Mesh-independent surface interpolation". *Geometric Modeling for Scientific Visualization*, Vol. 3, pp. 37–49. 2003.

Li, G., Liu, L., Zheng, H., and Mitra, N. J. "Analysis, reconstruction and manipulation using arterial snakes". *ACM Transactions on Graphics*, Vol. 29, N. 6, pp. 152:1–152:10. December, 2010.

Linsen, L. and Prautzsch, H. "Local Versus Global Triangulations". In *Proceedings of the Eurographics 2001 (Short Presentations)*, pp. 257–264. Manchester, England. September 5-7, 2001.

Lipman, Y., Cohen-Or, D., and Levin, D. "Data-Dependent MLS for Faithful Surface Approximation". In *Proceedings of the Eurographics*

Symposium on Geometry Processing (SGP'07), pp. 59–67. Barcelona, Spain. July 4-6, 2007.

Lipman, Y., Cohen-Or, D., Levin, D., and Tal-Ezer, H. "Parameterization-free projection for geometry reconstruction". *ACM Transactions on Graphics*, Vol. 26, N. 3, p. 22. July, 2007.

Lorensen, W. E. and Cline, H. E. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In *Proceedings of the SIGGRAPH 1987*, pp. 163–169. 1987.

McInerney, T. and Terzopoulos, D. "Topology Adaptive Deformable Surfaces for Medical Image Volume Segmentation". *IEEE Transactions on Medical Imaging*, Vol. 18, pp. 840–850. 1999.

Meyer, M., Desbrun, M., Schröder, P., and Barr, A. H. "Discrete Differential-Geometry Operators for Triangulated 2-Manifolds". 2002.

Miller, J. V., Breen, D. E., Lorensen, W. E., O'Bara, R. M., and Wozny, M. J. "Geometrically deformed models: a method for extracting closed geometric models form volume data". *SIGGRAPH Comput. Graph.*, Vol. 25, N. 4, pp. 217–226. 1991.

Mount, D. M. and Arya, S. "ANN: a library for approximate nearest neighbor searching". 2010.

Nagai, Y., Ohtake, Y., and Suzuki, H. "Smoothing of Partition of Unity Implicit Surfaces for Noise Robust Surface Reconstruction". *Computer Graphics Forum*, Vol. 28, N. 5, pp. 1339–1348. July, 2009.

NVIDIA. *NVIDIA CUDA C Programming Best Practices Guide*. 2009.

NVIDIA. *NVIDIA CUDA C Programming Guide*. 2010.

Ohtake, Y., Belyaev, A., Alexa, M., Turk, G., and Seidel, H. P. "Multi-level partition of unity implicits". *ACM Transactions on Graphics*, Vol. 22, pp. 463–470. 2005.

Osher, S. and Sethian, J. A. "Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations". *Journal of Computational Physics*, Vol. 79, N. 1, pp. 12–49. 1988.

Pauly, M., Gross, M., and Kobbelt, L. P. "Efficient simplification of point-sampled surfaces". In *Proceedings of the Conference on Visualization (VIS'02)*, pp. 163–170. Washington, DC, USA. 2002.

Press, W., Teukolsky, S., Vetterling, W., and Flannery, B., editors. *Numerical recipes*, chapter 9, 11, pp. 585, 464–466. Cambridge University Press, third edition. 2007.

Raman, S. and Wenger, R. "Quality isosurface mesh generation using an extended marching cubes lookup table". *Computer Graphics Forum*, Vol. 27, N. 3, pp. 791–798. May, 2008.

Samozino, M., Alexa, M., Alliez, P., and Yvinec, M. "Reconstruction with Voronoi centered radial basis functions". In *Proceedings of the Eurographics Symposium on Geometry Processing (SGP'06)*, pp. 51–60. Cagliari, Sardinia, Italy. June 26-28, 2006.

San Vicente, G., Buchart, C., Borro, D., and Celigüeta, J. T. "Maxillofacial surgery simulation using a mass-spring model derived from continuum and the scaled displacement method." In *Posters Proceedings of Annual Conference of the International Society for Computer Aided Surgery (ISCAS'08)*. Barcelona, Spain. June, 2008.

San Vicente, G., Buchart, C., Borro, D., and Celigüeta, J. T. "Maxillofacial surgery simulation using a mass-spring model derived from continuum and the scaled displacement method." *International journal of computer assisted radiology and surgery*, Vol. 4, N. 1, pp. 89–98. January, 2009.

Sánchez, J. R., Álvarez, H., and Borro, D. "Towards Real Time 3D Tracking and Reconstruction on a GPU Using Monte Carlo Simulations". In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR'2010)*, pp. 185–192. Seul, South Korea. October 13-16, 2010.

Sankaranarayanan, J., Samet, H., and Varshney, A. "A Fast k-Neighborhood Algorithm for Large Point-Clouds". In *Proceedings of the IEEE/Eurographics Symposium on Point-Based Graphics*, pp. 75–84. Boston, MA, USA. July 29-30, 2006.

Sethian, J. A. *Level Set Methods and Fast Marching Methods* (ISBN: 978-0521645577). Cambridge University Press. 1999.

Sharf, A., Lewiner, T., Shamir, A., Kobbelt, L. P., and Cohen-Or, D. "Competing Fronts for Coarse-to-Fine Surface Reconstruction". *Computer Graphics Forum*, Vol. 25, N. 3, pp. 389–398. September, 2006.

Torres, J. C., Cano, P., Melero, J., España, M., and Moreno, J. "Aplicaciones de la digitalización 3D del patrimonio". In *Proceeding of Congreso Internacional de Arqueología e Informática Gráfica, Patrimonio e Innovación: Arqueológica 2.0*. Sevilla, Spain. June 17-20, 2009.

Torres, J. C., Soler, F., Velasco, F., León, A., and Arroyo, G. "Marching octahedra". In *Proceedings of the Congreso Español de Inform´ática Gráfica (CEIG'09)*. San Sebastián, Spain. September 9-11, 2009.

Vasilakis, A. A. and Fudos, I. "Skeletal rigid skinning with blending patches on the GPU". Technical report, Department of Computer Science, University of Ioannina, Ioannina, Greece. October 12, 2009.

Vermeer, P. J. *Medial axis transform to boundary representation conversion*. PhD thesis, Purdue University. 1994.

Vorsatz, J., Rössl, C., and Seidel, H. P. "Dynamic remeshing and applications". In *Proceedings of the ACM Symposium on Solid Modeling and Applications (SMA'03)*, pp. 167–175. 2003.

Wang, J., Oliveira, M. M., Xie, H., and Kaufman, A. E. "Surface reconstruction using oriented charges". *Computer Graphics International 2005*, pp. 122–128. 2005.

Xu, C. and Prince, J. L. "Snakes, Shapes, and Gradient Vector Flow". *IEEE Transactions on Image Processing*, Vol. 7, N. 3, pp. 359–369. 1998.

Zhang, L., Liu, L., Gotsman, C., and Huang, H. "Mesh reconstruction by meshless denoising and parameterization". *Computers & Graphics*, Vol. 34, N. 3, pp. 198–208. June, 2010.

Zhao, H., Osher, S., and Fedkiw, R. "Fast Surface Reconstruction Using the Level Set Method". In *Proceedings of the IEEE Workshop on Variational and Level Set Methods*, pp. 194–202. 2001.

Zhou, K., Gong, M., Huang, X., and Guo, B. "Data-Parallel Octrees for Surface Reconstruction". *IEEE Transactions on Visualization and Computer Graphics*, 2010. To appear.