THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Tamy Boubekeur**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

——————

Hierarchical Processing, Editing and Rendering of Acquired Geometry

——————

Soutenue le : 21 Septembre 2007

Après avis des rapporteurs :

Markus Gross ...	Professeur ETH Zurich	
Roberto Scopigno	Directeur de Recherche au CNR de Pise	

Devant la commission d'examen composée de :

Markus Gross ...	Professeur - ETH Zurich	Rapporteur
Roberto Scopigno	Directeur de Recherche - CNR	Rapporteur
Claude Puech ...	Directeur de Recherche - INRIA	Examinateur
Pascal Guitton ...	Professeur - Université Bordeaux I	Président
C. Schlick	Professeur - Université Bordeaux II ...	Directeur de thèse

2007

Abstract

Digital representations of real-world surfaces can now be obtained automatically using various acquisition devices such as 3D scanners and stereo camera systems. These new fast and accurate data sources increase 3D surface resolution by several orders of magnitude, borrowing higher precision to applications which require digital surfaces. All major computer graphics applications can take benefit of this automatic modeling process, including: computer-aided design, physical simulation, virtual reality, medical imaging, architecture, archaeological study, special effects, computer animation and video games.

Unfortunately, the richness of the geometry produced by these media comes at the price of a large, possibility gigantic, amount of data which requires new efficient data structures and algorithms offering scalability for processing such objects.

This thesis proposes time and space efficient solutions for modeling, editing and rendering such complex surfaces, solving these problems with new algorithms sharing 4 fundamental elements: a systematic hierarchical approach, a local dimension reduction, a sampling-reconstruction paradigm and a point-based basis.

Basically, this manuscript proposes several contributions, including: a new hierarchical space subdivision structure, the Volume-Surface Tree, for geometry processing such as simplification and reconstruction; a streaming system featuring new algorithms for interactive editing of large objects, an appearance-preserving multiresolution structure for efficient rendering of large point-based surfaces, and a generic kernel for real-time geometry synthesis by refinement.

These elements form a pipeline able to process acquired geometry, either represented by point clouds or non-manifold meshes. Effective results have been successfully obtained with data coming from the various applications mentioned.

Résumé

La représentation des surfaces du monde réel dans la mémoire d'une machine peut désormais être obtenue automatiquement via divers périphériques de capture tels que les scanners 3D. Ces nouvelles sources de données, précises et rapides, amplifient de plusieurs ordres de grandeur la résolution des surfaces 3D, apportant un niveau de précision élevé pour les applications nécessitant des modèles numériques de surfaces telles que la conception assistée par ordinateur, la simulation physique, la réalité virtuelle, l'imagerie médicale, l'architecture, l'étude archéologique, les effets spéciaux, l'animation ou bien encore les jeux video.

Malheureusement, la richesse de la géométrie produite par ces méthodes induit une grande, voire gigantesque masse de données à traiter, nécessitant de nouvelles structures de données et de nouveaux algorithmes capables de passer à l'échelle d'objets pouvant atteindre le milliard d'échantillons.

Dans cette thèse, je propose des solutions performantes en temps et en espace aux problèmes de la modélisation, du traitement géométrique, de l'édition intéractive et de la visualisation de ces surfaces 3D complexes. La méthodologie adoptée pendant l'élaboration transverse de ces nouveaux algorithmes est articulée autour de 4 éléments clés : une approche hiérarchique systématique, une réduction locale de la dimension des problèmes, un principe d'échantillonage-reconstruction et une indépendance à l'énumération explicite des relations topologiques aussi appelée *approche basée-points*.

En pratique, ce manuscrit propose un certain nombre de contributions, parmi lesquelles : une nouvelle structure hiérarchique hybride de partitionnement, l'Arbre Volume-Surface (VS-Tree) ainsi que de nouveaux algorithmes de simplification et de reconstruction ; un système d'édition intéractive de grands objets ; un noyau temps-réel de synthèse géométrique par raffinement et une structure multi-résolution offrant un rendu efficace de grands objets.

Ces structures, algorithmes et systèmes forment une chaîne capable de traiter les objets en provenance du pipeline d'acquisition, qu'ils soient représentés par des nuages de points ou des maillages, possiblement non 2-variétés. Les solutions obtenues ont été appliquées avec succès aux données issues des divers domaines d'application précités.

Une traduction de l'introduction et de la conclusion résume cette thèse en fin de manuscrit.

Acknowledgments

First, I thank the "Best PhD Advisor of the World", Christophe Schlick, for all the discussions we had during the 3 years of my PhD. He brought me unnumbered scientific skills, passion for research, and inspiration for teaching.

This thesis has been reviewed by Marcus Gross and Roberto Scopigno, and presented to Claude Puech. They were all members of my PhD commity, and I want to thank them for taking interest in my work.

I also want to thank my team, IPARLA, for all the work and entertainment we had. In particular, I want to thank Pascal Guitton, the team leader, for his support and his ability to lead us in advanced research activities while maintaining a **really** fair ambiance in the group. Special thanks goes also to Patrick Reuter, Xavier Granier, Raymond Namyst, Carole Blanc and Pierre-André Wacrenier for their comments and help at many levels during my research and teaching activities at Bordeaux University.

I also want to thank my co-authors, Florent Duguet, Julien Hadim, Olga Sorkine and Julien Lacoste, for all the pleasant time I had working and discussing with them.

Wolfgang Heidrich, and the Imager group from the University of British Columbia, offered me a great experience during my long-term stay in Vancouver, Canada. Special thanks to Abhijeet Ghosh, Vladislav Kreavoy and Eric Broschu for their friendship during my canadian days.

Finally, I want to thanks my friends for the amazingly good time I had with them in Bordeaux and my family for their everyday support during my studies. And more than anyone, I want to thank my beloved Elisabeth, for all the love she gives me and the unconditionnal support she offered at all the steps of this thesis.

Contents

Chapter 1

Introduction

Digital modeling encapsulates objects and phenomena in a set of numerical values describing their properties. Among them, the *shape* has a fundamental importance in all applications involving a simulation, the simplest being the image synthesis capturing an approximation of the illumination undergone by the object, in other words: the *rendering*. Prior to this process, the *geometric modeling* of a tri-dimensional object uses a large variety of functions to represent this shape: they are structured by spatial, spectral or semantic links, and differ according to the target application, the time and memory constraints, the level of accuracy required or even the mandatory artistic rules.

Applications of geometric modeling and rendering range from scientific simulation to entertainment software, including reverse engineering, special effects, archaeological exploration, video games, education, training and computer animation. With the recent increase of digital technologies, widely supported by the development of Internet, all these fields have to face a growing demand on a short time schedule, resulting in complex multimedia systems like, for instance, *high-end 3D packages*, *graphics-physics 3D engines* and *flying simulators*. However, while 3D technologies are now able, in many situations, to quickly produce near-realistic images, there is still a lack in the creation of what really matters: the content.

For decades, digital 3D models have been created by Computer Graphics (CG) designers, using complex interactive tools for reproducing real-world objects and inventing imaginary ones. While computer animation and video games strongly rely on their artistic skills, CG designers cannot fulfill demands on rapid and precise surface modeling from real-world.

Recently, a new way to create 3D objects has emerged: *automatic modeling*, or how to generate 3D objects with a 3D scanner, just like pictures are taken by cameras. With these new devices, generating million of polygons sampling a human face can be done in a matter of seconds, and two or three engineers can produce digital models of a building, with a sub-millimeter precision, in few hours.

Unfortunately, this new source of content brings a bunch of new problems, coming from the two characteristics of sampled surfaces:

- acquisition is a discrete process that only gets a sampling of reality. Therefore, the notion of surface, intrinsically continuous, has to be reconstructed, involving more or less arbitrary decisions to reconnect the discrete samples

- the fine degree of accuracy induces huge data sets, which challenge even the most powerful computers for applications that require a quick feedback, like processing, editing and rendering.

In this thesis, we propose new algorithms for large 3D objects, designed for time and memory efficiency, and able to handle the complex shapes coming straight from the 3D acquisition pipeline.

1.1 Overview of Contributions

Throughout our research work, we had to solve various problems occurring in the geometric processing pipeline dedicated to acquired geometry. We present several original contributions in the fields of efficient processing, editing and rendering techniques for sampled 3D surfaces. Here is an exhaustive list of the main contributions:

Processing

- a new hierarchical space subdivision structure, the Volume-Surface Tree, which can replace the octree for efficient partitioning, offering a better error-driven split.
- a new fast surface simplification algorithm based on VS-Tree
- a new fast surface reconstruction algorithm based on VS-Tree
- a generic kernel for out-of-core simplification.

Editing

- a size-insensitive framework for interactive editing
- two kernels for transferring appearance and deformation between different sampling of an object.

Rendering

- an efficient multiresolution polygonal rendering algorithm for point-based surfaces
- an appearance preserving conversion for large objects rendering
- a generic kernel for real-time mesh refinement with arbitrary displacement
- a controllable refinement method for mesh smoothing with singularities
- an approximation of subdivision surfaces for real-time applications.

1.2 Outline

In the various topics that we address in this thesis, a large number of previous ideas and papers are discussed. In order to maintain a clear presentation, we do not concentrate the presentation of these numerous previous contributions in a single chapter, but spread out them, according to the context, at the beginning of each chapter. This thesis is organized in 3 parts:

Part I proposes new geometry processing and editing methods for large objects.

Chapter 3 introduces the volume-surface tree and its application to fast simplification and reconstruction of surfaces.

Chapter 4 describes our size-insensitive framework for arbitrary large model editing.

Part II offers a new multiresolution generation and rendering system for point-based surfaces.

Chapter 5 explains how to generate and render an adaptive polygonal structure for visualizing point-based surfaces.

Chapter 6 extends this idea by introducing the idea of attribute mapping in streaming for appearance preservation of large objects.

Part III introduces a new generic refinement kernel for interactive applications.

Chapter 7 proposes a new GPU kernel for adaptive mesh refinement, allowing arbitrary mesh refinement and displacement in a single pass, at vertex shader level.

Chapter 8 extends the original Curved PN Triangle refinement scheme by introducing scalar tags to control surfaces singularities such as sharp creases and tension.

Chapter 9 tackles the problem of real-time subdivision surface rendering by proposing a visually plausible approximation which avoids recursion, and allows adaptive sampling at very deep levels.

Each chapter starts by a motivation and a context, stating related state-of-the-art problems and solutions. Each contribution is systematically concluded with results and implementation details, as well as a discussion presenting limitations, particular notions, a summary and the perspectives related to the contribution.

Chapter 2

Background

The algorithms, data structures and techniques presented in this thesis aims at offering efficient processing of sampled geometry. Therefore, the background of this work takes place at the junction of several fields and we discuss the related work when necessary throughout the manuscript. However, several topics are transverse and presented in the following sections.

2.1 Acquiring Geometry

The raise of 3D scanners leverages the work of CG designers, but requires an engineering process for ending with high quality 3D surfaces. Basically, the idea is to combine several 2.5D sampling of an object, taken from various points of view, to form a 3D sampling, latter converted to a surface. One must note that only 3D samples capture the reality: any topological connectivity between these samples, either explicit or implicit, comes as an assumption made over the original surface. Usually, the 3D acquisition pipeline is composed of three main steps:

1. the **capture**, which samples intersections distances (or depth values) in a whole range of directions from a given point of view, outputting depth images (also called scan sheets),

2. the **registration**, which puts together a set of depth images, taken from different points of view, to form a 3D point cloud,

3. the **reconstruction**, which generates a surface (e.g. mesh) from the point cloud.

At each step, *data cleaning* must be applied for eliminating the various artefacts that may occurs. In this thesis, we will focus on the processing and use of these geometries. Nevertheless, in order to correctly appreciate the choice we made all along our work, we give a deeper description of this geometry acquisition pipeline (illustrated in Figure 2.1) in the following paragraphs.

We refer the reader to the course on 3D Photography by Curless and Seitz [CS00] and to the survey on the 3D model acquisition pipeline by Bernardini and Rushmeier [BR02] for a complete review of 3D acquisition technologies.
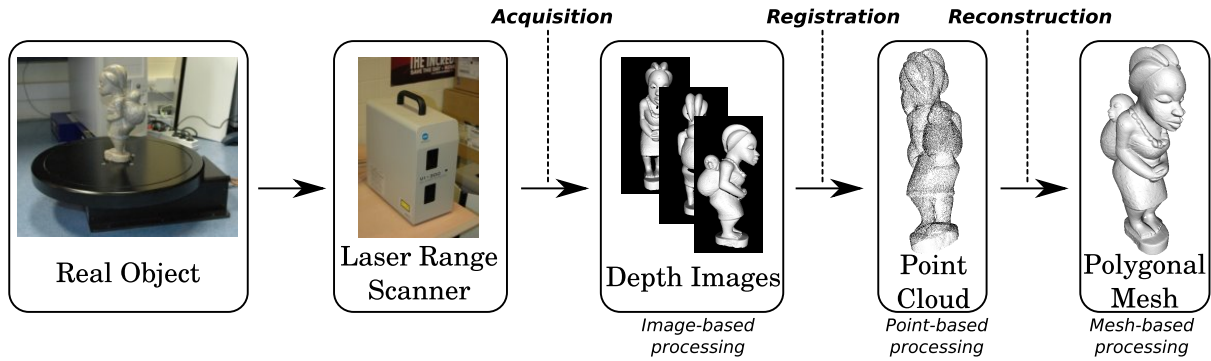
Figure 2.1: *The 3D acquisition pipeline.*

Raw data capture   The surface sampling is performed through a set of depth images $\{D_0, ... D_n\}$ that indicates, for a whole range of rays, at which depth arise the first intersection with the real model along the ray defined by $\{x, y\}$, providing a 2.5D sample $\{x, y, D_i(x, y)\}$ in the frame of the sensor. This capture can be performed with various devices, called *3D scanners*, which are classified in two main families:

- **Active 3D scanners**: a radiation (e.g. light) is emitted and the response produced by the scanned model is used to define its depth values. For instance:

  - *Laser 3D Scanners* use laser rays to sample the surfaces, either using a time-of-flight measure (for large scale distant objects, like buildings) or triangulation (a fixed camera is used for capturing the laser impact on the surface of smaller objects).

  - *Structured Light 3D Scanners* project 1D or 2D patterns using an LCD projector, analyze the deformation of the pattern produced by the surface projection and triangulate the depth value. Such acquisition is usually faster but less accurate than laser techniques.

- **Passive 3D scanners**: simple cameras can be used to capture different views; the set of resulting images is then analyzed using:

  - *Stereoscopy*, measuring the difference obtained from two near points of view.

  - *Silhouettes*, extracted by image analysis, and combined to approximate the convex hull of the model (concavities can be missed).

Active 3D scanners are more expensive, may not work in some particular lighting conditions, but still offers betters results than passive ones. Note also that while color may be acquired using the same process, the view/light-dependent nature of this property usually requires different systems for capturing the underlying material and reusing it in different view/lighting conditions (see Goesele's thesis [Goe04]). Finally, capturing transparent objects is not really possible with all these systems (see the topographic reconstruction of Trifonov et al. [TBH06]).

Registration   The registration searches for a set of transformation matrices $\{M_0, ..., M_n\}$ that align relatively to each other the 3D geometry defined by the set of depth image. Let $p = \{x, y, D_i(x, y)\}$ a 3D sample defined by D_i, the registration process generates a matrix M_i so that $M_i \otimes p_i^\top = M_j \otimes p_j^\top$, with p_i and p_j sampling the same surface point from their relative frame. Popular methods to register scans divide the problem in two steps:

1. a global registration that use a full scan analysis to roughly align two scans; this step is the harder one if no additional information is provided (it may be user-controlled)

15

2. a local registration, that use curvature-based feature correspondence to perform the precise align-ment. One efficient algorithm to do so is the Iterative Closest Point (ICP) [BM92], which can be enhanced to spread the residual error over the complete set of pair-wise aligned scans, minimizing the maximum error [Pul99].

At the end of the process, a 3D discrete point set P is output, so that $\forall p \in P, \exists \{x, y, i\} / p = M_i \otimes \{x, y, D_i(x, y)\}^\top$

Reconstruction   Finally, P is converted into a continuous surface S. Most of these algorithms end with a mesh and can be classified in three families:

- **explicit methods** generate directly a mesh interpolating P. Such methods include displacement-based methods [STKK99, JK02], deformable models [TM91, DQ01] and growing fronts [SLS*06]

- **combinatorial methods** use an intermediate combinatorial structure built upon P for selecting a part of the so-defined connectivity as the surface. Such methods can be based on the Voronoi diagram [BC00, ACK01, DGH01], its dual Delaunay Triangulations [GKS00, CSD02], the convex hull or the k-graph (graph linking each sample of P with k neighbors)

- **implicit methods** define a function $f : \mathbb{R}^3 \to \mathbb{R}$ so that one of its iso-surface (usually, the zero set) approximates or interpolates P [HDD*92, HDD*94]. Several basis exist for these func-tions, either using Radial Basis Functions (RBF) [SPOK95, CBC*01, TO02], quadric approxi-mations [OBA*03], or Poisson equations [KBH06]. Depth Image can also be directly used after registration for constructing a distance function [CL96]. These techniques often relies on some least square fitting, and may be:

    - global: P is entirely used for defining f in any point

    - local: only a small and compact subset of P is considered for a given point

    - *artificially* local: several functions interpolate globally subsets of P and are combined to-gether, using for instance the *partition of unity* [OBA*03, TRS04] for solving the global reconstruction.

  As meshes are ubiquitous in 3D applications, a final explicit solution must be extracted from the implicit form of f. In general, finding this solution is not possible through an analytical process, so a piecewise linear approximation is usually obtained with a 3D contouring method, such as the Marching Cubes [LC87, Blo88] algorithm, for creating a polygonal mesh.

In spite of the numerous papers published over the last 15 years, surface reconstruction — and its dual sampling theory — is still an open problem [Gro06] and all existing solutions have specific drawbacks and advantages, with theoretical guarantees that do not hold in practice. In Chapter 3, we will focus on fast reconstruction methods and show how the problem can be solved in most of the cases by an hybrid implicit-explicit reconstruction. We refer the reader to the work of Kazhdan [Kaz05] for a recent survey on surface reconstruction.

Data cleaning   Each stage of the acquisition pipeline introduces its own noise and artifacts. Therefore, specific data filtering must be employed a each level. Note that all these processes strongly benefit from additional user control [WPK*04], particularly in in under-sampled or highly noisy areas.

Image-based filtering   Raw data samples are organized in depth images. This indicates an acquisition "direction" for samples and image processing techniques can be employed. For instance, on very specular material, laser scanners may estimate a too small value for the depth of some samples, producing "peaks" in the direction of view. Therefore, a local Laplacian filtering can help to detect and remove these artifacts. Moreover, since depth images only represent partial capture, another global cleaning has to be performed after registration, for eliminating unwanted component (e.g., wall behind the scanned object)

Point-based filtering   After the second stage, the resulting point cloud may also exhibit some noise that needs to be removed, particularly in overlapping regions that have driven the registration. At this point, point-based methods are employed. In particular, the Moving Least Square projection is recognized as a good filter for non uniform point clouds. In the original operator [Lev98a, Lev98b], the surface defined by stationary projection, the Point Set Surface or PSS [ABCO*01, AK04a], is evaluated at any point $p \in \mathbb{R}^3$ by:

- collecting a set of neighboring samples N^p in the point cloud

- fitting a plane to N^p in the least square sense

- fitting a low degree polynomial parameterized on this plane, that minimize the L^2 error to N^p

This projection procedure can actually be replaced by a simple projection on an average plane when a normal estimation is provided [AK04b], still converging. This operator acts as a smoothing operator, and can preserve features when required [FCOS05]. Note that it is somewhat related to the *reproducing kernel particle approximation*. We refer the reader to our own work [RJT*05], not discussed in the present thesis.

Mesh-based filtering   Finally, once reconstructed, the resulting mesh can undergone various enhancement like hole filling, noise removal, anisotropic semi-regular remeshing and parameterization. We refer to the course of Botsch et al. [BPK*07] for an introduction to mesh processing.

One fundamental geometric processing is the simplification step, that allows to conform a surface resolution to a given budget of samples or polygons allowed by the final application. We will discuss this problem in Chapter 3.

About Normals   Once registered and before reconstruction, having an estimation of the normal vector for each sample of the point cloud is a very useful information. One solution for that is to estimate this vector directly in the depth image, using local differentiation. However, the resulting normal information remains an estimation as it is based on the depth image geometry, and it might be false on silhouettes (this can be prevent by either filtering the normals or removing border samples, which have low confidence anyway). An alternative method is to use *shape-from-shading* to "measure" the normal information, by using several photos with different lighting conditions, and which can be used with geometry-based estimation in an optimization process. We refer to the work of Nehab et al. [NRDR05] for additional information.

Last, when the point cloud comes without any information on the original depth images, Hoppe et al.[HDD*92] propose to consider, for each sample, the eigen vector associated to smallest eigen value of the covariance matrix of its nearest neighbors as an approximation of the direction of the normal. A minimum spanning tree is then used to make the orientation consistent. We will essentially consider *point-based surfaces* as point clouds equipped with normal vectors.

2.2   Meshes Versus Point-Sampled Surfaces

At the end of the third stage of the acquisition pipeline, the large repository of geometry processing, editing and rendering techniques is available and can be used. However, observing the second stage, several researchers have developed methods that act directly on the point cloud, before reconstruction. In the literature, they appear as *Point-based* of *Mesh-less* methods. Globally, point-based algorithms process shapes without considering any explicit topology information, such as the edges and the polygons of a mesh. They rather use weak topological estimators based on the k nearest neighbors (k-neighborhood), or all the samples contained in ball around the considered location (ε-neighborhood). In facts, many of point-based tools offer similar results that mesh-based ones. Sometimes, the results are even better, like for *surface simplification*, for which point-based methods allow more degrees of freedom for reducing the resolution of a shape.
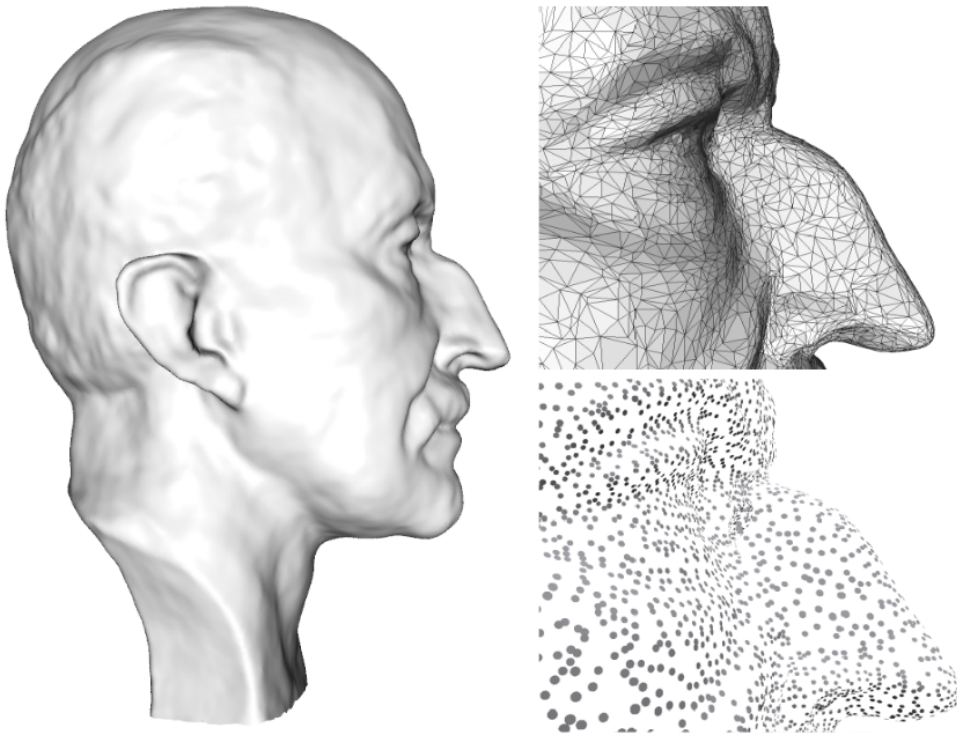
Figure 2.2:   Left: *The Max Plank model.* **Top right:** *Close-up on the mesh version.***Bottom right:** *Close-up on its point cloud version.*

Considering performances, both representations trade memory for speed: meshes encode additional data for polygons (and sometimes edges) but offer constant time neighbor access, while point sets, free from topological information, can only offer a linear access time to neighbors. A logarithmic access is possible at the price of the pre-generation of a space partitioning structure, such as kD-Trees, but this structures implies an additional memory overhead.

So neither efficiency nor quality can clearly help to distinguish situations where point-based surfaces should be preferred to meshes. Still, two elements vote for using point-based processing when possible. First, the surface reconstruction is far from being a straightforward task: it is time-consuming and many algorithms have side effects on the final geometry compared to the original sampling provided by the 3D scanner. So using point-based methods helps to maintain as long as possible the original sampled surface as a "ground truth". Second, even after reconstruction, most of surface reconstruction algorithms

cannot provide *guaranteed* manifold for the output mesh. This means than the explicit connectivity may not be consistent everywhere, resulting in a so-called *triangle soup*. Third, surface reconstruction **is** a more or less arbitrary answer to the question: "Which samples should be indexed by a given polygon and thus become privileged neighbors ?". Likewise, nothing is said about the choice made by the surface reconstruction algorithm when a mesh is submitted to a given process. In fact, topology information cannot be acquired statically, and is decided using more or less strong supposition during the sampling of the shape, an information usually unknown.

In this context, point-based methods appear as generic processing methods, able to process either manifold meshes, polygon soups and point clouds. Unfortunately, rendering hardware and algorithms are designed for polygonal surfaces, imposing meshes in most of commercial CG software. In Chapter 5 and 6, we will show that polygonal rendering technique can benefit to point-based surfaces, introducing a fast local meshing algorithm for interfacing **directly** point-based surfaces and polygonal rendering systems, allowing to combine flexible point-based modeling tools with efficient polygonal rendering, even with large models.

In this thesis, we will not make a strong difference between **acquired geometry**, **sampled surface**, **point cloud** or **point-based surface** (PBS). All these terms correspond to the set of samples coming from the acquisition pipeline before reconstruction. In general, we consider this set as processed (e.g., outliers removal, noise filtering) and in many cases, with sampled or estimated normal vectors.

2.3 Large Objects

With sub-millimeter precision in range scans [LPC*00], even a small real-world object can lead to tens or hundred of millions samples at the end of the acquisition pipeline. Being too big for most applications, it must be simplified to a target resolution fitting hardware and algorithmic capabilities. However, the simplification process is application-dependent and, for instance, special effects experts have a different definition of optimality than video games designers: the former use to deal with millions of polygons per objects, while the latter is expected to keep only few thousands. So, deciding to simplify an object before processing or editing it may results in loss of features that could be useful for future applications.

One solution to this problem is *out-of-core processing*. These techniques allow to process or visualize an object at its full resolution, using either streaming or external memory management. One popular example of such methods is simplification itself: too large objects do not fit in memory and require out-of-core methods for being simplified [Lin00]. However, out-of-core methods remains limited to slow offline processing or visualization of static shapes [RL00, CGG*04] after a long preprocess. One major contribution of this thesis, stated in Chapter 4 is a set of algorithms organized in a streaming system allowing interactive shape and appearance editing of large objects, keeping full resolution models on the output and opening a path to size-insensitive computer graphics.
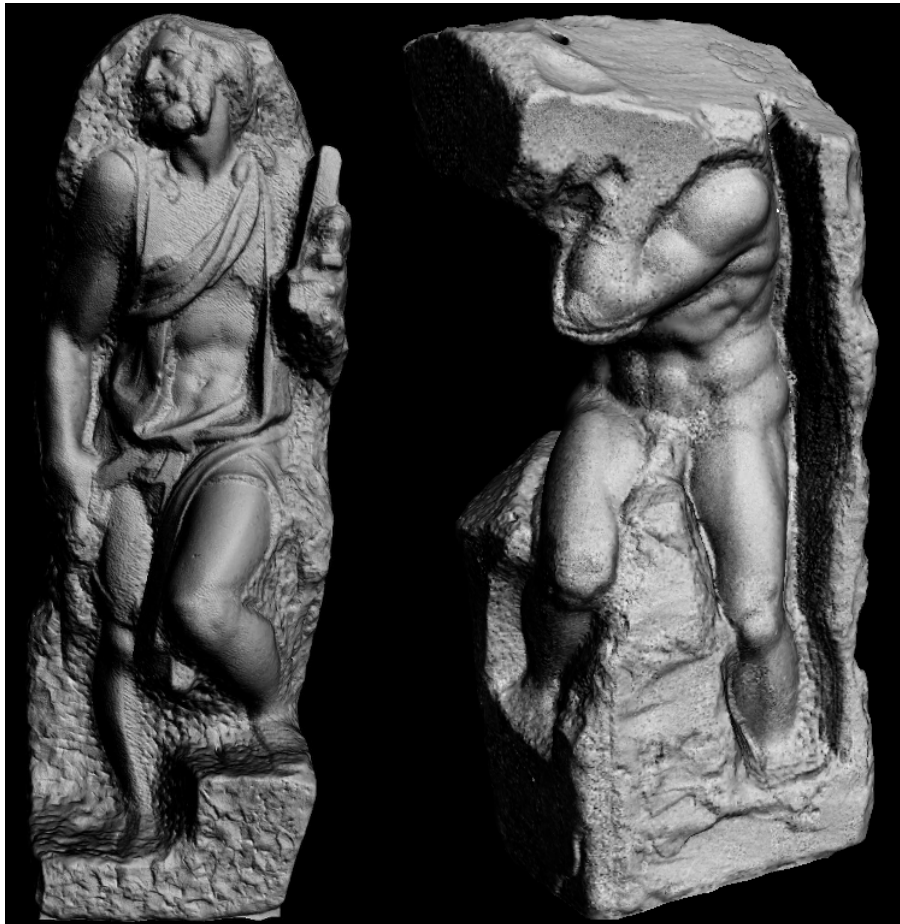

Figure 2.3: *The two largest publicly available sampled surfaces, the St Matthew (left) and the Atlas (right), are provided by the Digital Michelangelo Project and feature several hundred millions of samples. While their simple visualization is a already challenge, we will go a step further by allowing their interactive editing.*

Part I

Processing and Editing of Acquired Geometry

Chapter 3

Volume-Surface Geometry Processing

Hierarchical Space Subdivision Schemes (HS^3) are ubiquitous in computer graphics, and more particularly when efficient processing of acquired geometry is mandatory: simplification, reconstruction, compression, visibility, and many other processing steps are based on trees to partition and structure data sets. Their simple principle has made them popular: the initial space, often an axis aligned bounding box, is recursively subdivided until each cell satisfies a given error criterion. The root cell of the HS^3 can be either globally associated with the whole scene, or locally with each single object. Some of the most popular HS^3 are octrees, kD-Trees and axis-aligned BSP-Trees, which are easy to implement and to integrate in existing computer graphics frameworks.

Nevertheless, in the case of 3D surfaces, while HS^3 generate satisfying clustering at coarse subdivision levels, it is obvious that at finer levels, when the cells come closer to the surface, volume-based decomposition leads to imbalanced clustering in areas where the surface is not aligned with the main directions of the data structure (see Figure 3.1(a)).

(a) Octree clustering   (b) VS-Tree clustering
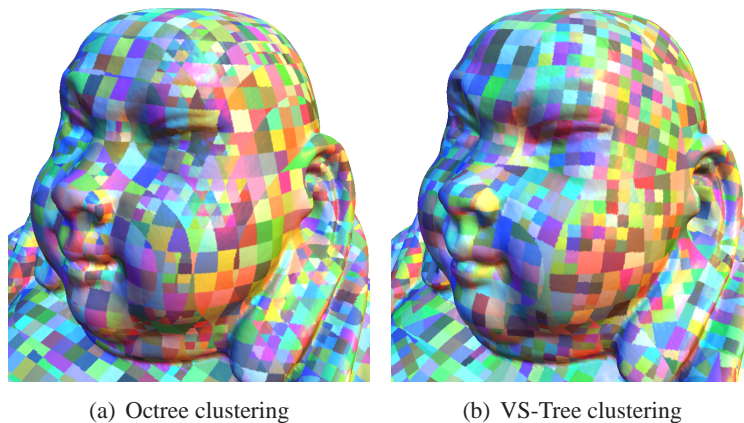
Figure 3.1: *Comparison between (a) octree clustering and (b) VS-Tree clustering. The local 2D scheme used by VS-Trees produces much better alignment of clusters and reduces the total number of clusters within a given error bound.*

In this chapter, we propose an alternative HS^3 which combines a 3D scheme for the first levels of the tree, and a 2D scheme as soon as the surface can be projected onto a plane without folding. We call such a tree a *Volume-Surface Tree* (or **VS-Tree**, for short). We show that VS-Trees achieve efficient

and elegant surface-based partitioning that can be applied to a variety of applications, such as *surface simplification* (Section 3.3) and *surface reconstruction* (Section 3.4).

3.1 Context: Simplification and Reconstruction

Hierarchical Space Subdivision Structures All HS^3 are based on a recursive subdivision of a root cell, as long as some user-specified criterion is not satisfied in every subcell. As outlined above, octrees, kD-Trees and BSP-Trees are by far the most popular HS^3. In the case of BSP Trees [FKN80], the space subdivision is dyadic, using a simple split plane, often chosen axis-aligned for the sack of efficiency. The kD-Tree data structure [Ben75] performs orthogonal space separation and stores additional data elements at internal nodes. Finally, quadtrees and octrees [JT80, Sam89], or more generally 2^d-trees, express the dimension of the subdivided space directly in their structure: a 1-to-4 scheme for quadtrees in 2D, and a 1-to-8 scheme for octrees in 3D, where an initial bounding cube is recursively subdivided in 8 equal cubes until satisfying a given criterion in each space partition, this criterion being related to the embedded geometry in our case. The very simple construction of the octree, as well as its fast convergence toward the shape of the embedded 3D surface, makes it very popular when geometry processing methods, such as *surface simplification* and *surface reconstruction*, need to be scaled toward large data sets.

Simplification by Clustering The goal of simplification methods is to reduce the resolution of an object, while maintaining as much detail as possible from the original shape [HDD*93, GH97, CSAD04]. Clustering methods are a particular subset of simplification techniques, which cast the problem as a partitioning problem, where each partition only keeps one single sample that minimizes the error, in a given metric [GH97, CSAD04], with the original surface. Hierarchical approaches, such as BSP-based methods [SG01] or octree-based methods [SW03], provide adaptivity in the surface partitioning. This adaptivity allows for more accurate simplification of non-uniformly sampled surfaces than regular grid partitioning methods [RB93, Lin00], while remaining almost as efficient. Such techniques have originally been developed for meshes, but they can also be directly applied on point clouds, when the sampling density is high enough [PGK02]. In practice, it appears that the quality of the mesh simplified by hierarchical clustering is strongly related to the subdivision scheme, and we will show how the local 2D scheme used by VS-Trees offers a much more regular sample decimation than the 3D scheme induced by octrees (see Section 3.3).

Surface Reconstruction To be as generic as possible, surface reconstruction techniques usually start from a sampling of the original surface in the form of a point cloud. Note that in addition to its position, each sample may also carry additional information, such as normal vector, that may (or may not) be exploited during the reconstruction. Since the seminal work of Hoppe et al. [HDD*92, HDD*94], various surface reconstruction methods have been proposed in the literature and it is out of the scope of this section to perform an exhaustive survey (see Section 2.1 for a brief summary).

Today's acquired point sets exhibits a sampling density that challenges reconstructions methods, so we focus on *speed-based* methods. Again, hierarchical data structures offer a simple and efficient framework to break the intrinsic complexity of surface reconstruction from dense point clouds. However, this induces a "divide-and-conquer" approach, that certainly speeds-up computation but also causes problems when a set of partial solutions have to be combined in a single surface. Therefore, implicit surfaces appear as the most suitable representation, since their volumetric definition can be easily obtained by simples operations (e.g., constructive solid geometry, polynomial blending) on many volumes.

For instance, an implicit surface reconstruction can be obtained by splitting the input point cloud with an octree, computing a separate implicit surface for each leaf of the octree, and finally *gluing* together the set of local implicit surfaces by using the *Partition Of Unity* method, where a compactly supported kernel weights the contributions of the different functions at a given point. This process has been successfully used for fast local polynomial fitting in the *Multi-Level Partition of Unity Implicits (MPU)* algorithm [OBA*03] as well as for *Radial Basis Functions* [TRS04]. Unfortunately, as usual with implicits, an explicit solution has to be provided at the end for processing and rendering purpose and the reconstructed implicit surface is converted into a mesh, which involves an expensive tessellation step [Blo94]. Moreover, the quality of the resulting mesh is generally poor (the grid-surface intersection involved in 3D contouring cannot output well-shaped triangles) and has to be improved using, for instance, an additional remeshing step, either based on parameterization [LSS*98], global optimization [Bot05] or fitting of subdivision surfaces [EDD*95, ZSS97, MK04]. Consequently, even if the computation of the implicit surface is efficient thanks to the space subdivision, the whole reconstruction process including the generation of an high quality semi-regular mesh becomes rather expensive.

In fact, surface reconstruction remains an ill-posed problem, even with slow global solutions, and we often resort to a painful try-and-test session. Thus, we propose in Section 3.4 a performance-oriented surface reconstruction technique that takes fully benefit of the volume-surface organization of the VS-Tree, to generate a semi-regular mesh of arbitrary genus over an unorganized point-cloud, dealing both with noise and non-uniform sampling. This algorithm is fast enough to be integrated in an acquired geometry processing pipeline, offering good results in most cases and letting ultimately the choice to users to switch to a slower techniques in difficult cases (e.g., large holes to fill, under-sampled surfaces).

3.2 Volume-Surface Tree

In this section, we introduce the VS-Tree as an hybrid hierarchical partitioning performing a volume-surface decomposition in its structure.
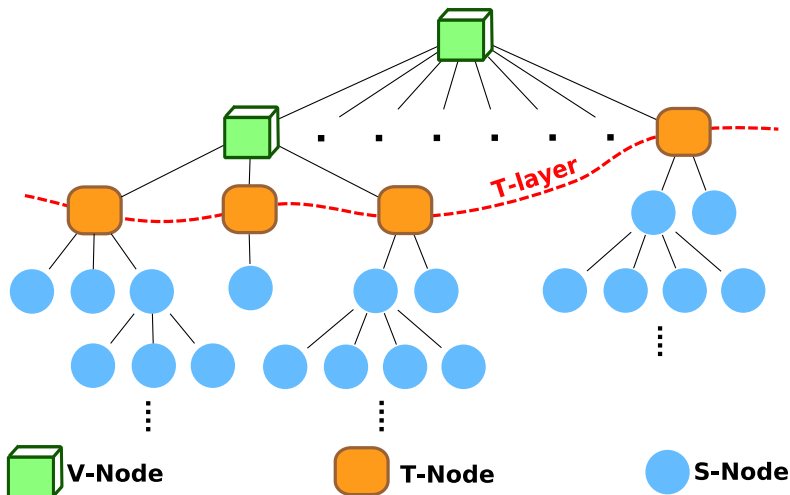
3.2.1 Definition

Figure 3.2: *The VS-Tree structure. Note in red the T-layer, capturing a 3D-2D interface in the hierarchy.*

26

A VS-Tree is a *surface-based HS*[3]. The basic idea is to combine an octree and a set of quadtrees to describe a discrete 3D surface. During the recursive split involved in the octree construction, we switch to a quadtrees as soon as the area of the surface associated with the current node is consistent with a scalar-valued function over a given ground plane (in other words, a *height field*). Figure 3.2 presents the three different kinds of VS-Tree nodes:

- **Volume Nodes** (V-Nodes): comparable to octree nodes. Each V-Node has 0 or 8 children, which can be V-Nodes or T-Nodes.

- **Transition Nodes** (T-Nodes): leaves of the 3D hierarchy which also are roots of the 2D hierarchies. Each T-Node has 0 or 4 children that are S-Nodes.

- **Surface Nodes** (S-Nodes): comparable to quadtree nodes. Each S-Node has 0 or 4 children that are S-Nodes.

Note that each T-Node carries a local frame that is used to align its corresponding sub-quadtree. The union of all T-Nodes defines the volumetric layer under which it becomes possible to implement 2D algorithms (see Figure 3.2); we call it the *Transition-layer* or **T-layer**.

VS-Trees are proposed in order to increase efficiency of geometric processing usually combined with simple and efficient hierarchical structures such as octrees. In order to maintain a behavior as similar as possible to octrees, the ideal structure should have the following properties:

- Purely recursive construction: popular hierarchical structures have the strong advantage to be instanced through a simple recursive call, which is easy to implement;

- Efficient construction: rigid organization of data, such as the 1-to-8 split of octrees, allows efficient traversal and refinement of an hierarchical structure;

- T-layer at low depth: switching to quadtrees as soon as possible reduces the memory overhead thanks to the 2 dimensional structure, and speeds-up traversals and tests. Inclusion tests for arbitrary points are performed in 2D using the quadtrees placed under the T-layer;

- Graceful degradation: in the worst case of very small or under-sampled topological features, such as iso-surface extraction from physical simulation, the structure should behave no worse than an octree.

3.2.2 Construction

There are a large number of possible 3D surface decompositions that lead to a collection of 2.5D pieces. We propose to use the following simple recursive construction method that is easy to integrate in existing application software.

Input   Let S be the set of samples defining the input surface. Each sample s_i of S is defined by a position p_i and a normal vector n_i. For dense meshes, S can be chosen, for instance, as the original vertices of the mesh, or as the barycenters of the polygons. S can also be a point-based surface, with normals approximated with a Principal Component Analysis (PCA) [HDD*92, GKS00] if not available.

Clustering   The construction of a VS-Tree begins with the computation of a bounding box B of S recursively subdivided with a 1-to-8 octree scheme. At each level, the current set of samples S_i associated

with the bounding box B_i at that level is classified against each child's bounding box. Let κ be a *height field indicator*, signaling whether S_i is consistent with a height field (i.e., it is 2.5D rather than 3D). This flag stops the recursive 1-to-8 subdivision process. When $\kappa(S_i)$ is true, the current node is set as a T-Node, and a local coordinate frame is computed. This local frame will strongly influence the final quality of the clustering, and must be carefully chosen. While it is generally impossible for a hierarchical structure to precisely recover all the anisotropic features present in the discrete surface, a well-aligned sub-hierarchy can often be computed by analyzing the underlying surface and considering its main directions (see Figure 3.1 and Figure 3.4). Thus, for constructing this local frame, we use a PCA on S_i, but rather than considering positions of samples [HDD*92, GKS00], we use their associated normals, a more relevant information when clustering surfaces [CSAD04].

Since we are looking for directions, we can perform the PCA in the normal space of S_i. The set of resulting eigenvectors is a good approximation of the tangent frame of the surface. We choose $\{n_i, u_i, v_i\}$ as a local frame, where n_i is the average normal of S_i, while u_i and v_i are the normalized projections of the two eigenvectors that minimize the dot product with n_i onto the plane Π_i defined by n_i and c_i (the centroid of S_i).

The set of samples S_i associated with the T-Node T_i is projected on Π_i. Finally, a bounding quad is computed for S_i and is recursively subdivided with a 1-to-4 quadtree scheme. The recursion is stopped when the error, computed over S_i, is below a threshold. Figure 3.3 shows the different steps involved in this construction. Note that the T-layer becomes independent of the discrete surface resolution when the sampling density is sufficient: typically, over-tessellating a dense mesh will not change the depth of the T-layer.

Height field indicator Evaluating if a piece of surface will exhibit folding during a lower dimensional projection can be done by numerically integrating the curvature over this area. Nevertheless, such a test is computationally expensive even in the case of regular meshes, and more complicated for non-manifold meshes or topology-free representations such as point clouds. In order to make our approach more general and efficient, various heuristics can be used to define the height field indicator κ for such a predicate. Pauly et al. [PG01] propose a normal-cone test for allowing the projection of a set of surfels using the miniball algorithm. We extend this idea by introducing an additional displacement threshold to detect scan misalignment in dense acquired point sets. Although a formal proof is not available, since it would depend on some form of density and/or topology criterion (not available in most practical cases),

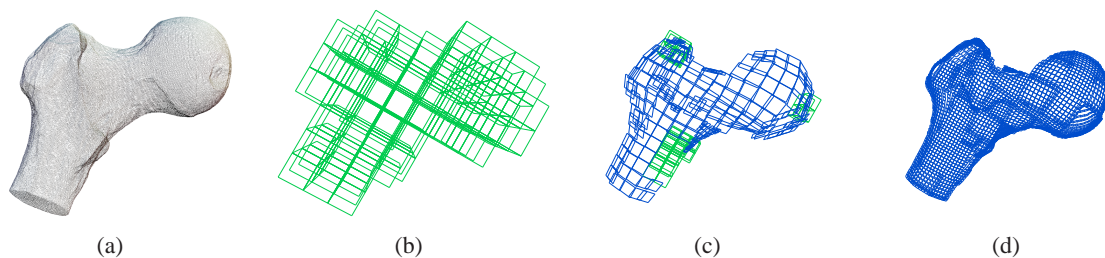
(a)   (b)   (c)   (d)

Figure 3.3: *Different levels of a VS-Tree. (a) The input discrete surface. (b) The upper levels of the tree are three-dimensional (in green). (c) The transition between 3D and 2D structure (in blue) is possible as soon as the surface can be locally expressed as a height field. (d) The lower levels of the VS-Tree are two-dimensional.*

Models	Samples	Octree	VS-Tree	Gain
Feline	49864	0.18s.	0.19s.	18%
Igea	134346	0.33s.	0.34s.	49%
Vase lion	200002	0.83s.	0.80s.	18%
Raptor	1000080	3.50s.	3.11s.	39%
XYZ dragon	3609601	11.82s.	9.88s.	52%
XYZ Statue	5000000	17.82s.	14.90s.	32%

Table 3.1: *Computation time to generate the HS^3 with L^2 error bounded at 10^{-4}. The gain is relative to the final number of partitions.*

this indicator κ gives convincing results in practice. So we define κ to be *true* when:

$$\forall j \in [0, k_i[\quad \begin{cases} n_{ij} \cdot n_i > \delta_a \text{ with } \delta_a \in [0,1] \text{ and} \\ \frac{|(p_{ij}-c_i) \cdot n_i|}{\max_{k_i}(||p_{ik_i}-c_i||)} < \delta_d \text{ with } \delta_d \in [0,1] \end{cases}$$

where k_i is the number of samples of the current cell i, n_i the average normal of the surfels in the cell, p_{ij} and n_{ij} are the position and the normal of the jth sample of the cell i. δ_a (angle deviation) and δ_d (displacement deviation) are user-provided thresholds. In our implement, $\delta_a = 0$ and $\delta_d = 1/6$ has provided satisfying results in all our tests. Note that by increasing δ_a and decreasing δ_d, it becomes harder for κ to be true, and thus the T-layer is conservatively dropped to a lower level of the hierarchy.

Error metrics   As usual with HS^3, an error metric L can be be defined to control the recursive subdivision with a simple geometric analysis. Good error functions should be monotonic and decreasing with the size of S_i. Obviously, any error metrics can be used with VS-Tree. Yet, we use a reduced set of such metrics, which have proved their qualities. In particular, we use the L^2 error function, which ignores small-scale high-frequency features in the partitioning, and which is discretized on sampled surfaces by:

$$L^2(S_i) = \sum_j ||p_{ij} - \Pi_i(p_{ij})||^2$$

with $\Pi_i(p_{ij})$ the orthogonal projection of p_{ij} on some average plane related to S_i (e.g., least square or $\{c_i, n_i\}$). We use also the Quadratic Error Function (QEF) introduced by Garland [GH97] for better capturing curved smooth surfaces. Last, as normal is a very relevant property of smooth surfaces, we often use the normal-based $L^{2,1}$ metric [CSAD04], which also better captures anisotropy:

$$L^{2,1}(S_i) = \sum_j ||n_{ij} - n_i||^2$$

More complex combined metrics, such as the Sobolev one, may also be used. In the case of large objects, simple approximated metrics, such as the local density, may be chosen for efficiency.

3.3   Rapid Simplification

In this section, we show how the VS-Tree structure improves prior surface simplification algorithms based on *hierarchical vertex clustering*.

3.3.1 Balanced clustering

Figure 3.1 illustrates the difference of vertex clustering obtained with an octree and a VS-Tree. The volume-based behavior of octree decomposition frequently leads to very imbalanced clustering, mixing small clusters (when the surface is located near the corner of the octree cell) and large ones (when the surface passes near the center of the octree cell). Moreover, the *cuts* generated by the octree cell boundaries can be clearly identified within the clustering (see Figure 3.1(a)). VS-Tree decomposition strongly reduces both artifacts, as it provides a much better alignment of the cluster boundaries with the embedded surface (see Figure 3.1(b)). A very low variance can be observed in the size of the clusters, basically because the clustering only depends on the planarity, but not on the orientation, of the surface locally associated with each T-Node. For instance, the variance in the number of samples per cluster has almost been divided by 2 between Figure 3.1(a) and Figure 3.1(b). Additionally, an almost regular quad-like clustering can be observed. The few remaining non-quad clusters primarily come from the volume-based decomposition created at the top levels of the VS-Tree.

Figure 3.4: *Hierarchical mesh simplification with L^2 error bounded at 2.10^{-3}. Left: Original object (7M triangles). Middle: Octree simplification (1.75 sec. - 62856 triangles). Right: VS-Tree simplification (1.20 sec. - 52024 triangles).*

3.3.2 Computation efficiency

In addition to providing more balanced clustering, the VS-Tree is also more efficient than the octree when computing the HS^3. Moreover the advantage of the VS-Tree over the octree increases with the size of the input data, as shown on Table 3.1. For large objects, a 16% improvement can be observed in the computation time, as well as a reduction of the number of clusters between 18% and 52% for the same bounded error. This may appear quite surprising as octree decomposition is generally considered extremely efficient. In fact, the speedup observed by VS-Tree decomposition comes from two different properties. First, when the size of the input data increases (e.g., very dense meshes), most of the data will be represented below the T-layer, and thus 1-to-4 splits will be much more frequent than 1-to-8 splits. To reach a given error threshold, the octree is thus usually much deeper, with significantly more empty cells compared to the corresponding VS-Tree. Second, below the T-layer, all the computations involved in the VS-Tree are done in 2D. When there is a large number of points in the sub-hierarchy of a given T-Node, these 2D computations more than compensate for the overhead involved in the projection to the local 2D frame.

3.3.3 Mesh simplification

We have implemented an hierarchical mesh simplification algorithm based on vertex clustering in the VS-Tree. The algorithm proceeds as follow:

1. a VS-Tree is computed for the input object,

2. all vertices are classified according to the S-Node they intersect (hierarchical test),

3. a representative vertex is computed for each leaf (centroid or QEF [GH97] origin for instance),

4. triangles that have their three vertices in different S-Node are re-indexed over the relative representative, others are discard.

Here again, the more balanced cluster sizes provided by the VS-Tree reduce the mismatch of features for a given error threshold, without imposing an overly conservative mesh density. Moreover, the local frame computed independently for each T-Node roughly captures the anisotropy of the underlying mesh, while the octree completely ignores it. For instance, see the cheek on Figure 3.4. As expected, the VS-Tree introduces fewer clustering artifacts in the mesh topology, and better captures the original geometry (see near the eye, for instance).

3.4 Fast Surface Reconstruction by Refinement

In this section, we describe a new efficient surface reconstruction algorithm based on the specific volume-surface decomposition of VS-Trees.

(a)   (b)   (c)
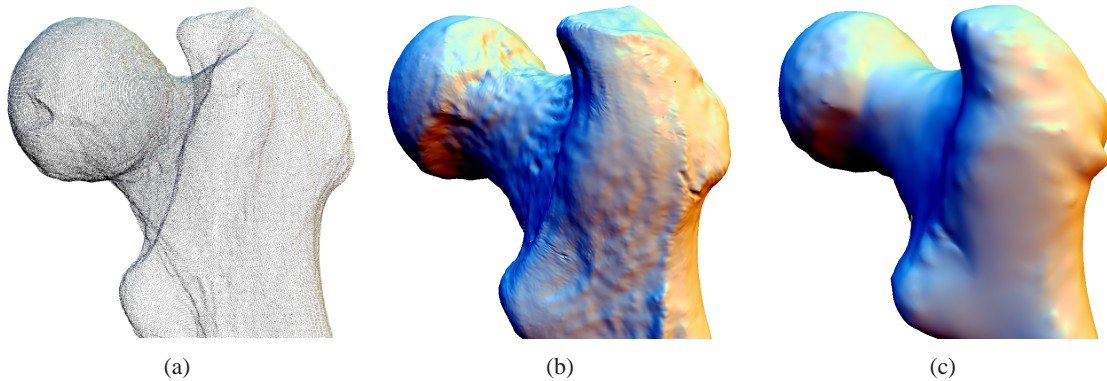
Figure 3.5: *Noise filtering. (a) Input point cloud (137063 samples). (b) Reconstruction with VS-Tree L^2 error bounded at 10^{-4} (1.758 sec, 125K triangles). (c) Reconstruction with VS-Tree L^2 error bounded at 10^{-3} (0.987 sec, 32K triangles).*

Obviously, meshes have become the de-facto standard for 3D geometry processing and rendering and we seek for a robust and efficient point-to-mesh surface reconstruction techniques. Several properties for such a reconstruction processes are mandatory: (1) dealing with arbitrary genus; (2) offering intuitive de-noising control; (3) avoiding final remeshing by directly providing a semi-regular mesh; (4) providing error controlled output; and, of course, (5) being as efficient as possible. We propose to use the advantages of the VS-Tree decomposition in order to develop a point-to-mesh reconstruction technique that fulfills these five properties.

Intuitively, most of the global topological features of the surface can be recovered at the T-Layer of the VS-Tree. Thus, the T-Layer helps us to split the problem into two main steps: a coarse mesh M_0 is generated during the first step, and refined during the second step, to account for all the details included in the input point cloud. This second step uses a displacement process driven by the quadtree corresponding to each T-Node. Algorithm 1 summarizes our approach.

Algorithm 1 Surface reconstruction using VS-Trees.

Require: PointSet S, Threshold t
  VSTree T ← buildVSTree(S)
  Mesh M ← extractMeshAtTLayer(T)
  while error(M, S) > t **do**
    M ← refinePN(M)
    M ← displace(M, T)
  end while
  return M

3.4.1 Base Domain Reconstruction

Globally, we follow the construction process presented in Section 3.2. Similar to Pauly et al. [PGK02], the high frequency noise typically present in scanned data [NRDR05], is directly addressed at the *point* level by simply specifying an L^2 error threshold driving the VS-Tree creation. While more formal noise removal solutions exist [PG01, SFS05], this simple technique nicely smoothes out the noise, as shown on Figure 3.5, and is intuitive enough to be easily tuned by the user.

Base mesh reconstruction: The remainder of the algorithm will inherit the global topology of M_0, and in particular its genus. Since the geometry of S-Nodes does not change the global topology of the shape, M_0 is created using only the T-Layer (see Figure 3.6(a)). However, the set of T-Nodes composing the T-Layer can be sparse (e.g., large areas with low curvature), which does not allow the use of Delaunay-based reconstructions for this base-mesh. Moreover, ideally, we would like a watertight 2-manifold, homomorphic to the input point-based surface. This naturally leads us to choose a simple implicit surface reconstruction defined, by a function $f : \mathbb{R} \to \mathbb{R}^3$, by just considering the half space defined by the oriented frame of each T-Node T_i (i.e., a linear polynomial acting as a distance function) and contouring it in similar fashion to Hoppe et al. [HDD*92]. However rather than directly contouring this simple localized distance function with a marching cube algorithm, we rather construct a smooth implicit surface using a *Partition of Unity* scheme:

$$f(p) = \sum_i \phi_i(p) Q_i(p)$$

with $\phi_i(p)$ a *Partition of Unity* kernel centered on T_i and

$$Q_i(p) = (p - c_i) \cdot n_i$$

the signed distance to its average plane, with $\{c_i, n_i\}$ the support plane of T_i (find using κ). The octree structure of the upper levels of the VS-Tree allows consistent generation of overlapping zones that can be used to blend the local distance functions, in a similar fashion as in the work of Ohtake et al. [OBA*03]:

$$\phi_i(p) = \frac{\omega_i(p)}{\sum_j \omega_j(p)} \quad \text{with} \quad \omega_i(p) = h\left(\frac{3|p - b_i|}{2r_i}\right)$$

with b_i the center of the T_i cell and r_i the radius of its bounding sphere. We replace the quadratic kernel proposed in the work of Ohtake by an Hermitian one, $h(t)$, for its better vanishing behavior (see

Section 4.4.1). The mesh is then generated by applying a Bloomenthal polygonization [Blo94]. In order to guarantee that no topological feature of the VS-Tree will be missed, we use a dual contouring grid and set its resolution to that of the deepest T-Node (see Figure 3.6(b)). Note that there is room for improvement here with the octree countouring methods recently proposed [SJW07, KKDH07]. Finally, this mesh is simplified by clustering it at the T-Layer level. This leads to the final coarse mesh M_0, which contains only one vertex for each T-Node (see Figure 3.6(c)).
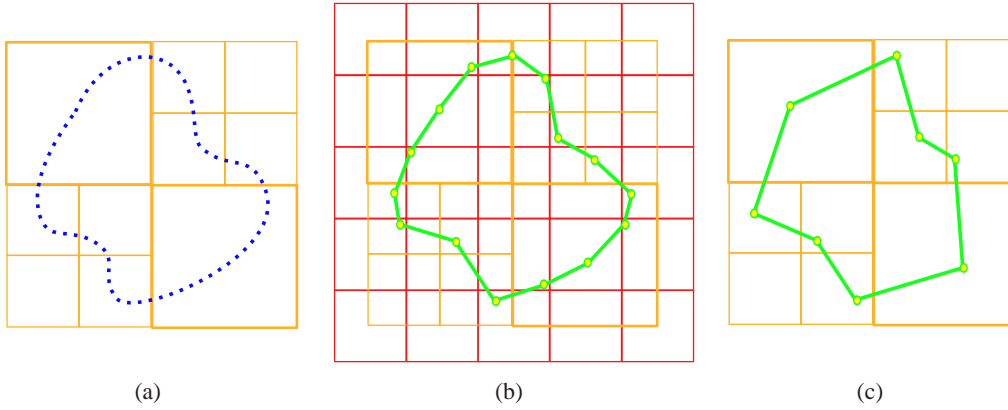

(a) (b) (c)

Figure 3.6: *Coarse mesh generation. (a) Input point cloud (in blue) clustered in a VS-Tree (T-Layer in orange). (b) Marching cube dual contouring at the resolution of the deepest T-Node (red grid). (d) Coarse mesh M_0 (in green) generated by simplifying the mesh at the T-Layer level.*

3.4.2 Mesh Refinement

The goal of this second stage is to iteratively refine the mesh, until the geometric features of the input point cloud are recovered according to a given error threshold. For triangular meshes, the approximating subdivision scheme proposed by Loop [Loo87] is known to provide high quality mesh refinement. But in our quest for efficiency, we need to find a trade-off between speed and quality. We have found that local subdivision based on *Curved PN-Triangles* [VPBM01] are well suited to our constructive approach. This leads us to the following efficient two-step refinement technique:

1. each triangle of the mesh M_i is refined into four sub-triangles and the newly inserted mid-edge vertices are translated according to the cubic Bezier triangular patch computed by the PN-Triangle scheme;

2. these three mid-edge vertices are translated to their final position, according to the geometry stored in the local quadtree (see Figure 3.7).
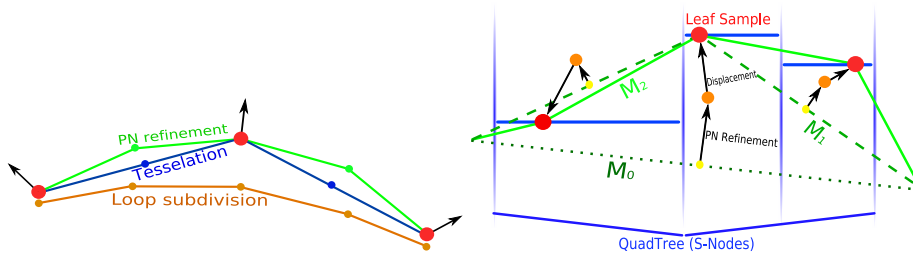

Figure 3.7: *Top: vertex insertion comparison. Bottom: VS-Tree refinement and displacement in a T-Node.*

This displacement procedure is the step that benefits most from the specific properties of our VS-Tree decomposition. Instead of having to define a smooth scalar field such as in implicit surface reconstruction methods, or a robust energy functional such as in dynamic model fitting [DYQS04], we simply use the quadtree defined at each T-Node to displace the inserted vertices accordingly. Let v denote an inserted vertex that has to be displaced. First, we find the highest S-Node s that only contains v. Then, we select the leaf l exhibiting the highest local variation in the quadtree built on s. Finally, we translate v toward the average sample carried by l. We mark l as locked, and will no more consider it for future displacement steps: as PN-Triangles provide an interpolating scheme, this vertex is now interpolated until the end of the refinement loop. This simple construction approximates the optimal displacement of v and avoids the mismatch of high-frequency features that would occur if a simple orthogonal displacement was performed (see Figure 3.7).

At each refinement step, the mesh is maintained *hole-free* since we only translate its vertices. In order to avoid the *surface aliasing* effect that could occur when many vertices are projected toward the same leaf, we do not displace v when no more leaf remains *unlocked* in the quadtree built on s. After each displacement pass, newly inserted vertices that have not been displaced are smoothed out according to the final position of neighbor vertices, using a simple tangential smoothing. Since the PN-refinement performs a 1-to-4 subdivision, each vertex v has at least two neighbors that have already been processed at a previous refinement step, and thus have reached their final position.
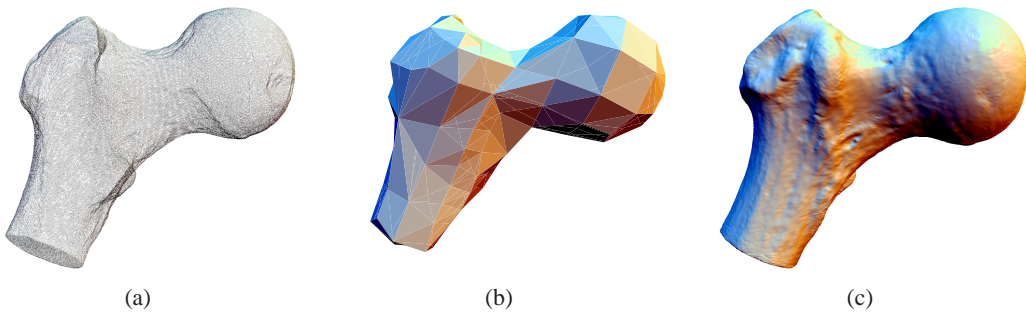

(a)   (b)   (c)

Figure 3.8: *Reconstruction of the ball-joint model (*137062 *points,* 1.758 *sec). (a) Input point set (b) Coarse mesh generated at the T-layer of the VS-Tree. (c) Final refined mesh.*

3.5   Results

Adaptivity to curvature variation   In the case of point clouds sampled from a surface that exhibits large variations of curvature, one may think that an adaptive refinement scheme [ZSS97] would allow a better capture of the global shape. However, both the efficiency of vertex insertion, as well as the final semi-regular topology of the mesh, would be lost by such an adaptive refinement. Efficient adaptivity to curvature variation can be easily included in our scheme by letting the user tune the δ_a and δ_d thresholds used in the height field indicator κ. Indeed, increasing δ_a and decreasing δ_d induces a deeper T-layer in high-curvature areas and thus, a larger number of T-Nodes. Since, M_0 is generated by T-Node clustering, M_0 is itself denser, leading to a final mesh with higher resolution in high-curvature areas (see Figure 3.10(a)). Although this solution may break down for some pathological cases, it remains far less expensive than, for instance, the optimization of the L^∞ error [MK04]. Figure 3.5, 3.8, 3.9 and 3.11 shows some examples of surface reconstruction obtained with our algorithm.

(a)   (b)   (c)
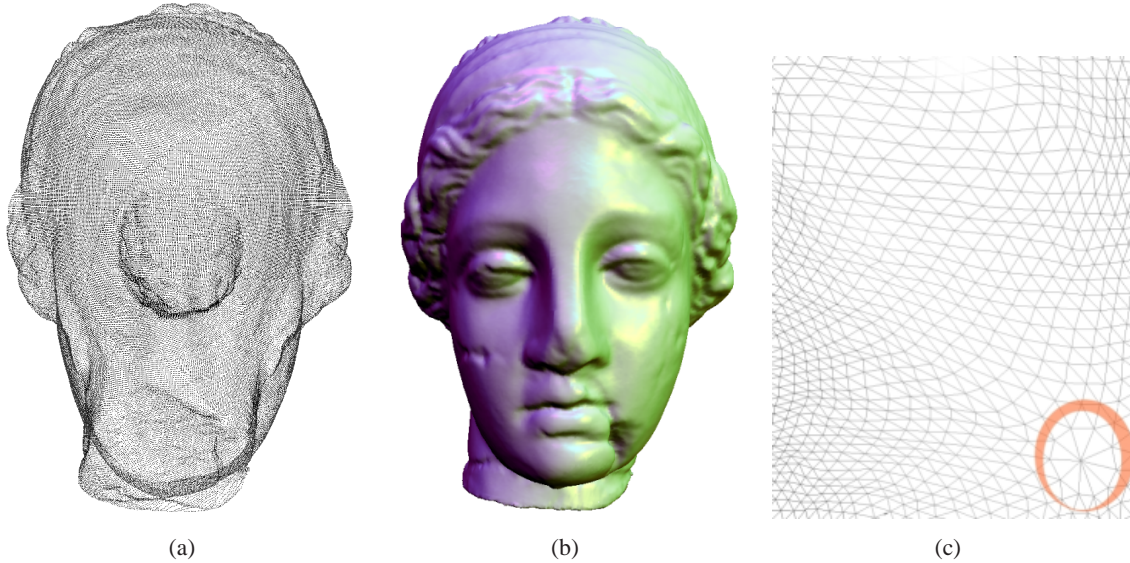
Figure 3.9: *Reconstruction of the Igea model. (a) Input point set. (b) Reconstructed surface (c) Close-up on the semi-regular mesh produced by our algorithm. Note, in the red circle, the limit of our technique which propagates high-degree vertices generated on M_0.*

Performances Table 3.2 provides some reconstruction timings for various models. The timing presented includes the VS-tree decomposition, the coarse mesh generation and the mesh refinement loop. Globally, this new algorithm is one order of magnitude faster than state-of-the-art *fast* surface reconstruction methods [OBA*03, GKS00], while directly providing a final mesh with semi-regular connectivity without any additional remeshing steps. For large point clouds, the VS-Tree construction becomes the bottleneck, since this is a non-linear operation. Figure 3.10(b) compares the final mesh quality of [OBA*03] to ours. In our implementation, the intensive use of pointers limits the size of in-core reconstruction. We are currently exploring out-of-core methods for performing the reconstruction with a constant and small amount of memory.

The mesh quality obtained by our technique is much higher as the one obtained by applying some octree-based tessellation on a reconstructed implicit surface (see Figure 3.10(b)) and approaches the quality obtained by mesh beautification techniques. However, they exhibit a few more extraordinary vertices, resulting from the initial clustering at the T-Layer level of the M_0 (see Figure 3.9). Nonetheless, it should be noted that the refinement process *does not* generate additional extraordinary vertices. So, if limiting the number of such vertices really matters for some specific application, one easy solution would be to apply mesh beautification on the coarse mesh M_0, which of course is dramatically faster than applying remeshing on the final dense mesh.
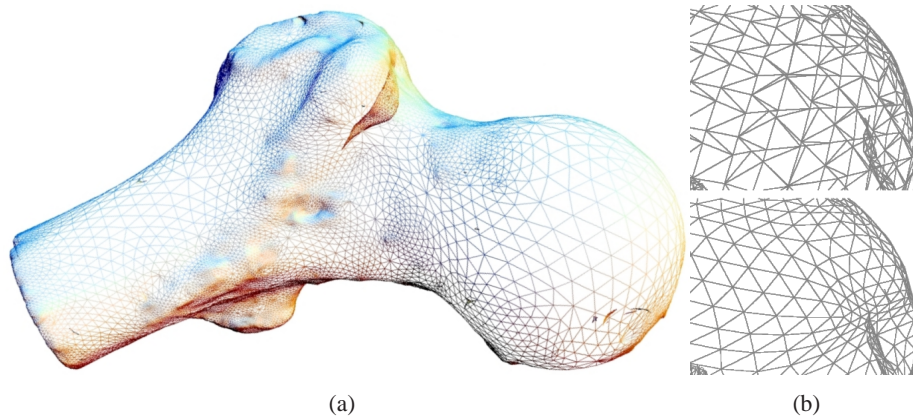
Figure 3.10: *(a) Adaptivity by T-Layer constraint. (b) Close-up of the mesh obtained when reconstructing the ball-joint model with the* Multiple Partition of Unity *(top) method with our VS-Tree based method (bottom).*

As said previously, even if our reconstruction technique generates high-quality meshes in almost every tested examples, we have biased each speed vs. quality tradeoff of our algorithm towards computation efficiency. Consequently, for difficult examples that exhibit poorly sampled areas with high curvature, slower reconstruction techniques based on implicit surfaces [OBA*03, TRS04], usually offer better recovering of thin features.

3.6 Discussion

Limitations: In spite of its efficient hierarchical analysis, the VS-Tree offers only a sub-optimal height-field decomposition. Its intrinsec 2^d-tree structure may impose over-splitting when searching lower-dimensional clusters. One typical example is the sphere, which will be cut in 8 pieces before switching to a 2D partitioning, while it is well know that the tetrahedron (i.e., 4 clusters) is the smallest non-degenerated polyhedra onto which a sphere can be projected. In fact, when an optimal height-field decomposition is mandatory, there is no efficient solution and one would resort to a global analysis, using for instance Mean Shift or K-Means methods. Nonetheless, there is clearly room for research in improving aggressive height-field indicators for (point-)sampled surfaces.

Summary: Hierarchical space subdivision schemes are the key ingredient to make efficient geometric processing methods in a large number of situations. In this chapter, we have proposed the VS-Tree, a surface-based partitioning structure combining an octree with local quadtrees. This simple structure improves the quality of simplified meshes generated by vertex-clustering, while maintaining similar computation time compared to conventional octrees. It can seamlessly replace octrees in a variety of situations, providing better results when dealing with dense surfaces.

We have also proposed an efficient point-to-mesh surface reconstruction algorithm based on the VS-Tree data structure. This algorithm combines the robustness of an implicit approach, for recovering the global topology of the surface using the upper levels of the VS-Tree, with the efficiency of an explicit one, for retrieving local geometric features by a simple and efficient local displacement scheme induced by the lower levels of the VS-Tree. As a result, manifolds of arbitrary genius can be reconstructed avoiding the computationnal effort involved with multiple polynomial fitting of a complex local geometry.

Perspectives: This structure and these algorithms can take place in in the 3D acquisition pipeline. In the following chapter, we will discuss the use of the VS-Tree in an out-of-core system for what usually follows the (semi-)automatic processing of acquired geometry: *interactive editing*.

Models	Samples	Our method.	MPU.
Bunny	35949	0.852s.	4.272s.
Dino	56195	1.026s.	5.010s.
Santa	75783	1.067s.	7.135s.
Igea	134346	1.813s.	6.890s.
Male	303382	2.798s.	55.008s.
Dragon	437647	5.400s.	60.176s.
Happy Buddha	543652	6.384s.	80.866s.
Youthful	1728305	20.621s.	200.527s.
XYZ Dragon	3609601	41.844s.	480.693s.
XYZ Statue	5000000	53.298s.	475.551s.

Table 3.2: *Timing for VS-Tree surface reconstruction (with error set to 5.10^{-4}) and MPU (with error set accordingly).*
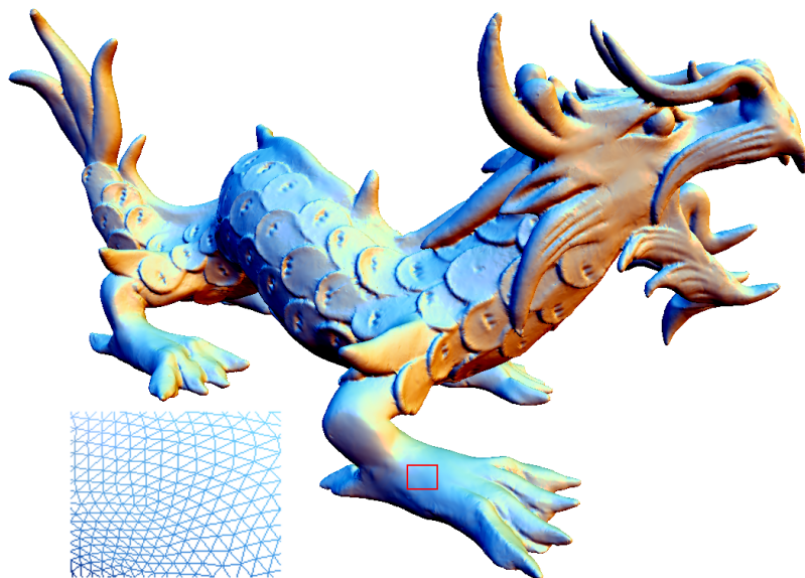

Figure 3.11: *Reconstruction of the XYZ Dragon model (3.6M points - 53.298 sec). Close-up on the semi-regular mesh produced by our algorithm.*

Chapter 4

Size Insensitive Interactive Editing

The previous chapter provides essential tools for semi-automatic processing of acquired surfaces. Combined with filtering tools, such as the MLS projection [ABCO*01], point clouds and can be simplified and reconstructed efficiently. However, the *ground truth* provided by the point sampling may need to be edited prior to any decision about its simplification and reconstruction. In particular, *appearance texture* and *shape deformation* may need to be applied. These two operations are the most common ones in the computer graphics industry, and both share a central constraint: they require an interactive response from the application.

In fact, interactive editing differs from the kind of algorithms described in Chapter 3, because there is not any a priori result: the user interactively explores the space of possible appearances and shapes, progressively refining its own idea on the result until obtaining the final 3D object, ready for animation and rendering. In this situation, the whole editing session must remain interactive, with enough frames-per-second for letting the user focus on his work rather than waiting for the computer. Observing state-of-the-art software tools, it appears that this **crucial** interactivity cannot be provided for dense surfaces, such as the ones coming from the acquisition pipeline: even few million polygons represent a bottleneck for rich surface painting and deformation tools.

Nevertheless, controlling interactively the shape and the appearance of surfaces, including acquired ones, is unavoidable in many computer graphics fields, for being able to adapt or modify real-world objects for a particular application. So far, the usual solution is to simplify the model, and then to edit it. In other words, the resolution and the accuracy of 3D surfaces does no more depend on the quality of the acquisition devices and its original precision, but depends on the capabilities of the software and the computer used for editing it. In practice, this means that, as simple has may be the modification (e.g., bending the arm of a virtual human), the simplification process has to be applied, loosing features originally acquired. This explains the difference we can observe between rendering of static shapes and animated ones for instance: the former have been preserved and finally rendered out-of-core, while the latter have undergone simplification for texturing and deformation reasons.

In a modern 3D content processing pipeline, we consider that the full scale geometry should be maintained as far as possible, being simplified only at the end of the pipeline, if required. Thus, we propose a new approach to 3D surface manipulation, the *sampling-reconstruction method*, for editing interactively both shape and appearance, whatever the size of the input/output object. The key idea behind this system can be stated as follow:

A modification of the sampling can be considered as a sampling of the modification

In other words, we claim that, for editing the shape and the appearance of a large object, we can:

- sample it at a lower resolution,

- edit interactively its shape and appearance,

- reconstruct a function that transfer these modifications to the original object, obtaining an edited large object.

So formulated, scalability becomes possible: both sampling and reconstruction are performed with *out-of-core streaming* algorithms, which makes them scalable to arbitrary object size, the intermediate sampling representing the only memory footprint required.

Our system is able to take either point clouds, polygon soup or manifold meshes as an input, and allows to use a large variety of popular editing method onto large objects (minimal assumption are made on the interactive editing system used). It is *size-insensitive*, upon a billion sample in our tests. Our framework is actually organized as a bracketing system for any interactive editing tool, and is based on two fundamental *stream* algorithms: an **out-of-core simplification preprocess** and an **out-of-core reconstruction post-process**. The former extracts a sampling from the original object, while the latter transfers the modification (color and deformation) undergone by this sampling to the original (large) model. The size of the sampling is chosen according to workstation capabilities and user demand (see Figure 4.1).
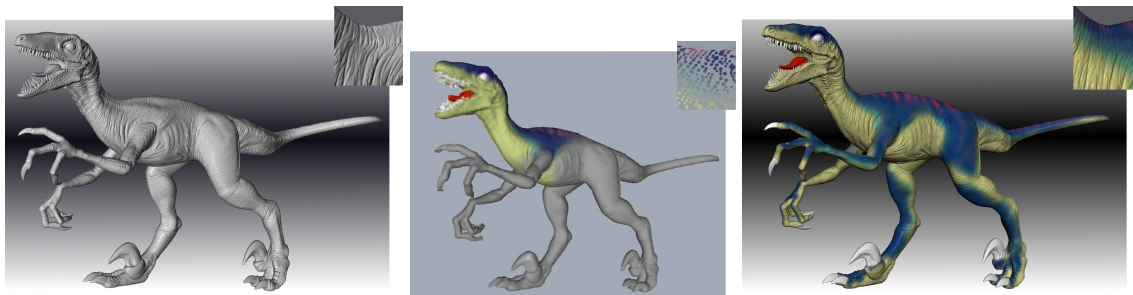
(a) Color Editing
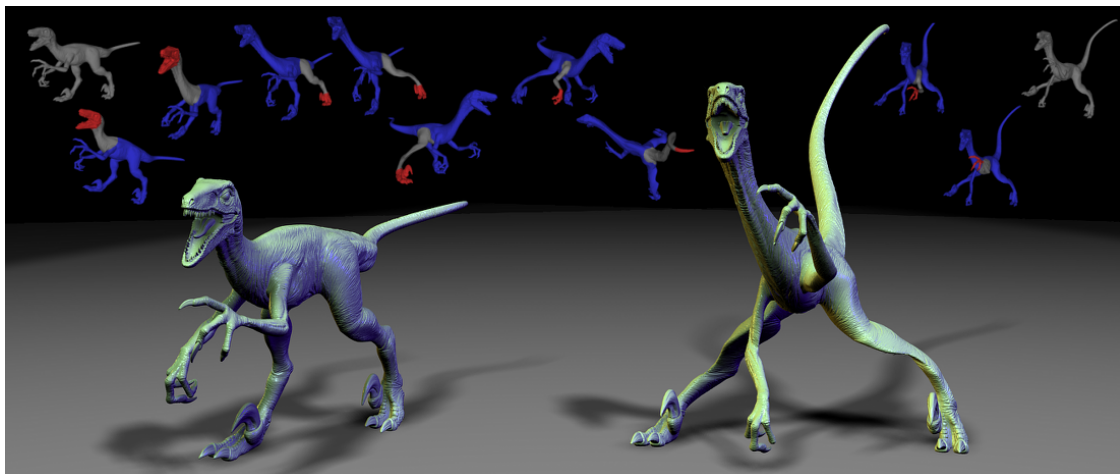
(b) Shape Editing

Figure 4.1: *Interactive editing of large objects: the Raptor model (8M triangles). The initial geometry is adaptively downsampled through a streaming process, to get a simplified meshless version (146k points) that can be edited interactively with on-demand local refinement (blue snapshots). Afterwards, a second streaming process performs a color and deformation transfer to the original geometry.*

4.1 Context: Interactive Manipulation of Large Objects

We focus on the interactive manipulation of the two essential properties of a 3D object: appearance and shape.

About the former, we propose an interactive painting tool: color information gained from acquisition is usually hard to exploit in final images because appearance is a view-dependent property. Note that even if some scanners provide the color information during the scanning process, this information can rarely be directly exploited since the captured appearance strongly depends on the lighting conditions. Actually, similarly to the geometry acquisition pipeline that we discuss in this thesis, *appearance acquisition methods* exist: we refer the reader to the thesis of Goesel [Goe04] for a recent survey. Unfortunately, such systems are complex and cannot be used in all the situation where 3D scanners are useful. As a result, appearance (e.g. color texture) has to be partially or completely edited afterwards.

Concerning shape, the 3D acquisition pipeline provides a base shape that may need to be modified. Freeform Deformation (FFD) techniques offers various way to perform this task, but none can handle large objects. Our approach solves also this problem. In the following paragraphs, we discuss the related work in these areas.

Appearance Texturing   The interactive texturing of 3D objects is a key step in the editing of the final object appearance in computer graphics productions. As usual with interactive tools, the size of the in-core model must be kept low since the dynamic information added during the interactive editing process would break any highly-optimized data-structures, from on-GPU vertex buffer objects to out-of-core representations of large objects.

Direct interactive texturing of 3D objects has been an issue in computer graphics for many years. One of the first complete framework for interactive 3D painting was the WYSIWYG painting tool of Hanrahan and Haeberti [HH90]. Their system allows the user to interactively paint colors and materials directly on a 3D model, introducing a simple brush metaphor. The authors were already pointing the usefulness of such a system for 3D scanned models.

Recently, the idea that 3D textures could be an interesting alternative to usual 2D textures in a painting tool has been independently developed by DeBry et al. [gDGPR02] and Benson and Davis [BD02] who introduced the idea of *octree textures*. The main idea is to set a per-node color at each level of the octree hierarchy and use it to color an object embedded in its volume. Note that octree textures may be interactively constructed or sampled from an existing texture [LHN05], without requiring any parameterization. This is particularly interesting in the case of acquired surfaces: their poor topological guarantees coupled with there high density make them hard to unfold in the plane, avoiding the use of 2D textures. Another great advantage of octree textures is their local control, which is not usual with solid textures, that are often globally defined by some procedural function. For a complete survey on textures in computer graphics, we refer the reader to the book of Heckbert [Hec86a].

The color texture model that we develop in this chapter, the *Point-Sampled Texture* (PST), is volumetric and even more flexible than octree-texture. It has been inspired by the simple construction of a *space-to-color* function from samples, as introduced with the *reaction-diffusion* method of Turk [Tur91], who efficiently obtained a color evaluation at a given location using a simple weighted average of the neighboring samples, an idea later used in the Photon Mapping [Jen96]. PST inherits the parameterization-free nature of point-based techniques, which merge appearance and geometry of samples in the same entity (i.e. the surfel) [PZvBG00, ZPKG02, AWD*04].

Note that several commercial packages propose 3D brushes for texturing and modeling [Ali06, Rig06, Pix06] but do not address the problem of applying them on huge objects.

For the sake of simplicity, we will essentially discuss here the construction of one single color texture. But, as usual with texturing tools, complex textures may be built incrementally by assigning different textures for different material channels, to get more complex shading (appearance composition with specular, ambient, emissive and/or diffuse textures, see Figure 4.12).

Freeform Shape Deformation   In the field of geometric modeling, FFD encompasses a large family of techniques [SP86, Coq90, MJ96] where intuitive constraints are interactively applied to a shape in order to deform it. Earlier FFD techniques were based on a 3D space deformation induced by a dummy object (e.g., lattice, curve, mesh): the user moves the vertices of this dummy object, inducing a smooth space deformation applied to the embedded shape (see [Bec94] for a survey). Such a dummy object is no longer required with recent FFD techniques, where the deformation is directly defined on the initial shape: a part of the surface is frozen, another one is moved, and the shape of the remaining part is smoothly deformed using some fairness constraints. Most recent methods formulate surface deformation as a global variational optimization problem [BK04, SLCO*04, YZX*04, BPGK06], although the same interface can be used with space deformation [BK05]. These techniques offer a precise control of the deformation area (arbitrary boundaries) but remain less efficient than multiresolution editing tools [ZSS97, LMH00, ZS00]. A comparative presentation of recent deformation techniques can be found in [Sor06] and [BS07].

All these techniques are powerful and flexible tools for interactive smooth 3D shape editing. However, while interactivity is the key constraint for the usability of such tools, it cannot be maintained when the complexity either of the 3D model or of the applied deformation exceeds a given workstation-dependent threshold (the notion of "large" strongly depends on both the workstation and the FFD method used since even few hundred thousand samples may be too much in some cases). This is somewhat in conflict with acquired geometries, that may contain several hundreds millions of samples, capturing accurately extremely fine-scale geometric features. In this chapter, we solve this scalability problem with our *sampling-reconstruction* framework, allowing users to define interactively the shape of arbitrarily large 3D models with most FFD tools, at full resolution, without doing any conversion, nor loosing any sample and opening the use of advanced deformation metaphors to models ranging from million to billion samples. Our system also offers the ability to work on models that fit in memory but overcome the capabilities of a given FFD tool.

Handling Large Objects   In spite of the continuous growth of hardware capabilities, even the mere visualization of large models is a complex challenge. Thus, large object management has been essentially studied in the context of static visualization. We will discuss this particular topic in Section 6.1. Concerning appearance, Christensen et al. [CB04] have showed that the irradiance can be stored with cache-coherent out-of-core structures. Their system is not intended for interactive manipulation but already gives some ideas on how to organize appearance data for large models. Note also that the specific issue of terrain visualization takes benefit from out-of-core techniques (see Losasso et al. [LH04] for a recent survey). Unfortunately, all these systems have been designed for static objects rendering only and cannot be used in a dynamic context such as texturing and FFD, inducing the simplification problem discussed earlier. Indeed, many fine-scale features are missed during this resolution reduction and when dealing with acquired shapes, the benefit offered by an accurate scanning process is partially wasted, loosing real-world object details which are acquired but not preserved for editing reasons. Similar problems arise when very complex synthetic models are created with specific techniques such as *displacement painting* (e.g., ZBrush [Pix06]).
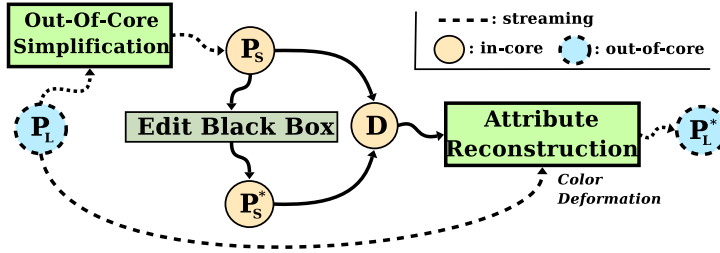
41

Figure 4.2: *Our Sampling-Editing-Reconstruction framework for interactive color and shape editing of large objects.*

4.2 A Sampling-Reconstruction Framework

Editing the color and the shape of large objects imposes one strong constraint on the software architecture: as the whole object is potentially too large to fit the in-core memory, the only truly generic and efficient implementation is to perform *out-of-core data streaming*. Moreover, as interactive shape editing typically involves semi-random access to the database, specific in-core data structures have to be designed, that allow efficient *dialog* with the out-of-core object, in the spirit of recent work by Isenburg et al. [IL05, ILSS06]. However, in order to avoid painful editing sessions, only a reduced number of streaming passes should be performed, since each pass may take several minutes for large objects.

The key idea developed in this chapter is that the color and the shape of a dense object can be precisely edited through the interactive modification (i.e., painting and FFD) of a simplified in-core version. Thus, we propose the three-fold *sampling-editing-reconstruction* system described on Figure 4.2:

1. *Sampling:* an efficient adaptive out-of-core downsampling is performed on the original large object P_L during a streaming process, to get a simplified version P_S with a size compatible with interactive texturing and deformation (see Section 4.3).

2. *Editing:* an interactive texturing and deformation session is then applied on P_S to obtain a modified simplified object P_S^*. During this session, local upsampling may be achieved at any time, when additional precision is required for a given manipulation.

3. *Reconstruction:* Finally, another streaming process performs an accurate feature preserving color and deformation transfer from P_S^* to P_L, in order to generate the final edited large object P_L^* (see Section 4.4).

This three-fold approach exhibits several features:

Size-independent interactivity: Interactivity is an essential property of any painting or deformation tool since the target result is not precisely known a priori and is usually reached by interactively exploring the space of possible shapes and appearances. Our system offers interactivity, for any size of the initial object, by performing adaptive downsampling to fit the workstation and software capabilities. Both the sampling and the reconstruction steps work in streaming and only involve local and linear processing, which guarantees memory and computation efficiency.

Pure meshless processing: as mentioned earlier, efficient 3D acquisition enables only greedy reconstruction techniques in practice, and it is well known that such algorithms do not provide strong guarantees about the consistency of the resulting topology. To overcome possibly non-manifold input data, we have chosen to simply ignore the underlying topology, and only employ meshless techniques [AGP*04], which allows our system to process unorganized point clouds, polygon soups or manifold meshes in

a similar way, both for data input and data output. This also lead us to develop the volumetric *Point-Sampled Textures* instead of regular bi-dimensional ones.

Compatibility with arbitrary painting and FFD tools: as recalled in Section 4.1, a rich palette of 3D editing methods has been proposed during the last twenty years; each of them has specific strengths and weaknesses and each CG designer has her own preferences among them. To preserve this variety, we do not impose any particular method, but rather propose a surrounding system allowing the use of **any** interactive texturing and deformation tool with large objects. Our system considers the editing step as a black box between P_S and P_S^*. It only requires a one-to-one correspondence between the samples of the simplified model P_S and those of its edited version P_S^*, which is trivially provided by most of tools.

On-demand local refinement: One possible weakness of manipulating downsampled geometry is precision reduction of some deformation or texturing tools (e.g., the user may want to precisely outline the frozen area on the original object, before manipulating the deformation handler). Whenever the user requests an improved precision of a specific area, our system performs efficient local upsampling of the in-core model by fetching additional samples from the original geometry and transferring to them the deformation defined so far.

Having all these features combined, our system can be considered as the first *interactive out-of-core multi-scale modeling and texturing system*, compatible with a vast repository of existing 3D editing tools.

4.3 Sampling by Adaptive Out-Of-Core Simplification

The first step of our system aims at efficiently generating a convincing simplification of the original, possibly gigantic, model during a streaming pre-process. As mentioned earlier, topology inconsistencies often present in such objects lead us to work in a meshless context. The large size of P_L requires an efficient simplification algorithm, and the temporary nature of P_S allows a non-optimal geometry. In this context, *vertex clustering* appears as a good choice. Such methods can run out-of-core [RB93, Lin00, SW03] and can handle non-manifold surfaces. We recall that the idea is to generate a partitioning of the space embedding the object, and compute one representative sample for each partition, using various error metrics to control the hierarchy depth. At the end of the process, the set of representative samples can be considered as a downsampling of the original object.

The adaptivity and the efficiency of vertex-clustering simplification algorithms strongly relies on two key elements:

- the *space partitioning structure* (e.g. 3D grid, BSP, octree)

- the *error metric* (e.g. Quadric L_2, $L_{2.1}$, etc).

For instance, Schaefer and Warren [SW03] have used octrees combined with a *quadric error function*[GH97] defined over the original geometry. Their algorithm can be used with large objects thanks to a dynamic split-collapse function over the octree structure. However, we have shown in Chapter 3 that the Volume-Surface Tree offers a better adaptive vertex clustering than octrees, as it requires less samples for an equivalent error bound. So we propose to extend the VS-Tree vertex clustering algorithm
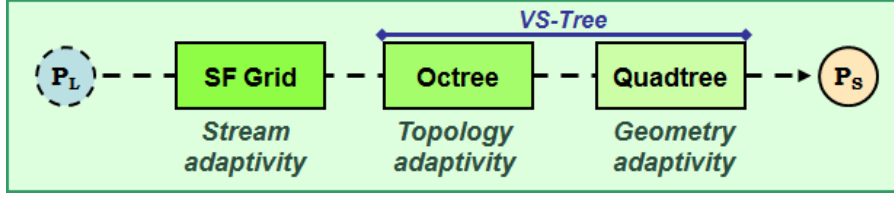
Figure 4.3: *Multi-structure Out-of-Core Simplification in Streaming.*

presented in Section 3.3 in order to make it out-of-core, which can be done in two ways:

- *External memory management* maps partially a structure from hard to main-memory, and synchronizes both versions.

- *Streaming computation* considers only a stream of samples which are maintained in main memory for a limited time, and processed on the fly.

The former allows more flexible data access but may be slow, while the latter is fast, but limited in its "view" of the object. As our algorithm need to be used on an interactive tool, we seek for efficiency, and propose a *VS-Tree simplification in Streaming* (see Figure 4.3). Actually, we can take benefit from the coherency present in large acquired objects, by using *spatial finalization* [ILSS06] for maintaining a low memory footprint. The basic idea of *spatial finalization* is that the order of sample in a streamed geometry mostly corresponds to its acquisition order, which means that two samples with similar locations in the stream, have similar positions in the space. This induces that clustering the stream spatially produces a time-coherent partitioning: the first and last sample of a partition have only a small difference in their stream rank. Building on this idea, we propose to generate a set of temporary VS-Trees structured in a coarse 3D grid and use them to locally downsample the surface. This VS-Tree forest avoids the memory challenge of one global data structure and behaves particularly well with gigantic objects which exhibit strong *acquisition coherency*, as mentioned by Isenburg et al. [ILSS06].

Algorithm 2 Streaming VS-Tree Simplification

Require: P_L the out-of-core large sampled surface
Require: r the grid resolution and ε the error driving the simplification
Require: P_S the empty in-core sampling
  Grid $G \leftarrow GridElement[r][r][r]$
  for each sample p streamed from P_L **do**
    $\{i,j,k\} \leftarrow$ coordinate of p in G
    $G[i][j][k].count \leftarrow G[i][j][k].count + 1$
  end for
  for each sample p streamed from P_L **do**
    $\{i,j,k\} \leftarrow$ coordinate of p in G
    $G[i][j][k].samples \leftarrow G[i][j][k].samples \cup p$
    $G[i][j][k].count \leftarrow G[i][j][k].count - 1$
    if $G[i][j][k].count = 0$ **then**
      $P_S \leftarrow P_S \cup$ VSTreeSimplification $(G[i][j][k].samples, \varepsilon)$
      free $(G[i][j][k].samples)$
    end if
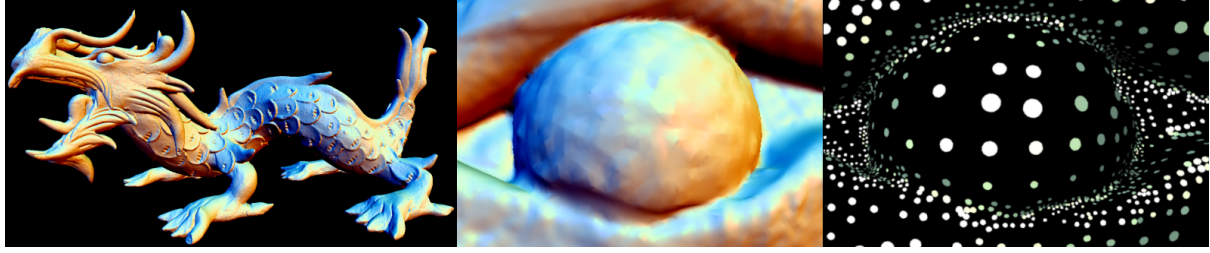 **end for**
  return P_S

Figure 4.4: Left: *XYZRGB Dragon (7.2M triangles).* **Middle:** *Close-up of the eyeball on the original geometry.* **Right:** *Adaptive downsampling by out-of-core VS-Tree clustering.*

Algorithm: Our simplification process, described in Algorithm 2, uses two streaming passes:

1. *First pass:* all the samples of P_L are streamed through a coarse 3D grid G. During this pass, each cell of G simply counts the number of samples falling in it and keeps the counter for the second pass (a preliminary pass is required if the bounding box is not known).

2. *Second pass:* all the samples are streamed again through G. This time, each read sample is stored in the intersected cell C, and the counter of this cell is decreased. Once the counter of C drops to zero, we know that there are no additional samples belonging to C in the remaining stream. So, we simplify the set of samples stored in C by using VS-Tree clustering (Section 3.3).
   The resulting set of representative samples is then concatenated to P_S, and the content of C (i.e. original set of samples and VS-Tree) is discarded. If required by editing tools, a mesh can be obtained either by reindexing polygons in the case of polygons soup inputs, or using fast meshing techniques in the case of point clouds or indexed mesh vertices (see Secton 5).

In practice, this algorithm only requires a small fraction of the original surface to be present at the same time in the main memory. The observed memory footprint ranges from 10% for million sized objects to less than 1% for billion sized ones. When the large objects are provided with both positions and normals for each sample, we can use a product of the L_2 and $L_{2.1}$ error metrics [CSAD04] to drive the VS-Tree clustering. Otherwise, a simple density measure is used, and an aggressive flatness test replaces the volume-surface transition predicate. When P_L is a point-cloud, the normals of P_S are estimated using a PCA on the local neighborhood [HDD*92]. The resolution of the coarse grid G is user-defined, ranging in our tests from 16^3 to 128^3 according to the size of P_L. Usually, a large number of cells speeds up the simplification process, while a small number improves the adaptivity of P_S (see Section 4.6).

Note also that for future local refinement that may be required during the interactive editing session (see Section 4.5), we keep, for each cell C, two values about this pre-processing:

- the starting and ending indices of the samples that belong to C in the input stream; this will enable partial streaming for local refinement of the cells intersecting the area marked to be refined;
- the VS-Tree structure; this avoids error computation and recursive split for levels already tested during the initial simplification. In order to keep a negligible memory footprint, when the tree is too deep, it is not cached and will be rebuilt from scratch if upscale is required.

Figure 4.4 illustrates the downsampling quality from P_L to P_S obtained with our approach. Obviously, this algorithm inherits the lower-dimensional structure of the VS-Tree and significantly reduces the number of final representative samples for a given error bound, thus also reducing the whole processing time. Compared to octrees, we have observed a gain of 15% to 25% both in time and memory, which is a significant benefit in the context of gigantic objects. The divide-and-conquer structure offered by *spatial finalization* makes this algorithm particularly well adapted to multi-core CPUs, which become more present on current workstations. Finally, when the Spatial Finalization heuristic fails, it can be replaced by one additional pass to split the input model into a set of files clustered on a per-cell basis.

4.4 Out-of-Core Attribute Reconstruction

The second out-of-core streaming process of our system takes place after the interactive editing session, where the initial point-based simplified geometry P_S has been transformed into its textured and deformed version P_S^*. The goal of this section, is to explain how to efficiently and accurately transfer these appearance and shape modifications to the original gigantic object P_L, to get the final edited object P_L^*. For each sample with position $p \in P_L$, we have to extract two functions from $\{P_S, P_S^*\}$: a *colorization* function that will define its color c and a *deformation* function that produced a new position p^*. As this attribute reconstruction is performed in streaming, we have developed a local solution, that only requires the analysis of a small and compact neighborhood. Since we use a point-based representation, we do not have any explicit neighborhood information. Nevertheless, a conservative set of neighbors can be collected with the *k nearest neighbors* [PGK02]:

$$N_k(p) = \{q_0, ..., q_{k-1}\} \text{ with } q_i \in P_S$$

We also consider the associated color c_i and deformed position q_i^*, both defined in P_S^* during the interactive editing session.

This neighborhood must be large enough to offer a correct filtering, but also small enough to remain accurate in areas of large curvature. In practice, we use $k \in [10, 30]$ according to the density of P_S. Note that $N_k(p)$ can be efficiently computed using a static kd-tree on P_S, generated once for all just before streaming P_L.

Let $f_{P_S}^A$ be the family *point-sampled functions* defined over the point set P_S and reconstructing the attribute A: $f_{P_S}^C(p)$ gives the corresponding color attribute to stream and $f_{P_S}^D(p)$ states the deformed position.

4.4.1 Streaming Colorization

Once the point set has been textured, we propose to consider itself as a *point-sampled 3D texture* or PST, to color its high resolution version in streaming. Thus, the question is: "How to extrapolate the set of samples in order to use it at a higher definition?". Actually, this problem frequently arises in the field of *surface reconstruction*. In particular, *variational implicit surfaces* methods are ubiquitously recognized as quality approximation methods for a set of samples with attributes [TO02]. Usually, an iso-surface is finally extracted after fitting a function $f : {}^3 \to$ to the set of samples. In our case, the problem is simpler as we do not need iso-surface extraction, and just keep the function defining the implicit surface as a 3D texture.

Several function basis are available for filling the space with point-sampled attributes. *Radial Basis Functions* or *Moving Least Squares* [AGP*04] provide smooth 3D fields and can be evaluated locally. Unfortunately, in our case, the final evaluation of the function may potentially be done several hundred million times for either coloring the original file or directly shading pixels during ray tracing for instance. Thus, we rather adopt a simpler and more efficient approach that takes advantage of a very important feature of our PST: contrary to implicit surfaces used for geometric reconstruction, we do not need a signed value. In this case, a variation of the seminal idea of Turk for pattern creation [Tur91] can be adapted to our more general problem.

We define the PST $f_{P_S}^C(p)$ as an *Partition of Unity* filtering process of P_S:

$$f_{P_S}^C(p) = \frac{\sum_{i=1}^k \omega(p, q_i)\, c_i}{\sum_{i=1}^k \omega(p, q_i)}$$

Note that the size of the k-neighborhood influences the support radius of the reconstruction: a large value of k will smooth out the so-defined attribute function and can be used as an intuitive global *filtering* parameter for users (see Figure 4.5). In our implementation, k is user-defined. The function $\omega(p,q_i)$ is a *decay* function that weights the influence of neighbors samples attributes. It is well-known that a Gaussian kernel is a good choice for ω. Nevertheless, in the context of large object texturing, selecting a less computationally intensive function is often interesting. We choose the standard uniform cubic Hermite polynomial, usually recognized as a good and fast approximation of Gaussian-based kernels:

$$\forall t \in [0,1] \quad h(t) = 1 - 3t^2 + 2t^3$$

The kernel function $\omega(p,q_i)$ uses the previous polynomial simply adapted to $N_k(p)$, and is hence defined as:

$$\omega(p,q_i) := h\left(\frac{|p-q_i|}{\arg\max_j(|p-q_j|)}\right)$$

Note that this *Hermitian Partition of Unity* kernel function is extremely inexpensive, but may filter out some high frequency details present in P_S. When this is an issue, *singular weight kernels* (i.e., Dirac behavior near zero) can be used [ABCO*01, GG07]. Alternative feature-preserving kernels may also be chosen among the huge set of kernels developed over the years, in the image processing community.
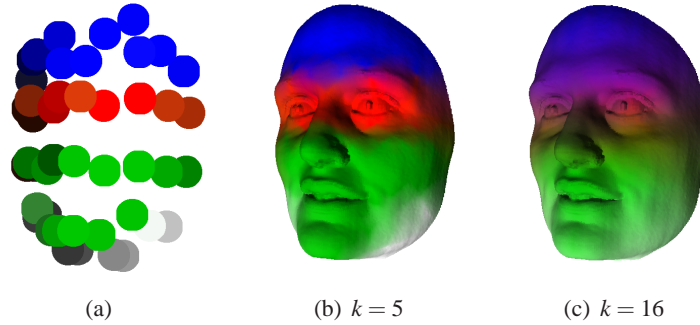

(a) (b) $k = 5$ (c) $k = 16$

Figure 4.5: *Point-sampled texture filtering. (a) A simple point-sampled texture. (b,c) Color texturing on a human face. The k-neighborhood used for space-filling intuitively drives the* smoothness *of the PST.*

4.4.2 Streaming Deformation

While colorization can be cast as a filtering process, deformation is a tedious task: considering that P_L has been scanned, we want to keep all as much as possible its original geometric features, avoiding the use of a simple (low-pass) filtering process.

Point Sampled Deformation Function: By manipulating one or several deformation tools during the interactive editing session, the user implicitly defines a continuous space deformation function $f_{P_S}^D$. But, as the editing step is considered as a black box by our system, the actual function $f_{P_S}^D$ is unknown. However, the simplified geometry P_S and its deformed version P_S^* form a discrete displacement field $\{p_i, p_i^*\}$ which can be interpreted as a point sampling of the continuous function $f_{P_S}^D$ (see Figure 4.6). Therefore, our goal is to reconstruct the continuous deformation function $f_{P_S}^D$ from the discrete field $\{p_i, p_i^*\}$. Here, the difference with the colorization of Section 4.4.1 is that the original geometric feature carried by p_i must me preserved under deformation. Since smooth function reconstruction methods are slow and too global, which make them prohibited in the context of large objects, we propose a purely local method, based on a *normal displacement* representation combined with a new efficient point-based coordinate system.

Figure 4.6: **Left:** *Initial in-core geometry obtained by out-of-core VS-Tree clustering.* **Middle:** *Deformed in-core geometry after an interactive FFD session.* **Right:** *Discrete displacement field generated by linking the initial (green extremum) and the final (red extremum) position of each sample point. This displacement field performs a sampling of the continuous space deformation function at the scale at which it has been edited.*

Average Plane and Normal Displacement: Reconstruction of a continuous function from point samples always involves some assumptions about the smoothness of the initial function between the samples. In our case, the simplified point-based geometry P_S has been computed from P_L by VS-Tree clustering which includes several geometric error bounds to guarantee a reasonable smoothness of P_L between neighboring samples of P_S. In other words, P_S can be considered as a low-pass filtered version encoding large-scale geometric components of P_L, while the difference P_L-P_S encodes fine-scale geometric details. Since the interactive editing tools provide P_S^*, we can smoothly reconstruct the deformation function corresponding to the low frequency part of P_L: we use again an aggressive interpolation based on Hermitian filtering. The remaining part is the preservation of the features carried by P_L-P_S, particularly when rotation is undergone. To relax this constraint, we propose to encode this details in the normal direction of some local average plane H [LSLCO05, KS06]. So each sample $p \in P_L$ can be expressed as:

$$p = p' + d \cdot n$$

where n is the normal vector of H, p' the orthogonal projection of p on H and d the signed distance from p to H (see Figure 4.7). Note that efficient computation of the average plane H is vital, as it has to be done for each sample $p \in P_L$. Moreover, the variation of H from one sample to its neighbor should be smooth as p' is not supposed to include geometric high frequencies.

Similarly to colorization, we propose to compute H using a local *partition of unity* filter weighted by an Hermitian kernel:

$$n = \frac{\sum_{i=1}^{k} \omega(p, q_i) \, n_i}{\sum_{i=1}^{k} \omega(p, q_i)}$$

The same kernel is also used to compute the center of H. The low-pass filtering can be intuitively controlled by the user, by increasing or decreasing the geometric extent of the kernel.
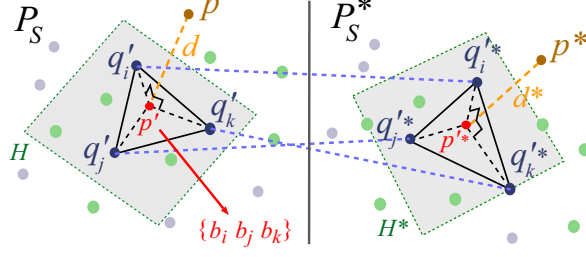
Figure 4.7: *Projected barycentric coordinates.*

Projected barycentric coordinates: Similarly to $p \in P_L$, any point $p^* \in P_L^*$ can also be expressed as a normal displacement relative to an average plane H^*:

$$p^* = p'^* + d^* \cdot n^* .$$

Reading this equation backwards provides a simple algorithm to transfer deformation from p to p^*:

1. compute H from local neighborhood of p in P_S

2. compute p' and d from p according to H

3. reformulate p' *intrinsically* in P_S

4. reproduce p'^* simply by switching from P_S to P_S^*

5. compute H^* from local neighborhood of p'^* in P_S^*

6. compute d^* from d, accounting for a possible scale factor

7. finally, compute $p^* = p'^* + d^* \cdot n^*$

Differential representations [SLCO*04, LSLCO05] provide an elegant solution to obtain rotation-invariant intrinsic coordinates. Kraevoy and Sheffer [KS06] introduce the *pyramidal coordinates*, where p' is replaced by its *mean value coordinates* [Flo03] in its 1-ring-neighborhood. However, these solutions require explicit topology and remain computationally expensive, which makes them prohibitive in our context.

We propose an alternative intrinsic encoding which can be considered as an approximation tuned for efficiency, more suitable in the context of gigantic objects. Let us consider a 3-neighborhood $T(p) = \{q_i, q_j, q_k\}$ for p and its projection $T'(p) = \{q'_i, q'_j, q'_k\}$ on H. We can use the projected barycentric coordinates $B(p) = \{b_i, b_j, b_k\}$ of p' in the triangle $T'(p)$ as intrinsic coordinates of p on H (see Figure 4.7). $B(p)$ can be directly computed from p using the fast evaluation (49 ops) proposed by Heidrich [Hei05]. So the switch from p' to p'^* (step 4 of the algorithm) can simply be expressed as:

$$p'^* = T'^*(p) \cdot B'(p)^\top .$$

We now have to face a topology problem: how to correctly select $T(p)$. For symmetry and robustness reasons, $T(p)$ should be as equilateral as possible, and to ensure fidelity of deformation, it should be as small as possible. To efficiently fit these constraints, we need to select neighbors according to their distribution [LP03, GBP05], so we introduce the notion of *angular voting*: the first neighbor $q_i \in N_k(p)$ is selected as the nearest sample to p. This neighbor will discard a conical space C_i starting from p, centered in the direction of q_i and with a solid angle $2\pi/3$. The second neighbor q_j is selected in the subset of $N_k(p)$ contained in $R^3 \backslash C_i$. This neighbor discards another conical subspace C_j. Finally, q_k is selected in the subset of $N_k(p)$ contained in $R^3 \backslash (C_i \cup C_j)$. If a test fails (for instance, an empty set in the

remaining space, which is frequent on surface boundaries) the angle is divided by two and the selection process is restarted. As a result, this algorithm searches for the smallest triangle centered on p (i.e., capturing accurately the deformation) with as large as possible edge angles (i.e., increased robustness avoiding numerical degeneration in projected barycentric coordinates).

The last element that we need to define is the signed distance d^* (step 6 of the algorithm). Thanks to the intrinsic coordinates, the local rotation undergone by the geometric details during the deformation has been accounted for, but an eventual scaling has not. So we propose to simply scale the distance d by the size ratio of the surrounding triangles, before and after deformation:

$$d^* = d \cdot r^*/r,$$

where r (resp. r^*) is the circumcircle radius of $T(p)$ (resp. $T^*(p)$). By putting all the elements together, we obtain the final expression for our reconstructed deformation function $f_{P_S}^D$:

$$\forall p \in P_L \quad p^* = f_{P_S}^D(p) = T'^*(p) \cdot B'(p)^\top + d^* \cdot n^*.$$

Figures 4.1, 4.8 and 4.14 present various examples of the accurate deformation of small features with our fast deformation function.

Streaming Deformation Reconstruction: Once the kd-tree required for neighborhood queries has been set up, the purely local behavior of our colorization and deformation functions enable streaming, as each sample p is processed individually: the position of samples of P_L are read on the input and new position and color are written on the output. Moreover, these per-sample operations can fully exploit multicore CPUs. In order to optimize the cache usage when collecting the local set of neighbor candidates in P_S, spatially coherent input buffers can be built by using the structure induced by G again (see Section 4.3).

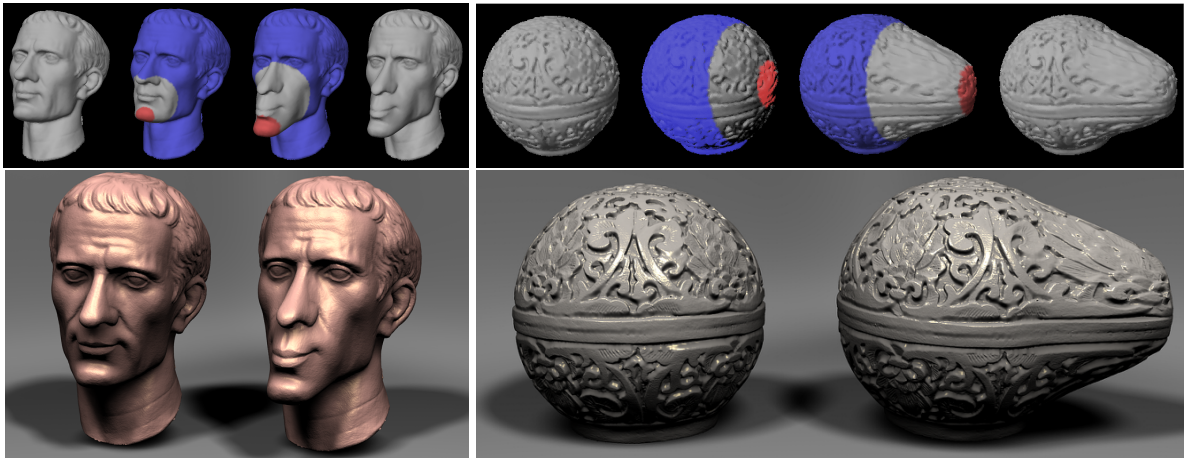**Figure 4.8: Left:** *Julius Caesar (800k triangles) interactively deformed using a 30k downsampling.* **Right:** *Bumpy Sphere (1.4M triangles) interactively deformed using a 25k downsampling. Note that both versions exhibits small scale features which are strongly blurred on the simplified version during the interactive session, but are adequately reintroduced by the deformation reconstruction on the original geometry.*
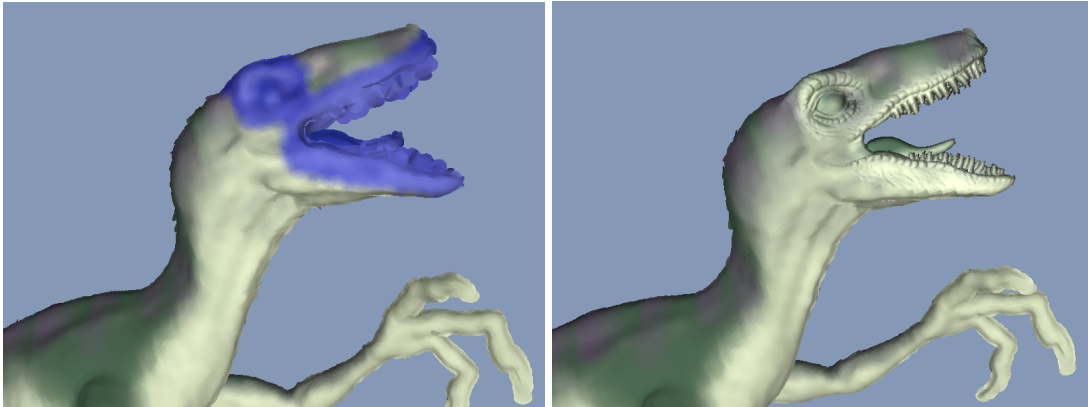
4.5 Interactive Out-Of-Core Multi-Scale Editing

Figure 4.9: *Out-of-core Multi-Scale texturing.* **Left:** *After having roughly painted on it, the user selects an area (in blue) of the low-res sampled object.* **Right:** *A local refinement is performed in streaming, by up-sampling the selected area from the original large model. Newly inserted samples are textured according the current PST defined by the in-core point set; the user can now paint smaller features.*

One weakness of manipulating downsampled geometry is the possible lack of precision for a particular color or deformation feature. To avoid this possible issue, our system is able to perform interactive upsampling, whenever the user requests improved precision on a specific area for finer editing. This *progressive* interaction works as follows:

1. The user selects the area which require higher sampling ratio and a set smaller error bound than the initial sampling;

2. Sub-parts of the original objects, corresponding to the index range of each cell of the grid G (see Section 4.3) intersecting the selection, are streamed.

3. Upscaling is performed by clustering the streams in the cached VS-Trees, set with the new error bound.

4. Each additional representative sample p is then concatenated to P_S, and its corresponding color c and deformed position p^* — computed by applying the colorization and deformation functions defined so far — is concatenated to P_S^*.

When using this procedure, it becomes quite natural to start with a very coarse in-core geometry P_S to define large scale deformation and rough color texture, and then refine some regions of interest and define more accurate deformations and finer color features. This principle can be used recursively until the desired precision is reached, which reproduces a very similar workflow as the one provided by subdivision surfaces. Figure 4.9 illustrate the application of this principle for color editing and Figure 4.10 gives an example of multi-scale deformation. This figure also shows another property of our system: since it is not method-dependent, several tools can be mixed; here, we first used the global deformation tool by Pauly et al. [PKKG03], and then, we switched to an *inflating* displacement tool.
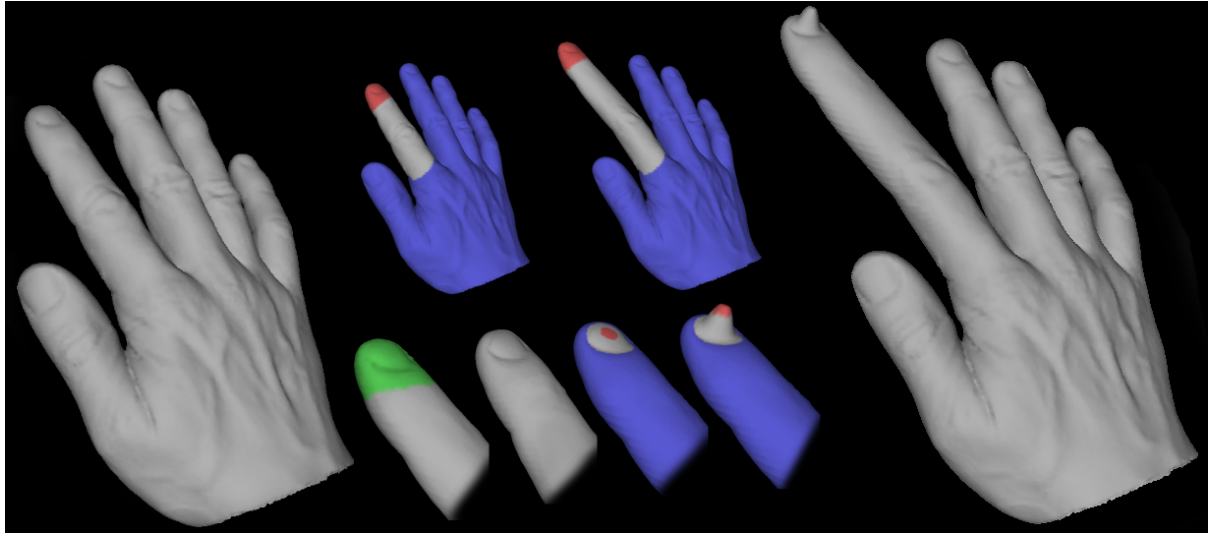
Figure 4.10: *Out-of-core multi-scale FFD on the Hand model (1.5M tri.).* **Left:** *Initial coarse sampling for interactive editing (50k samples).* **Middle Top:** *FFD performed at a large scale.* **Middle Bottom: Local upsampling of the in-core geometry (green area) with our system: the additional samples fetched from original geometry are moved according to the deformation performed so far, and FFD is enabled at a finer scale (125k samples), with the same or another tool.* **Right:** *Final multi-scale deformation.*

4.6 Results

Implementation: We have implemented our system on a standard workstation (P4 3.4GHz, 1.5GB RAM, 36GB SCSI and 200GB SATA HD) running GNU Linux 2.6, using C++, OpenGL, POSIX threads, the GNU Scientific Library and Qt.

Performances: Table 4.1 presents the timings of both streaming processes (adaptive simplification and attribute reconstruction) for various models[1]. In all cases, no memory swapping has been observed, thanks to the spatial finalization. Since deformation is more expensive than colorization, we report only the deformation timing. Colorizations exhibit in general two time faster post-processing.

The pre-processing streaming is mostly bottlenecked by the physical capabilities of the I/O device. Note that, to speed-up processing of very large objects, only vertices are read and streamed: since ordering is preserved in the stream, the final deformed point set remains compatible with the original topology (provided by triangle indices) and point-based editing can be used safely [ZPKG02, PKKG03]. Alternatively, triangles can also be streamed, if mesh-based FFD are employed.

The final streaming is more computationally intensive. To exploit multi-core architectures, now widely available, multiple threads are used to process the I/O buffer. Note also that the second pass of the pre-process benefits from multithreading for largest objects, using a specific thread for each active cell of G. In practice, this means that the thread scheduler has to deal with 20 to 200 simultaneous threads.

---

[1]The scalability of our system has been intensively tested on the Digital Michelangelo gigantic models. As there are strong legal restrictions on shape editing for these models, we only present the timings, but not the resulting pictures.
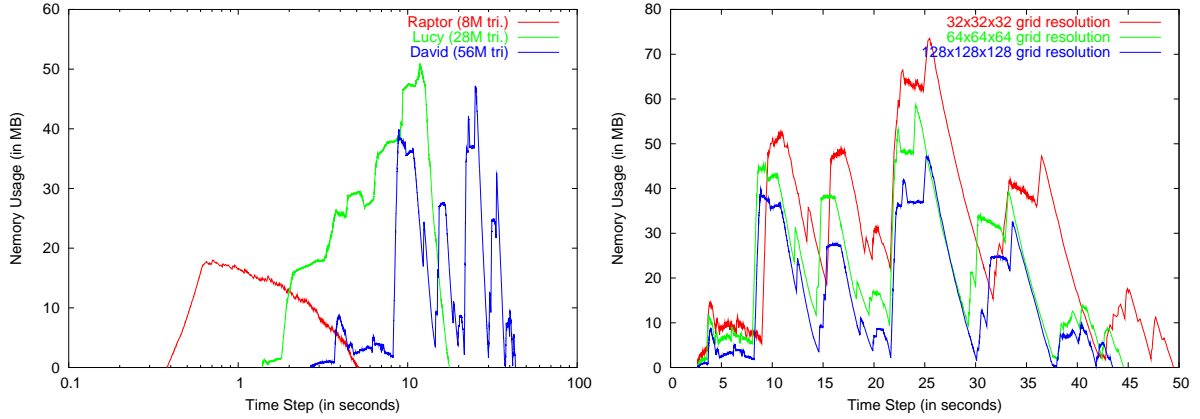
Figure 4.11: Left: *Evolution of memory footprint during out-of-core simplification for various models (error bounds have been set to obtain about 200k samples).* **Right:** *Influence of the coarse grid resolution on the footprint for the David model (56M triangles).*

The k-neighborhood queries, intensively used during the output streaming, are implemented onto a static kd-tree, built once, just before streaming P_L. The typical size of P_S ranges from 20k (after out-of-core simplification) to 500k (worst case observed after all the local refinements involved in a whole editing session). This induces a very fast generation of such a tree (less than one second in all our tests). The computation workload is essentially concentrated in this attribute reconstruction (see Table 4.1) and more particularly the deformation, which emphasizes the use of our aggressive but efficient *projected barycentric coordinates*.

The main part of the memory footprint during the interactive session is reduced to P_S plus P_S^*. Note that making P_S itself out-of-core is actually fairly easy as discussed in the next section. Figure 4.11 measures the evolution of the memory footprint involved throughout the simplification process. The peak memory usage is reached during the second pass of the simplification preprocess. Fortunately, tt clearly appears that this footprint is largely independent of the model size (both Lucy and David present a memory peak about 50MB) and is rather linked to the geometric complexity: for instance, high surface genus may require deeper VS-Tree samplers. The influence of the grid resolution is more complex. A finer resolution for G reduces the global memory footprint, but as there are more simultaneously active VS-trees, it involves additional over-clustering in areas of low sample density. In practice, we rather

Models	Original Size		R/W data	Sampling	Pre-process		Post-process
	vertices	triangles			1st pass	2nd pass	
Julius Caesar	387K	774K	8.8 MB	30K pts	0.35 s	0.43 s	1.70 s
Bumpy Sphere	701K	1.4M	16 MB	25K pts	0.81 s	0.98 s	1.32 s
Hand	773K	1.54M	17.6 MB	50K pts	0.81 s	1.11 s	1.41 s
XYZRGB Dragon	3.6M	7.2M	82.3 MB	160K pts	2.25 s	2.98 s	5.06 s
Raptor	4M	8M	91.5 MB	146K pts	2.07 s	3.07 s	5.94 s
Lucy	14M	28M	320.4 MB	202K pts	7.21 s	10.5 s	37.9 s
David	28M	56M	640.8 MB	209K pts	13.3 s	30.2 s	120 s
St Matthew	186M	360M	2.07 GB	189K pts	60.4 s	122 s	338 s
Double Atlas	500M	1G	5.59 GB	270K pts	143 s	775 s	1121 s

Table 4.1: *Pre-process (adaptive simplification) and post-process (deformation transfer) performances for various models.*

advocate a medium resolution for G, around 64^3 in our experiments, which nicely balances sampling quality and reasonable memory consumption.

Many examples provided in the paper have been created using the point-based editing tool proposed by Zwicker et al. [ZPKG02] and Pauly et al. [PKKG03] in the *PointShop 3D* environment. We have also experimented our system with *Blender* to provide interactive mesh-based out-of-core editing of gigantic objects (e.g., displacement painting).

Complexity Let l be the size of P_L and m the size of P_S. The theoretical complexity of the post-streaming is $O((l+m)\log m)$, due to kD-Tree construction and k-neighbors queries. In practice, we have $m << l$ and the cache coherent access to samples in the stream exhibit an almost linear behavior for the range of object size we study. The theoretical complexity of the pre-streaming cannot be worst than the one of a quad-tree clustering which is $O(l\log_4 l)$: this worst case corresponds to an height-filed directly detected by the κ predicate. Another "bad" case would be an an object P_L composed of samples with random location in the volume (no surface coherency), and a spatial finalization grid resolution of one. In this case, the complexiy is bounded by $O(l\log_8 l)$ (no surface detection). The practical complexity is hard to estimate in general, as it is geometry-dependent. Nevertheless, we can consider a complexity of $O(l\log_4 l^*)$, with $l^* << l$ in the case of large and dense sampled objects: a geometric error function is used for driving the tree clustering, and it is clear that, in the case of smooth, "sufficiently" sampled surfaces, an higher sampling ratio does not involve a deeper tree (i.e. a single leaf represents a whole piece of surface, as soon as it is sufficiently sampled).

Examples: Figure 4.1 shows a complete out-of-core texturing and FFD session with the Raptor model. Figure 4.12 shows a combination of several color PST applied on various channel (diffuse, specular, ambient, etc) of the appearance of the Vase Lion model and Figure 4.8 illustrates the deformation obtained with mid-sized objects. Multi-scale texturing and modeling are illustrated on Figures 4.9 and 4.10. Finally, the absolute scalability of our system, for either shape or appearance modeling of very large models, is presented on Figures 4.13 and 4.14. Note that in all cases, interactivity has been preserved, while on the same workstation, major commercial modeling packages are no longer interactive above one million triangles, and just fail when trying to load objects around 10 millions.

Figure 4.12: *Point-sampled texture for high-quality rendering.* **Top left:** *Original mesh (6.5M polygons).* **Bottom left:** *Multi-channel interactive out-of-core texturing with our system (50k point samples).* **Right:** *Offline rendering of the original mesh (6.5M textured polygons) with our point-sampled textures (diffuse and specular component).*
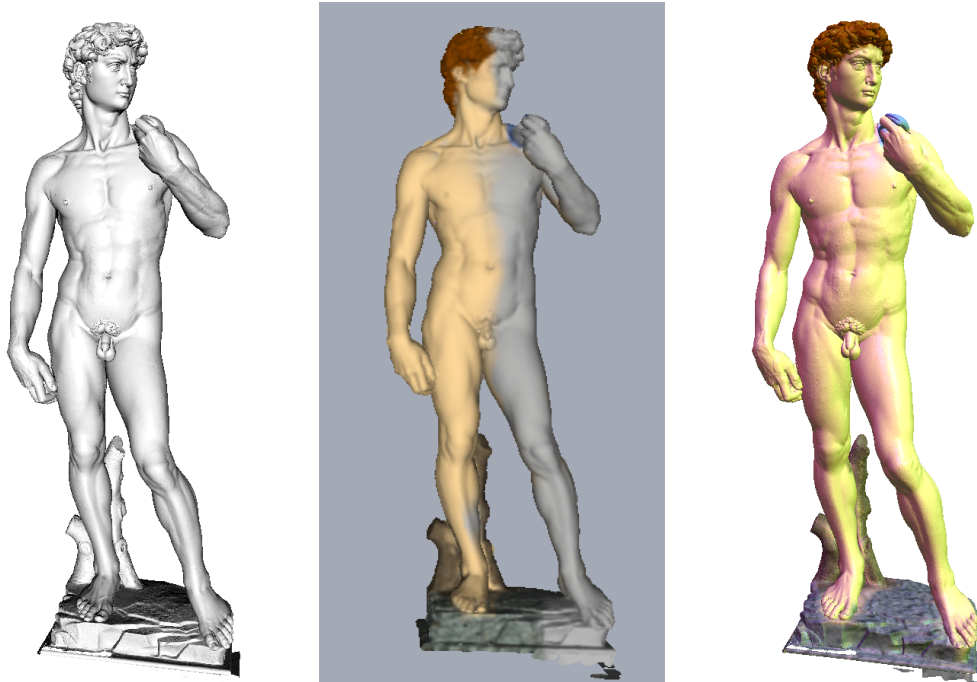
(a) Michelangelo's David (56M polygons) - 200k samples for interactive texturing
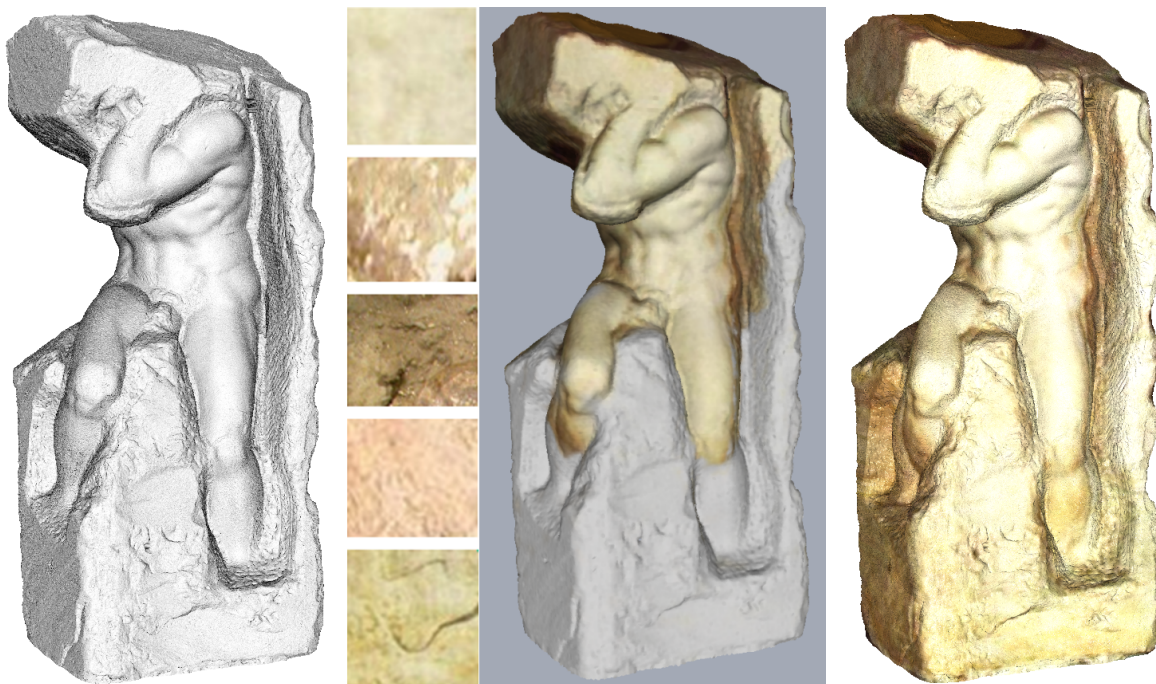
(b) Michelangelo's Atlas (500M polygons) - 278k samples for interactive texturing

Figure 4.13: *Interactive texturing of very large models.* **Left:** *Original large mesh.* **Middle:** *Interactive multi-scale texturing with our system.* **Right:** *Application of the PST to the original model and real-time out-of-core rendering. Multi-scale editing is used, and textures are composed from photos and stones patterns.*

Figure 4.14: *Interactive freeform deformation of very large models.* **Top:** *XYZRGB Dragon (7M triangles) - 160k samples for interactive deformation.* **Bottom:** *Lucy (28M polygons) - 300k samples for interactive deformation. Interaction snapshots are displayed in grey, red and blue. Each full session with adaptive simplification, interactive editing and deformation transfer took less than 5 minutes.*

4.7 Discussion

Comparison To our knowledge, this is the first system that permits interactive multi-scale feature-preserving shape texturing and editing of gigantic objects, as well as opening the use of costly editing method to medium size objects. However, the three pieces of system can be easily compared to existing methods.

First, we have propose a new out-of-core simplification algorithm, which can be compared to the octree method of Schaefer et al. [SW03]. In term of efficiency, our out-of-core VS-Tree simplification reaches the user-defined error threshold quicker than octrees, generating less samples for the same error. Concerning memory, the spatial finalization allows to discard most of the structure overhead along the time, which reduces the memory footprint, while the octree method maintains a single complex octree during all the simplification. This makes also our algorithm easily portable on PC cluster and multi-core/CPU workstations.

Second, our *point-sampled textures* are sampled volumetric textures, and can thus be compared to *octree textures*. Basically, the main advantage of PST over octree textures is to allow the user to interactively refine directly from the original surface, without being constrained to the grid topology induced by octrees (See Figure 4.13). Simple point sets allow greater flexibility and very quick variation in the density of sampling (which are very frequent when the user wants to texture a given area more accurately [gDGPR02]) where a very deep octree would have been necessary. Last but not least, octree textures cannot efficiently represent fine color features which are not axis-aligned. However, the uniform structure of octree textures allows efficient on-GPU implementations [LHN05, LKS*06], which is more difficult for PST. Of course, in such a situation, our PST can be easily resampled in an octree texture for real-time shading. But, we rather focus on very large objects, for which the color is usually encoded in the data-structure, on a per-sample basis, for efficient rendering [RL00, DVS03, GM05].

Third, a variation of our system could be to simplify a model with an arbitrary out-of-core method, edit it, and stream the original samples through a volumetric variational colorization and deformation field constructed on the simplified model, such as the one based on radial basis functions by Botsch et al. [BK05]. Compared to such an approach, our system offers at least two benefits. First, the spatial finalization structure built during the sampling allows to efficiently and locally upsample the model during the interactive session. Second, our color and deformation reconstructions are fast, avoiding any global variational minimization (for which, for instance, local editing with displacement painting may require too many constraints), while providing visually accurate and plausible results: in the context of large objects, this speed comes as a key property in a time-scheduled professional context. Lastly, one could consider making a specific or texturing or FFD method size-insensitive . This is possible, for instance, with volumetric deformation fields. However, we believe that large objects should not impose a particular modeling method. Our system is generic, which means that not only arbitrary editing techniques can be used for manipulating the shape and the appearance of the large object, but also that several can be **mixed** within the **same** session: for instance, by only considering the couple $\{P_S, P_S^*\}$, we allow the user to start her work by a globally smooth deformation, then to continue with bone skinning for articulated parts, before ending with displacement painting such as in recent popular 3D tools (e.g., ZBrush). This flexibility, and the possibility to upsample on-demand specific areas is the strength of our system.

Limitations During the development of this system, we have almost systematically traded accuracy for efficiency. Consequently, at least three limitations can be exhibited. First, our streaming deformation may cause local self-intersection on highly deformed areas, which is an issue with many existing multi-scale editing techniques. Second, the quality of the initial downsampling strongly influences the

smoothness of the final color and deformation. This is one reason for which we have included on-demand local refinement, as it is difficult for the user to predict the number of samples ultimately required during the interactive session. Note also that all geometric prediction (e.g. curvature) can fail why colorization, since high frequency color variations may not be correlated to the geometry. Finally, the major drawback of our color editing method is also its strength: this is a *parameterization free* tool for texturing large objects, which means flexibility and efficiency as demonstrated throughout this chapter, but which also implies that its "3D painting metaphor" is slightly different from usual 2D painting software [Ado06] and requires for CG designers and artists to change their habits. This is also the reason why 3D painting is still an active research field: retrieving in 3D the accuracy of popular 2D painting packages **is** a challenge that would also induce new interaction metaphors.

Summary   We have proposed a *size-insensitive framework* to interactively apply texturing and FFD techniques to large objects. By size-insensitive, we mean that the in-core memory footprint does not depend on the size of the original object, but rather on the complexity of the user-requested modification.

This system features 3 new algorithms:

- an adaptive out-of-core simplification algorithm based on *VS-Trees* and *Spatial Finalization*,

- a smooth colorization method providing a fast color transfer between model at different scales without parameterization,

- a feature preserving deformation, the projected barycentric coordinates, able to transfer a deformation from a simplified model to its original version.

One main advantage of our system is its ability to directly work full resolution meshes and point clouds, without requiring any (possibly long and feature-missing) conversion to other representations (e.g., subdivision surfaces). The choice of meshless techniques for both out-of-core streaming processes not only ensures efficiency but also provides flexibility, as they can be seamlessly used on point sampled data, standard manifold meshes, as well as polygon soups (topologically inconsistent shapes made of multiple disjoint surfaces, that are quite frequent in CG applications for the entertainment industry).

Perspectives   Defining a totally scalable system means that the simplified models themselves become out-of-core. Even though we have not encountered the case, one could imagine that when applying numerous local refinements, P_S and P_S^* would ultimately be too large to fit the in-core memory. Thus, we are working on a system that reuses the spatial finalization to implement a kind of *Least-Recently-Used* caching system between the in-core and the out-of-core memory. This mechanism is compliant with the usual workflow for interactive shape editing: first apply global texturing and deformation on a coarse in-core model, then recursively refine the model to apply more and more localized modifications, that does not involve the whole object. Actually, we have already a satisfactory solution for colorization, as described in Annex 10.
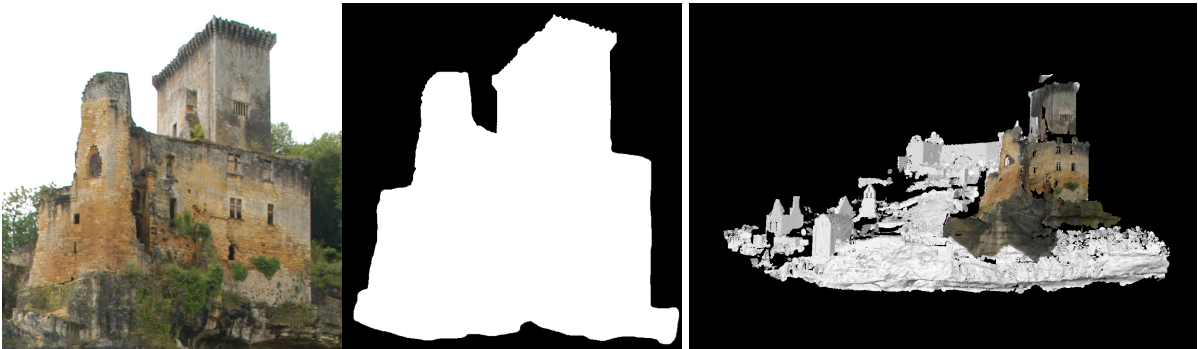
Finally, our system inherits an important limitation of FFD tools: the global topology of the object cannot be edited. Removing this issue is one of our future research directions: an important step in this direction has been made in [JZH07].
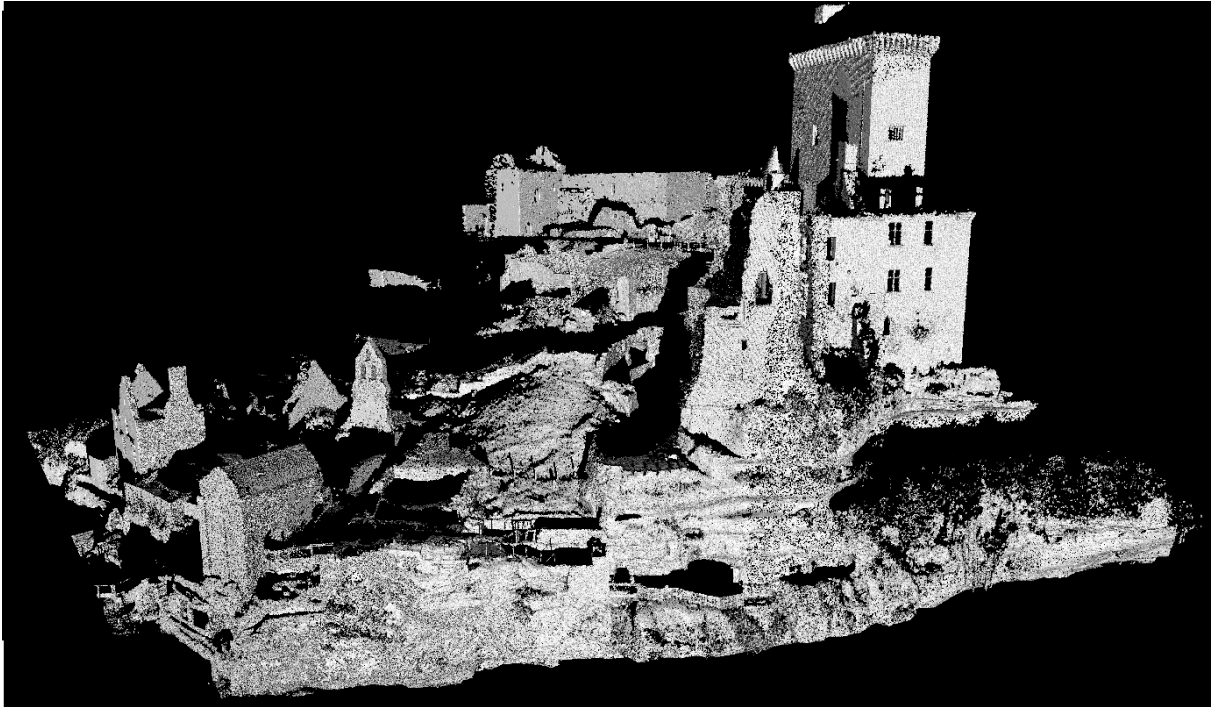
(a) Laser Range Scan of the Chateau
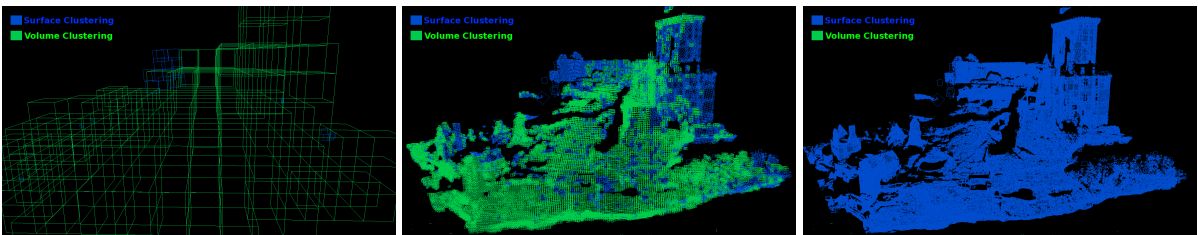
(b) Volume-Surface Clustering

(c) Interactive-Out-Of-Core Texturing

Figure 4.15: Application to large scale environnement editing. *(a) 7 millions point-samples coming from the registration of 6 scans of a castle. Each scan has been obtained using a time-of-flight scanner, suitable for distant and large scale objects. (b) VS-Tree clustering: man-made objects quickly appear during the clustering. (c) Interactive out-of-core texturing: using several photos and some texture patterns of stone, wood and grass, the environnement model is enhanced with color-information for each point in a full size-independent stream process.*

Part II

Rendering of Acquired Geometry

Chapter 5

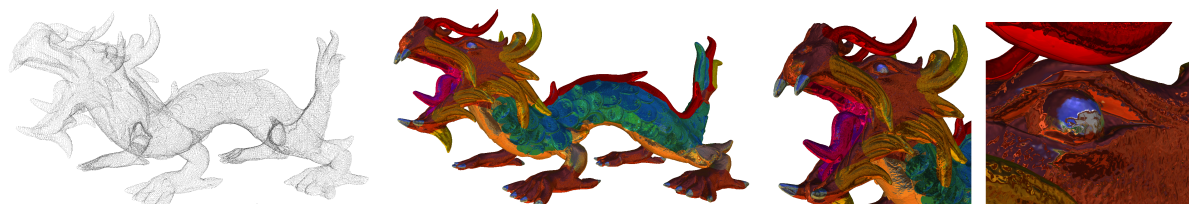# Point-Based Surface Rendering with Surfel Strips

Figure 5.1: *The Asian Dragon point-based surface can be rendered with surfel stripping at its full definition of 3.6M points, with antialiased 2D texturing and cube mapping, at 31 frames per second at a display resolution of 1600x1200 pixels.*

This chapter addresses the second problem often present in the acquisition pipeline: the visualization of Point-based Surfaces (PBS). Such surfaces are directly obtained after point-based processing, such as noise filtering, and require *hole-filling* rendering methods due to their possibly high sampling rate. Prior work in PBS visualization has essentially focused on the design of a new kind of rendering algorithms, called *point-based rendering*, which peform an image-space surface reconstruction by considering the surfel as the unique rendering primitive. This class of algorithm has been extensively studied over the last few years, and various GPU implementations offer reasonnable results. Unfortunately, such techniques induce two drawbacks: first, the performances are not competitive with polygonal rendering, due to the native support of polygons by graphics devices for polygons; and second, since no object-space entities exist beyond the points, a large part of the huge repository of polygonal rendering techniques is not compatible with point-based rendering.

In this chapter, we propose a new hardware-friendly approach to the problem of visualization of PBS: a *polygonal* approach. The goal of our work is to efficiently merge 3D models represented as point clouds in state-of-the-art high quality polygonal 3D renderers, providing an additional layer between point-based modeling and polygonal rendering.We claim that a polygonal interface can be generated and maintained efficiently between the point-based surface and the hardware. More precisely, we present a new technique for fast local meshing and multiresolution rendering of PBS called Surfel Strippingn where *Surfel Strips* are topological entities — composed of small triangle strips that interpolate the PBS — designed for efficient generation and GPU rendering.

Basically, the idea is to generate polygons upon the PBS as efficiently as possible to *feed* the GPU,

producing hole-free rendering. There are two major contributions which build upon the general ideas of fast hierarchical partitioning and lower dimensional geometry processing defended in this thesis.

First, at loading time, we equip the PBS with a weak topology targeting visualization only. This is done by first generating a set of overlapping small triangular meshes that interpolate the PBS using a *lower dimensional Delaunay triangulation*. We then remove redundant triangles and finally strip the small triangular meshes by using a cache-friendly stripping method. All these operations are performed by using an octree data structure.

Second, we reuse this data structure for providing a multiresolution interactive visualization of the surfel strips at rendering time. Since *Surfel Stripping* is local and very fast, it can be used in a lot of situations as an object-space alternative to the image-space surface splatting and thus be considered half way between point-based rendering and surface reconstruction. Rendering Surfel Strips is very efficient since it neither requires multi-pass rendering nor time-consuming vertex/fragment shaders compared to surface splatting. We show also how to exploit the locality of the surfel strips for maintaining compatibility with point-based modeling tools, such as local deformations of surfaces. We finally give some examples of well known visual enrichments developed for polygons, directly applied to PBS thanks to surfel strips.

5.1 Context: Visualization of Point-based Surfaces

The interest in PBS visualization has grown significantly in recent years in the computer graphics community. Several authors have already explained the reasons of this popularity [AGP*04], e.g. the widespread use of 3D acquisition devices that directly generate PBS, or the riddance of connectivity management that greatly simplifies many algorithms and/or data structures.

The basic idea to use points as rendering primitives can be attributed to the seminal paper of Levoy and Whitted [LW85]. However, rendering a sufficiently large amount of points at interactive framerates only became feasible when an efficient point-based rendering system was presented by Grossman and Dally [GD98]. Their work initiated a highly growing interest towards point-based graphics, and we refer the reader to [AGP*04, KB04] for a complete survey of point-based rendering. It is now widely admitted that when including additional information at each point [KV03], such as normal vectors, colors or material properties, and using specific rendering techniques (mainly to efficiently fill the holes that may appear between the points), PBS can become as flexible as the ubiquitous polygonal surfaces. Following Pfister et al. [PZvBG00], such enriched points are commonly called *surfels*.

A large variety of rendering techniques for PBS have been presented in the literature and all have to solve the central problem of hole filling when points are projected on the screen. They can basically be classified in three families (see also Figure 5.2):

- **Surface Splatting** which runs in the images space by blending ellipsoids centered on points;
- **Raytracing** which cast rays through pixels, intersecting a continuous approximation of the PBS;
- **Patching** which perform a local object-space reconstruction, enabling direct rasterization.
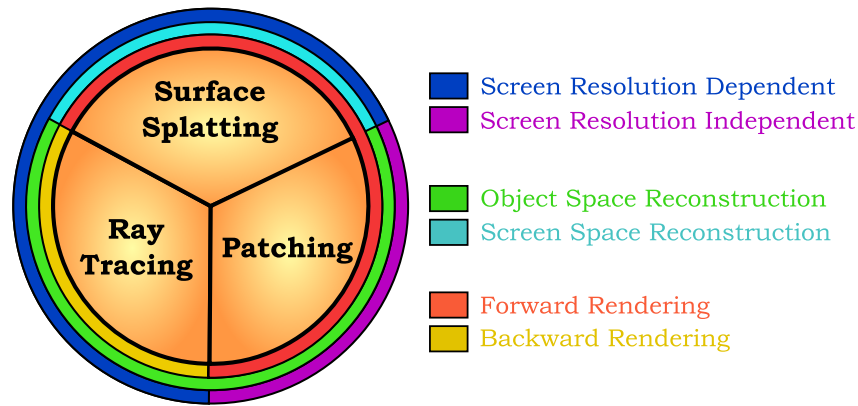
Figure 5.2: *Point-based rendering classification.*

Surface Splatting   Most of point-based rendering methods are based on *splatting*, where a reconstruction kernel (e.g. gaussian convolution) is centered at each projected point to fill the neighboring pixels. The accumulation of the contributions from all the kernels can be considered as an image-space surface reconstruction that is generated on the fly. This approach has a lot of advantages, such as filtering and antialiasing, and thus enables high-quality rendering. Unfortunately splatting also involves a totally different graphics pipeline, compared to the one used in current 3D graphics hardware. As a consequence, even advanced hardware implementations of splatting techniques [BSK05, GBP06] have to resort to expensive combinations of vertex shaders, fragment shaders and multi-pass rendering to finally obtain a surface that could have been rendered directly if its equivalent polygonal expression were available. Moreover, the intrinsec image-space reconstruction makes harder the scalability of such techniques to high-definition display.

Surface Splatting methods can be divided in two groups: *quality-oriented* methods and *efficiency-oriented* methods.

One of the early papers in the former group is undoubtedly the work by Pfister et al. [PZvBG00], who first introduce the idea of surfel and local screen filling around screen-projected points (i.e. "splatting"). This work has then been extended by Zwicker et al. [ZPvBG01, ZRB*04], with the *EWA Surface Splatting*, one of the most popular point-based rendering techniques, which is based on the screen space formulation of the Elliptical Weighted Average (EWA) filter, initially proposed by Heckbert for antialiased texture mapping on polygonal meshes [Hec86b]. EWA splatting enables high-quality anisotropic filtering and EWA splats can be implemented on programmable GPUs [RPZ02, BK03, GP03, BSK04, BSK05] and even directly as special hardware devices [WHA*07]. However, surface splatting suffers from limitations due its image-space accumulation principle. One example of such weakness is the case of transparent surface rendering requires complex ordering in the drawing calls [YZ06, GBP06]. Similarly, depth-of-field [KZB03] and deffered shading [GBP04, BSK05] has to be redesigned for fitting surface splatting.

Second, there are performance-oriented approaches, which are mainly based on specific data structures for efficient rendering of very large point sets, such as 3D scanned objects. The early member of this family is the QSplat technique developed by Rusinkiewicz et al. [RL00] as part of the Digital Michelangelo Project [LPC*00]. This kind of technique has also been used in hybrid point-polygon rendering systems [DVS03, CN01, CAZ01, DH02, CH02, GM05]. Actually, these algorithms do not propose a solution to the so-called hole filling problem: their basic principle is rather to use a point-based representation to provide an efficient level-of-detail rendering for complex polygonal meshes, than to provide a true rendering solution for point-based surfaces.

Ray Tracing   Ray tracing of PBS induces the non trivial question of intersecting point cloud with a line. Obviously, the probability of intersecting a line with a point in a 3D space is infinitely small. The simplest solution is to replace rays with cones, shafts or cylinders: the surface intersection point would thus be obtained by considering the closest point to the ray origin in the cone/cylinder/shaft. Unfortunately, this solution produces view-dependent intersections, leading to poor image quality [SJ00].

To overcome this problem, most of PBS ray tracing algorithms locally approximate the point set with a continuous surface and consider the intersection with this substituted surface [AA03, Wal05]. Usually, MLS projection is a good choice for such an approximation [ABCO*01]. The polynomials basis used for evaluating the surface can be precomputed and cached in a kD-Tree, then used for maintaining a logarithmic intersection cost. Alternatively to moving least squares, weighted least square can be used for deep enough trees (i.e., dense enough PBS). In the case of animated PBS, the kD-Tree is replaced by a bounding sphere hierarchy [AKP*05], allowing a progressive update of pre-cached data.

Patching   The algorithm proposed in this chapter can be considered as a *patching* process. There is very little work in this field, but the basic idea of generating a set of object-space patches "onto" the point clouds has several advantages in term of fast rendering. Combined with their introduction of *Point Set Surfaces* based on the MLS approximation, Alexa et al. [ABCO*01] implemented a first point-based rendering technique quite related to ours, rendering a PBS as a collection of overlapping two-dimensional parametric patches that locally approximate the surface. For every patch, a quad mesh is generated by sampling the parametric domain of the underlying bivariate polynomial. Since the patches are generated independently, it is obvious that the resulting surface is not even C^0 continuous. Moreover, as neighboring patches do not share common normal vectors and colors on their boundaries, a *visual smoothness* for the rendered surface is only achieved when employing a very large number of patches, which actually never interpolate exactly the point cloud. Thereafter, Linsen et al. have proposed the Fan Clouds [LP03], which are triangle fans constructed on surfels k-neighborhoods. This method is somewhat related to the idea of lower dimensional meshing presented in the context of surface reconstruction by Gopi et al. [GKS00]. However, these solutions do not propose a complete rendering solution for PBS and their k-neighborhood basis avoids a larger area analysis in the lower dimension, as we will discuss further. More recently, Wicke et al. [WOG05] have proposed a conversion of point-based surfaces to polygonal surfaces with textures. In a way, this work as a similar goals to ours: providing an interface to polygon-based software and rendering techniques. Unfortunately, their global approach requires a heavy preprocess (more than 20 minutes for half a million points).

We propose an efficient object-space patching method based on a set of small pieces of triangulated surfaces that we call *Surfel Strips*. Surfel Strips can be quickly generated while loading the PBS either from a local disk or from some network, and are stored in a specific octree-based data structure, the *Stripping Tree*. It is important to notice that, despite the use of triangles for rendering, Surfel Stripping is not a point-to-mesh reconstruction technique (a complete discussion on this topic can be found in Section 5.4), since we preserve the integrity of the underlying PBS, by only generating indexed polygons over it. In other words, the core representation of objects is still the PBS. The Surfel strips are used to fill quickly the topology naturally required for polygon rasterization, and since they are purely locally generated, they can be locally updated during some point-based modeling session, where common point-based tools are used to modify the shape of the 3D object (see Chapter 4).
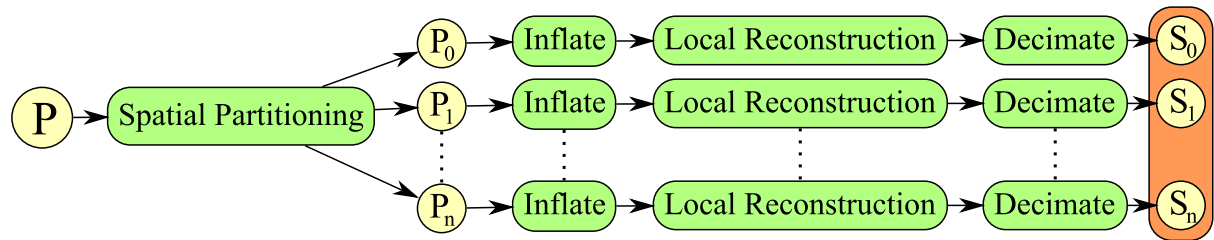
5.2 Surfel Stripping

Figure 5.3: *Overview of the algorithm.*

As said above, the basic principle of Surfel Stripping is to convert the initial PBS into a set of rendering primitives, called *Surfel Strips*, indexed onto the PBS. In fact, the rendering of polygonal primitives, as performed on current graphics hardware, requires two arrays: a *geometry buffer*, usually defined by listing the connectivity of vertices, and which can be filled by the PBS in our case, and an *index buffer* (topology), made of polygons in the case of meshes, and by definition missing for PBS. The goal of Surfel Stripping is precisely to provide an efficient way to fill the index buffer in the case of PBS.

Definition Since *triangle strips* [ESV96] are the most efficient 3D primitives in current hardware, we define a Surfel Strip as a small 2-manifold strip of triangles that locally interpolates a subset of a PBS (see Figure 5.4). We recall that a triangle strip is lossless compression of triangle list based on the local shared ordering induces by common edges: for instance two adjacent triangles are usually represented as list of six indices:

$$\{v_0, v_1, v_2, \ v_1, v_2, v_3\}.$$

Strips exploits the partial duplication that exists in this list for encoding a triangle as the last two indices plus a single new one, leading in our example to the list:

$$\{v_0, v_1, v_2, v_3\}.$$

In order to keep a single reference per surfel strip, we use *degenerated* strips: two strips can be joined by duplicating the last index of the first and the first index of the second one, creating a primitive with empty geometry but enabling a unique list for disjoined pieces of surfaces.

When the original PBS includes additional information at each point, such as colors or texture coordinates, the Surfel Strip automatically inherits them on a per-vertex basis.
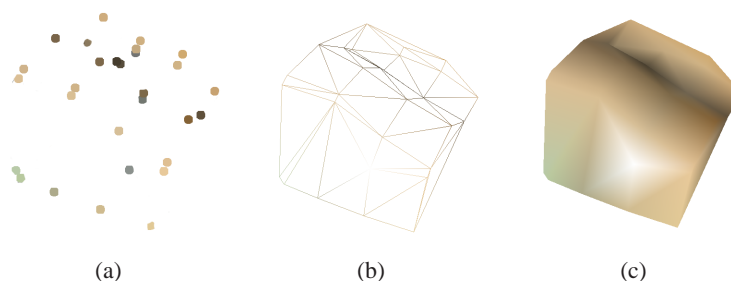

(a)   (b)   (c)

Figure 5.4: *The Surfel Strip principle: (a) small subset of the initial surfel set, (b) local connectivity information is computed, (c) resulting Surfel Strips rendered with Gouraud shading by using a per-vertex normal and color.*

This latter behavior is an important characteristic of Surfel Stripping: *all* the data that exists in the original PBS is exactly transmitted to the Surfel Strip structure. In other words, there is no compression or low-pass filtering as in usual splatting techniques [ZPvBG01, BSK04]. Of course, filtering *is* sometimes interesting, mainly when there is some noise in the initial PBS. But in our opinion, it is preferable to remove noise at the point-based level, with for instance [PG01], rather than spending computational effort at *each* rendering frame to low-pass filter the point set.

In addition to its ability to efficient hardware rasterization, such a localized primitive also provides a coarser granularity for many aspects of the rendering process: a large amount of operations (e.g. discarding tests for culling, see Section 5.2.5), can be performed at the Surfel Strip level, instead of at the point level, reducing the number of different tests to perform in a space-coherent fashion.

Once the idea of using local triangle strips for a hardware-friendly visualization of surfels is set, there are still three fundamental problems to solve to get an efficient and accurate system:

- How to efficiently generate each individual Surfel Strip? This can further be divided into two sub-problems: the efficient computation of the local connectivity and the efficient generation of the triangle strip from the connectivity.

- How to guarantee that no holes will be visible between neighboring Surfel Strips? In other words, we want an object-space hole-filling algorithm, similar to the image-space hole-filling provided by conventional splatting techniques.

- How to take benefit of the data structures constructed at loading time in order to propose an efficient rendering and in order to locally update the "visualization layer" provided by the *surfel strips*.

The next section details the algorithm that we propose to solve these two problems.

(a)   (b)   (c)   (d)   (e)
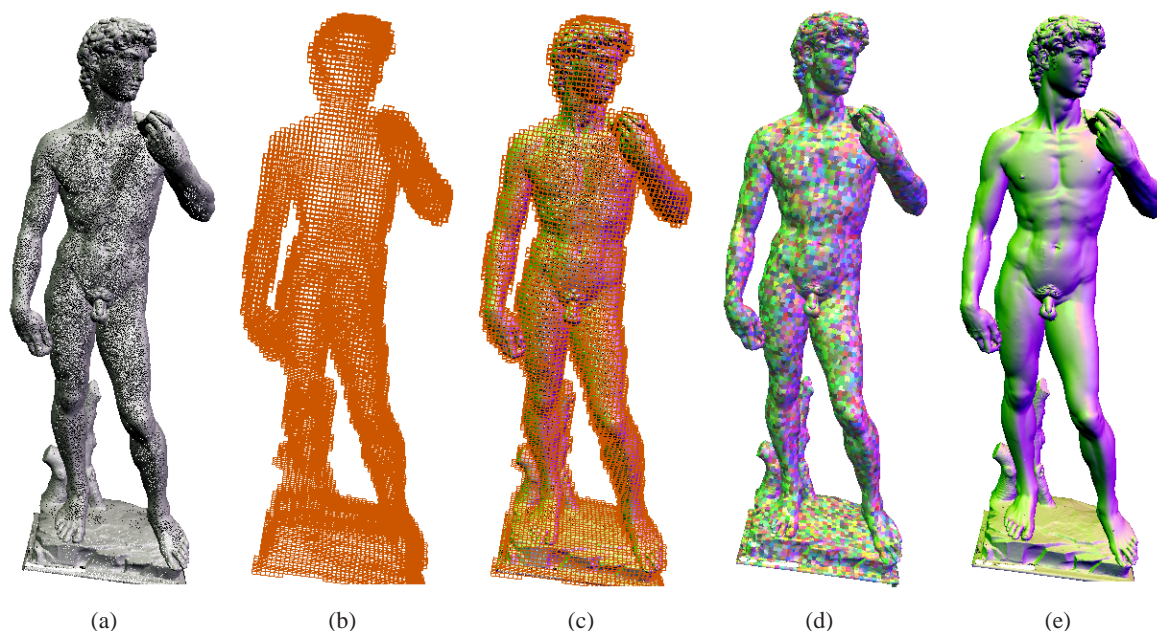
Figure 5.5: *The different steps involved in Surfel Stripping: (a) initial surfel set, (b) corresponding Stripping Tree space-partitioning data structure, (c) a Surfel Strip is generated at each leaf of the Stripping Tree, on an* inflated *local surfel set, (d) after* decimation*, most of the overlappings have been discarded, (e) real-time rendering using 3 colored light sources.*

5.2.1 Lower Dimensional Triangulation

The *Surfel Stripper* is the core of our system: it can be seen as a blackbox that inputs a small subset P_i of the initial PBS and outputs a triangle mesh patch S_i, the ground topology of an upcoming surfel strip. A canonical method to create triangles from an unstructured set of points in an n-dimensional space is the Delaunay triangulation. However, using a true 3D Delaunay triangulation to reconstruct a 2-manifold in 3D is usually not very efficient, as this process generates a lot of interior (i.e. volume) triangles that have to be find and removed to keep only the triangles that lie on the surface. This is not triavial in the case of non uniform point sets and actually a waste of time in our case.

In order to generate only "surface" triangles, we propose to perform a 2D Delaunay triangulation by projecting P_i on a lower dimensional object, i.e. an average plane Π_i. Indeed, this process greatly speeds-up the meshing but imposes another constraint in the partitioning: P_i must be consistent with a *height map* representation (i.e. each point can be expressed as an elevation along the normal of an average plane). We will explain latter how to reach this constraint during the hierarchical partitioning. This 2D approach reduces the generation time by about one order of magnitude (see Figure 5.6).
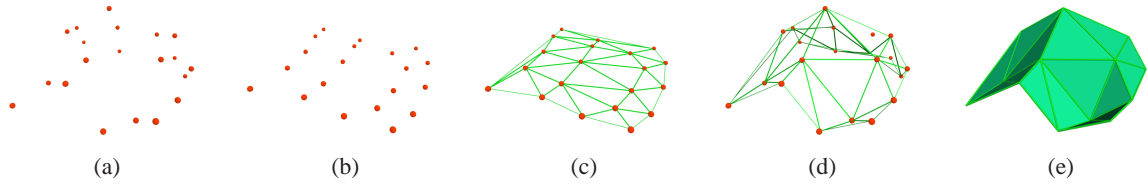

(a)   (b)   (c)   (d)   (e)

Figure 5.6: *Local surface reconstruction performed by the Surfel Stripper. (a) Initial partition P_i. (b) Projection onto Π_i. (c) 2D Delaunay triangulation. (d) 3D projection. (e) Surface patch S_i indexing P_i.*

We define Π_i by the centroid of P_i and a normal vector that can either be obtained by using *Principle Component Analysis* on the covariance matrix of the surfel positions of P_i (the eigenvector associated with the minimum eigenvalue), or by simply averaging the normals of P_i when they are available. We use an adapted version of the *Incremental Randomized Delaunay Triangulation* [Dev98] on the projection of P_i (see Algorithm 3).

Algorithm 3 Incremental Randomized Delaunay Triangulation

Require: $P_i \in PBS$
  $S_i \leftarrow \text{boundingTriangle}(P_i)$
  for each p randomly choosen in P_i **do**
    for each $t \in S_i$ **do**
      if $p \cap circumCircle(t)$ **then**
        $S_i \leftarrow S_i - t$
        $T_p \leftarrow T_p + t$
      end if
    end for
    $E_p \leftarrow \text{all edges in } T_p \cap S_i$
    for each $e \in E_p$ **do**
      $S_i \leftarrow S_i + Triangle(p, e)$
    end for
  end for
  return S_i

Thanks to the random insertion of samples, this algorithm exhibits $\Theta(n \log n)$ complexity where n is the

number of surfels in P_i. A typical size of n in our implementation is between 20 and 40, which offers the best overall performance for the entire Surfel Stripping process.

The connectivity information generated by this 2D triangulation forms a patch S_i indexed over P_i. However, a *strict* partitioning would lead to a set of disjoint patches, with holes in-between their borders. We propose an *inflate-and-decimate* approach to solve this problem.

5.2.2 Inflate-and-Decimate

The set of surfels submitted to the Surfel Stripper is determined by the initial partitioning that will be detailed in the next section. In order to avoid holes between Surfel Strips, we improve the local triangulation by proposing an efficient two pass technique, called *inflate-and-decimate*, which reduces the set of useless triangles while still maintaining a hole-free visualization.

Inflation The *inflation* pass takes place before the Delaunay triangulation: we extend S_i by including the nearest surfels from neighboring partitions of P_i (see Section 5.2.4). The inflation can be *conservative* by including all the surfel of the neighboring space partitions, or *aggressive* when a density estimation is provided. This inflated surfel set P_i is then triangulated using a 2D Delaunay algorithm as detailed above. As a result, obtaining overlapping surface patches, we fill the holes in the object space (see Figure 5.7).
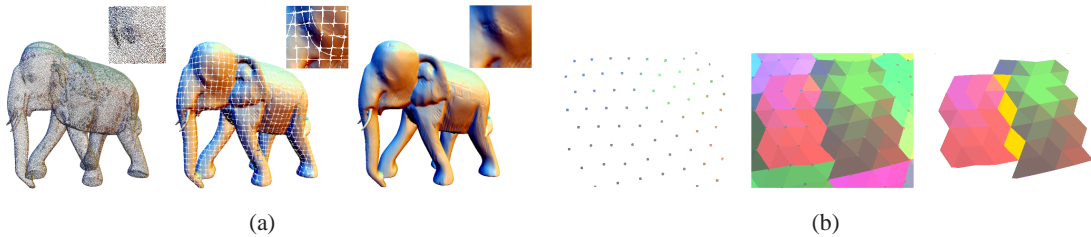

(a)   (b)

Figure 5.7: *Hole filling through overlapping. (a) (a) In yellow, the overlapping zone between the two neighboring surfaces. (b) From left to right: the original point-based surface, the aggregation of generated surfaces respectively without and with overlapping. Even under a strong close-up, the visual continuity is maintained.*

Decimation The *decimation* pass is done after the triangulation: we compare the resulting triangles of S_i with the neighboring patches that have been generated so far and discard useless triangles in overlapping zones. This decimation pass is based on a classification of the triangles. In this classification, established for its low computational cost, a triangle can have one of the four following states:

- **outer**: the triangle does not share any surfel with the original surfel set of P_i,
- **redundant**: more than one instance of the triangle is present in the overlapping zone (i.e.perfect overlapping, very frequent thanks to the Delaunay triangulation),
- **dual pairs**: the triangle forms, with a triangle sharing a common edge, the dual configuration of two triangles present in a neighborhoring partition,
- **valid**: in all other cases.

(a) Overlap   (b) **outer** removal   (c) **redundant** removal   (d) **dual pairs** removal
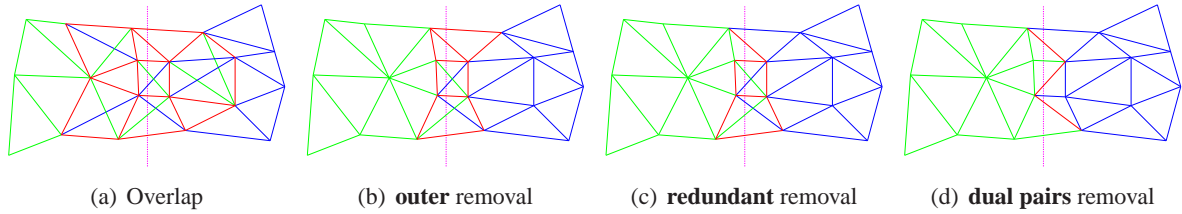
Figure 5.8: *The decimation pass: two overlapping triangulations with shared edges shown in red.*

Discarding *outer* triangles ensures that the overlapping zone will be only a thin band of triangles in the worst case. An instance of a triangle is removed of the current inspected cell when it is *redundant*. The *dual pairs* of triangles representing geometrically the same quad have not to be kept to ensure a hole-free vizualisation (see Figure 5.3 and 5.7). The *valid* triangles are maintained and are used for the rest of the algorithm. This set of *valid* triangles, quickly detected by the use of this classification, does not certify a watertight triangulation, but considerably reduces the number of overlappings between the small neighboring triangulations. We have made this choice in order to keep the global processing as fast as possible. A finer classification and an additionnal local remeshing rule could lead to a watertight triangulation under some sampling criteria, but this is not useful for our visualization purpose and is also time-consuming. Indeed, one nice property (observed in experiments) of this *inflate-and-decimate* process is that it leads to patches with boundaries that match perfectly in more than 99% of the cases. This surprisingly good result can be explained by the local uniqueness of the Delaunay triangulation, that resists quite well under projection in medium curvature areas. So, very often, the same set of triangles are generated in the overlapping zones of two neighboring inflated patches and the decimation process will then perfectly remove the overlapping triangles. A typical example is shown in Figure 5.8.

Note that using "neighboring" Surfel Strips may appear somehow in contradiction with our claim that we do not generate explicit connectivity between the strips. In fact, there is no real contradiction here because we only use the connectivity of the space-partitioning cells and do not explicitly stitch the strips together. Finally, the only annoying case where the decimation step cannot totally remove the overlapping, arises when the sampling density vs. curvature rate is too small. In this case, a different connectivity may be generated for surfels that belong to the overlapping zone of neighboring inflated strips. This is due to the very different orientation that may occur for the average planes that are computed in two neighboring cells in such high curvature areas. When this case arises, we simply keep the triangle of the inflated Surfel Strip to maintain a hole free visualization without strong artefacts (see the close-up view on Figure 5.15).

This inflate-and-decimate process is efficient, robust and very easy to implement. The usual approach, developed in computational geometry [CSD02], to stitch boundaries of partial triangulation by computing an adjacency graph, is much more complex, requires a precise computation, and has to examine a large set of configurations to find the case where neighboring triangles must collapse. As we only seek for a hole free visualization, the proposed technique perfectly fits our requirements.
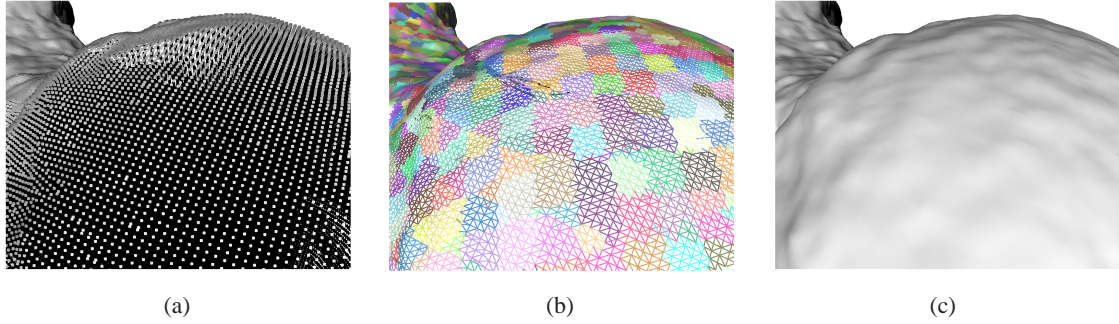
Figure 5.9: *The Surfel Stripper: (a) the initial PBS, (b) the collection of Surfel Strips with random colors, almost every overlapping triangles have been discarded (c) the final Gouraud shading does not suffer from the remaining overlappings.*

5.2.3 Fast stripping

In order to speed-up rendering and compress the patch topology overhead, each patch is stored as a triangle strip rather than individual triangles. Several approaches have recently been proposed to perform a direct stripping during the Delaunay triangulation [VK03]. Nevertheless, due to the decimation step involved in our approach, it does not make sense to generate strips before the final set of triangles is actually known. We have found that the fast-stripping algorithm proposed in [RBA05] works extremely well to strip our small sets composed of about 50 triangles (e.g., 5.10^{-5}sec. to strip 50 triangles on a P4 1.8 GHz). For every leaf node of the Stripping Tree, a cache-friendly *half-edge data structure* is computed by storing the 3 half-edges at each triangle as a vector. This nicely aligns the half-edges in memory and reduces each half-edge access to one pointer de-referencing. The stripping is then done in a similar way to STRIPE [ESV96]. Note that since the strips are computed separately in each leaf, they are constrained to the local space-partition of the leaf. Of course, this makes the strips smaller and so less optimal concerning data overhead, but as a result the strips will be more "culling-friendly" than usual long strips which may be visible from many viewpoints and thus limit the ability of the rendering system to perform a tight hierarchical back-face and frustum culling (see Section 5.2.5).
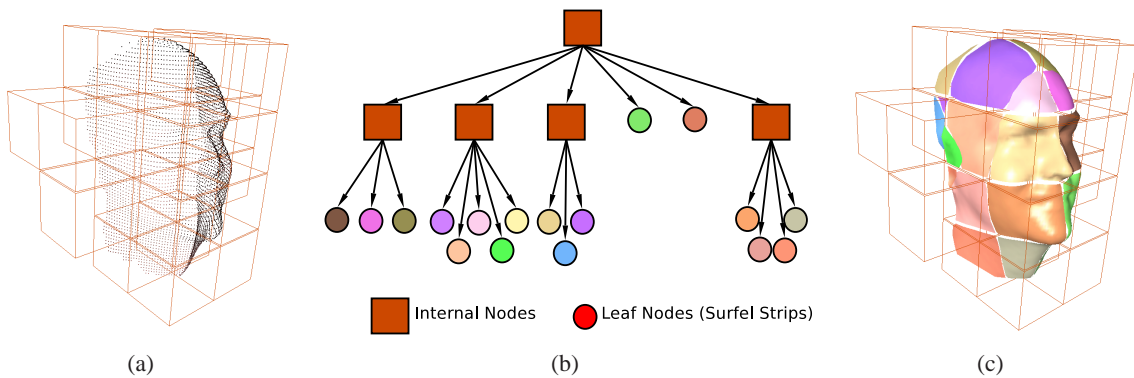

Figure 5.10: *The Stripping Tree structure: (a) the partitioning of the input surfel set, (b) the adaptive tree with the Surfel Strips on its leaves. (c) a Surfel Strip is generated for each cell (with a random color for each cell).*

71

5.2.4 The Stripping Tree

After having detailed the Surfel Strip primitive and the Surfel Stripper algorithm, the last component to focus on is the *Stripping Tree* data structure that is used to efficiently subdivide the initial PBS in a way that it is consistent with the constraints required by the Surfel Stripper. Actually, almost any usual space partitioning technique (bounding sphere hierarchy, BSP-tree, kD-tree, octree) may be used, as long as a consistent split criterion can be defined. In our current implementation, we use an octree-based bounding box hierarchy. Each internal node of this hierarchy contains:

- the bounding box of the whole set of surfels belonging to its subtree,

- the average position, normal and color of its subtree,

- a cone of normal vectors used for fast culling,

- 2 to 8 references to its children nodes

Each leaf node contains a Surfel Strip (see Figure 5.10).

The generation of the Stripping Tree for the PBS is based on the main constraint of the Surfel Stripper: a Surfel Strip can only represent a height field. Consequently, we have to partition the PBS into a collection of height fields. The recursive construction is based on this local property. A node with an associated surfel set that does not satisfy this property is subdivided into 8 new nodes. We use the same criterion as the one described in Section 3.2.2. Note that a VS-Tree can perfectly be substituted to the octree here, preserving the remainder of this chapter unchanged.

The described construction has the advantage to quickly converge towards the PBS since the local height field property is reached after less subdivision steps compared to when using BSP trees or bounding spheres hierarchies. As explained in Section 5.2.1, the inflate-and-decimate process used by the Surfel Stripper implies the availability of neighboring space-partitioning cells. Instead of using a topological approach based on the tree to find the neighboring cells, we have found it more efficient to simply use a geometric predicate: the epsilon box-collisions with the current cell (i.e. a test whether the box distance is smaller than epsilon) are computed between other cells in a top-down process. Then any leaf cell that passes the test is added to the list of neighbors of the current cell, and its surfels are added to the inflated surfel list. To speed-up the process, a distance threshold may be employed to add only neighboring surfels that are close enough to the current cell either using an input density estimation or a heuristic. In our implementation, the distance threshold is set to 25% of the cell diameter.

In order to guarantee a good performance of the Surfel Stripper, the space-partitioning must also ensure that each leaf of the Stripping Tree does not have to handle too many surfels. This means that in addition to the height field criterion, we also include a *population criterion* that ensures that no leaf node contains more than k surfels. We have determined experimentally that constraining $k \in [20, 40]$ provides a good trade-off for the whole preprocessing step on almost every tested model, a tradeoff between:

- too large surfel strips, which are expensive to compute as the complexity of 2D Delaunay triangulation is not linear and does not provide good hierarchical culling, and

- too small surfel strips, which would lead to bad memory performance and too much overlapping proportionnaly to Surfel Strips size.

In the case of quite uniformly sampled PBS, this population criterion also constrains the geometric extent of all resulting Surfel Strips to be very similar, as can be seen in the random color visualizations (Figures 5.5, 5.9 and 5.11). This feature also offers some good properties for downsampling and LODs as will be discussed in section 5.2.6.

5.2.5 Rendering Surfel Strips

The Surfel Strip collection can be directly submitted to standard graphics APIs without the use of specific vertex/fragment shaders or multipass rendering. During the rendering step, the Stripping Tree is traversed top-down, and the per-node normal cone and bounding box are used for hierarchical backface and view-frustum culling according to QSplat [RL00]. As illustrated in Figure 5.11, hierarchical backface culling can reduce the number of rendered Surfel Strips by almost 50%, even performed at the surfel strip resolution. In other words, we test the leaves (Surfels Strips), which are the finer entities for our hierarchical culling and never test the triangles individually.
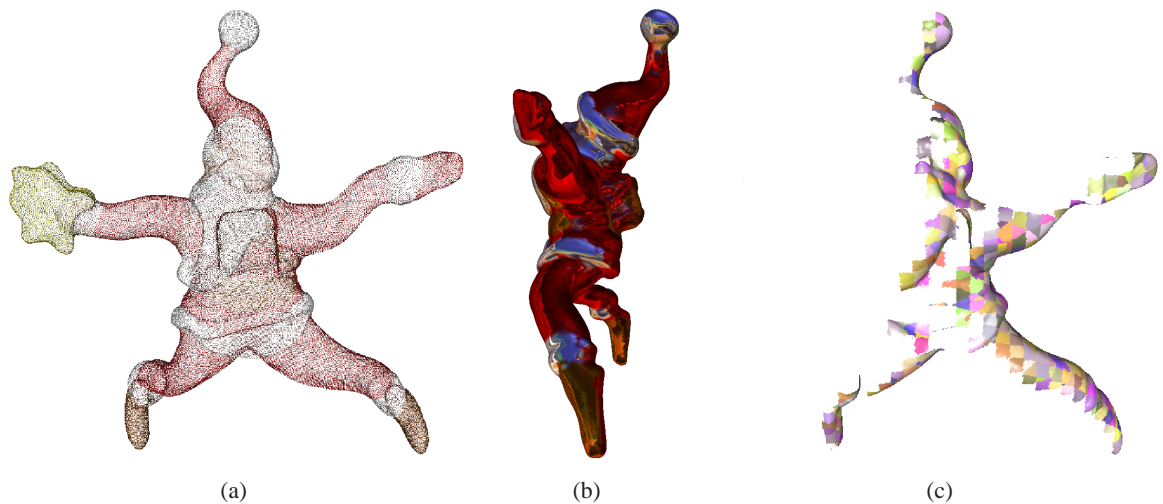

(a)　　　　　　　　　　(b)　　　　　　　　　　(c)

Figure 5.11: *Hierarchical culling of the Surfel Strips: (a) the initial surfel set, (b) the Surfel Strip rendering, (c) the actual subset of Surfel Strips that has been used (i.e. non-culled) for the rendering done in (b).*

5.2.6 Multiresolution Levels-Of-Detail

The main strength of Surfel Stripping is to be able to display complex point clouds on very high resolution displays while providing interactive framerates, which is of major importance in many different application fields like, for instance, precise archeological studies of scanned artefacts, or model validation in reverse engineering. So considered, there is currently no competitive point rendering technique that would be able to display the full resolution 3.6M antialiased textured and environment mapped point model presented in Figure 5.1 at 31fps on a 1600x1200 display (see discussion in Section 5). On the other hand, having only one high resolution representation of a given PBS is sometimes wasteful. Consequently, being able to switch between several levels-of-detail (LODs) would be a valuable extension of Surfel Stripping. In this section we present two different approaches for including multiresolution in the surfel stripping system.
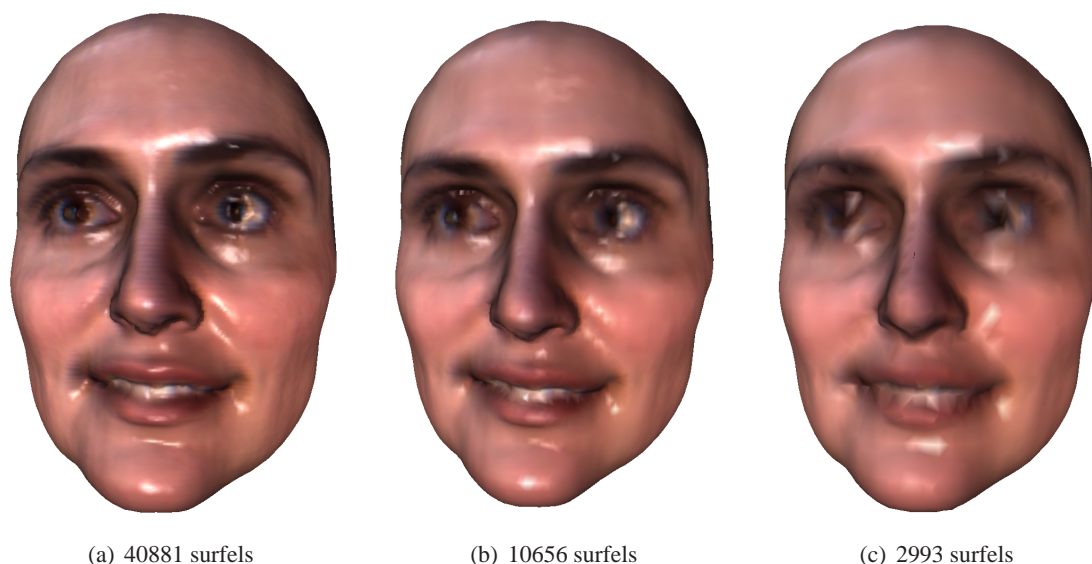
| (a) 40881 surfels | (b) 10656 surfels | (c) 2993 surfels |

Figure 5.12: *Surfel stripping for a PBS at different levels of details.*

Multi-resolution at generation time One of the main adavantages of point-based surfaces is their ability to quickly produce different levels of details of a shape. Rather than constructing a set of discrete levels of detail starting from the surfel strips at full resolution, one could prefer to take advantage of this good property of PBS by constructing a set of LOD directly on the point cloud, and then using the surfel stripping for each of these discrete levels. Near-optimal levels can be constructed using the different techniques presented in [PGK02]. Nevertheless, in order to speed up this process, we use a hierarchical simplification based on the stripping tree constructed at full resolution, by clustering points in a similar fashion to the algorithm described in Section 3.3. This fast approach offers convincing results in usual cases (see Figure 5.12). Its only weakness is that the preprocessing time and the memory footprint is increased by about 33% as with usual mip-mapping (each inner level contains approximatively 1/4th of the strips of its child level). As usual with discrete LOD, the selection of the current level is simply based on a distance criteria.

This solution does not involve any modification in the Surfel Strips rendering. However, being performed at generation time, it cannot provide a true view-dependent adaptivitiy.

Multi-resolution at rendering time Following [RL00] and [DVS03], we have integrated a multiresolution rendering scheme in the hierarchical traversal of our structure, performing a hybrid viewpoint-dependent point-strip rendering. This avoids unuseful *complete* drawing of too small or too far surfel strips and does not require any additional preprocessing.

As described previously, each internal node of the stripping tree carries a *representative surfel* — with position, normal and material attributes computed as an average of its children — and a bounding sphere enclosing all its leaves. During the depth-first traversal of the stripping tree, we compute the projected size of the bounding sphere of each of the nodes. When this size is less or equal to a pixel, we draw the *representative surfel* as a single shaded point, otherwise we continue to traverse the structure top-down, performing culling as mentioned above (see Figure 5.13).
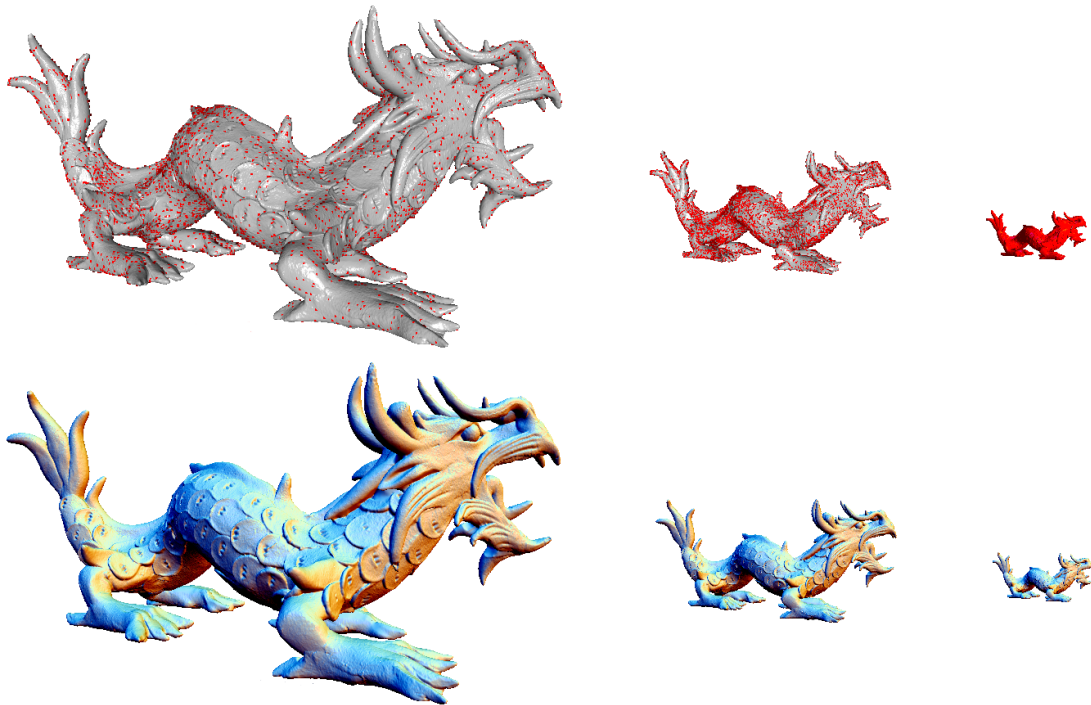
Figure 5.13: *Hybrid point-strip Multiresolution Rendering. Top row: the surfel strip and internal nodes drawn as simple point are displayed in red. Bottow row: final rendering, the aliasing is reduced thanks to the average color and normal used for representative surfels of internal nodes.*

Our experiments have shown that performing too expensive tests to decide **very** precisely when we have to render a single point (i.e. a pixel) or a rasterized primitive (i.e. a triangle strip) cannot offer the same framerates than our approach, because of the highly optimized rendering pipeline present in today's GPU, with which it is sometimes more efficient to render a small object than to decide whether we have to render it. Our approach represent a good trade-off, since:

- the tests performed will never reach the triangle level, but will be limited, in the worst case, to test if a *whole* surfel strip leaf partition (e.g. about 50 triangles) has to be fully rendered, or has to be simply replaced by a point; this induces a sub-linear complexity, even in the worst case;

- the *population* criterion mentioned above ensures a fine enough selection in practice.

While we do not perform the selection on the GPU, our mixed point-strip rendering reaches high framerates in practice (see Figure 5.1), preserving a low CPU workload and letting the vertex shader instruction set free for other tasks.

5.2.7 Interactive surface deformation

The ability of surfel strips to be generated considering only a small local set of surfels makes it possible to incrementally update the collection of surfel strips. For instance, this allows local point-based freeform deformations. Let us consider the Figure 5.14: on the right, the Santa model (75 783 surfels) has been loaded and a stripping tree has been constructed on-the-fly to provide a direct rendering of the model.

By using conventional point-based modeling tools [PKKG03], we have locally deformed and up-sampled

75

the top of the model, such as shown on the right of the figure. In order to keep an interactive framerate, we keep all the surfel strips which have not been modified, and recompute the surfel strips only for the top of the model. During the interactive deformation, the modified points are classified against the stripping tree, according to the following process for each modified point:

1. Each leaf cell containing the point is marked.

2. When the height children of a node are marked, we propagate this information bottom-up in the tree and the node is marked.

This allows to reduce the number of full traversals of the tree: during the classification of a given point in the stripping tree, we stop the top-down traversal as soon as a marked node is encountered. After having processed all the modified points, we recompute the cells marked as modified, and update in a bottom-up fashion the *representative surfels*, normal cones and bounding spheres of internal nodes.

A slight modification of the original stripping tree generation (Section 5.2.4) is necessary for allowing the user to enlarge some part of the model: the original bounding box used for the octree-based decomposition of the point cloud must be over-scaled, and we ensure that all the deformations applied to the model fit inside this enlarged bounding box. Note also that during the deformation, some points can move to "empty" space. In this case, the stripping tree will be refined in location where, at the beginning, no cells were present.

The updating time is 0.18 seconds in the example of Figure 5.14, and the original surfel stripping performed at loading time has taken 2.67 seconds. Note that, even if it is possible, we have **not** stretched the original surfel strips of the deformed zone, but completly recomputed them. This incremental update of the stripping tree reduces the computation in the case of freeform deformations. Of course, for particularly well identified deformations, such as bone-based skinning of characters, more efficient approaches can be used to limit the number of local surfel strip regenerations. Finally, the global interactivity, during the user freeform deformation, can be increased: following [PKKG03], a *lazy* update of our structure can be performed when deforming the object (in our case by simply "stretching" the strips for instance), and the *true* update is performed only once the deformation is finished.
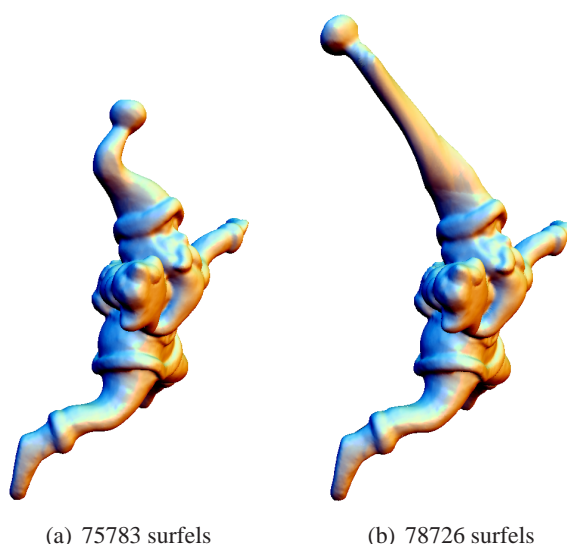

(a) 75783 surfels   (b) 78726 surfels

Figure 5.14: *Interactive deformation of the underlying point-based surface.(a) Original Surfel Stripping. (b) Local update.*

5.3 Results

We have implemented our visualization system under Linux with OpenGL. Running times and framerates are given for an Intel P4, 3.4 GHz with an nVidia Quadro FX 4400 GPU. All tests have been done by using vertex buffers.
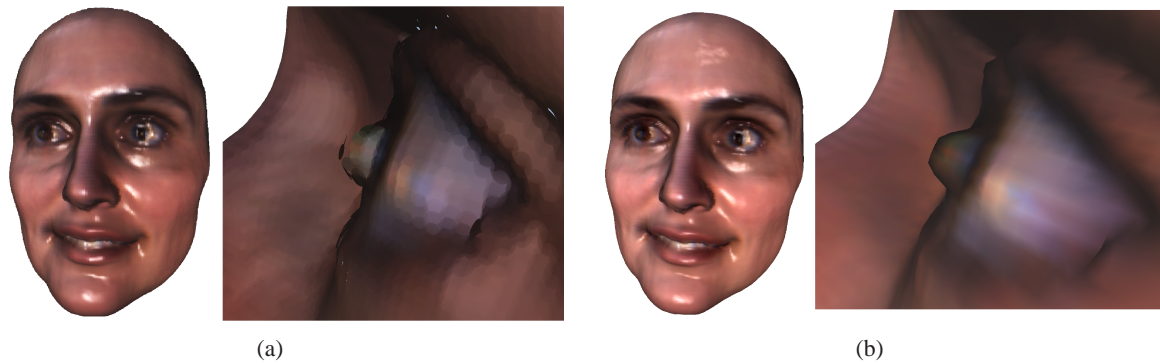

(a)   (b)

Figure 5.15: *Comparison of the visual quality. (a) The high quality EWA rendering (note the strong EWA artefacts on close-up views: lack of continuity for silhouettes and visible splat boundaries). (b) The same object at the same resolution rendered with Surfel Strips.*

Visual Quality   As pointed out by Botsch et al. [BSK04], Zwicker et al.'s EWA splatting [ZPvBG02] can be compared to Gouraud shading of polygons in terms of quality, since both techniques only blend colors and do not use per-pixel normal interpolation. As far as signal theory is concerned, it is true that both shading techniques have the same limit when the number of surfels/vertices grows to infinity, but actually the convergence rate is quite different: for a given number of surfels/vertices, Gouraud shading is closer to the limit shading than EWA splatting. This appears clearly on the left part of Figure 5.15: for the same number of points, EWA applies a stronger low-pass filtering and thus cancels much more details than the Gouraud shading provided by Surfel Stripping. Moreover, for close-up views, strong visual artefacts such as silhouette discontinuities and visible splat boundaries appear very often with EWA splatting (see the eyeball and the eyebrows on the right part of Figure 5.15). Although we did not perform comparison, this argument should remain true when comparing with Phong shading for both techniques [BSK05].

Another advantage of Surfel Stripping over EWA splatting appears when rendering non-uniform point clouds: Surfel Stripping takes benefit of the Stripping Tree to perform an adaptive reconstruction in undersampled areas, and generates a hole free surface with well distributed triangles, thanks to the underlying Delaunay triangulation. On the contrary, the hole filling approach of EWA splatting is based on an adaptive per-vertex radius. So, in order to be conservative, a large radius has to be used in undersampled areas, which produces a strong blurring effect in transition zones between undersampled and well-sampled areas.

In terms of quality of PBS rendering, Surfel Stripping should also be compared to *Phong splatting* [BSK04], as both techniques propose to generate a meso-structure for the rendering of a small set of surfels. A Phong splat strongly reduces its underlying surfel set by averaging the color information and by encoding the normal variation wih a quadratic function over the splat. Surfel Stripping offers much more flexibility as it interpolates (and thus preserves) *all* the position/orientation/color details included in the original point cloud, which is desirable in many applications. Furthermore, Surfel Stripping always

Model	Face	David	Bouddha	Asian Dragon
Points	40881	258332	543654	3609601
Surfel Strips	1612	17861	28757	89356
Preprocess	2 s	12 s	26 s	131 s
FPS	>200	167	121	31

Table 5.1: *Preprocessing time and rendering framerate for various models (rendering is done with antialiased 2D texture, cube mapping and 3 light sources, on a 1600x1200 screen resolution)*

keeps the true geometry of its surfel set, resulting in a better silhouettes preserving behavior.

However, splatting methods offers an high quality filtering when several surfels belongs to the same pixel. In particular, this reduces this aliasing effect. Considering that the Surfel Stripping is a polygonal method, the only alternative is to use super-sampling, which may significantly shrink the framerate and cannot offer competitive filtering with EWA splatting.

Finally, both approach can benefit from Phong interpolation and deffered shading [BSK05] when rich and expensive fragment shaders are used.

Performance   We achieved two different kinds of performance measurements: first, the preprocessing time required by the Stripping Tree and the generation of Surfel Strips by the Surfel Stripper, and second, the framerate that is obtained during the rendering by including the hierarchical culling and multiresolution rendering.

We performed tests on many different models up to a few millions surfels (only in-core models are allowed with our current implementation) and the framerate *never* fall below 31fps on a 1600x1200 resolution, even when simultaneously activating antialiased 2D texturing, cube mapping and 3 lights sources (see Table 5.1 and Figure 5.18).

The critical step for the preprocessing is the Delaunay triangulation. Initially, we thought that the popular Fortune's algorithm[For87] would provide better results than the incremental randomized one, but for small surfel sets, better performance cannot be clearly established. The choice of an incremental triangulation also allows progressive visualization combined with progressive data transmission.

Figure 5.18 illustrates the robustness of Surfel Stripping for various PBS, with different densities and complex features. Our experiments have realized a *visually* perfect, crack-free and hole-free rendering for every tested model.
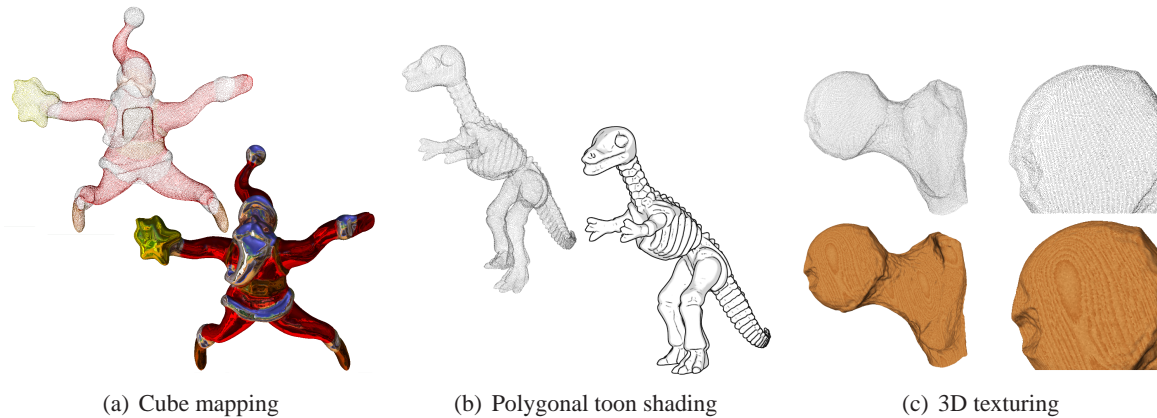
(a) Cube mapping   (b) Polygonal toon shading   (c) 3D texturing

Figure 5.16: *Surfel Strips can naturally* **directly** *benefit from the rich collection of polygonal rendering techniques, with many hardware-supported ones.*

Polygonal Rendering Techniques   Modern graphics hardware offers various extensions for specific rendering tasks. As Surfel Stripping is a pure object-space approach, all these specific hardware rendering techniques are automatically available. Figure 5.16(a) shows the reflection produced by using cube environment mapping when rendering the strips. Figure 5.16(c) illustrates another hardware-supported feature offered to PBS with of Surfel Stripping: volumetric textures which have a density distribution unlinked to the PBS one. This is an interesting propery when large flat parts (that can be represented geometrically with few surfels) require a higher definition for the appearance.

Note also that the framerate does not suffer from these additional effects, since they are hardware-supported and mainly take place in the rasterization unit of the GPU. Our approach also enables a large variety of alternative polygonal rendering techniques, such as non photo-realistic ones (see Figure 5.16(b)).

A last advantage of Surfel Stripping compared to image-based techniques is to be perfectly adapted for an easy integration of PBS in current rendering engines. Figure 5.17 shows the direct use of shadow maps with antialiased **Phong Shading** in scenes that combines polygonal models, spline models and point-based models.
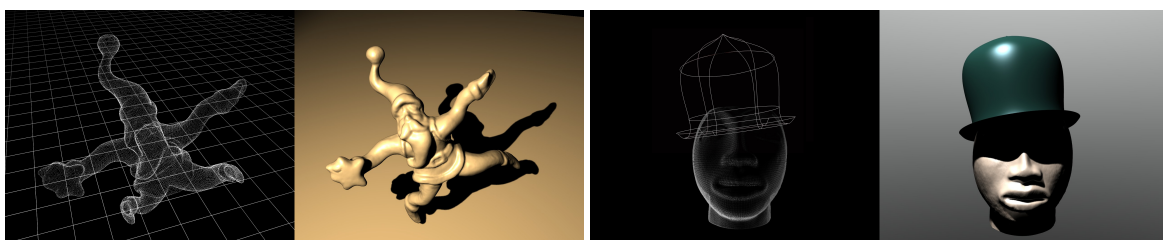
Figure 5.17:   *Surfel Stripping enables direct use of PBS in standard polygonal rendering engines. Here, two examples of antialiased Phong shading with shadow maps, merging PBS, meshes and spline surfaces.*

79
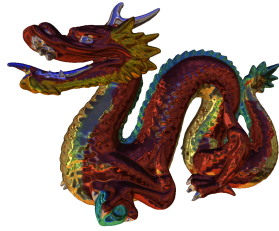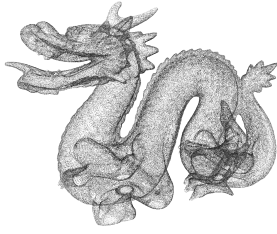
(a) Stanford Dragon with per-surfel color and cube-mapping (437 646 points).
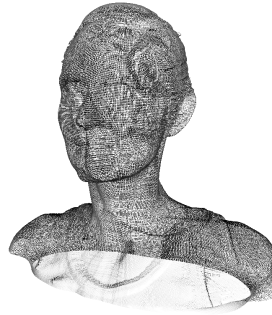
(b) Woman face (309 737 points)
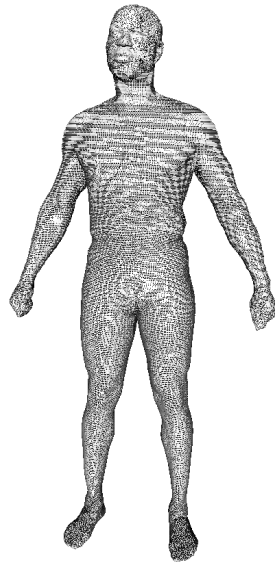
(c) Man body (146 616 points)
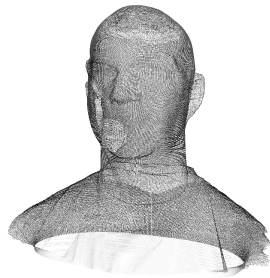
(d) Man face (303 382 points)

Figure 5.18: *Realtime OpenGL rendering of surfel strips (right) converted from colored point clouds (left).* The artefacts in shoulders are not produced by the surfel stripping, they were already present in the input data.

(a) Other Examples
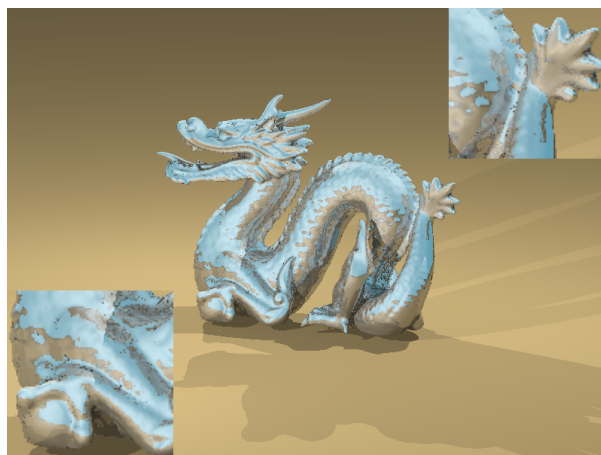
(b) Raytracing Surfel Strips

5.4 Discussion

The Surfel Stripping is somewhere inbetween point-based rendering and surface reconstruction techniques. On one side, it allows to quickly obtain a mesh indexed over a point-cloud by considering the strip topology. Even if there is no guarantee on the watertight nature of the resulting set of manifolds, this can represent a good enough results for many applications (e.g. entertainment industry). On the otehr side, Surfel Stripping provide a minimum (i.e visually continuous) topology for submitting a point cloud to the polygonal graphics pipeline, without adding or removing points. Furthermore, its intrincsec hierarchical principle offers at no cost a simple and efficient view-dependent adaptive rendering.

To our knowledge, there are at least two previous papers that include a similar idea of local (non watertight) triangulations: the visualization system proposed for point set surfaces by Alexa et al. [ABCO*01], and *Fan Clouds* introduced by Linsen and Prautzsch's [LP03]. Compared to the former, the rendering quality offered by Surfel Stripping is much higher, as it uses the position, normal and color information that exists at every single surfel, which is not the case for Alexa's technique, where the C^{-1} boundaries of the patches are apparent, since neighboring patches do not share common attributes such as normal information. Compared to the latter, both the rendering speed and the rendering quality offered by surfel stripping is higher: first, triangle strips offer better caching better triangle fans in the hardware graphics pipeline, second, fan clouds do not propose any hierarchical structure to generate efficient culling and third, the local Delaunay triangulation is more robust and respect better the geometry than a simple k-neighborhood fan construction, reducing the final number of overlappings to get hole-free rendering..

Scalability and GPU Friendliness   The standard pipeline used in 3D graphics hardware has been developed to scale efficiently when the screen resolution is increased. Thanks to the incremental computation involved in triangle rasterization, the framerate that can be achieved by hardware rendering is only slightly affected when switching from, say, 800x600 to 1600x1200. Unfortunately, the complex per-pixel operations involved in image-space splatting techniques, such as EWA splatting, break this nice property. This means that the user has to systematically find a trade-off between high-resolution rendering at low framerates and low-resolution rendering at high framerates.

This is not the case by using our approach, since it is totally based on the standard triangle rasterization, and very high framerates are obtained even for high resolutions (typically 120 fps at 1600x1200 for a PBS with 400k surfels). Another major feature of our approach, thanks to the standard pipeline, is that the rendering time of a single frame is relatively view-independent for a given number of surfels. The only component that can speed-up or slow-down the rendering time in that case, is the culling step that may discard a significantly different number of Surfel Strips from one frame to the others.

But as already said above, the main advantage of Surfel Stripping compared to standard point-based rendering is its GPU friendliness. The process only requires one standard rendering pass, which frees graphics hardware resources to include additional visual effects by using popular multi-pass rendering effects, such as *shadow maps*, *motion blur*, *depth of field*, etc. Actually, this was our initial goal when we developed our approach: be able to smoothly merge the rendering of PBS in current high performance 3D engines, such as the one developed for video games, with as little specific processing as possible. Finally, the Surfel Stripping is somehow for PBS what marching cube si for implicits and blobs, what recursive sampling is for subdivision surfaces and what parameter iteration is for NURBS: a ***fast tessellator for rendering***.

Limitations   Essentially, the Surfel Stripping fails in two situations:

- very non-uniform sampling of the surface: in this case, the surfel stripping will not be able to fill too large holes,

- very dynamic surfaces, such as fluid simulations: in this case, the very frequent updates of the strips can lead to a complete regeneration of the *Surfels Stripping* structure.

In our opinion, the first case is a sampling problem, and belong to geometric preprocessing, even with conventional splatting. The second limitation is still the big advantage of usual point-based rendering (see Keiser et al. [KAG*05]), even if ther per-surfel radius (for splatting) or the continuous approximaion (for ray tracing [AKP*05]) update belongs to some local neighborhood analysis and/or caching to ensure and fast hole-free visualization.

Summary In this chapter, we have presented both a fast stripping method for Point-Based Surfaces and a rendering system tuned for hardware rendering at interactive framerates. Our system provides an additional object-space layer between point-based surface and polygonal rendering, represented as small triangular strips, the *Surfel Strips*, organized in an efficient hierarchical structure. The main advantage of this system is its ability to be locally generated and updated, the natural preservation of the surfel properties such as position, normal and color, and the direct reuse of conventional polygonal rendering methods.

We have shown that, in various cases, Surfel Stripping represents an efficient alternative to existing high quality rendering of PBS that have been developed in recent years, since it neither requires a specific multi-pass rendering process, nor some expensive combination of vertex/fragment shaders. Basically, our combination of hierarchical culling, multiresolution rendering and strip-based rasterization provides, at high screen resolution, a significant speed-up of the rendering framerate, compared to current state-of-the-art high quality point rendering techniques.

Surfel Stripping can also be seen as an alternative to complete point-to-mesh surface reconstruction offering a fast solution to import colored PBS into standard 3D applications.

The Stripping Tree has been developed to quickly space-partition a PBS and offers an efficient access to neighboring cells. At rendering time, it provides an efficient hierarchical multiresolution rendering, particularly interesting for models made of more than one million surfels. Surfel Stripping is currently not the best solution for highly dynamic surfaces, such as fluid simulation, but a convincing solution in all other cases.

Perspectives In spite of its various optimizations and trade-offs, the Surfel Stripping remains limited to in-core models and the size of models coming straight from the acquisition pipeline can exceed its capabilities. Therefore, in order to provide an efficient visualization of **huge** point-based surfaces, we explain in the following chapter how to perform an efficient out-of-core appearance preserving conversion of such large sampled models, outputting *normal-mapped surfel strips*.

Chapter 6

Appearance Preserving Rendering of Large Point-Based Surfaces

Very dense point-based surfaces, composed of hundreds millions of samples, are more and more frequent and tend to become the typical output of modern geometry acquisition pipelines [LPC*00]. Offering an interactive visualization of such objects is fundamental in various situations, including preview at high screen resolution, merging in polygonal 3D engines and 3D databases browsing. Indeed, many of these applications share a common context: the rendering has to be performed in real-time and to remain as visually plausible as possible. Considering the rich surface description provided by large models, *appearance-preserving methods* are particularly well adapted: these techniques perform a conversion from high resolution geometries to lower resolutions ones, equipped with high-resolution normal and color maps. They often provide a very similar rendering to the original models, while their low polygon count ensures high framerates at high screen resolution. Unfortunately, such methods usually rely either on a global parameterization of the object or a full resolution mesh representation, which are both incompatible with direct visualization of large sampled surfaces. In this chapter, we address the lack of efficient visualization techniques for large sampled surfaces without preliminary full-resolution reconstruction.

We propose a fast processing pipeline enabling real-time appearance-preserving polygonal rendering of large point-based surfaces. Our goal is to reduce the time-slot required between a point set made of hundred of millions samples and a high resolution visualization taking benefit of modern graphics hardware, tuned for normal mapping of polygons. Our approach can be divided in two steps:

1. We starts with a combination of two elements already presented in this thesis: the out-of-oore simplification in streaming presented in Section 4.3 and the Surfel Stripping discussed in Chapter 5, for providing a polygonal rendering of a lower resolution version of the large object.

2. Therefore, the resulting coarse geometric representation is enriched by applying a set of maps which capture the high frequency features of the original data set. We choose as an example the normal component of samples for these maps, since normal maps provide efficiently an accurate local illumination. We call the resulting rendering primitive a *Normal Surfel Strip*. Nevertheless our approach supports straightforwardly other surface attributes, such as color.

The main contribution takes place in the second step, during which we efficiently reconstruct the normal maps from a stream of sampled normals. Sampling issues of the maps are addressed using an efficient diffusion algorithm in two dimensions. As a result, we obtain a set of enriched Surfel Strips, recovering the essential part of the original feature wealth present in the large data. This out-of-core process is

able to directly handle large unorganized point-based surfaces without the time-consuming full resolution meshing or parameterization steps, required by current state-of-the-art high resolution visualization methods. Contrary to pure point-based methods, our approach takes fully benefit from the hardware graphics pipeline, particularly efficient with normal mapping. One of the main advantages is to express most of the fine features present in the original large point clouds as textures in the huge video memory usually provided by graphics devices, using only a lazy local parameterization. Our technique has been tested on various very detailed scanned objects and statues, for which an interactive visualization has been obtained with a global preprocessing time that represents only a couple of minutes.

6.1 Context: Large Object Rendering

Visualization systems designed for large 3D objects (tens or hundreds of millions samples) can be divided in two categories: mesh-based and point-based systems. We refer the reader to [Tol99, Lin03] for an introduction to out-of-core methods and visualization.

Mesh-based system These systems require two specific preprocesses: first, the point cloud has to be reconstructed at full resolution, which can require upon days of computations, and second, they have to be converted in an out-of-core format allowing fast disk-to-GPU updates. For instance, the Adaptive Tetra-Puzzles of Cignoni et al. [CGG*04] propose to construct a diamond-based hierarchy over a large mesh, storing on disk set of geometric attributes directly in the GPU format, and fetching them on demand during the interactive visualization which can even be real-time (60 FPS) with models like the David (56M triangles). Since we rather seek for a system able to handle directly the point-based surfaces, we refer the reader to this paper for additional references.

It is interesting to note that even if working in a mesh context, several systems take benefit from the point primitive for saving rendering workload. For instance, since many neighboring samples of a gigantic object are often rendered over a single pixel, Gobetti et al. [GM05] have developed the *Far-Voxel* system, a hybrid mesh-point rendering system, where a cluster of polygons can be rendered as a single point, with a specific shader approximating the appearance of the underlying surface area. Raytracing has also been used for interactive exploration, but it often requires a cluster of computers [WDS04] or a very long preprocess for out-of-core level-of-details construction [YLM06]. Whatever the case, the framerate is not competitive with rasterization, and the main benefit of raytracing, such has advanced visual effects produced by secondary rays, restricts inevitably both framerates and screen resolution.

Point-based system QSplat [RL00] was the very first system designed for rendering large data set coming from 3D scanners. The algorithm starts by clustering the input set of samples in a bounding sphere hierarchy, which is a binary tree carrying a representative surfel, a bounding sphere and a visibility cone on each of its nodes. The nodes attributes are quantized in a multiresolution fashion over a 32 bits word (48 bits with color), and stored on-disk in a cache coherent order. Then, at rendering time, sub-parts of the tree are loaded according to the point-of-view and drawn using OpenGL point primitives. According to the computer capabilities, the tree is more or less refined at interactive framerates, offering a surfel for each single pixel after few seconds. Note that the idea of compressed point format for rendering has then inspired several systems, using octree [BWK02, DD04], hexagonal [KSW05] or wavelet [GM04] quantization.

The main advantage of the QSplat algorithm is its scalability, since even an object composed of several hundreds of millions nodes can be visualized at interactive framerate on almost any computer. However,

its main drawback is its highly dynamic nature, not easily amenable to a good GPU support. In fact, apart of the final point drawing, QSplat is intrinsically a software/CPU rendering system. As a result, even on a powerful computer, the rendering quality remains poor when moving the viewpoint, and the user has to stop its interaction for obtaining an high resolution rendering after several refinements.

To overcome a part of this problem, Dachsbacher et al [DVS03] have introduced the Sequential Point Tree, which performs most of the hierarchical culling on the GPU. Unfortunately, since all the information is kept at the original format, the size of the object is restricted to in-core models. Therefore, Wimmer and Scheiblauer [WS06] have proposed an out-of-core implementation, using a nested octree for managing an out-of-core forest of enhanced Sequential Point Trees, able to perform rendering without the normal information.

These GPU rendering systems clearly outperform the original QSplat rendering, but inherits its main weakness: their pure tree-based multi-resolution rendering does not allows the GPU to run in an optimal context, leading to either poor framerates or visible temporal artifacts.

Appearance-Preserving methods   In the case of objects exhibiting an important surface coherency such as human bodies, statues, cars, and so on (a tree with a low sampling rate being a perfect counter example), it is possible to perform a local analysis for converting the sampled geometry into a more GPU-friendly format: an *appearance-preserving* representation. Contrary to previous rendering techniques, appearance-preserving methods [COM98, CMRS98] do not try to render the whole original geometry. Instead, the object is dramatically simplified, and the features which are lost in this process are reintroduced in a set of high resolution normal maps, which will offer a very similar shading to the original object. This principle can be applied directly on a single simplified object, or combined with a multi-resolution structure, such as the *Progressive Meshes* [Hop96, SSGH01], for enhancing the different level-of-details.

Nowadays, the main advantage of appearance-preserving methods is their GPU-support: considering the fragment shader level [TCS03], the only thing to do to use normal maps is to fetch normal fragment from on-board texture memory and to use it for shading the fragment instead of the interpolated Gouraud value or Phong vector. Concerning adaptivity, normal map rendering offers two advantages: 2D normal maps can be compressed more efficiently than 3D geometries, using for instance the differential encoding proposed by Munkberg et al. [MAMS06] or their fixed rate compression scheme [MOSAM07], and multiresolution filtering can be cast as a simple mip-mapping process. Of course, using normal mapping involves meshes and some kind of parameterization, two notions present at the other extreme of the spectrum when dealing with PBS. In this chapter we fill this gap in a matter of minutes for scanned objects composed of several hundred millions samples.
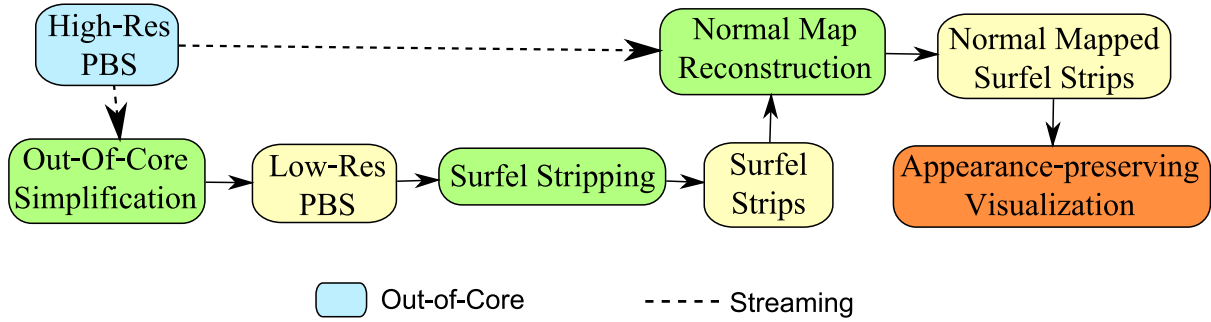
Figure 6.1: *Overview of our approach for interactive visualization of large models. The usual expensive step, the* meshing*, is only performed on a very reduced point cloud, and is no more the bottleneck. Most of the fine details are expressed through the normal maps, generated on a per-surfel strip basis with* 2D diffusion*, a faster process than 3D geometric reconstruction.*

6.2 Appearance Preserving Surfel Stripping

6.2.1 Overview

Our algorithm is described on Figure 6.1 and runs out-of-core with the following two-passes sequence:

- **pass 1:** Out-of-core simplification and fast meshing
    1. we perform an out-of-core simplification of the huge model using vertex clustering
    2. the resulting simplified point based surface is quickly converted into *Surfel Strips*, organized in a *Stripping Tree*, following the fast lower dimensional meshing of Chapter 5.

- **pass 2:** Normal field streaming and reconstruction
    1. *all* the points of the original model are *streamed* through the stripping tree and distributed to their corresponding leaves, where the point normal is projected onto a quad texture associated to each Surfel Strip (see Section 6.2.3). This streaming process is the key step of our technique, as its output sensitivity allows us to handle large models with limited memory. At the end of this step, each leaf of the Stripping Tree contains a low resolution Surfel Strips and a high resolution *sparse* normal map
    2. the holes present in sparse normal maps are filled by a *diffusion* algorithm, reconstructing a continuous normal fields which interpolates the original normals of the large point-based surface (see Section 6.2.4).

The resulting rendering primitive, present on each leaf, is called *Normal Surfel Strip* (i.e a coarse piece of mesh plus a high resolution continuous normal map) and is rendered in a similar fashion to usual surfel strips, with an additional fragment shader for exploiting the normal map.

6.2.2 Out-of-Core Simplification and fast meshing

Out-of-core simplification has already been discussed in Section 4.3. After many experiments, we have found that two algorithms can be used in the present context:

- when the input topology is simple and the sampling density uniform enough, a grid-based clustering, similar to the work of Borrel [RB93] and Lindstrom [Lin00], offers a convincing enough

result. This is the fastest simplification and it is known that it results sometimes in a poor quality. However, this not a problem in our case, since the resulting sampling is not used "as-is" but is then enriched with normal maps, which will be responsible for most of the final appearance

- when the input PBS has a more complex topology and density distribution or when an higher quality is mandatory, an adaptive sampling performs better, although in a longer time, and we reuse the simplification algorithm presented in Section 4.3 (VS-Tree forest built using spatial finalization).

Actually, a precise down-sampling is not mandatory in our particular case: we are not looking for the *n best* points to represent a given large model, we just need a point cloud capturing reasonably the original shape of the large PBS and which can be quickly tessellated with Surfel Stripping. So, we use mainly grid-based simplification, for which we have determined that a grid resolution of $2^{\lfloor \log_{10}(n) \rfloor}$, with n the total number of points, offers good results in practice and is extremely fast, as it processes up to 5 millions points per second on our workstation.

Note that in all the cases, we maintain a counter of the original point samples that have intersected a given cell: this information will then be used for choosing a normal map resolution in the second pass.

Finally, the resulting simplified point-based surface is augmented with Surfel Strips, distributed on the leaves of a Stripping Tree (see Figure 6.2).

| (a) | (b) | (c) |
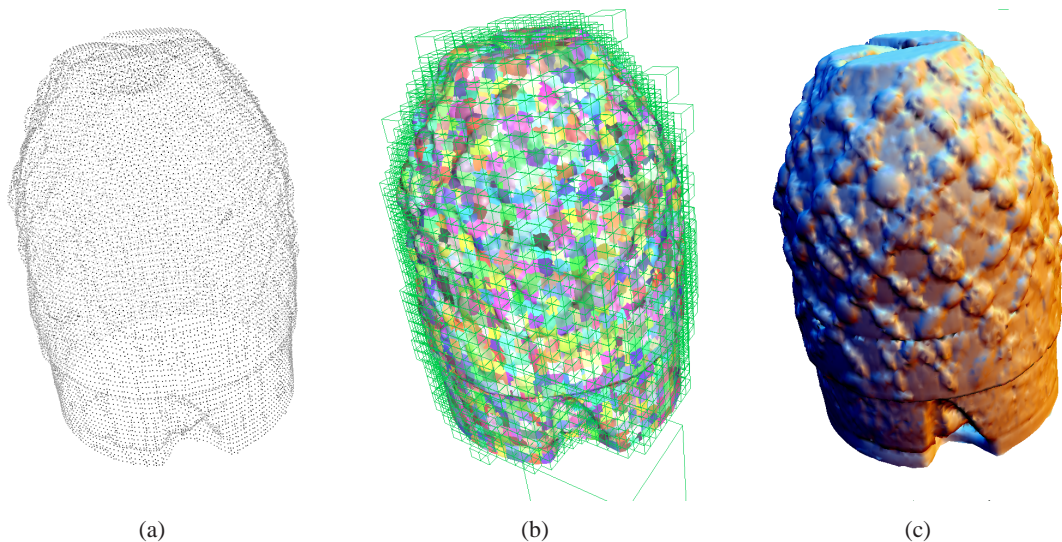
Figure 6.2: *First pass of our appearance preserving surfel stripping process. (a) The simplified PBS obtained after the out-of-core simplification. (b) The set of local meshes quickly generated thanks to the Stripping Tree partitioning (in green). Each colored patch corresponds to a surfel strip, locally generated in 2D. (c) The coarse polygonal representation obtained, made of local triangle strips interpolating the input points.*

6.2.3 Streaming Normals

At this point, the Stripping Tree of the down-sampled point-based surface is available, and can be visualized as a coarse representation of the original model (see Figure 6.2). In order to recover the original appearance of the large model, a *normal map* will be associated to each Surfel Strip. These normal maps are generated during a *normal streaming* process, where all points of the initial large object are streamed

through the Stripping Tree, performing an hierarchical classification to quickly find the set of Surfels Strips they belong to. This second streaming of the large model is purely local, dealing with one single point sample at the same time in the main memory.

One key idea in this part is that we generate local *disjoint* normal maps (one for each Surfel Strip), and ensure a *visual* continuity by enlarging their support: similarly to the *inflate-and-decimate* process used in the surfel stripping, we consider, for each leaf partition, an *inflated* normal field, made of all the normals belonging to the considered leaf, plus some normals belonging to neighboring clusters. In practice, we actually use only *aggressive* inflations, by just scaling leaf bounding boxes until embracing their whole associated Surfel Strip, including the overlaps with neighbors. Consequently, a single normal vector may belong to more than one Surfel Strip. Latter, at rendering time, the Z-buffer tests will selects one instance of this vector in overlapping zones for shading the corresponding pixel. As can be observed in Section 6.3, the thousand pixels coming from these overlapping areas do not suffer from visual artifacts. Algorithm 4 summarizes the normal streaming:

Algorithm 4 Normal Streaming

Require: T the stripping tree at low resolution
Require: $\{p,n\}$ a sample from the input stream
  Leaf node $c_p \leftarrow$ depthFirstTraversal (p, T)
  Project n on c_p normal map
  for each Leaf node c_p^i in the ε-neighborhood of c_p **do**
    if $p \cap inflatedBBox(c_p^i)$ **then**
      Project n on c_p^i normal map
    end if
  end for

Normal maps For each intersected leaf, the local parameterization of the point relative to its Surfel Strip is computed by projecting the point on the average plane used for creating the strip (see Section 5.2.1). Actually, we parameterize the projected point according to a bounding quad of the inflated partition and aligned to the two eigen vectors associated to the two highest eigen values of the covariance matrix, previously computed and stored on leaves during Surfel Stripping. This parameterization is used to fill the relative pixel value of the associated normal map with the normal vector of the streamed surfel. We use floating point textures, so if more than one normal is projected onto the same pixel, we just add the normal vector value to the existing pixel, and normalize all the normal maps after having processed all the points of the original model. This also prevents from aliasing artifacts that may occur. We refer the reader to the frequency analysis of Han et al. [HSRG07] for a discussion on better choices than average normals.

The resolution of each normal map is proportional to m the counter mentioned earlier and stored on each leaf, but is also rescaled according to the user-specified texture memory budget T. We define the average side resolution s for a texture map as:

$$s = \sqrt{\frac{m}{n} \cdot T}$$

with n the total number of samples. Similarly, its aspect ratio a_r is equivalent to the bounding rectangle of its Surfel Strip, leading to the final resolution $w \times h$:

$$w = s \cdot a_r \text{ and } h = \frac{s}{a_r}$$

Using a flat parameterization of a non-flat Surfel Strip may generate some distortions, especially in areas of high curvature that would result in a global loss of details. Nevertheless, in practice, the constraints

imposed during the construction of the stripping tree lead to close to planar Surfel Strips, which limit distortion and no artifacts were visible in our experiments. Since the normal maps are generated on a quad basis, they are easily packed into few large textures, eventually compressed [MAMS06, MOSAM07] and stored in the graphics card memory.
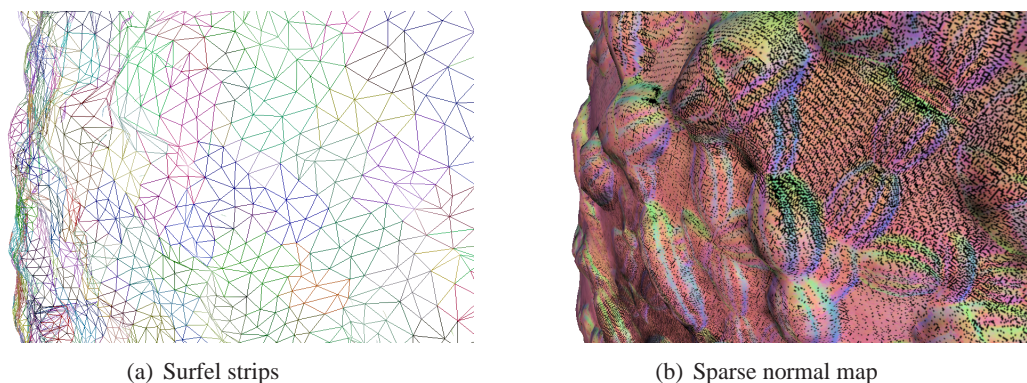

(a) Surfel strips   (b) Sparse normal map

Figure 6.3: *After the normal streaming step, a sparse normal map is attached to each Surfel Strip. (a) Coarse topology computed from the sub-sampled point cloud. (b) Color visualization of the spare normal map: pixels color is set with the XYZ coordinates of the normals. Black points corresponds to pixels of the normal map where no surfel as been projected.*

6.2.4 Normal Map Reconstruction

After the normal streaming process, each surfel strip is enriched with a sparse normal map since several pixels may not have been filled by projected normals (as shown in Figure 6.3). For using this map as a texture for our coarse surfel strips, holes need to be filled (black pixels in Figure 6.3). Many approaches have been developed over the years to fill holes in an image, which is a basic operation for image repairing. Exploration-based approaches such as [BWG03] directly compute an illumination value for a pixel given by exploring its neighborhood. On the other hand, iterative PDE-based approaches such as [PGB03], spread existing color in the image using PDEs such as Poisson equation, or diffusion equation. We use the PDE-based diffusion technique presented in [XP98], for its guarantees of continuity and smoothness. The implementation is based on a multigrid resolution scheme that first solves the problem at a coarser resolution, and then uses this coarse result to initialize the algorithm at finer resolution:

Solve (h, $Ax_h = b$)

1. Pre-smoothing steps: $Ax = b$

2. Downsample: $x_{h-1} = Dx_h$

3. **Solve (h-1, $Ax_{h-1} = b$)**

4. Upsample: $x_h = Ux_{h-1}$

5. Post-smoothing steps: $Ax = b$

where $Ax = b$ corresponds to the matrix formulation of discrete diffusion equation with finite differences, and h corresponds to the quadtree level associated with the resolution of the processed image (see also the Push-Pull algorithm in [GGSC96]).

The approach of [PG01] corresponds to a multigrid iteration with no pre-smoothing step, a single post-smoothing step and a specific down-sampling algorithm that only takes into account existing samples.

We inspire from [PG01] by skipping the pre-smoothing, and only using existing samples for down-sampling, but run the post-smoothing iterations until the convergence criterion is met. Indeed, without these extra iterations, some blocky interpolations are present in the texture we obtained, especially around holes.

The multigrid resolution algorithm proved to be very efficient in practice (see Table 6.5), only a few iterations (e.g. 5) were needed for convergence with 10^{-3} error bounds in most cases. Note that the same approach can be used to create maps for other scalar or vectorial sampled values, like colors or displacement vectors.

Figure 6.4 shows the resulting set of high resolution normal maps attached to the coarse surfel strips. Our reconstruction-by-diffusion process provides normal maps that express the essential part of the original large model appearance. These maps are stored as textures in the GPU memory, and benefit from the **automatic filtering** provided by the hardware mip-mapping. This property is quite interesting, since it can be interpreted as both an anti-aliasing process and an hardware supported multiresolution rendering, thanks to the different levels of the mip-mapping. Again, the recent studies in the frequency domain [HSRG07] open a higher fidelity filtering to normal maps.
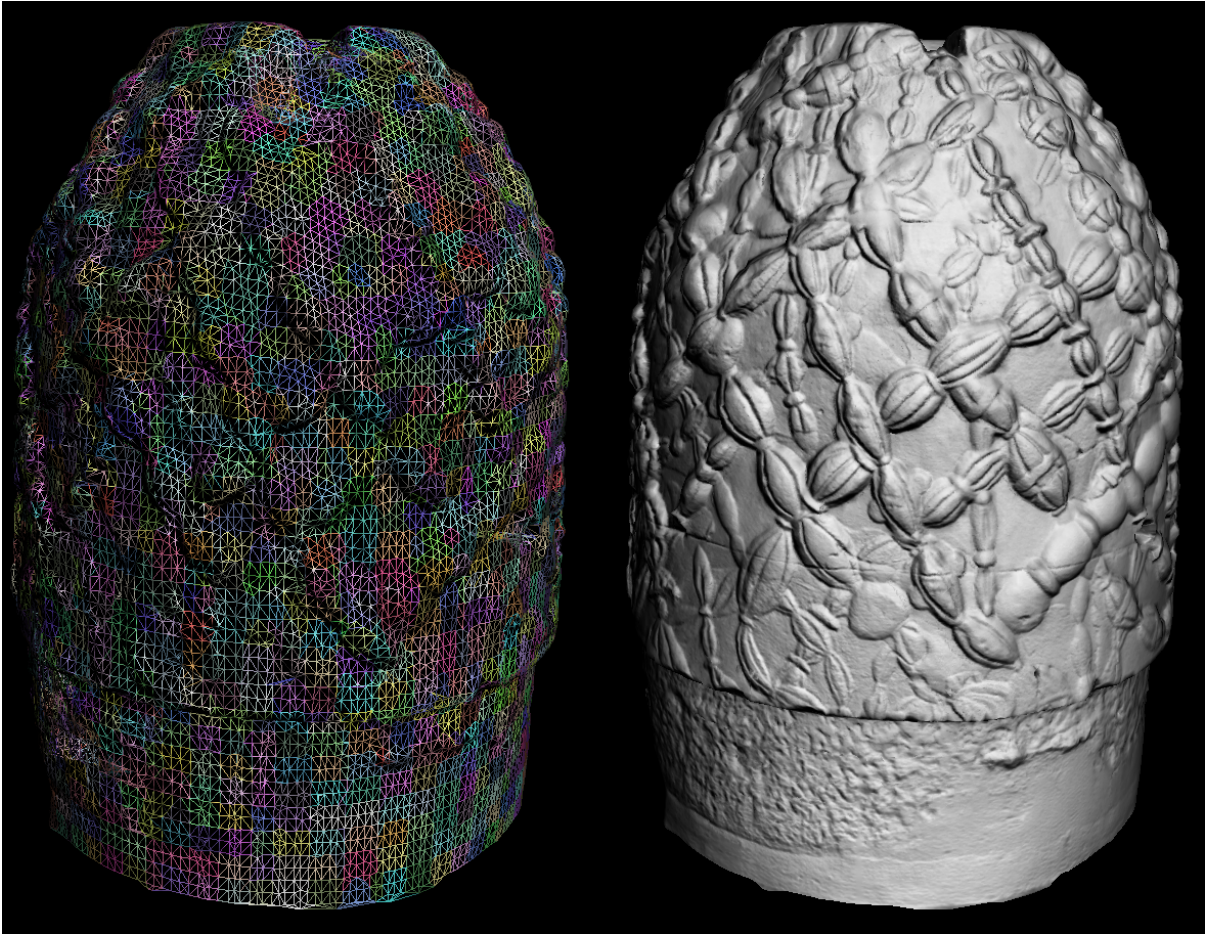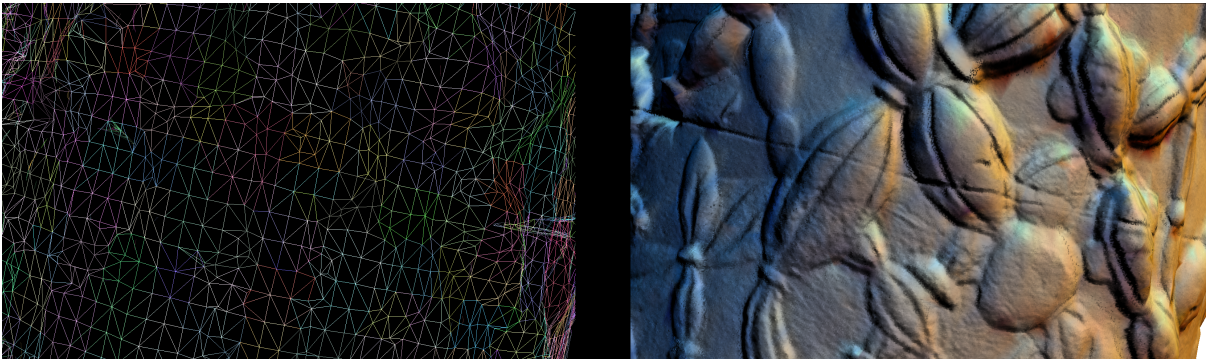
Figure 6.4: *Diffused normal map rendering. (a) The Omphalos model (11 664 466 points). (b) Close-up. Left: coarse surfel strips quickly generated after the out-of-core decimation of the large point cloud (random per surfel strip color). Right: real-time rendering, with per-pixel illumination using the reconstructed normal maps. Note the nice automatic filtering of the model appearance, thanks to the intrinsic hardware mipmapping of the normal maps.*

6.3 Results

We have implemented our system under Linux on an Intel PIV 3.2 GHz, 1GB RAM, 160GB UDMA HD, NVidia Quadro FX 4400. We use C++ and the OpenGL Shading Language (for normal mapping). We consider input binary files where points are encoded as an unorganized list of chunks of 6 floats (3 for the position and 3 for the normal). Table 6.5 summarizes the preprocessing times of our system. Figures 6.6, 6.10 and 6.12 shows the real-time rendering obtained on various large objects with our approach.

| Models	Omphalos	Column	Dancers	St Matthew	Atlas
Num. of points	11 664 466	22 877 845	31 620 449	186 810 938	250 000 000
TIMINGS					
Simplification	5 s	10 s	14 s	61 s	81 s
Surfel stripping	2 s	4 s	5 s	7 s	7 s
Normal streaming	45 s	151 s	213 s	667 s	890 s
Normal reconstruction	35 s	35 s	34 s	152 s	170 s
Total	100 s	201 s	274 s	887 s	1148 s
RENDERING					
Num. of Surfel Strips	1602	1721	2013	2457	2516
Num. of triangles	45 504	51 012	66780	79030	79967
Textures memory (MB)	68	71	94	185	201
Frames per second	> 200	> 200	198	165	164

Figure 6.5: *Preprocessing time and rendering framerate for various large models. The total timing represents all the steps needed for the preprocessing, starting from an unorganized point cloud on disk up to a ready-to-render data structure in memory. The framerates are given for 1600x1200 screen resolution.*

It appears that the normal map initialization is the main bottleneck. Obviously, tree-traversal and local projections involved in this out-of-core streaming remain costly since they are performed for each sample of the large model. Nevertheless, all the different stages involved in our approach are highly parallelizable (each point sample is treated separately), and can take benefit from recent multi-core CPUs (an improvement factor of 1.5 can reasonably be considered for dual-core CPUs). Note also that we use a pointer-based implementation of the Stripping Tree, which could be enhanced. Our resolution criteria for the normal maps works quiet well in most of the cases. Actually, even when a high density variation occurs inside a leaf of the tree, aliasing is prevented in the normal map thanks to the iterative diffusion step (see Figure 6.7). Note that the memory usage for textures is measured without compression. The hard-drive latency strongly influences the performances of *normal streaming* and *simplification* passes (we measure performances with the grid-based simplification, we refer to Section 4.6 for the adaptive case) . Better performances can be reached by using high-speed hard-drives (U-SCSI) and a dedicated workstation, where useless processes are stopped (usually between 20 and 30 on our Linux system).

The excellent framerates given in Table 6.5 are reached thanks to the highly optimized polygonal hardware graphics pipeline, particularly adapted to display low resolution polygonal models with high definition textures at high screen resolution.
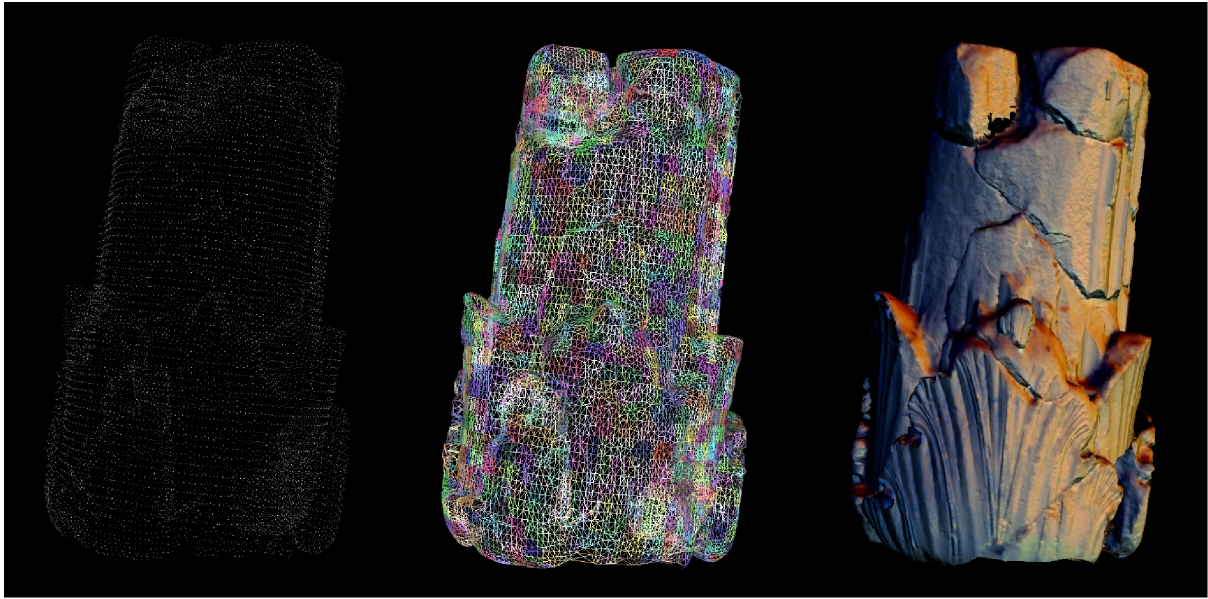
(a) Column (22 877 845 points)
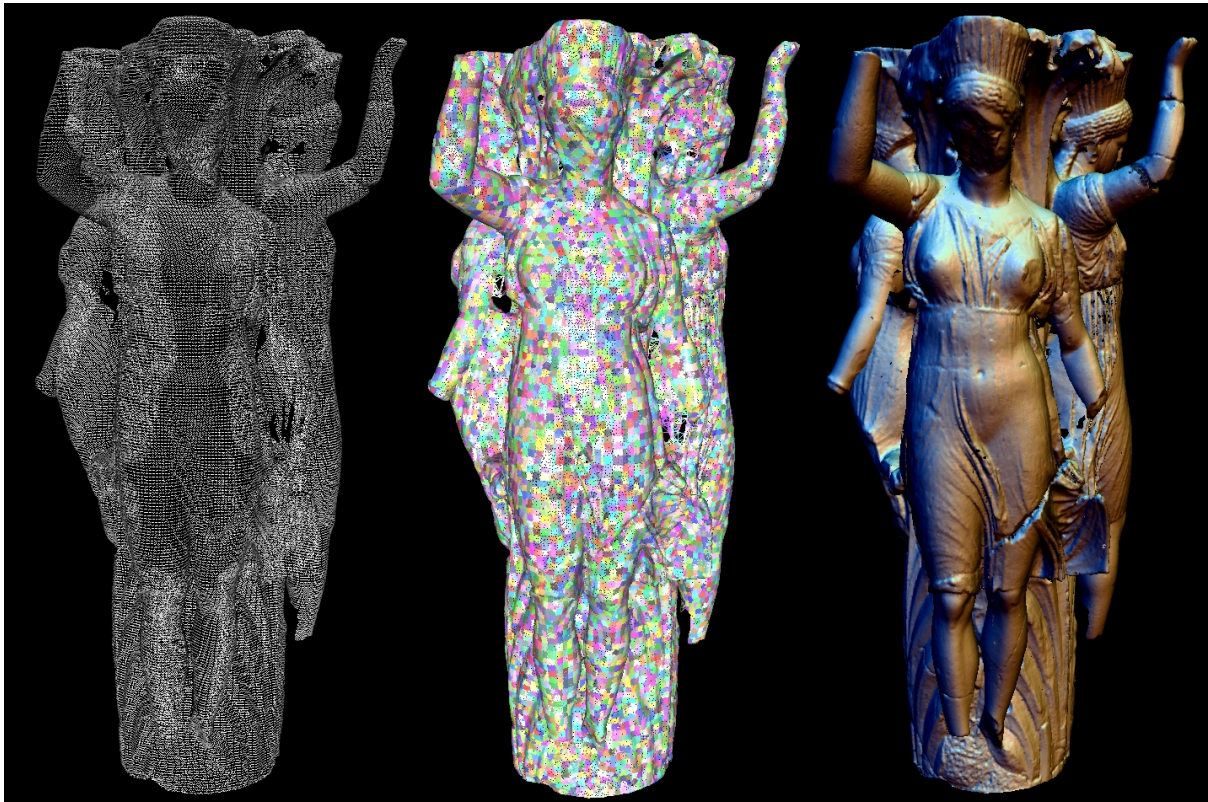
(b) Dancers (31 620 449 points)

Figure 6.6: *Visual quality for various large models. Antialiased rendering with 3 color light sources on a 1600x1200 screen resolution.* **Left:** *the sub-sampled point cloud decimated at the first out-of-core reading pass.* **Middle:** *the coarse Surfel Strip collection.* **Right:** *the interactive rendering of this collection, enhanced with normal map expressing the fine details, generated during the second reading pass of the point cloud (models courtesy EDF).*

6.4 Discussion

Comparison The critical point in our work was to reduce as much as possible the pre-process time needed for obtaining a convincing visualization of large point clouds. Compared to *QSplat* [RL00], our preprocessing is faster (one order of magnitude in the worst experimental case) and it does not require a previous surface reconstruction (huge additional processing time). Compared to the *Layered Point Clouds* [GM04], although we did not implement it, our preprocessing seems to be about ten times faster, according to the paper timings. Of course, these multiresolution methods are conservative, and do not perform a low pass filtering on the geometry such as ours, but from the visualization point of view, we keep the essential appearance thanks to high resolution normal maps reconstructed in 2D (see Figure 6.10).

Figure 6.7: *Upper part of the St Matthew model with our method rendered at 165 FPS, without (left) and with (right) the normal maps. The maps are recreated* **directly** *from the point cloud, providing a convincing appearance, while using less than 80k triangles (left image). Most of the "appearance" information carried by the original point cloud is directly stored through these normal textures on the GPU memory (185 MB) and used for the per-pixel lighting.*

Note also that our system provides a **polygonal** rendering, highly optimized on today's GPU. This allows us to reach high framerates at high resolution. Clearly, our approach provides results that also confirm [PGK02]: performing decimation on the point cloud and then applying reconstruction methods is definitely more efficient than meshing and optimizing the full resolution point cloud, at least for our visualization purpose.

Finally, the diffusion process of normal maps can be seen as a kind of surface reconstruction, where not the geometry, but the normal field is reconstructed from points, in the lower dimension (the average plane of the leaf node). Figure 6.8 shows our normal mapping reconstructed directly from original samples: the same order of visual quality is reached when comparing to prior art methods where a full resolution surface reconstruction and parameterization were necessary before performing the appearance preserving simplification.
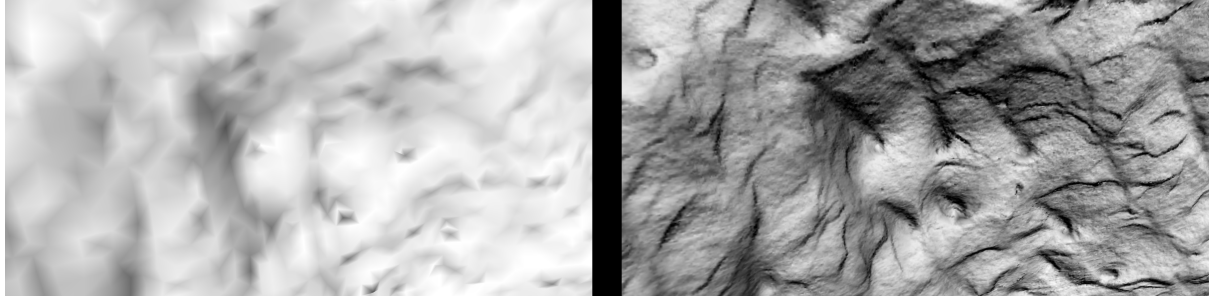
Figure 6.8: Left: *Surfel Strips rendering.* **Right:** Normal *Surfel Strips rendering. High frequency details of large scans models are preserved, using detailed normal textures instead of huge polygons sets. Globally, our approach provides similar results to usual appearance preserving methods that require full resolution tessellation, parameterization and simplification, while we deal only with the point samples.*

Limitations We have made the choice to use a very simple a grid simplification scheme in most cases at the beginning of our algorithm. This choice has been made after various experiments with real data sets, which show essentially that most of the time, large scans are dense enough to support, in the particular case of appearance-preserving conversion, this *regular* decimation, allowing a faster pre-process. Nevertheless, complex topologies and highly varying density distributions require to switch to an adaptive simplification, such as the one we proposed in Section 4.3, which usually means a much longer pre-process (upon twice longer in our experiments).

The reader must also note that our approach is still a "simplification" one, which exhibits drawbacks and advantages. On one hand, even if most of the fine visual details are kept thanks to the high resolution normal maps (see Figure 6.9), a slight shrink effect can appear in silhouettes because of the coarse mesh definition. This is the price for reducing the time preprocessing and improving the rendering framerate compared to "multiresolution" approaches such as QSplat or Sequential Point Trees. On the other hand, this low-pass filtering has frequently removed the *registration noise* present in our examples. Figure 6.9 shows the rendering of the St Matthew model with the publicly available QSplat software. We can observe that our method provides a globally equivalent appearance, with a much higher framerate, even under a strong close-up. Note that we have compared with QSplat because it is the only publicly available software for large dataset.

Note also that QSplat is not tuned for recent graphics hardware, which explains the poor framerate obtained. One of our future experiments will be to compare our results with a combination of the DuoDecim compression scheme of Krueger et al. [KSW05] and the GPU splatting of Botsch et al. [BSK05], which should be the state-of-the-art large point-based surface rendering system.

Finally, our algorithm works for objects which exhibit an important surface coherency, for which a single normal map can be shared by numerous neighboring sample for capturing the normal field they defined. For instance, our approach is not adapted for complex objects poorly sampled, like trees, for which our method would require several hundreds of samples by leaf for running correctly.
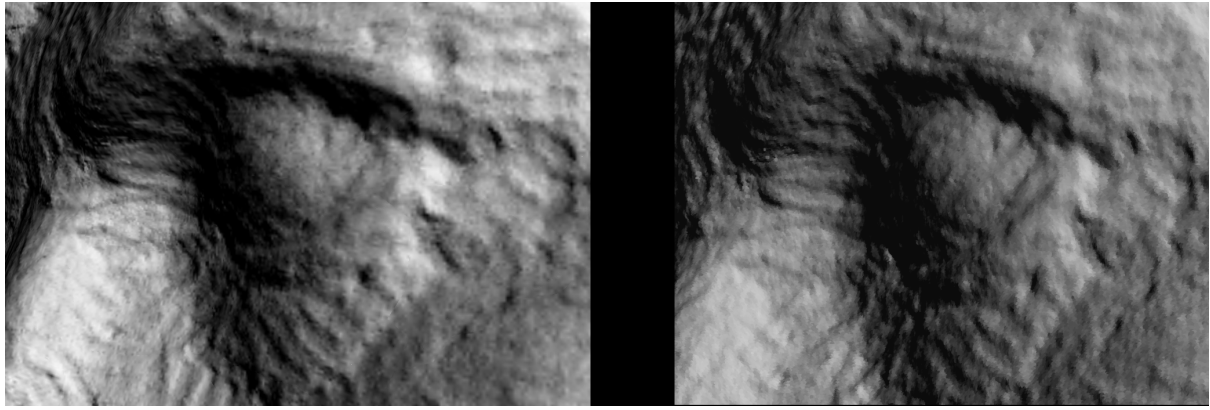
Figure 6.9: *Eye of the St Matthew. Visual quality comparison between our approach at 165 FPS (left) and the QSplat rendering (right), obtained at 0.3 FPS. Even under a strong close-up, our method keeps the fine visual details as well as the QSplat system, but with a much higher framerate.*
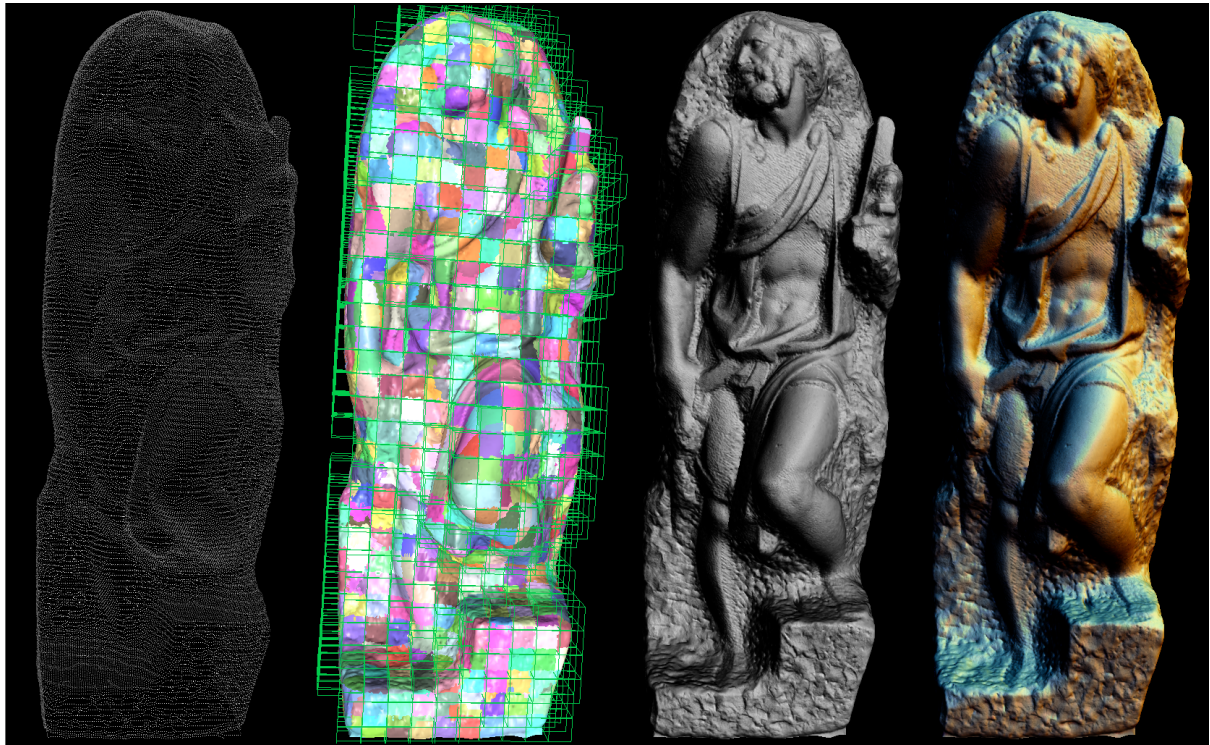
Figure 6.10: *Real-time visualization of the St Matthew model (186 810 938 points).* **Left:** *the sub-sampled model.* **Middle left:** *the coarse polygonal representation generated in the leaves of the Stripping Tree (in green).* **Middle right:** *final Normal Surfel Stripping under one white light source.* **Right:** *under 3 colored light sources.*

Applications   Our method has been intensively used on scanned archaeological artifacts, as shown all along this chapter. Their high resolution and need for quick interactive visualization represent a typical application of our system (see Figure 6.11). However, although originally designed for appearance preserving visualization of large point-based surfaces, we have often experiments alternative uses:

- **Appearance preserving visualization of large meshes:** our algorithm performs very well by considering the set of vertices of a large mesh as the input point cloud. In the case of a polygon soup, triangles can be reindexed over the simplified point cloud for generating the Surfel Strips connectivity in place of the Delaunay triangulation. Consequently, we obtain a fast way to convert large meshes to coarse ones with normal maps.

- **Color preserving visualization:** when a per-sample color is provided, we reconstruct color maps on top of normal maps, and this represent a convenient way to obtain low-resolution textured models from large scans.

- **Conversion for interactive applications:** obviously, Normal Surfel Strips succeed as well as original Surfel Strips in merging acquired objects in interactive applications and polygonal rendering engines. In particular, by specifying a fixed amount of authorized memory, our conversion process is output sensitive, and can fulfill the constrains of a particular application, ranging from preview on mobile devices toward high-quality offline rendering on PC clusters. Our system can also offer a fast transition to common mesh-based software, by keeping just the connectivity of Surfel Strips as a mesh and exporting normal textures. Finally, our method can fully run in streaming, including the progressive output of normal maps, using final spatialization for establishing as soon as possible when a given normal map can be reconstructed and streamed on the output (i.e., no more sample coming). This solution entitles a low memory footprint, is highly parallelizable and can be seen as tweak of the *sampling-reconstruction* principle presented in Chapter 4.
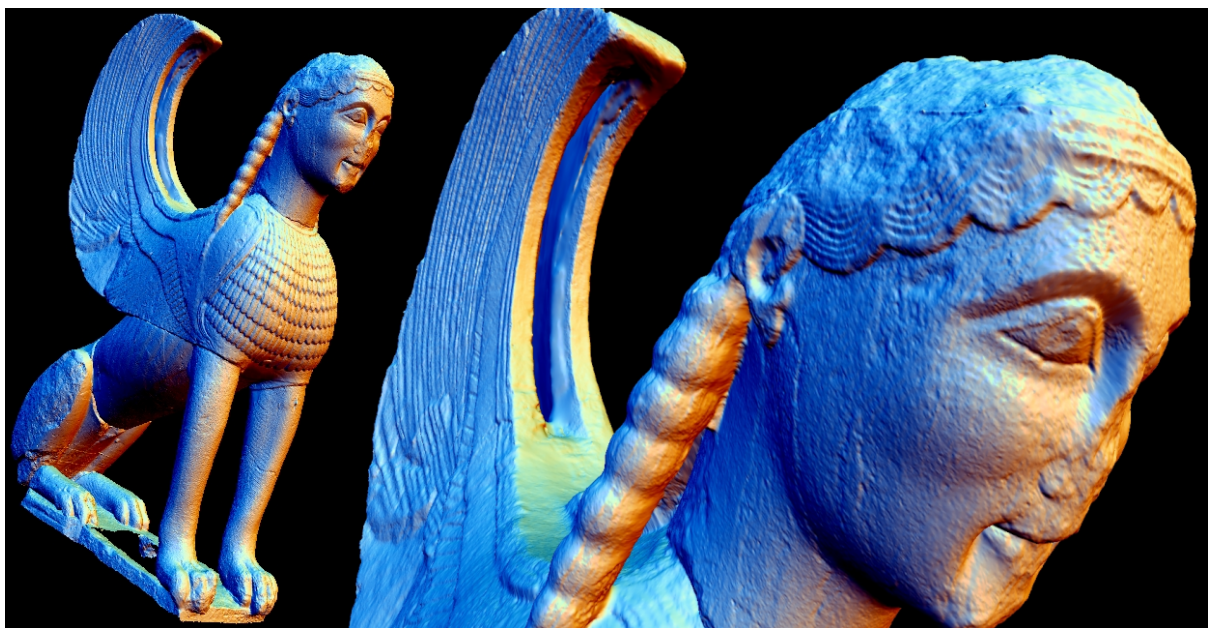

Figure 6.11: *Application to archaeological visualization: interactive inspection of the Naxian Sphinx, originally captured with 15M point samples (model courtesy Ausonius).*

Summary We have proposed an efficient conversion process to obtain an interactive appearance-preserving visualization of large 3D objects represented by point clouds which exhibit a surface coherency. The main advantages of our technique are:

- Direct processing of unorganized point clouds, avoiding any kind of surface reconstruction of the large model.

- No complex data structure or complex processing is needed on the large model.

- The idea of inflated support for normal map reconstruction allows consistent overlapping and avoids global parameterization of the model, enabling appearance preserving normal map conversion of large point-based surfaces and (non-manifold) meshes.

- The pre-process is very fast as it basically only requires two out-of-core passes, which makes it usable in various applications where quick preview is mandatory

- The final in-core model is entirely stored on the GPU memory, large enough on today graphics devices to handle efficiently appearance attributes of hundred millions of samples, through, for instance, normal textures.

- Since all details are stored as normal maps, the rendering takes automatically benefit of the hardware mip-mapping for filtering details at a given screen resolution.

- The output sensitive nature of our approach, as well as its polygonal approach, allows to tailor precisely the size of the final appearance-preserving representation, which allows to rule correctly the inclusion of scanned objects in interactive applications.

The whole pipeline is easy to implement, and has provided very convincing results when applied on a large variety of acquired point-based surfaces. We hope that it can become a good complement to existing high quality but slower visualization methods of large models.
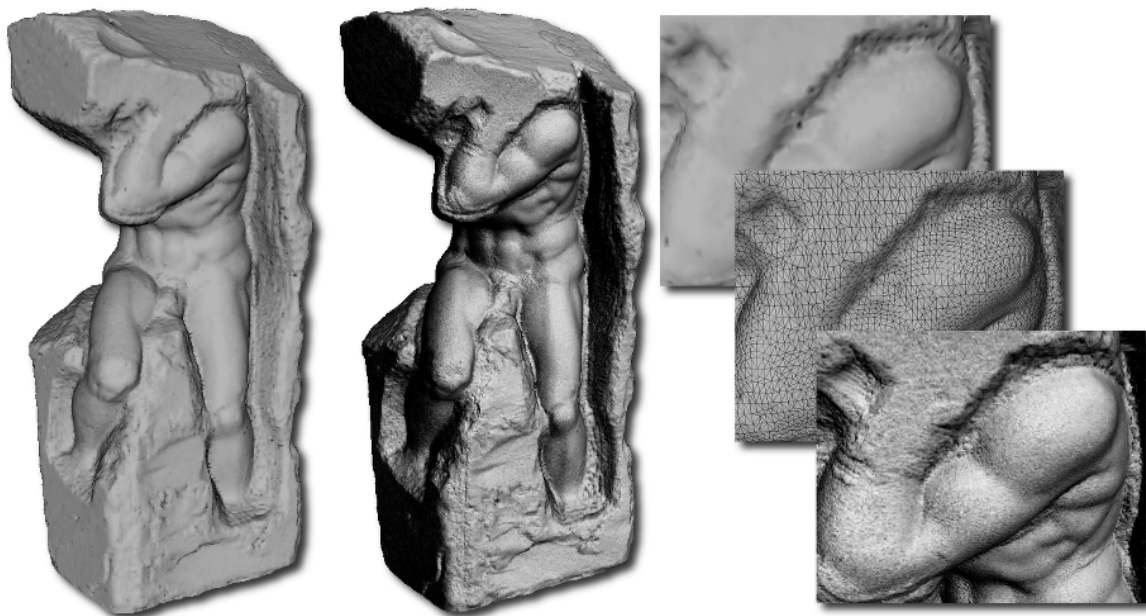
Figure 6.12: *Our largest data set, the Atlas, featuring half a billion samples, demonstrates the scalability of our appearance preserving method for large objects.*

Perspectives   As mentioned in the limitations, our approach still performs a simplification on the original data. In fact, normal maps offer convincing shading and are adapted to modern GPUs, but the ultimate rendering solution for large objects would be to perform an adaptive *geometry synthesis* prior to rendering, performed on the fly according to the various rendering parameters (geometry, point-fo-view, hardware capabilities, semantic, etc...). This would be possible with our system by replacing normal maps by displacement maps, for truly recovering the geometry, and obtaining precise silhouettes and shadows. Our reconstruction by diffusion supports such attributes, by simply storing the vectors from the projected to the original points in the textures. Unfortunately, one fundamental problem remains: if we want to keep a coarse polygonal representation, we have to perform a **real-time mesh refinement**. We address this problem and its applications from a more general point of view in the last part of the this thesis.

Part III

Toward Real-time Geometry Synthesis

After having developed new efficient solutions for processing, editing and rendering large sampled models, it appeared that the next step in the understanding and use of automatically acquired 3D shapes was **Geometry Synthesis**. *Looking back to our appearance preserving conversion, the extraction of a displacement map is straightforward. But its use is not. In fact, the main challenge is related to* **real-time mesh refinement** *and is more general than the precise topic of this thesis. Thus, we have decided to address directly this general problem, and our results are stated in this third part.*

Chapter 7

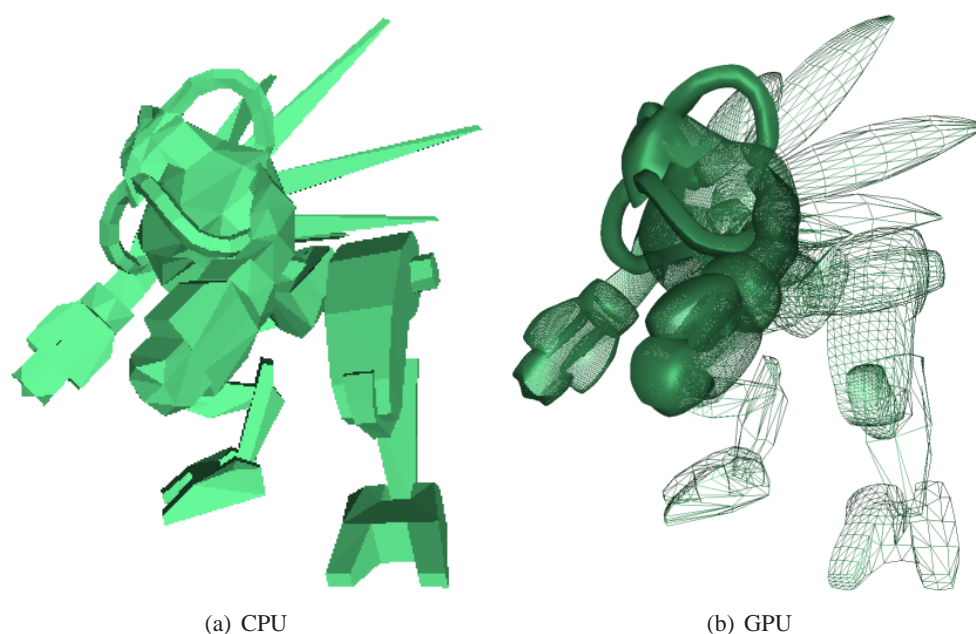# Generic Mesh Refinement

(a) CPU   (b) GPU

Figure 7.1: *By using only a dynamic coarse mesh (1246 triangles) animated on the CPU (left), our GPU kernel generates an adaptive frame-by-frame tessellation and displacement (right), and provides an extremely detailed rendering (1.1M triangles at 263 FPS).*

For the wide range of applications discussed in this thesis and beyond, image synthesis techniques leverage the amount of information required for creating realistic animated pictures. In particular, for real-time rendering, the application has just to provide a set of polygons describing the geometry of a scene, and the graphics hardware will automatically produce a coherent grid of pixels through the usual rasterization pipeline. However, *flat* descriptions of surface, such as surfel lists or polygons meshes, quickly exceed the capabilities of the rendering hardware, since each single frame requires to browse and display the whole list. In the two previous chapters, we have shown how a multi-resolution structure can be generated and enriched with high-resolution textures, to display efficiently large sampled surfaces. Now, going further in the direction of high-quality interactive shape reproduction, we believe that recovering not only the appearance, but also the geometry of large models at rendering time is a key problem. The is even a more general problem than the precise context of this thesis: the geometry

of a model may not be static, and the bandwidth bottleneck between the application and the graphics hardware limits the size of geometric description that can be transmitted for real-time rendering, and thus also limits the realism of the rendered pictures.

On-the-fly geometry synthesis addresses this issue by allowing an additional level of abstraction in the graphics pipeline. For interactive applications, geometry synthesis is usually cast as a *mesh refinement process*. Rather than enumerating the huge number of polygons that would be required to get an accurate discrete approximation of a complex shape, mesh refinement techniques split the surface representation into a *coarse polygonal mesh combined with a continuous displacement function*. Then, at rendering time, mesh refinement basically performs two successive operations on the coarse mesh: a *tessellation* step followed by a *displacement* one.

During the first step, a refined mesh topology is generated at a target level-of-detail, simply by splitting each coarse polygon into a set of finer ones, without any actual geometric modification. Then, during the second step, each newly inserted vertex is translated to its final position, obtained by sampling the continuous displacement function. Many existing computer graphics techniques can be expressed under this paradigm, such as spline-based or wavelet-based surface representation, subdivision surfaces, hierarchical height fields, etc. The key feature that makes this process work well, is that the continuous displacement function can usually be defined by providing a smaller amount of data compared to the size of the huge refined mesh. Examples of such additional data include subdivision masks for smooth surface generation, bitmap textures for displaced meshes, or a bunch of numerical data for procedural geometry synthesis.

However, performing a full GPU implementation of this two-step process remains a problem with current devices. While graphics hardware offers a flexible *vertex shader* stage that allows an efficient implementation of the displacement step, the lack of geometry creation on GPU makes the implementation of the tessellation step really tricky. Last generation devices, launched at the end of 2006, embed a *geometry shader* stage [Bly06] which has been specifically designed for geometry upscale. Unfortunately, even if the geometry shader clearly represents a step in the right direction, it does not provide the ultimate high-level flexible solution demanded by many applications. One of its main limitation, is that the geometry shader cannot output (i.e. generate) more than a fixed amount of floating point numbers (1024 in the original specification), which means that only about 2 or 3 levels of refinement can be applied on each coarse triangle. If deeper refinement is required, multi-pass geometry shading has to be employed, which obviously reduces overall performances.

The lack of flexible geometry synthesis on GPU, has led some researchers to cast the mesh refinement problem as a general purpose computation problem, using a GP-GPU approach [GPG06]: by converting the coarse mesh as a standard rectangular image, the tessellation step becomes a simple image upscaling operator, and the displacement step can be implemented in the *fragment shader* stage. However, such an approach induces several strong restrictions. First, it requires an additional preprocessing step to convert the mesh into an adapted image format. Second, it involves intensive use of multi-pass rendering and fragment shading, while the vertex shading stage is greatly under-exploited, as it only has to process a few full-screen quads. Third, the whole process has potentially to be restarted for each frame in the case of dynamic meshes. Fourth, additional hardware pipelines (e.g. physics simulation hardware) are not directly compatible with such an approach, since no object space geometry is really produced. And last, multiresolution and adaptivity cannot be easily handled by such a process.

In this chapter, we propose an alternative approach called *adaptive refinement kernel* (**ARK**), based on three key features. First, a flexible control of the adaptive level-of-detail is obtained by a simple and generic *depth-tagging process*. Second, a set of *adaptive refinement patterns* (**ARP**) is employed to allow crack-free adaptive multiresolution refinement. And third, a specific single-pass vertex program,

called *adaptive refinement shader* (**ARS**), performs both tessellation and displacement steps involved in mesh refinement. By combining all three ingredients, we obtain a flexible kernel for adaptive on-the-fly mesh refinement on GPU.

This kernel does not involve any preprocessing of input coarse meshes, as it directly processes the basic mesh representation used in low-level APIs, such as polygon soups or indexed triangle sets, without requiring additional high-level data structures (e.g. half-edge representation). With our kernel, the final mesh is never generated on the CPU, never transmitted on the graphics bus, and even never explicitly stored on the GPU. All the refinement is performed by our single-pass generic vertex program, which totally frees the fragment shaders for including additional visual enrichments. This kernel offers a flexible way to perform geometry synthesis based on displacement maps extracted from acquired geometry. It is also particularly well-suited for dynamic meshes which are deformed on a frame-to-frame basis (animation of characters, physics simulation, etc.) and for procedural shapes that usually include high frequency features and require fine tessellation at rendering time.

7.1   Context: Real-time Mesh Refinement

Existing mesh refinement methods can basically be divided in two main categories: either *direct* or *indirect* refinement.

Direct Refinement   This first category includes pure geometry synthesis approaches, where the input coarse mesh is directly refined in object-space, without additional conversion steps. Multi-scale rendering of numerical models of terrains are maybe the most classical examples of on-the-fly direct refinement [AH05], but the involved algorithms are usually limited to height-field configurations. Another well-studied topic includes all the techniques that target an efficient GPU implementation of subdivision surfaces [ZS00], as pioneered by Pulli and Segal [PS96]. They introduced a memory-efficient depth-first algorithm for refining an arbitrary triangle mesh toward the subdivision surface it defines. They use *precomputed tables of basis functions* for a prefixed refinement depth, one for each possible configuration of the one-ring neighborhood. At rendering time, these tables are used for each coarse triangle according to its one-ring neighborhood. A uniform triangle refinement at a prefixed depth is then performed, and the generated vertices are projected on the limit surface. Such a refinement is specific to each subdivision scheme, and can benefit from low level implementations, using either SIMD instructions of modern CPUs[BS02] or programmable GPUs [BS03]. Unfortunately, these *direct* approaches are limited by the the set of precomputed tables, restricted in term of topology and does not address the problem of tessellation (pre-tessellated coarse meshes are usually stored on graphics memory). Specific hardware has also been proposed in order to reduce the bandwidth between CPU and GPU [BKS00, dRBAB02].

Alternatively to true subdivision surfaces, Vlachos et al. [VPBM01] have proposed *Curved PN Triangles*, a fast spline-based mesh smoothing based on the 3 positions and normals of a triangle. We will show in the next chapter how Curved PN Triangles can be implemented with our kernel, and how they can be controlled by scalar tags.

Indirect Refinement   This second approach casts mesh refinement as a kind of image processing algorithm [BW06]. Before the introduction of recent unified architectures, fragment processing was much more flexible than vertex processing. Thus, several algorithms have been proposed, again mostly focused on subdivision surfaces, which consider meshes as textures rather than geometry. Basically, these methods work in three steps: first the input mesh is converted on CPU to an image-based representation.

For instance, Shiue et al.[SJP05] start with a two-step subdivision of the initial mesh on the CPU (basically to sufficiently separate vertices with extraordinary valence), and then, unfold each original vertex with its two-ring neighborhood in a 1D texture. Similarly, Bunnell [Bun05] breaks the original surface into small pieces, projecting them on 2D textures which provides a limited GPU support for displaced subdivision surface [LMH00]. With such an approach, the "geometric" texture can then be upscaled, by applying scaling and filtering operations (multi-pass rendering) corresponding to the usual tessellation and averaging steps in subdivision [Kob00, WS04], which is implemented using a render-to-texture function and replacing the usual image filtering kernel by the mesh subdivision one. This is done recursively until reaching a given depth or an error bound. Finally, upscaled images are converted back to geometry, rasterized and rendered on screen. These algorithms works well for small refinement depths, but inherit the intrinsic limitations of GP-GPU approaches: they require a conversion of the input model to a specific format and employ intensive multi-pass rendering. When the input is not a mesh but an object with a global parameterization, such as NURBS or T-Spline surfaces, the indirect method proposed by Guthe et al. [GBK05, GBK06] is more efficient, as the parameterization already acts as image coordinates. Note that the *unified shader architecture* of recent GPUs, that offers efficient vertex texture fetch, coupled with the additional topology information provided by recent APIs [Bly06] now permits to process geometry directly at vertex/geometry shader level, without any mesh-to-image conversion.

Adaptivity and Local Control   Including adaptivity within mesh refinement can strongly improve the overall performance, by reducing the number of polygons in areas classified as less important (e.g. flat areas, far areas, partially hidden areas). Multiresolution mesh representation [Hop96] is based on this notion. Kähler et al. [KHS03] have proposed an interesting curvature-based approach for CPU adaptive mesh tessellation. Nevertheless, adaptive refinement methods are not easily amenable to GPU implementation, due to their highly dynamic adjacency information.

Local control of a given mesh refinement process has been frequently solved by including additional per-[vertex/edge/face] boolean or scalar tags, which can be used to edit the shape (e.g. crease, tension, bias, etc) of the refined surface around the tagged simplex [BS95, BMZB01]. Here, we introduce a similar tagging scheme, but this one is not intended to control the geometry but rather the topology of the refined mesh. This per-vertex tagging scheme is then used to select adaptive tessellation in the parametric domain (barycentric coordinates of the triangles), "mapping" it implicitly onto each original polygon.
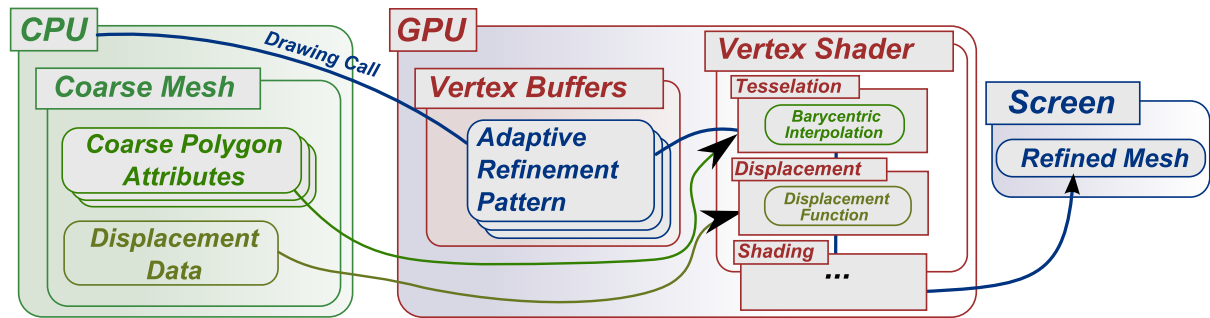
7.2 Adaptive Refinement Kernel

Figure 7.2: *Architecture of our adaptive refinement kernel (ARK). For each coarse polygon to refine, we first transmit its geometric attributes as well as the displacement function attributes to the adaptive refinement shader (ARS). Second, a drawing call is performed, that selects the correct adaptive refinement pattern (ARP) according to the desired level-of-detail. All the triangles included in the selected ARP (implemented as a vertex buffer object) are then translated by barycentric interpolation from the polygon attributes, and warped according the displacement attributes. Finally, the set of so-mapped refined triangles are rasterized and passed to the fragment shaders for rendering on screen.*

7.2.1 Overview

The *Adaptive Refinement Kernel* (ARK) presented in this chapter offers the following properties:

- Standard geometry structures used by rendering APIs (polygon soups or indexed triangle sets) can be employed as-is, without any preprocessing (e.g. global or local parameterization) nor any additional data structures often required by refinement techniques (e.g. half-edge structure).

- Only the coarse mesh is transmitted from the CPU to the GPU. The only required additional data is a simple per-vertex scalar attribute, called *depth-tag*, that indicates the level-of-detail desired in the vicinity of each vertex. Note that this depth-tagging may be generated either automatically or under user supervision.

- As mesh refinement is performed on-the-fly, on a frame-by-frame and triangle-by-triangle basis, arbitrary level-of-detail can be obtained, even for animated meshes.

- The whole two-stage adaptive mesh refinement (tessellation and displacement) is performed on the GPU, by a single-pass vertex program, which totally frees the fragment shaders for additional visual enrichments.

The workflow architecture used by our ARK is described in Figure 7.2. The key idea is to precompute all the possible refinement configurations of one single triangle, for various per-vertex depth-tags, and encode them using barycentric coordinates. Each possible configuration is called an *adaptive refinement pattern* (**ARP**) and is stored, once for all on the GPU, as a vertex buffer object. Then, at rendering time, the attributes of each polygon of the coarse mesh, as well as the attributes of the displacement function are uploaded to the GPU and the adequate ARP is chosen according to the depth-tags. Finally, the vertex program simultaneously interpolates the vertices of the current coarse polygon, and the displacement function, by using the barycentric coordinates stored at each node of the ARP to "map" the refined connectivity on the coarse one. The first interpolation generates the position of the node on the polygon (i.e. tessellation step) and the second one translates it to its final position (i.e. displacement step).
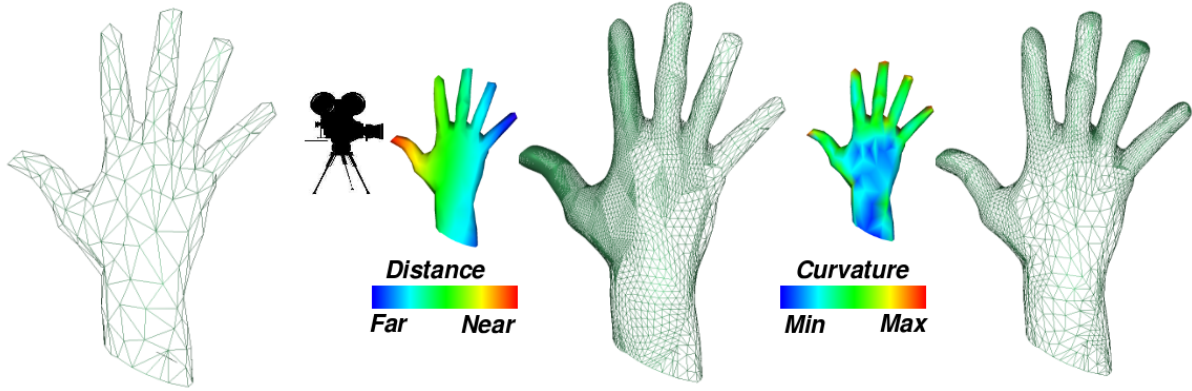
Figure 7.3: *Examples of depth-tag configurations (color code) and adaptive refined topology generated on the GPU.* **Left:** *Initial coarse mesh transmitted from CPU to GPU.* **Middle:** *Adaptive refinement using distance-based depth-tagging.* **Right:** *Adaptive refinement using curvature-based depth-tagging.*

7.2.2 Topology Control with Depth-tagging

On the CPU-side, the application specifies the usual per-vertex attributes of the mesh (position, normal, color, etc) as well as a specific one: the *vertex depth-tag* that indicates the level-of-detail desired in the vicinity of each vertex. The depth-tagging process can either be performed once for all for static meshes, or dynamically recomputed at each frame for animated meshes.

Once this vertex depth-tagging has been set, it is employed at rendering time to adaptively refined each coarse polygon, according to a set of precomputed configurations. More precisely, the depth-tags will be used for selecting a per-edge tessellation rate. To ensure crack-free refinement, the tessellation must be consistent on the two sides of a given edge. Thus a consistent *edge depth-tag* is computed by simply taking the arithmetic mean of the two adjacent vertex depth-tags. Moreover, to easily manage general non-triangulated meshes, a centroid split is performed for each coarse polygon with n vertices to get a set of n triangles. The depth-tag of the centroid, called *face depth-tag* is computed as the mean of the n surrounding edge depth-tags.

Such a tagging approach is very generic, as the tag values can be set according to any metric. In this article, we do not propose new metrics, but rather show how to set the depth-tag according to any existing one. For instance, Figure 7.3 shows a static tagging generated by using a modified version of the curvature estimator proposed by Rusinkiewicz [Rus04], as well as a dynamic tagging generated by using a simple camera-to-vertex distance metric.
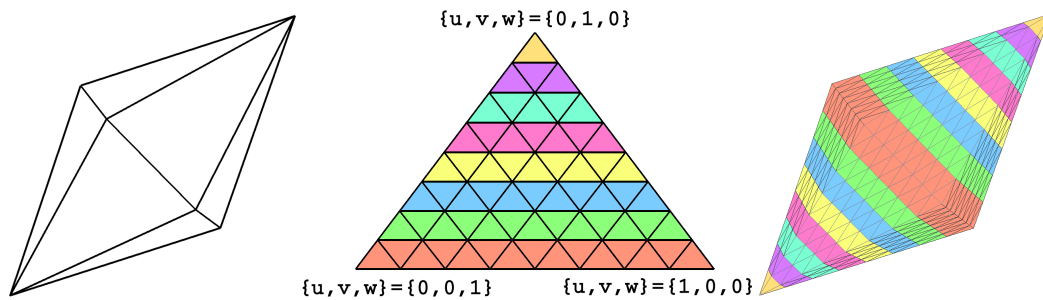
7.2.3 Refinement Patterns

Figure 7.4: *Principle of Uniform Refinement Patterns. (a) Coarse mesh stored on CPU. (b) Uniform Refinement Pattern (URP) stored as a vertex buffer object on GPU, where each node is stored as barycentric coordinates. (c) Final refined mesh rendered on screen. The URP is used to tessellate all triangles at a uniform level-of-detail. In this example, the URP is a tessellated triangle encoded as a single degenerated strip, composed of 8 different regular parts (each part has a different color).*

According to the classification of Shiue et al. [SJP05], our technique can be considered as a patch-based refinement. We start by explaining the principle in the uniform case and then give a generalization to adaptive refinement.

Uniform Refinement Patterns (URP)   In the case of uniform tessellation rate, our approach use *one single* Uniform Refinement Pattern for the whole mesh. This refinement pattern is transmitted once for all, from the CPU to the GPU, as a vertex buffer containing a few strips (see Figure 7.4).

Let A_T be the set of attributes of a coarse triangle T. Typically A_T contains the 3 vertex positions, 3 vertex normals, 3 vertex colors, and 3 texture coordinates. We propose to render the refined mesh with the following algorithm:

```
GLuint URP;
void precomputeURP () {
  generateAndStripURP (URP);
  sendURPVertexBufferToGPU ();
  bindVertexBuffer ();
  sendURPIndexBuffersToGPU (URP);
  bindIndex (URP);
}
void render (Mesh M) {
  for each CoarseTriangle T of M do {
    sendToGPU (A(T));
    drawElement ();
  }
}
```

Basically, at rendering time, the attributes of each coarse triangle are uploaded to the GPU and the URP is drawn *instead of* the coarse polygon. The barycentric coordinates stored at each node of the URP are used to interpolate the per-vertex attributes (e.g. positions, normals) of the coarse triangle, and to output each refined vertex in the *graphic context* of the currently processed coarse triangle. This virtually generates vertices on GPU and can be seen as a procedural *instanciation* method for refinement purpose.

Another way to see this method is to consider that the URP is "mapped" onto each coarse polygons, enriching their connectivity.

More formally, let suppose that a functional $f_{A_T} : [0,1]^2 \rightarrow R^3$ can be constructed over A_T, the simplest case being the identity function, i.e. linear refinement, without displacement and corresponding to the tessellation. To evaluate f_{A_T} at each vertex V of URP, we have to recover its parameterization $\{u,v\}$ onto T. Actually, we need to know the position of a refined vertex "relatively" to the original triangle. Since the URP is only used for the topological storage of the tessellation, we propose to use to encode the use *barycentric coordinates* of V as its position vector: $V_{xyz} := \{w,u,v\}$ where $w = 1 - u - v$.

Now, during the vertex shading pass, the GPU can clearly identify the parameterization $\{u,v\}$ for each vertex V of RP, and thus evaluate its functional value $f_{A_T}(u,v)$. Of course, each attribute in the set A_T may eventually be interpolated by a different functional.

Let us consider the position attributes $\{P_0, P_1, P_2\}$ of the current coarse triangle drawn and the parameterization $\{w,u,v\}$ (encoded as the position of inner vertices) of each vertex V of RP. In order to perform the tessellation, we just have to interpolate between $\{P_0, P_1, P_2\}$ to obtain the output position V_{xyz} of V:

$$V_{xyz} := wP_0 + uP_1 + vP_2 = V_x P_0 + V_y P_1 + V_z P_2$$

The URP technique strongly reduces the CPU-GPU bottleneck, as only the coarse mesh is transmitted, while the GPU synthesizes the high-resolution mesh on a per-triangle basis. This approach is particularly well-suited for dynamic objects that cannot be refined and stored on the GPU once for all, as well as for procedural displacement textures, that usually require highly tessellated meshes. In this case, the URP technique enables to stream more geometry toward the screen than could even be stored on the CPU or the GPU.

Unfortunately, providing only uniform refinement is a major drawback for most applications, as it is almost impossible to avoid either over-tessellated or under-tessellated meshes, even in the easy case of a moving camera in a static scene. Therefore, we propose to extend this approach by generating a set of *Adaptive Refinement Patterns* (ARP).

Adaptive Refinement Patterns (ARP)   Basically, the idea of ARP is to precompute all the different topological configurations of a refined polygon both for regular and irregular situations, still encoding the nodes in the plane parametric space with barycentric coordinates. Then, at rendering time, the low-level API can select the correct ARP, according to the depth-tag configuration of the coarse polygon.
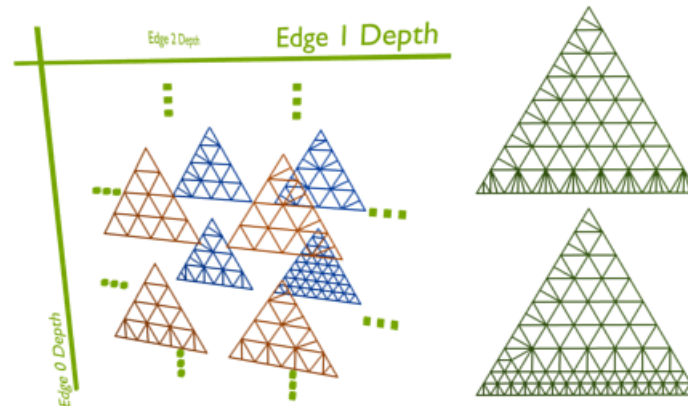

Figure 7.5: **Left:** *The matrix (or pool) of adaptive refinement patterns, in the barycentric coordinates system, stored as vertex buffer objects on GPU, with its 3 dimensions corresponding to the 3 depths tags of a coarse triangle.* **Right:** *Two different ARPs with different support sizes for the adaptive topology of a triangle. The largest support offers better transitions between the different edge resolutions, but requires more vertices.*

ARP for Triangular Meshes   For triangular meshes, it is possible to encode all configurations up to an upper bound of the refinement depth. Since different tessellation rates may appear for different edges of a triangle, the set of ARPs is implemented as a matrix of l^3 patterns, with l being the deepest refinement level allowed (left part of Figure 7.5). This matrix is precomputed and uploaded to the GPU once for all. The quality of *adaptivity* for a given refinement scheme is usually rated with its support size [Kob00]. The larger is the support, the "smoother" will be the transition between two different tessellation rates, but additional vertices are required (see right part Figure 7.5). The pseudo-code of the algorithm used on the CPU-side for triangular meshes is presented below:

```
GLuint ARPPool[MaxDepth][MaxDepth][MaxDepth];
void precomputeARPs () {
  generateAndStripARPs (ARPPool);
  sendARPVertexBufferToGPU ();
  bindVertexBuffer ();
  sendARPIndexBuffersToGPU (ARPPool);
}
void render (Mesh M) {
  if (dynamic)
    for each Vertex V of M do
       V.tag = computeRefinementDepth (V);
  for each CoarseTriangle T of M do {
    sendToGPU (A(T));
    bindIndex(ARPPool[T.v0.tag][T.v1.tag][T.v2.tag]);
    drawElement ();
  }
}
```

Our system thus allows any kind of adaptive transition, as soon as its support fits in the area of the coarse triangle. Possible adaptive refined topologies range from *border-split* patterns to *variational angle-maximizing* one. In most cases, simple *border-split* topologies as the one presented at the upper-right corner of Figure 7.5 offer good results. Note that the APRs might be harder to convert into triangles strips (lossless topology compression) than regular ones. Thus, an automatic *stripping* is performed using the STRIPE algorithm [ESV96, RBA05]. Algorithm 5 gives a border-spliter adaptive tessellation.

Algorithm 5 Border-splitter Adaptive Refinement Generation

Require: input triangle $T = \{[0,0],[1,0],[0,1]\}$ // barycentric coordinates
Require: mandatory refinement edge depth $\{d_0^e, d_1^e, d_2^e\}$
  $d_{min} = min(d_0^e, d_1^e, d_2^e)$
  $T^r \leftarrow$ uniform refinement of T at depth d_{min}
  for each edge i **do**
    $r_i \leftarrow d_i^e - d_{min}$
    for each refined triangle t on edge i **do**
      $T^r \leftarrow T^r - t$
      $T^r \leftarrow$ split t r_i times along edge i
    end for
  end for
  return T^r

ARP for General Polygonal Meshes   While the memory footprint remains low when storing refinement patterns for triangles, it becomes a problem for more general polygons. If no care is taken, the number of different configurations to store may quickly become impractical when the tessellation level ℓ increases. Indeed, as each edge of the polygon includes its own tessellation rate defined by its depth-tag, the number of different tessellation patterns is ℓ^3 for a triangle, ℓ^4 for a quadrangle and more generally ℓ^n for a polygon with n vertices or edges. One possibility which strongly reduces the total number of configurations is to use *constrained depth-tagging*, for which the variation among the depth-tags for each polygon is clamped to one level up or one level down. Unfortunately, constrained depth-tagging requires additional non-trivial work on the CPU-side, which may have to be repeated for each frame, in the case of dynamic tagging.

We propose a alternative solution for efficient encoding and processing of the set of ARP without requiring any limitation on the vertex depth-tag configurations, and only involving very limited CPU overhead. This solution is illustrated on Figure 7.6. Let us take the general case where the CPU has to manage a polygon with n vertices. First, each couple of adjacent vertex depth-tag (δ_k, δ_{k+1}) is converted into an average edge depth-tag $\bar{\delta}_k$ by computing the arithmetic mean. An average face depth-tag $\bar{\delta}$ is also computed from the set of $\bar{\delta}_k$. This double averaging acts as smoothing process of the initial vertex depth-tags, which will naturally soften abrupt variations of the tessellation rate. Second, the polygon is split into a set of n triangles by linking each pair of adjacent vertices to the centroid of the polygon. The depth-tag of each inner edge of these triangles is set to $\bar{\delta}$. This guarantees that each triangle only contains two similar depth-tags $\bar{\delta}$ and $\bar{\delta}_k$ because they have been smoothed by double averaging. The inner part of the triangle (green area on Figure 7.6) will be uniformly tessellated at the rate provided by the face depth-tag $\bar{\delta}$, while the outer part strip will generate a crack-free junction between level $\bar{\delta}$ and level $\bar{\delta}_k$. All the (very reduced) number of possible configurations for this adaptive triangle strip are concatenated at the end of the uniform tessellation of the inner part of the triangle, and the whole data is stored on the GPU as a single index buffer. Each specific configuration can thus be simply retrieved by providing an offset in that buffer.

Basically, with our solution, one single strip of tessellated triangles at the outline of the initial coarse polygon is used to manage the crack-free junction between different adaptive levels, while most area of the polygon is tessellated according to the face depth-tag. In other words, we solve the adaptivity problem on a per-polygon basis, which can thus be done without complex high-level data structures to encode the neighboring topology for each polygon. For pathological cases where $\bar{\bar{\delta}}$ and $\bar{\bar{\delta}}_k$ differ too much, two border strips instead of one may be employed to create smoother transition between coarse and fine tessellation, and thus better avoid elongated triangles. Finally, note that since all ARPs are precomputed and uploaded once for all on the GPU, rendering one polygon with uniform tessellation, and one with adaptive tessellation, takes exactly the same time, for a equivalent tessellation rate. This is far from being true with existing adaptive mesh refinement techniques.
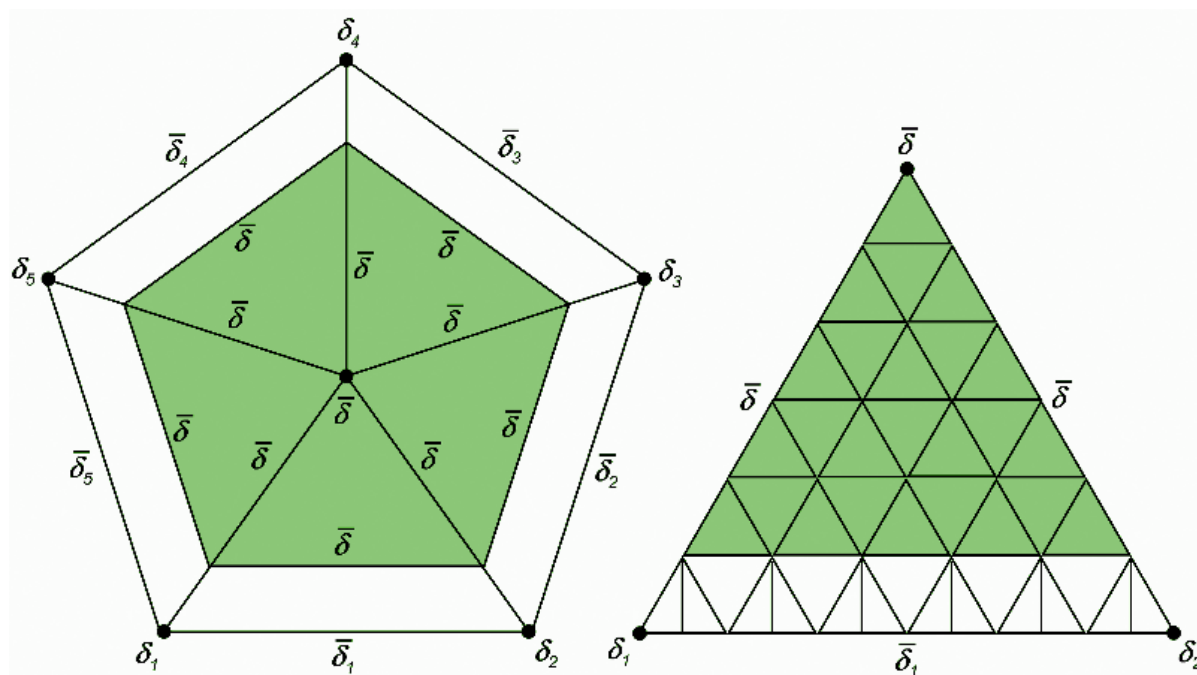

Figure 7.6: *ARP factorization for non-triangular patterns.*

7.2.4 Adaptive Refinement Shaders

Our kernel uses a specific single-pass vertex program called *Adaptive Refinement Shaders* (ARS), that successively performs the tessellation and the displacement steps. During the tessellation step, the coordinates of the current *ARP* are used to generate a barycentric interpolation of the standard per-vertex attributes (position, normal, etc). Then, during the displacement step, the resulting vertices are displaced using additional attributes (e.g. textures for displacement mapping).

Note that since all ARPs are encoded in the barycentric space, refinement shaders are totally independent of the topology of the patterns. So, the same shader is used, whatever the given ARP. Here is an example in GLSL [KBR04] of a refinement (vertex) shader which performs a simple procedural refinement with linear tessellation:

```
const uniform vec3 p0, p1, p2, n0, n1, n2;

float displace (vec3 p) {...}

void main (void) {

  // Tessellation by barycentric interpolation
  float u = gl_Vertex.y;
  float v = gl_Vertex.z;
  float w = gl_Vertex.x; // 1-u-v
  gl_Vertex = vec4 (p0*w + p1*u + p2*v, gl_Vertex.w);
  gl_Normal = n0*w + n1*u + n2*v;

  // User Defined Displacement
  float d = displace (gl_Vertex.xyz);
  gl_Vertex += d * gl_Normal;

  // Shading and Output
  ...
}
```

Note that the barycentric coordinates may be used for non-linear interpolation (e.g. quadratic interpolation for normals [VPBM01]). Moreover, in addition to vertex displacement, the same process can further be used to interpolate any other per-vertex attribute during the refinement process. Finally, as the refinement is totally performed on a per-polygon basis, meshes with arbitrary genius and even non-manifold can be directly processed (see Figure 7.7).
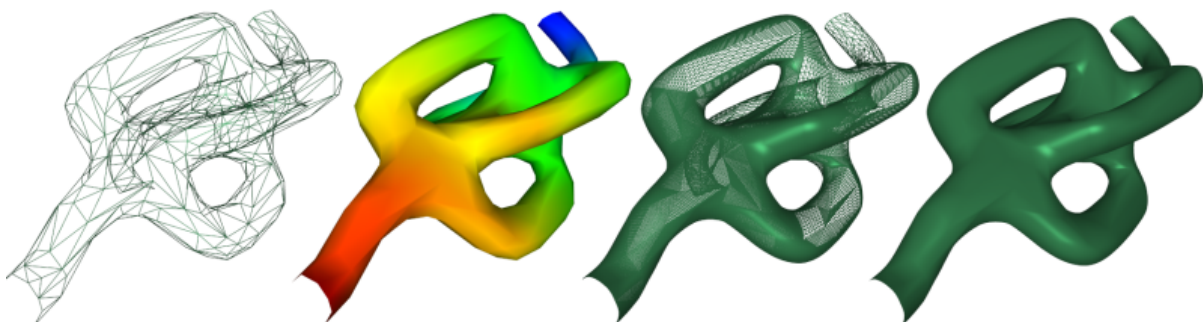

Figure 7.7: *Adaptive refinement of a deformable genius-4 shape. The refinement provided by the ARK is not restricted to a particular topology, nor manifold conditions.*

7.3 Refinement Zoo

In this section, we present various examples of on-the-fly mesh refinement algorithms which have been implemented with our kernel.
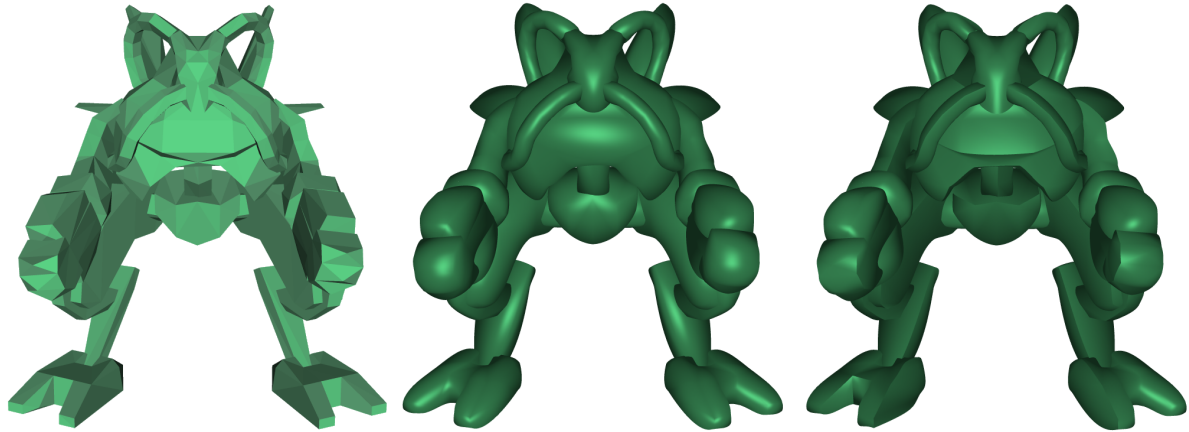
7.3.1 Bézier Smoothing

Figure 7.8: **Left:** *Coarse mesh (1246 triangles on CPU).* **Middle:** *Adaptive interpolated smoothing by Curved PN Triangles (1.1M generated triangles on GPU).* **Right:** *Sharp features, tension and bias control with Scalar Tagged PN Triangles (similar number of generated triangles on GPU).*

Curved PN Triangles [VPBM01] are an efficient alternative to usual subdivision surfaces. This method generates an interpolated "visually" smooth refinement over an arbitrary mesh just by taking into account positions and normals stored at each triangle vertex. The basic idea is to define a cubic displacement field and a quadratic normal field, each of them being defined by a simple triangular Bézier patch. Scalar Tagged PN Triangles, presented in Chapter 8, improve this scheme by allowing accurate control of sharp creases, local tension and bias with additional vertex attributes. The computation of the corresponding Bézier control points can be done on CPU and transmitted to the GPU as additional vertex attributes. But, as the involved computation is very light and does not involve specific data structures, the whole process can be implemented on the vertex shader. Figure 7.8 shows two results obtained with our GPU implementation of these techniques. It should be noted that compared to benchmarks provided by our graphics device manufacturer, the framerates we obtain for deep refinement show that the ARK saturates the GPU vertex processing horsepower, which means that no bottleneck appears neither on CPU nor on the graphics bus.
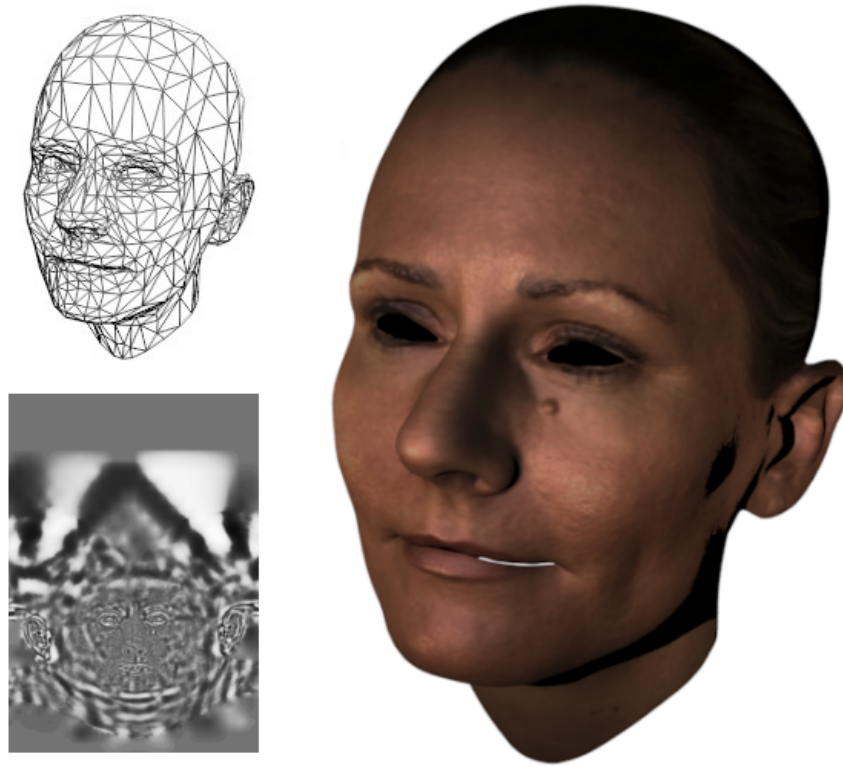
7.3.2 Full GPU Displacement Mapping

Figure 7.9: *Real-time displacement mapping.* **Top Left:** *Coarse mesh streamed from CPU (*1914 *polygons).* **Bottom Left:** *Displacement map stored on GPU.* **Right:** *Displaced Adaptive PN Triangles, generated on the fly in real-time by our GPU Kernel (3.6M polygons). This final rendering (58 FPS) includes the use of displacement map with our kernel on the vertex shader, as well as normal, color and shadow maps on the fragment shader (data courtesy Cyberware).*

Recent graphics hardware allows vertex-texture fetches [Fer05]. This means that *displaced subdivision surfaces* [LMH00] can be easily implemented by storing the displacement in a floating point texture, and accessing it in the second stage of the refinement shader. However, GPU evaluation of subdivision surfaces can be expensive on the vertex shader because it requires complex computation for vertices with high valence (we address this particular problem in Chapter 9). Fortunately, in the work of Lee et al. [LMH00], the subdivision process is only used for smoothly sampling a base domain for vertex displacement, while the final geometric continuity is expressed by the displacement and not the subdivision. In this case, Curved PN Triangles [VPBM01] can provide a smooth enough base domain in many cases compared to genuine subdivision surfaces, with the additional benefit that no local neighborhood has to be transmitted to the vertex shader to achieve the refinement of a given coarse triangle. Figure 7.9 gives an example of the rendering of such *Displaced PN Triangles*.
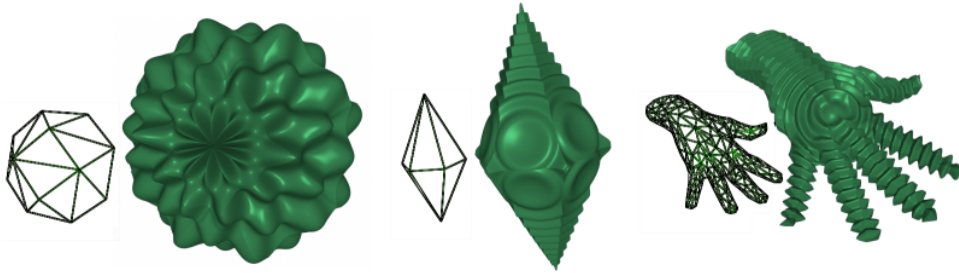
7.3.3 Procedural Refinement

Figure 7.10: *Few examples of complex shapes defined by a simple mesh with an high frequency procedural displacement. Deep refinement can be reached efficiently.*

Geometry synthesis by procedural refinement is clearly one of the best examples that enlightens the strength of our ARK. These techniques often define a very coarse mesh, with complex displacement functions, potentially requiring a very high tessellation rate to correctly sample all high frequency features. Figure 7.10 shows several examples of such refinement, which only require to transmit a small set of user-defined parameters to define the corresponding procedural displacement function.

7.3.4 Adaptive Terrain Rendering
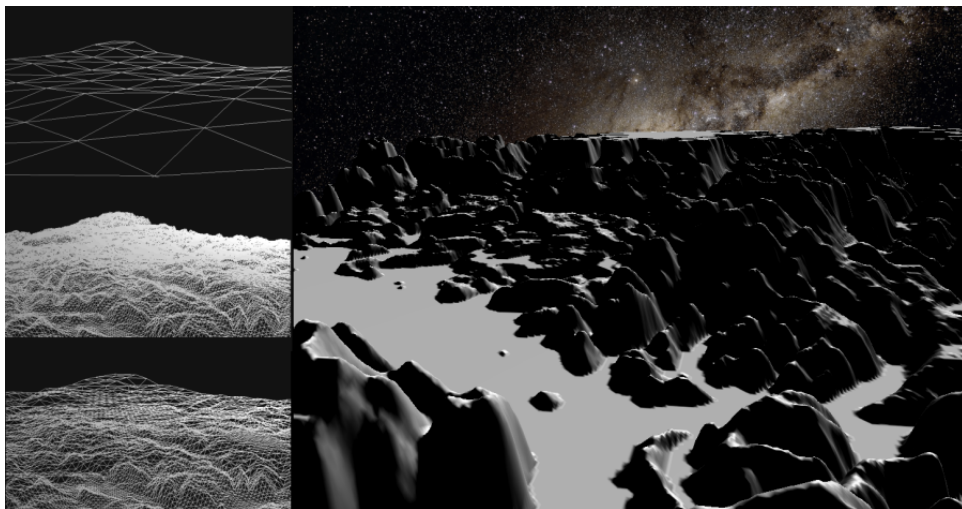
Figure 7.11: *This terrain has been rendered at an average framerate of 44 FPS (6M tri.), by using a single height-map texture to displace the refined tessellation. The refinement is driven by a view-dependent depth-tagging.* **Left top:** *Topology for input ground.* **Left middle:** *uniform on-the-fly refinement with the URPs.* **Left bottom:** *adaptive on-the-fly refinement with the ARPs.* **Right:** *Final adaptive real-time rendering.*

While dedicated systems exists for efficiently adaptive rendering of terrains [AH05, LC03], the ARK allows very simple adaptive refinement of height-field models. We use a basic ground made of few hundreds polygons as a coarse mesh, and upload an high resolution height-field as a floating point texture to the GPU memory. Then, at rendering time, we tag the vertices of the coarse ground using a

view-dependent distance metric. Finally, the coarse ground is adaptively tessellated on-the-fly by the ARK and displaced using vertex texture fetch from the height-field texture (see Figure 7.11).

7.3.5 Animated Mesh Refinement
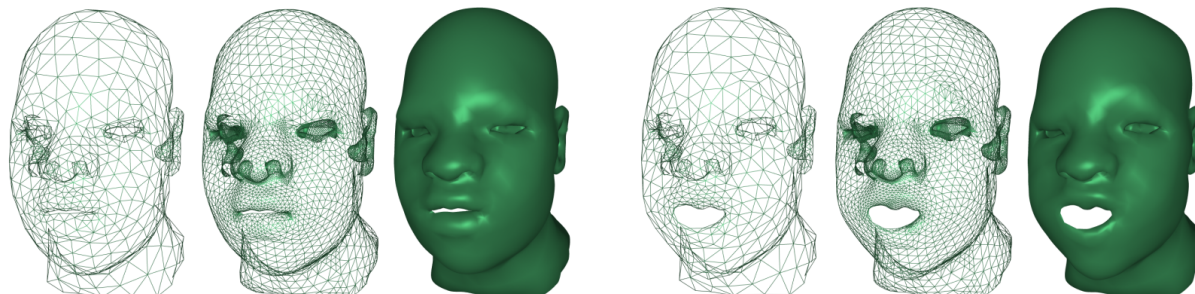
Figure 7.12: *Dynamic refinement of an animated mesh.* **Left:** *Frame 1.* **Right:** *Frame 12. The coarse mesh is animated on the CPU, and the GPU maintains an adaptive refinement driven by dynamic tracking of curvature modifications.*

Animated meshes are another important application that could significantly benefit from our ARK. Indeed, as mesh refinement is performed on-the-fly, without storage and without specific per-object pre-computation, an animated mesh just requires a frame-by-frame update of its depth-tag configuration, in addition to usual vertex position update by the application. An adequate adaptive refinement will then be generated at each frame. Figure 7.12 presents two frames of a face animation sequence with dynamic adaptive refinement. The depth-tagging is based on a local curvature estimation performed frame-by-frame, while the refinement process uses smoothing by Curved PN Triangles over the coarse mesh.
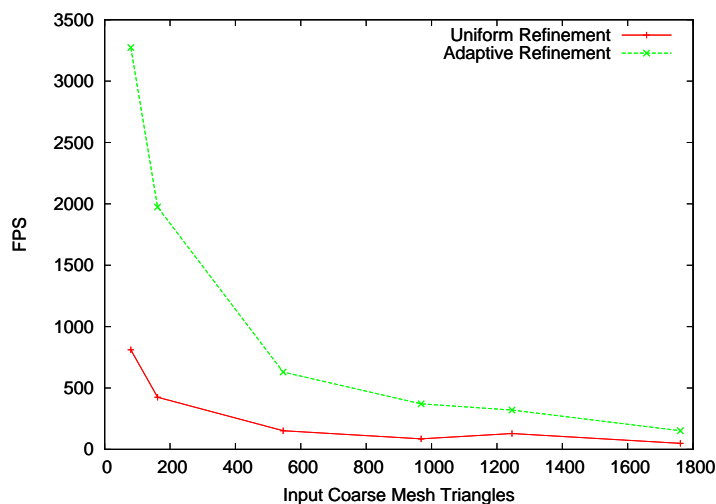
Figure 7.13: *Comparison between the frame rate obtained with uniform mesh refinement (URP) and our new adaptive refinement (ARP). For the largest coarse meshes (1800 on-CPU triangles), more than two million triangles are generated on-the-fly by the vertex shader. Note that our method is between one and three orders of magnitude faster than the equivalent CPU-based adaptive refinement.*

121

7.4   Implementation and Performance

Our implementation runs under Linux, using OpenGL and GLSL. All tests have been performed on an nVidia GeForce 8800 GTX with 768MB of memory, on an Intel P4 2.4GHz with 1GB of memory.

GPU Implementation of ARP: The ARP is the central structure of our system. In order to tightly reach the maximum performance at rasterization time, the ARP is encoded as an *indexed vertex buffer* of degenerated triangle strips [SWND05], directly on the GPU memory. Moreover, because we use dyadic refinement, each refinement level is actually a superset of the previous one, so we can further reduce the global memory footprint by separating the geometry from the topology. A *vertex buffer* is used to encode all the geometry by storing the set of barycentric coordinates for the nodes that belong to the deepest uniform ARP. Then the topology of any given ARP is encoded by using an *index buffer*, as an indexed strip over this maximum configuration. So, at rendering time, when the application selects a given ARP for refining a coarse triangle, the only action performed by the API is to bind the corresponding index buffer and set the correct offset, while always keeping the same vertex buffer, which guarantees cache-friendly access.

Regarding memory usage, on the CPU side, the only memory overhead comes from the storage of the set of ARP identifiers. This overhead is extremely small and totally independent of the current 3D scene. For instance, if the maximum refinement level is set to 10 (which offers a maximum refinement of 1024*x*1024 sub-triangles for each coarse triangle), the precomputation (all ARP generation) time is less than half a second and the main memory overhead is less than 4kB. On GPU side, the memory overhead required to store the set of ARP at this resolution is about 26MB.

For either uniform or adaptive refinement patterns, we have observed that in the case of deep refinements, rendering performances were very close to the one obtained with static meshes (i.e refined during a preprocessing step and stored on the GPU one time for all). This can be explained by the small memory requirement of our method, which maintains a good cache coherency.

Note that in restricted conditions, with 16-bit precision (e.g. PDAs), our ARP encoding allows a maximum refinement level of 256×256 for each coarse triangle. At the other extreme, with a modern GPU and very high resolution displays, we have experimented real-time performance when using up to 2048×2048 tessellation for each coarse triangle. Even higher resolutions can easily be reached, since the ARK fully runs in object space.

Figure 7.13 presents the rendering frame rate obtained for various models. The measure integrates the tessellation step and a simple procedural displacement step for an animated mesh. A dynamic adaptive refinement has been performed frame-by-frame, based on an approximated local curvature estimation, combined with a view-dependent refinement bound. Compared to our URP technique, our ARP scheme offers a gain ranging from 250 to 460% depending on the model, while providing the same final image quality. This can be explained by a finer gradation of the tessellation, avoiding rendering of unnecessary small triangles (e.g. flat areas, far areas). In many cases, the quality is even better, since the aliasing of over-tessellated meshes (more than one triangle for a pixel) is strongly reduced. In extreme cases, the gain can even reach one order of magnitude, when the depth-tagging is static and the input mesh is very coarse (see Figure 7.10, for instance). Compared to our optimized CPU implementation of adaptive refinement, our GPU refinement kernel improves the frame rate between one and three orders of magnitude, depending on the overall complexity of the refinement.

Figure 7.14 shows the frame rate obtained for a target refined mesh made of 1M triangles, under various *input size vs refinement depth* ratios. It clearly appears that coarse meshes with high refinement

depth totally outperforms medium meshes with low refinement depth, for the same total number of triangles. This comes from the fact that for the latter, transmission of input polygons attributes becomes a bottleneck on the graphics bus. At the other extreme of the spectrum, when the target shape can be described by a very coarse mesh with deep refinement, the ARK runs in an optimal context and completely saturates the GPU vertex processing horsepower.
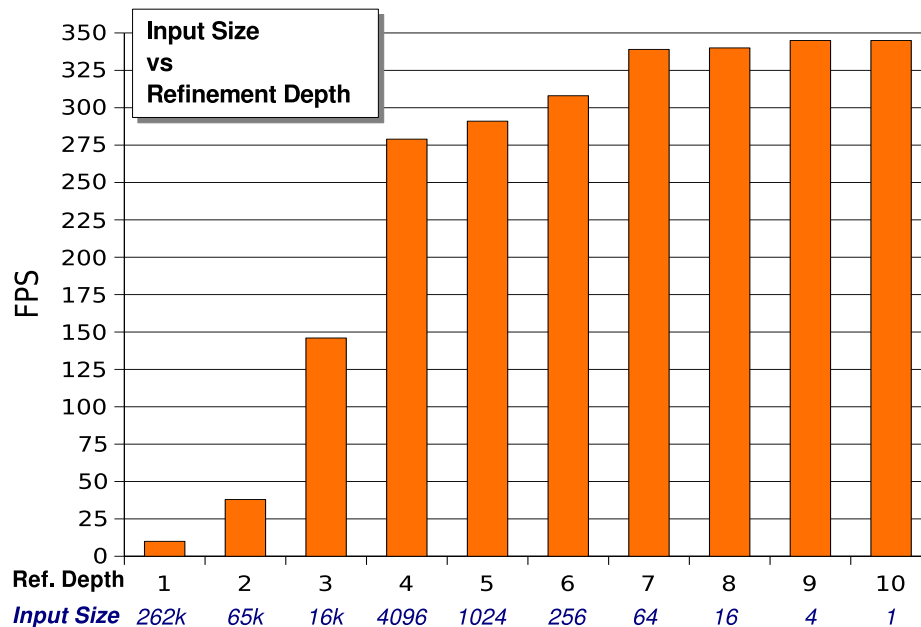

Figure 7.14: *This diagram shows frame rate measures for a target refined mesh resolution of 1M triangles under various input size vs refinement depth ratio.*

7.5 Discussion

Limitations The technique presented here has essentially two main limitations. First, the refinement depth must be sufficient to avoid the bottleneck involved in the transfer of the per-vertex attributes. Second, on elder graphics hardware, vertex texture fetch is slow, which limits applications such as terrain rendering and displacement mapping. Fortunately, this restriction has recently disappeared with the introduction of *unified shader architectures* on graphics hardware. For instance, the terrain render application at Figure 7.11, which uses intensively texture access from vertex shader, runs at about 2 FPS on an nVidia Geforce 6800 and 44 FPS on an nVidia Geforce 8800 (unified shader architecture), for an average refinement depth of 8, which produces about 6M polygons.

Another concern may be the question whether the depth tagging should be better performed on the GPU instead of the CPU. This could be done by using a preliminary rendering pass that would store vertex depth-tags in a texture. However, this would involve a strong limitation on the kind of depth-tagging that can be implemented, as many useful information may only be available for the application running on the CPU. Moreover, as the depth-tagging is performed on the coarse mesh, the computation overhead remains negligible, particularly in the case of deep refinements.

Refinement Kernel vs. Geometry Shader DirectX 10 technology [Bly06] has introduced a new *geometry shader* (**GS**) stage in the hardware rendering pipeline. The first graphics devices including

123

these functionalities have been launched at the end of 2006. Even if the GS can obviously be used to perform mesh refinement, its features are quite different from the way we have structured our ARK. The main limitation when using the GS to perform mesh refinement is that the level of geometry upsampling is hardware limited and fixed. For instance, only 1024 floating point numbers can be output with current specifications [Bly06]. This is far from being able to tessellate up to 2048×2048 triangles per coarse polygon for instance, as with our ARK. Multi-pass GS rendering may be employed to reach deeper refinement, but it would obviously strongly reduce overall performance. In practice, as mentioned by hardware manufacturer [Gre06], it is not even possible to reach the single pass upper bound, without observing a huge performance degradation.

Even without the limit of geometry upsampling, implementing adaptive mesh refinement with the GS would also require to correctly manage crack-free junctions between different tessellation rates. With our precomputed ARPs, this problem is solved once for all and stored, while the GS would have to generate consistent topologies on-the-fly and thus require complex shader code. Notice that, as the GS implements a superset of the vertex shader functionalities, the solution provided by the ARPs can straightforwardly be implemented on the GS.

Actually, we consider that the GS stage represents a complement to the ARK, rather than an alternative. By combining both approaches, one may generate more complex refinement in a two stage process. First, at the VS stage, the ARK tessellates and displaces a base domain (e.g. apply Bézier smoothing on very coarse meshes) and then additional vertices are inserted at the GS stage (e.g. local extrusion to create hairy objects). We can also imagine using the GS for low refinement depth where the ARK is less efficient, and then switch to ARK to get high refinement depth when needed.

Low level API extensions   The presented kernel can be integrated at the driver level, in any standard graphics API such as OpenGL, without any additional hardware capabilities. In this case, the control interface is a reduced set of functions:

- `glARKinit(GLuint maxLevel)`: builds the set of ARP (special indexed vertex buffers) on GPU, and stores the corresponding identifiers, indexed by depth-tags.

- `glEnable(GL_ARK)`: when activated, the ARK refinement will replace any triangle drawing call by the corresponding ARP.

- `glDisable(GL_ARK)`: restore usual OpenGL behavior;

- `glDepthTag1i(GLuint d)`: set the current vertex depth-tag state (for upcoming vertices).

The fixed pipeline would provide a simple linear refinement, which can then be tuned by setting user specific adaptive refinement shaders. With this set of functions plus some additional commodity callbacks, the use of the ARK is totally transparent to programmers (direct port of existing source code). Alternatively, finer control of refinement can be provided through specific functions (`drawARP()` for instance) in order to mix refined and regular drawing calls without switching the mode.

Summary   We have presented a simple and efficient GPU kernel for *adaptive geometry synthesis by mesh refinement* based on a generic depth-tagging process, that makes it suitable for any refinement control that can be expressed on a per-vertex basis (e.g. curvature, view-dependent LOD, area of interest penalty, local estimation of displacement variation, etc). We have introduced *Adaptive Refinement Patterns* and *Adaptive Refinement Shaders*, and have shown that their combination allows the implementation of various kind of dynamic refinement, with almost no modification of the rendering loop at

application level. The CPU processing is reduced to the transmission of (dynamic) coarse meshes to the GPU, eventually combined with additional dynamic data for driving the refinement.

The kernel allows very deep adaptive refinement, using a single-pass vertex program. It does not impose any conversion of input mesh (such as local or global parameterization) and allows further on-GPU geometric processing, since it consistently performs geometry synthesis in object space. The solution is more efficient and even more flexible than prior software-based methods. In practice, the benefit of the ARK is proportional to the depth of the adaptive refinement. The kernel permits to "saturate" the GPU with geometry to draw, and with its intrinsic CPU-to-GPU streaming principle, it is possible to draw a refined surface almost independently of the amount of available memory, either on CPU or GPU (the ultimate limitation is represented by the storage on GPU of the set of ARP required for the chosen depth).

Our refinement kernel exhibits an interesting collaboration between the CPU and GPU (the global analysis at coarse resolution is let to the CPU for depth-tagging, while local fast refinement is performed on the GPU, driven by these tags). This corresponds to the idea of shared rendering workload between powerful multi-core CPUs and GPUs, as stated by Pharr [Pha06]. Future graphics hardware and API developers seek for refinement methods based on generic barycentric interpolation, as mentioned by Sloan [Slo06]. Thus our kernel can also be considered as a first step, performing a software emulation of such future on-board *refinement shader* stage.

Perspectives: With this kernel in hand, we can address various applications of geometry synthesis. First, we have shown how easy and efficient can be the PN triangle refinement with the ARK, so we will discuss the local control that can be performed on that kind of surface in Chapter 8. Second, one important mesh refinement method that need to be developed for interactive application is mesh subdivision.
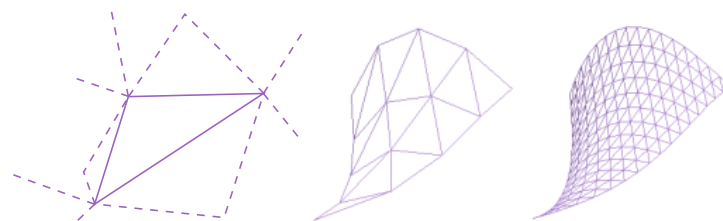
Figure 7.15: *Refinement of Loop subdivision surface with the ARK. From left to right: the input coarse triangle (with its neighbors in dashed lines), the synthetized piece of subdivision surface at depth 2 (1-16 refinement) and 4 (1-256).*

As already mentioned above, genuine subdivision surfaces are very different to mesh smoothing techniques based on Bézier patches, as the refinement of each coarse polygon is usually implemented in a recursive procedure depending on its one-ring (or even two-ring neighborhood). We have studied two single-pass approaches to this problem:

- **exact** *subdivision surface rendering*: as stated by Stam [Sta99], an exact evaluation of limit surfaces at arbitrary parameter is possible by tilling the parameter values domain in a set of triangular patches and performing an eigen analysis of the so-defined parameterization. Unfortunately, in the case of a triangle indexing an extraordinary vertex, the implementation requires a huge amount of additional data for each triangle, which is no more compatible with efficient rendering. So, we have tried to develop a hybrid CPU-GPU implementation which delays a large part of the computation on the ARK, keeping the horsepower of modern multi-core CPUs for non regular

cases. Figure 7.15 gives a preliminary example of our current work on a Loop subdivision with our kernel. However, this solution is still development, and seems to not be competitive with the second one.

- **approximate** *subdivision surface rendering*: we have developed an approximation of subdivision surfaces for interactive applications, which can be implemented efficiently with the ARK, reaching real-time performances for millions of polygons output while being visually very similar to exact subdivision surfaces.

We discuss this application in the Chapter 9.

Chapter 8

Controllable Mesh Smoothing with Scalar-Tagged PN Triangles

This chapter presents a new fast mesh enhancement technique based on smoothing by refinement. This technique can be applied on the simplified geometry obtained at the end of the acquisition pipeline for improving *on-the-fly* the surface quality for rendering. We improve the principle proposed by Vlachos et al. in their "Curved PN-Triangles". The key idea is to assign to each mesh vertex, a set of **three scalar tags** that act as shape controllers. These scalar tags (called sharpness, bias, and tension) are used to compute a procedural displacement map that enriches the geometry, and a procedural normal map that enriches the shading. The resulting technique offers two majors features: first, it can be applied on meshes of arbitrary topology while always generating surfaces with consistent behaviors across edge and vertex boundaries, second, it only involves operations that are purely local to each polygon, which means that it is very well suited for GPU implementation, with for instance the ARK presented in the previous chapter.

8.1 Curved PN Triangles

Compared to true subdivision schemes [ZS00] or mesh smoothing techniques, *Curved PN-Triangles*, a totally local refinement scheme introduced by Vlachos et al. [VPBM01], is much better suited to hardware implementation, since no adjacency information between triangles has to be stored and managed. More precisely, starting from an input coarse mesh equipped with vertex normals, an interpolating refined mesh is generated on-the-fly at rendering time by replacing each coarse triangle with a Bézier patch driven by the 3 positions and normal of the triangles vertices. The most innovative idea of PN-Triangles, compared to previous work, is to relax the constraint of high-order geometric continuity, and to show that a simple *visual smoothness* is sufficient for several applications. This visual smoothness is obtained by computing, simultaneously but independently, a displacement field, defined as a cubic Bézier patch, used to enrich the geometry of each triangle, and a procedural normal field, defined as a quadratic Bézier patch, used to enrich its shading. Note that an hardware implementation can be easily designed for such an empirical local smoothing method [CK03b, CK03a]. In order to offer a greater control on the initial coarse mesh, this chapter proposes to assign to each vertex of this coarse mesh, a set of *three scalar tags* that act as intuitive shape controllers, namely sharpness, tension and bias. The area of influence of these shape controllers is very local but is sufficient to guarantee consistent local surface features, such as curvature values around vertices or tangent plane discontinuities across edges.

8.2 Description of Scalar Tags

8.2.1 Local surface analysis

Indexed faces sets have become the most common data structure to store polygonal meshes, as it avoids the duplication of the vertex coordinates. But it has also one major consequence: the only adjacency relationship stored in the data structure are the indices of common vertices shared by two neighboring polygons. Thus the only way to get a consistent behavior of the surface across polygon boundaries is to store the shape parameters on a per vertex basis and to ensure that the influence of all the shape parameters is strongly localized around each vertex.

Unfortunately, if a per-vertex storage is well-adapted to per-vertex shape parameters such as local tangent plane or local curvature, it is much less adapted to per-edge features that may exist in the geometry, such as creases or straight lines. So, to be able to correctly account for per-edge features, we impose some constraints on the *one-ring* neighborhood of each vertex. Let us consider Figure 8.1: a crease passes through the vertex O and cuts the underlying triangle fan in two sub-fans. An average normal vector N^+ and N^- can be computed for each sub-fan, by simply averaging the normal vectors of the included triangles. The sharp crease is then implicitly defined by the three tagged vertices A, O and B. The corresponding normal discontinuity can be simply encoded, by applying a kind of Haar filtering on the two normal vectors N^+ and N^-: we store the average normal vector $N = N^+ + N^-$ (which is normalized to unit length) and a difference vector $\Delta = N^+ - N^-$. So $\Delta = 0$ corresponds to a smooth vertex. In the remainder of the chapter, we will use the word "tagged" to specify a vertex with a non-null Δ and the word "untagged" otherwise. To be able to always keep a local decision about per-edge features, we

(a)   (b)
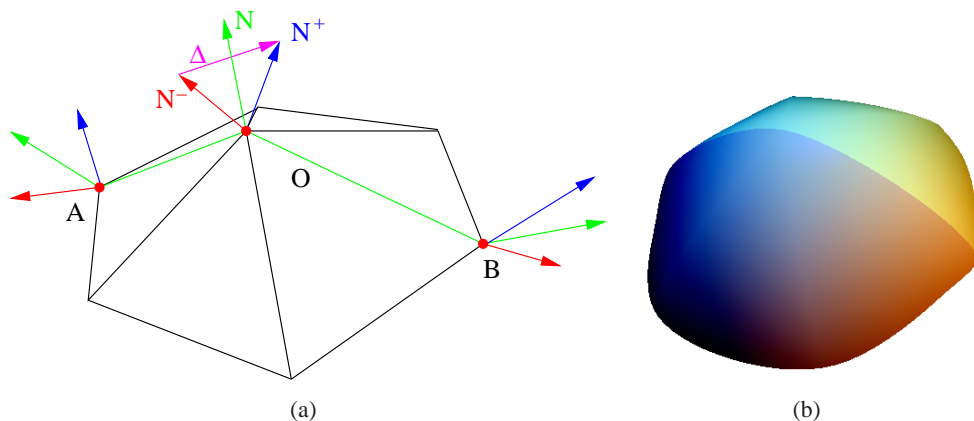
Figure 8.1: *(a) At the vertex level, the green sharp crease can be encoded by a simple vector Δ. (b) This additional per-vertex data locally controls the underlaying per-triangle smooth surface generation.*

impose the following restriction on the local configuration:

1. A tagged vertex can have 2 tagged neighbors at most.

2. A triangle can have 2 tagged vertices at most.

The first restriction ensures that only one crease passes through a given vertex. If not, this would mean that we need more than one vector Δ to encode the normal discontinuity at this vertex. As an extension, encoding several vectors Δ would allow to represent multiple sharp creases, but at the price of a more important per-vertex data set. The second restriction make unambiguous the difference between two distinct creases that are separated by only one triangle, and a crease that loops around a single triangle. Note that some simple local remeshing step can remove this limitation (split for instance).

8.2.2 Shape parameters through scalar tags

Compared to the original PN-Triangle model, the inclusion of the normal discontinuity vector Δ allows us to generate different displacement fields and normal fields on both sides of a crease edge. We propose now to define additional per-vertex scalar values (i.e. *scalar tags*) to offer an even more accurate control of the local geometry. We have selected three shape parameters that are particularly well adapted to the control of sharp creases. We first describe these three shape parameters independently of the underlying refinement technique.

The first scalar tag $\sigma \in [0, \infty[$ is called *sharpness*: it defines the divergence of normal vectors across the two sides of the crease, by interpolating between totally smooth ($\sigma = 0$) and totally sharp ($\sigma = \infty$) configurations (see Figure 8.2(a)).

The second scalar tag $\theta \in [-1, 1]$ is called *tension*: it corresponds to the usual tension parameter that has been defined in the spline literature [BB83, Far02]. It is used to locally control the curvature of all Bézier boundary curves that are starting from on a given tagged vertex (see Figure 8.2(b)), and allows to interpolate between three different configurations: tensed Bézier ($\theta > 0$), standard Bézier ($\theta = 0$) and relaxed Bézier ($\theta < 0$).

The third scalar tag $\beta \in [-1, 1]$ is called *bias*: it also corresponds to the usual bias parameter that has been defined in the spline literature [BB83, Far02]. It is used to locally control the direction of all Bézier boundary curves that are starting from a given tagged vertex (see Figure 8.2(c)). Here again three different configurations are interpolated: bias toward N^+ ($\beta > 0$), no bias ($\beta = 0$) and bias toward N^- ($\beta < 0$).
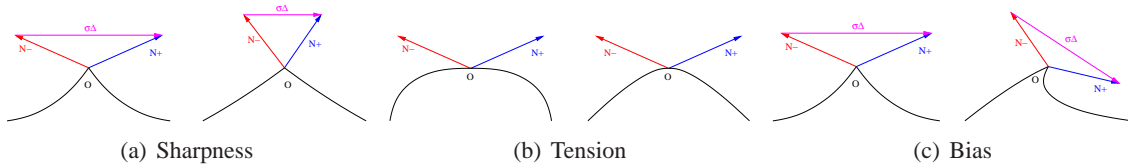

(a) Sharpness (b) Tension (c) Bias

Figure 8.2: *The role of scalar tags. (a) Sharpness controls the normal deviation on sharp creases. (b) Tension controls the curvature of boundary curves in the vertex neighborhood. (c) Bias controls the direction of boundary curves in the vertex neighborhood.*

These three scalar tags are used as shape controllers and they completely drive the mesh refinement. Our experiments have shown that these values, defined by the user, are very intuitive and predictable, even for users not familiar with geometric modeling software.

To sum up, an enriched coarse mesh (*ST Mesh* in the remainder) can be defined by using a set of two tables V and T. Each line of table V stores all the data relative to a vertex: the position P, the average normal vector N, the normal discontinuity vector Δ, and the three scalar tags σ, θ, and β. Similarly, each line of table T stores only the three vertex indices (i, j, k) relative to a triangle.

8.3 Mesh generation

8.3.1 Combining shading and smoothing

As said above, our technique is strongly based on the PN-triangles presented by Vlachos et al. The reader unfamiliar with this work may refer to [VPBM01] for details on the construction of PN-triangles.

In order to obtain a coherent effect of the shape parameters defined above, their influence has to be accounted both for the shading and the geometry of the surface generated during the rendering process. As shown in Figure 8.3, in the case of a sharp crease, this approach ensures a coherent behavior, both on the silhouette and at the interior of the object.
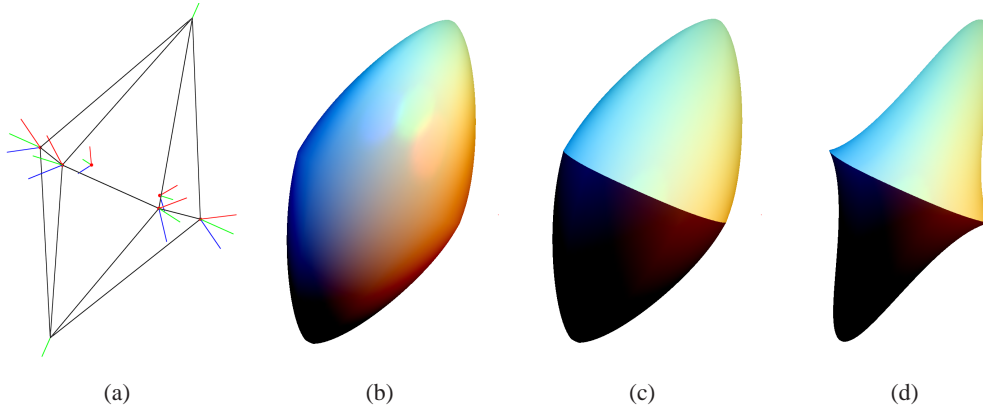

(a)   (b)   (c)   (d)

Figure 8.3: *(a) Coarse mesh with a ring of vertices tagged as sharp ($\sigma = 0.7$). (b) Result obtained with standard PN-Triangles. (c) Result with sharpness only in shading. (d) Result with sharpness both in shading and geometry.*

The shape factors described in the previous section can now be used to efficiently generate a surface with sharp features. Globally, we can make a distinction between:

- the sharpness value σ which mainly acts on the shading,
- the bias β and the tension θ which mainly act on the silhouette of the object, and so on the underlying geometry.

We propose to generate a coherent shading for ST-Meshes with a *procedural normal map* constructed with the modified normals, and a smoothing algorithm for the geometry that can be formulated as a *procedural displacement map*. Both of them are computed with one triangular Bézier patch, similarly to PN-Triangles. The combination of these two procedural maps produces a real-time piecewise smooth visualization that is accurately controlled by the simple per-vertex scalar tags of the ST-Mesh.

8.3.2 Generation of the normal field

The normal field constructed across a triangle has to be smooth in the interior of the triangle, continuous across an untagged edge (i.e. without normal discontinuity) and consistent to the discontinuity encoded by the σ values of tagged vertices.

To account for tagged vertices, the three original normal vectors are modified in the following way: since Δ_i represents the direction of the discontinuity at vertex i, we define $N_i'^+$ (resp. $N_i'^-$), by $N_i'^+ = (N_i + \sigma_i \Delta_i)/\|(N_i + \sigma_i \Delta_i)\|$ (resp. $N_i'^- = (N_i - \sigma_i \Delta_i)/\|(N_i - \sigma_i \Delta_i)\|$). The choice of $N_i'^+$ or $N_i'^-$ is made according to the classification of the triangle against the triangle-fan split introduced by a sharp crease (see Figure 8.1). A linear or quadratic (Bézier) interpolation between these normals can produce a visual smoothness over the refined mesh [VPBM01]. In the remainder of the chapter, we will note N_i', the normal of the *current* side of a crease for vertex i.

8.3.3 Generation of the displacement field

As proposed by Vlachos et al., the displacement field will be computed by defining a triangular Bézier patch. But the shape modifications generated by the scalar tags at each vertex have to be accounted for, when generating the displacement field, so the process has to be slightly modified.

A set of 10 Bézier control points have to be computed to define a cubic triangular patch (see Figure 8.4), to define the displacement field $b(u,v)$:

$$
\begin{aligned}
b(u,v) \;=\; & b_{300}w^3 + b_{030}u^3 + b_{003}v^3 \\
& + 3b_{210}w^2u + 3b_{120}wu^2 + 3b_{201}w^2v \\
& + 3b_{021}u^2v + 3b_{102}wv^2 + 3b_{012}uv^2 \\
& + 6b_{111}wuv
\end{aligned}
\tag{8.1}
$$

To simplify the upcoming notations, we propose to decompose each control point as:

$$
b_i = d_i + e_i
\tag{8.2}
$$

where d_i corresponds to the position of the control point when the patch is in a flat configuration (i.e. all control points are lying in the plane) and e_i is the displacement vector when d_i is projected onto the plane defined by the normal and the position of the closest vertex. As in [VPBM01], we classify the control points into 3 main categories:

- **vertex coefficients**: b_{300}, b_{030}, b_{003}

- **tangent coefficients**: b_{210}, b_{120}, b_{021}, b_{012}, b_{102}, b_{201},

- **center coefficients**: b_{111} is procedurally obtained by the formulation proposed by Farin to ensure quadratic precision [Far02].
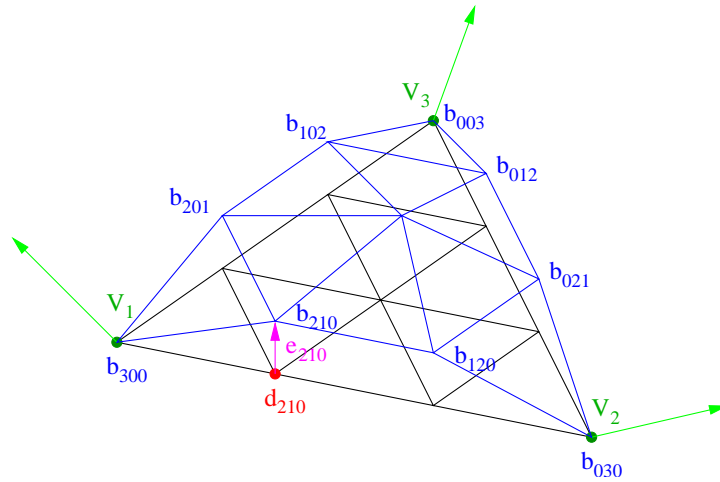
Figure 8.4: *A cubic triangular Bézier patch replaces each input triangle. Each control point of this patch can be decomposed in a parameter position d_i and a displacement e_i, defined using vertex positions and normals, and in our case, additional scalars tags.*

The scalar tags should neither affect the vertex coefficients (as we always want an interpolating surface) nor the center coefficient (as Farin's formulation always maintains a nice shape for the interior of the patch). So, we propose to reduce the geometric expression of the scalar tags only through the tangent coefficients. Moreover, to preserve coherence across triangle boundaries, the scalar tags carried by a

vertex will only affect the two nearest tangent coefficients. For example, the scalar tags of V_1 will only affect the coefficients b_{210} and b_{201}.

In the remainder of this section, we consider the case of a coefficient b_i, which is computed using the scalar tags of $V_j = (P_j, N_j, \Delta_j, \sigma_j, \theta_j, \beta_j)$, and the position of the opposite edge vertex P_k. We have also to determinate whether the coefficient is on a sharp edge or not. For this, we use a predicate δ_i which is true if the two relative edge vertices are tagged.

Let $\Pi(p, n, q) = -n.(q - p)$ be the signed distance operator of projection of q onto the plane defined by the point p and the normal n. We can write the Equation 8.2 as:

$$b_i = d_i + \Pi(P_j, N_j, d_i)N_j$$

with $d_i = P_j + (P_k - P_j)/3$.

For instance, with the coefficient b_{210} of Figure 8.4, we have $j = 1$ and $k = 2$; δ_{210} will be true if V_1 and V_2 are tagged, false otherwise. The formulation of this coefficient becomes:

$$b_{210} = d_{210} + \Pi(P_1, N_1, d_{210})N_1$$

with $d_{210} = P_1 + (P_2 - P_1)/3$. Let us now describe how to modify the geometric definition for a tangent coefficient b_i associated with a tagged vertex V_j.
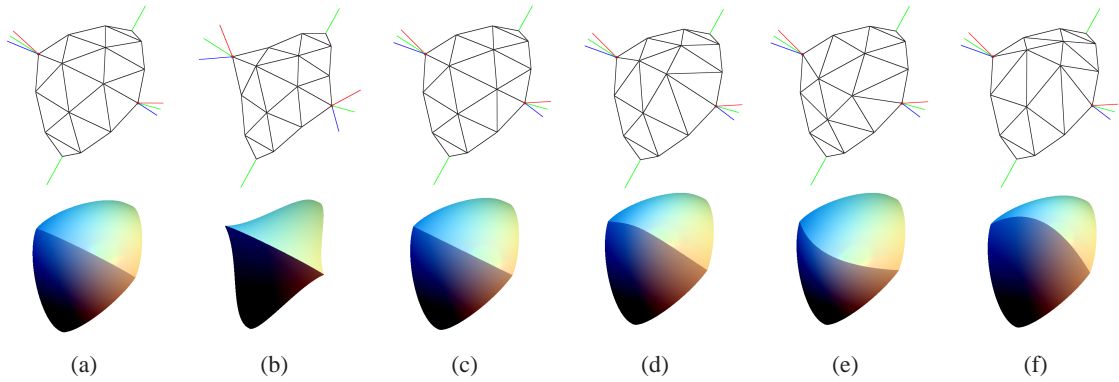

Figure 8.5: *Transmission of the scalar tags of two vertices to adjacent Bézier patches. (a) $\sigma = 0.2$, $\theta = 0$, $\beta = 0$ (b) $\sigma = 1$, $\theta = 0$, $\beta = 0$ (c) $\sigma = 0.2$, $\theta = 0.2$, $\beta = 0$ (d) $\sigma = 0.2$, $\theta = -0.6$, $\beta = 0$ (e) $\sigma = 0.2$, $\theta = 0$, $\beta = -1$, (f) $\sigma = 0.2$, $\theta = 0$, $\beta = 1$*

Sharpness: To get a consistent silhouette for the refined surface, we have to transmit the sharpness value σ of a vertex to its relative tangent coefficients. This means that b_i has to express the "flatness" of the Bézier patch near the sharp crease (see Figure 8.5(a) and 8.5(b)), which is actually the only important aspect for its perception. So, we just have to act on the projection, by constraining b_i to the plane of the "sharp" normal, according to σ, with the following formulation:

$$e_i = (1 - \delta_i)\Pi(P_j, X_j, d_i)X_j \text{ with } X_j = \frac{(1 - \sigma_j)N_j + \sigma_j N'_j}{\|(1 - \sigma_j)N_j + \sigma_j N'_j\|}$$

If $\sigma_j = 0$ we are in the "smooth" case. Otherwise, the modified normal will flatten the patch near the tagged edge by reducing the elevation produced by e_i.

Tension: As usual in tensed Bézier splines, the tension around a vertex V_j will be controlled by the distance between its associated tangent coefficients b_i and its position P_j (see Figure 8.5(c) and 8.5(d)).

With our formulation, this leads to simply translate d_i before evaluating the projection e_i. We want the tension to be maximal when $d_i = P_j$, so the tangent coefficient will simply be computed by:

$$d_i = P_j + \frac{(1 - \theta_j)}{3}(P_k - P_j)$$

Bias: The bias factor is independent of the crease side: two triangles sharing a common vertex V_j of a sharp crease have to conform their Bézier patches in the same direction, defined by the Δ_j (see Figure 8.5(e) and 8.5(f)). This time, we want the bias to be expressed only for sharp edges, and we propose to act again on e_i, by using a projection direction that directly takes into account Δ_j. We obtain:

$$e_i = \delta_i \Pi(P_j, N_j, d_i) Y_j \text{ with } Y_j = \frac{N_j + \beta_j \Delta_j}{\|N_j + \beta_j \Delta_j\|}$$

By stitching all together, we obtain the following final formulation for the tangent coefficients:

$$
\begin{aligned}
b_i &= d_i + e_i \\
d_i &= P_j + \frac{(1-\theta_j)}{3}(P_k - P_j) \\
e_i &= (1 - \delta_i)\Pi(P_j, X_j, d_i)X_j + \delta_i \Pi(P_j, N_j, d_i)Y_j
\end{aligned}
\tag{8.3}
$$

The remainder of the process is totally similar to the one used with PN-triangles: the 10 Bézier control points do totally define the continuous displacement field. This field, and the associated normal field, can thus be directly sampled in real-time by the ARK (Chapter 7).

8.4 Summary

In this chapter, we have shown how to easily control some useful local surface singularities through a reduced set of scalar shape parameters encoded in a per-vertex basis. This work enriches the original PN-Triangle model, and allows the user to design more complex shapes at a coarse level that will be dynamically refined preserving these shape parameters, which is an interesting property for real-time applications and compression. For instance, the models obtained after the various simplification algorithms proposed in the first chapters of this thesis can be enhanced easily.

However, PN triangles remain en empiric approach to surface refinement: in particular, the final quality is not competitive with true subdivisions surfaces [ZS00]. In the following chapter, we propose a new mesh refinement process which approximates true subdivision surfaces, producing convincing, visually plausible, rendering and very fast to compute once implemented on GPU with the ARK.
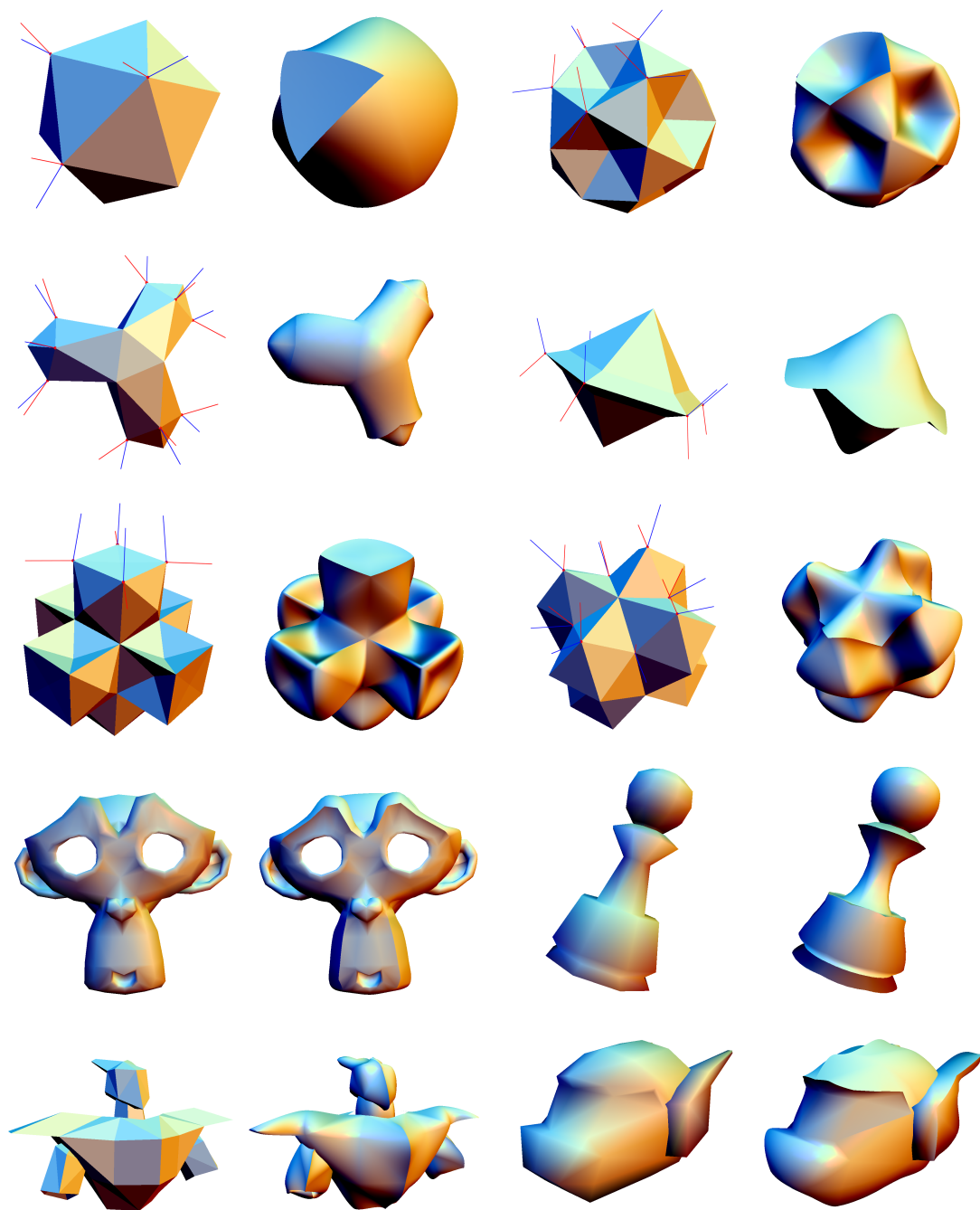
Figure 8.6: *Original meshes (left) and realtime refinement (right) expressing the scalar tag configuration.*

Chapter 9

Real-time Quadratic Approximation of Subdivision Surfaces

Subdivision surfaces are undoubtedly the most flexible smooth geometric representation. By only manipulating a carefully designed low-resolution mesh, an high-resolution smooth version is automatically generated using a set of local recursive rules applied on each input coarse polygon. However, while being intensively used in CAD and SFX industries, they have not yet gained a significant interest for interactive and real-time applications. In fact, their recursive definition imposes a non-trivial CPU overhead, difficult to hide in interactive applications. We propose to avoid this recursion by introducing an efficient approximation of subdivision surfaces which offers a very close appearance compared to the true subdivision surface, while providing at least one order of magnitude faster rendering. Our technique uses enriched polygons, equipped with edge vertices, and replaces them on-the-fly with low degree polynomials for interpolating positions and normals. By systematically projecting the vertices of input mesh at their limit position on the subdivision surface, the visual quality of the approximation is good enough for imposing only a single "true" subdivision step, allowing real-time performances even for million polygons output. Additionally, the parametric nature of the approximation allows an efficient adaptive sampling for polynomial adaptive rendering and displacement mapping.

9.1   Context: Subdivision Surfaces for interactive rendering

Subdivision Schemes: A *subdivision scheme* [ZS00] defines a smooth surface using a coarse mesh M^0 and a subdivision operator S, that combines various refinement rules (odd vertex, even vertex, border, crease, etc). For most of subdivision schemes such as Loop [Loo87] or Catmull-Clark [CC78], these rules are local, and only require the one-ring-neighborhood for subdividing each polygon of the coarse mesh, quickly converging to the limit surface. Thus, the application of the refinement rules is done recursively, generating a set of mesh $\{M^0, M^1, ..., M^n\}$ with $M^{k+1} = S(M^k)$ until the chosen depth n. The linear combination of neighboring vertices for computing the next position of a given vertex are usually illustrated with a subdivision mask. For stationary schemes, limit masks exist that directly provide the projection of a vertex on the limit surface.

Efficient Rendering of Subdivision Surfaces: Since a decade, subdivision surfaces have been intensively used for offline rendering and high end modeling, and have progressively replaced NURBS in many areas, as they are able to represent smooth shapes with arbitrary topology. With the increasing demand in realism for interactive applications, efficient rendering of subdivision surfaces has become

a major research area in recent years. However, the lack of geometry generation on GPU, as well as the reduced knowledge about local neighborhood allowed in the graphics hardware pipeline have led researchers to tackle efficient rendering of subdivision surfaces with two different approaches. We have already stated the main classification of refinement methods in Chapter 7, and according to this classification, direct real-time synthesis of subdivision surfaces is done with *precomputed tables of basis functions* [PS96, BS02, BS03], while indirect synthesis is done using *images-based methods* [SJP05, Bun05]. These methods are able to reproduce exactly subdivision surfaces but remains slow.

In Chapter 8, we have discussed how a *visually smooth* refinement can be tailored at low cost by using triangular Bézier patches locally generated on triangles. This kind of refinement is very efficient and purely local but its empirical generation only provides poor to average visual quality.

The method we propose in this chapter combines a low computationnal cost, even better than PN Triangles, with a visually plausible approximation of true subdivision, offering a very similar rendering quality compared to exact subdivision schemes [Loo87], far better than PN refinements. We use limit projections for driving a local polynomial approximation of the surface, which allows a direct evaluation at arbitrary location without recursion, in the spirit of the work done by Stam [Sta98, Sta99], but efficient enough to be done in real-time. In fact, by considering both positions and normals, we produce a visually smooth rendering adapted to interactive applications using simple quadratic Bézier patches, which makes adaptive sampling straightforward. We call our approximation "QAS" for "Quadratic patches for Approximation of Subdivision surfaces".

Figure 9.1: . **Left:** *Coarse mesh (546 triangles).* **Middle** *Our real-time GPU approximation of the subdivision surface (527 FPS - depth 5 - 500k triangles).* **Right:** *True Loop subdivision performed on CPU at same depth.*
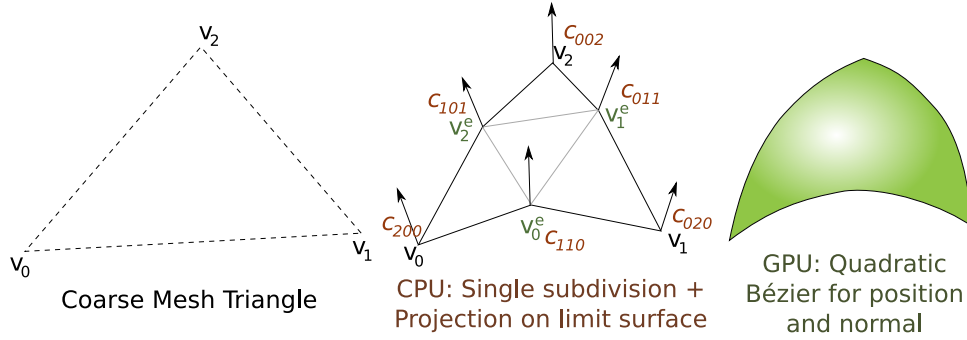
Figure 9.2: *Approximation principle.* **Left:** *Coarse triangle T of M^0.* **Middle:** *Enriched hexagon H^T sampled on M^∞.* **Right:** *Final smooth patches $\{P_T, N_T\}$ generated on GPU.*

9.2 Approximated Subdivision

9.2.1 Principle

The very first subdivision step provides a crucial information on the target smooth surface, particularly when using *limit rules*: it indicates in which direction the surface will converge for all its edges. By studying the different subdivision schemes developed over the years, we can observe that the variation they produce on edges is a good indicator of their smoothness and curvature quality. This information is even more accurate with limits masks (i.e. when projecting each vertex at its limit position).. We propose to use this initial guess of the first subdivision step performed on the CPU to compute a local quadratic Bézier approximation on the GPU. Instead of using an empirical estimation [VPBM01] of the Bézier coefficients for producing the *visual continuity*, we fit two Bézier patches on the limit positions (resp. normals) provided by the single subdivision step with projection on the limit surface M^∞ (see Figure 9.2). With these two patches in hand, we can sample (uniformly or adaptively) the piece of subdivision surface belonging to each input coarse triangles using either the *vertex shader* with our ARK (Chapter 7) or the *geometry shader* [Bly06].

9.2.2 CPU Support

The algorithm starts by applying a single subdivision step using limit masks. Each triangle T is thus split into 4 sub-triangles, with vertices on the limit surface. These sub-triangles share 6 vertices (Figure 9.2) and the sub-mesh can thus be organized in an hexagonal shape $H_T = \{v_0, v_1, v_2, v_0^e, v_1^e, v_2^e\}$ with $v_i = \{p_i, n_i\}$ being the limit positions and normals (using tangent masks for instance) at this location. This structure is adapted to recent graphics hardware including a geometry shader stage, which allows to transmit triangles with edge neighbors: here we transmit edge vertices inserted by the subdivision pass instead. Note that we focus on triangle meshes, since they are ubiquitous in interactive applications. Thus, we use the Loop scheme [Loo87] as a basis QAS. The Modified Butterfly scheme [ZSS96] can be used when the interpolation of the coarse mesh is mandatory. We perform this step on CPU in our implementation. However, a GPU implementation can be considered.

9.2.3 GPU Polynomial Approximation

Once H_T is transmitted to the GPU, a shader (either vertex shader on old devices or geometry shader on recent ones) automatically fits 2 *triangular Bézier patches* to H_T: $P_T(u,v)$ for positions and $N_T(u,v)$ for
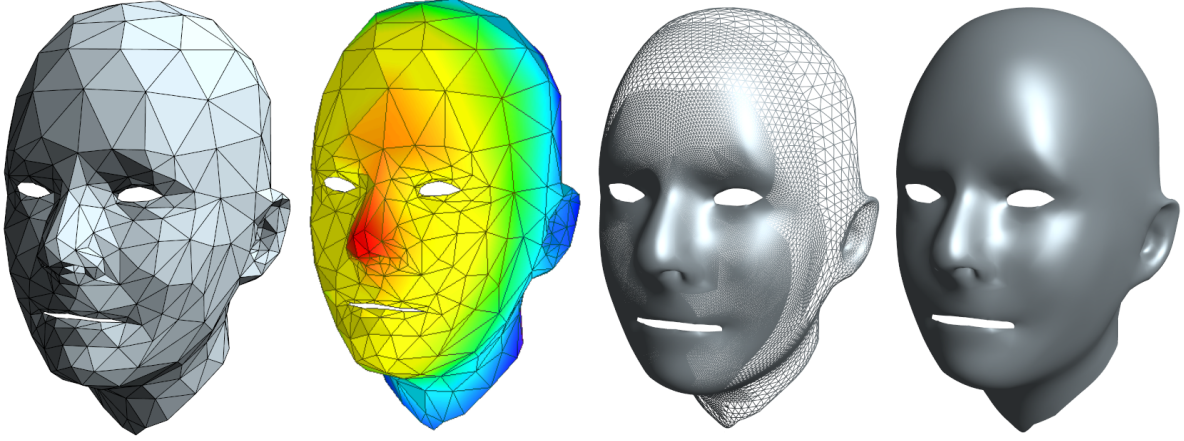
Figure 9.3: *Adaptive Subdivision Rendering.* **Left:** *Input coarse mesh (703 triangles).* **Middle Left:** *View-dependent distance-based adaptive depth tag.* **Middle Right:** *Underlying adaptive topology produced on-the-fly (620k triangles at 499 FPS)* **Right:** *Final rendering.*

normals. In other words, we produce a *procedural displacement map* and a *procedural normal map* that approximate the variation of the limit subdivision surface. Both patches are defined by:

$$Q(u,v,w) = \sum_{i+j+k=2} b_{ijk}^2(u,v,w)c_{ijk}$$

with

$$b_{ijk}^2 = \frac{2!}{i!j!k!}u^i v^j w^i \text{ and } w = 1-u-v$$

In practice, c_{ijk} is replaced by p_{ijk} or n_{ijk} (see Figure 9.2). We use quadratic patches as they provide a good trade-off between curvature reproduction and computational cost.

Now, we have to define the 6 control points required by both Bézier patches, such as they interpolate the limit vertices (either positions or normals). These control points are organized as an hexagon (middle part of Figure 9.2): three of them correspond to the original vertices $\{v_0, v_1, v_2\}$ projected at the limit and are naturally interpolated by the triangular Bézier patches at control points $\{c_{200}, c_{020}, c_{002}\}$, while $\{c_{110}, c_{011}, c_{101}\}$ correspond to edge vertices $\{v_0^e, v_1^e, v_2^e\}$ and are **not** interpolated. So, we need to define them such as the actual geometry defined by P_T (resp. N_T) interpolates the edge positions (resp. normals). Actually, a linear collocation is possible in this case. For instance, considering the first edge vertex p_0^e, we have to solve:

$$P\left(\frac{1}{2}, \frac{1}{2}, 0\right) = \frac{1}{4}(p_0 + p_1 + 2p_{110}) = p_0^e$$

which implies that

$$p_{110} = \frac{1}{2}(4p_0^e - p_0 - p_1)$$

Other edge coefficients are simply obtained in a similar fashion, and the same principle is used for computing the Bézier patch for normals. Separating the position field and normal field defined for each patch allows a local computation of the approximation (on a per-hexagon basis), without dealing with high order cross-edge continuity [VPBM01]. By interpolation, the normal field defined by N_T is guaranteed to be C^0 on edges, which produces a visually smooth shading.

139

9.2.4 Adaptive Rendering

By substituting recursive rules with Bézier patches, we can directly evaluate the surface approximation at arbitrary parameter values. So not only uniform tessellation is done without recursion, but adaptive refinement is also made easier. This adaptivity can be performed by setting a per-vertex subdivision depth, either on CPU or GPU, using for instance a view-dependent metric (e.g. coarse triangle to camera distance) or a view-independent one (e.g. curvature approximation). Then, adaptive tessellation can be obtained with either two implementations:

- **Geometry Shader**: H_T can be directly transmitted to the GS using the DX10 pipeline [Bly06]. A simple loop evaluates points and normals using P_T and N_T and output a stream of triangles. Unfortunately, this solution only holds for low subdivision depth, as the size of the GS output is hardware limited.
- **Vertex Shader**: For higher subdivision depth (3 and more), the adaptive refinement kernel introduced in Chapter 7 offers an efficient way to render our subdivision surface apprixation. Note that the transfer cost of H_T is not a bottleneck for deep subdivision levels.

Figure 9.3 gives an example of an adaptive on-the-fly QAS rendering. In the following listing, we provide a generic GPU implementation of QAS in GLSL for on-the-fly Bézier patch fitting and adaptive sampling. This simple shader runs on any GPU equipped with vertex shading capabilities:

```
const uniform vec3 n0, n1, n2, p0, p1, p2;
const uniform vec3 ne0, ne1, ne2, pe0, pe1, pe2;
vec3 edgeCP (vec3 e, vec3 p0, vec3 p1) {
  return (e * 4.0 - p0 - p1) * 0.5;
}
vec3 Q (float u, float v, float w,
        vec3 p0, vec3 p1, vec3 p2, vec3 e0, vec3 e1, vec3 e2) {
  vec3 n200 = p0, n020 = p1, n002 = p2;
  vec3 n110 = edgeCP (e0, p0, p1);
  vec3 n101 = edgeCP (e2, p0, p2);
  vec3 n011 = edgeCP (e1, p1, p2);
  return w * (n200*w + n110*2*u) +
         u * (n020*u + n011*2*v) +
         v * (n002*v + n101*2*w);
}
vec3 P (float u, float v, float w) {
  return Q (u, v, w, p0, p1, p2, pe0, pe2, pe2);
}
vec3 N (float u, float v, float w) {
  return Q (u, v, w, n0, n1, n2, ne0, ne2, ne2);
}
void main(void) {
  float u = gl_Vertex.x; // barycentric coordinates
  float v = gl_Vertex.y; // as position in the
  float w = 1.0 - u - v; // RP drawn
  gl_Vertex.xyz = P (u, v, w);
  gl_Normal = normalize (N (u, v, w));
  [...] // Shading
}
```

Property	Shiue's kernel	QAS
CPU Preprocess (Subdivision)	2 passes (1x16)	1 pass (1x4)
GPU Input process (CPU)	2-ring unfold	none
Number of Rendering pass	depth × num. of tri.	1
GPU Workload	FS	VS/GS
Reproduction	Exact	Approximate
Adaptive Refinement	Difficult	Trivial
Type of Coarse Polygons	All	All with pre-tessellation
Subdivision Scheme	All	Dyadic with limit masks
Performances (4k tri., depth 5)	Interactive	Real-Time

Table 9.1: *Comparison of QAS with Shiue's kernel [SJP05] for the subdivision of a dynamic mesh.*

9.3 Results

We have implemented QAS on an AMD Athlon 3500, with 2GB of memory and an nVidia Geforce 8800 GTX, using C++, OpenGL and GLSL. While being geometrically only C^0, the resulting surface has an appearance almost indistinguishable from the equivalent true subdivision surface (see Figure 9.4). This is due to the combined fitting of positions and normals, which ensures both a smooth shading and curved silhouettes. Considering performances, our technique outperforms existing solutions [BS03, SJP05] for three reasons: we only perform a single true subdivision pass on CPU, we use a single rendering pass on GPU whatever the depth (i.e., constant processing cost per-vertex) and there is no geometry-to-texture conversion. Note also that the mesh is always synthesized on-the-fly, either using Geometry or Refinement Shaders, without storing the topology of the high resolution mesh. As a result we obtain real-time performances (more than 120 FPS) for objects composed of several thousands of coarse polygons, subdivided at depth 5 (more than 2.5M tessellated triangles). Performances degrade linearly with the number of triangles created and transmitted at CPU level. As a limitation, note that the higher is the vertex valence, the less accurate becomes QAS. However, this can be prevented by performing remeshing. Last, the direct adaptive rendering allowed by our technique, combined with its low CPU overhead makes this approximation particularly suitable for high quality interactive applications, as it offers much better results than purely empirical smoothing methods. Figure 9.5 gives additional examples of our approximation: we can observe that high framerates can be reached even with deep refinement levels, since our pure parametric evaluation does not access texture memory.

9.4 Discussion

Comparison: We compare QAS to the GPU kernel of Shiue et al. [SJP05] as it is one of the best solution so far. Table 9.1 states advantages and weaknesses of our approach compared to their. One interesting property of our implementation is its single pass vertex shading principle: thus, recent graphics hardware with unified architecture will automatically allocate additional shader units for vertex shading to obtain optimal balance between vertex and fragment processing, avoiding the usual conversion required by fragment-based processing of geometry.

The local nature of our kernel makes it also easily comparable to Curved PN Triangles [VPBM01]. Formally, the two approaches differ in the computation of Bézier control points: an empirical estimation based on tangent plane for PN Triangles, and a true limit subdivision surface interpolation in our case. As a result, we obtain a far better quality since limit projection may create larger, smoother and more

consistent variation of the geometry that the simple normal-based approach (see Figure 9.4). This also allows us to simply use a quadratic polynomial instead of a cubic one.
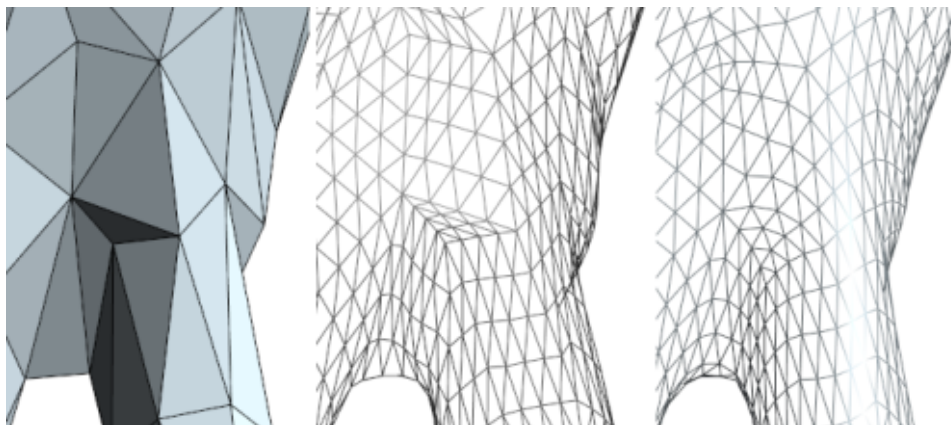

Figure 9.4: *Comparison with Curved PN Triangle Smoothing.* **Left:** *Coarse Mesh.* **Middle:** *Curved PN Triangles (cubic patches).* **Right:** *QAS (quadratic patches).*

Summary   We have proposed QAS, a simple and visually convincing approximation of subdivision surfaces using a combination of single limit subdivision pass on CPU and quadratic Bezier patch fitting on GPU. Our method is easy to implement, avoids recursion and reaches real-time performances for several thousands of input polygons per-frame, outputting millions of tessellated triangles. Our method is generic in the sense that arbitrary depth and arbitrary vertex valence can be handled and adaptively subdivided. This approximation imposes less CPU workload, less graphics bus bandwidth and is more efficient than exact GPU subdivision kernels, while providing better visual results than empirical smoothing [VPBM01] and lower memory footprint than table-based methods. While CAD applications may benefit from more precise and more costly approximation techniques such as the recent work of Loop and Schaefer [CS07], QAS represents a solid choice for interactive applications, such as video games and virtual reality software, and can also be considered for special effects, as a large upsampling can be done adaptively. As an application, high resolution displacement mapping takes benefit from this efficient approximation for sampling the maps that can be extracted directly from large point-based surface with the algorithm presented in Chapter 6.

Perspectives   As future work, we plane to perform the limit projection at GPU level still preserving a single pass rendering. It is interesting to note that we have solved a performance problem of subdivision schemes using local polynomial fitting, while Levin [Lev06] has recently solved a quality problem (continuity) of these schemes near extraordinary vertices using a similar approach of local polynomials fitting. We believe that such hybrid polynomial-subdivision representations are a major research direction in the field of smooth surface modeling.
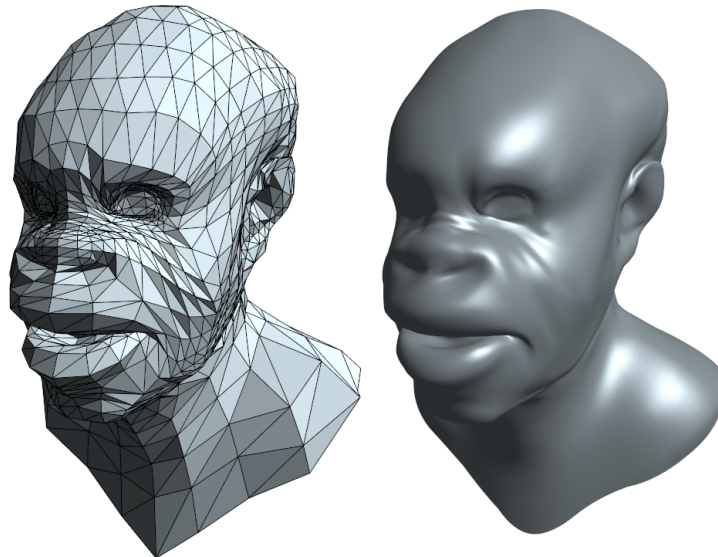
(a) 2720 coarse triangles refined at depth 5 (2.7M triangles) - 113FPS
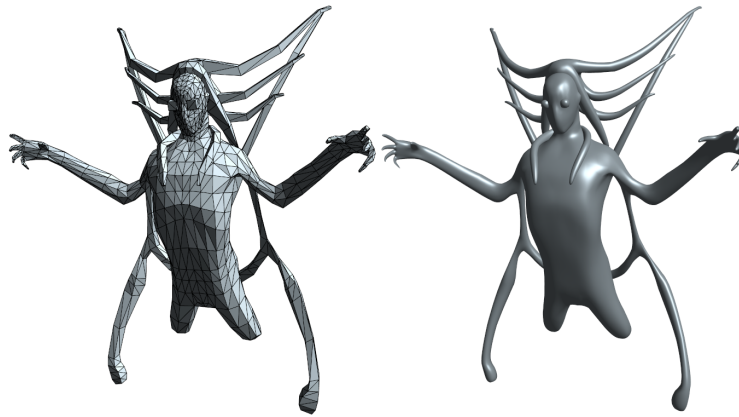
(b) 2516 coarse triangles refined at depth 5 (2.5M triangles) - 128FPS
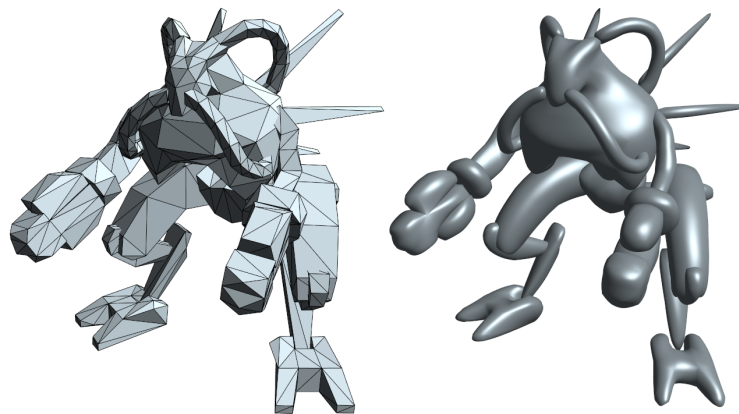
(c) 1246 coarse triangles adaptively refined at depth 6 (2.6M triangles) - 132FPS

Figure 9.5: *Additional examples of real-time approximation of subdivision surfaces.* **Left:** *On CPU dynamic coarse mesh.* **Right:** *Realtime QAS geometry synthesis on GPU. Note that all input meshes are dynamic.*

Chapter 10

Conclusion

In this thesis, we proposed techniques for fast processing, editing and rendering of acquired geometry. These techniques offer memory and computational efficient solutions to various problems that occur when using acquired geometry in computer graphics applications. They all build on a set of new concepts and data structures that are generic enough to be used in other contexts as well.

First, we have introduced the Volume-Surface Tree, and show that a hierarchical space partitioning structure can better take into account the geometry of a 3D surface by using an hybrid partitioning scheme, melting 3D and 2D decomposition. As hierarchical partitioning is a fundamental tool in geometry processing, we have then been able to apply this structure to the problems of fast surface simplification and fast surface reconstruction. The resulting algorithms increase the quality-over-speed ratio compared to state-of-the-art methods and clearly illustrate that not only the geometric measure and quantities, but also the underlying architecture of a partitioning structure has to be considered when performing hierarchy-based surface processing. Furthermore, the Volume-Surface Tree is general enough to be used in other surface processing, including texturing and compression.

Second, we have presented the first interactive editing system which allows to perform high level interactive modifications, such as appearance and shape editing, of large models with or without the connectivity information. This is obviously particularly interesting in the context of acquired geometries for which the accurate local features captured by the scanner need to be preserved, even when interactive editing is mandatory. This system is based on a sampling-reconstruction principle, where two algorithms ensure the "dialog" between the out-of-core large model and a given standard texturing or deformation tool. The first algorithm acts as pre-process and performs an out-of-core adaptive simplification of the large model. Then, the simplified model is deformed and textured interactively. The second algorithm acts as a post-process and applies the modification undergone by the simplified model onto the original large one. Both algorithms work in streaming and use point-based methods, which enables the manipulation of very large, unstructured sampled surfaces. Moreover, this system remains active during the interactive session and allows to refine the in-core geometry on-demand, providing virtually an out-of-core multiscale layer to any interactive editing tool. Note that the various experiments presented in this thesis show also that point sets are well suited to capture and transmit surface properties (appearance and deformation). Therefore, they represent a solid alternative to usual regular 2D textures, which becomes obvious in an out-of-core context.

Third, we have addressed the problem of point-based surface rendering, which is necessary for visualizing acquired models before reconstruction. Indeed, we have shown that the point-sampled surface representation can be interfaced at low cost with polygonal rendering systems and hardware, performing a fast lower dimensional meshing organized in a multi-resolution structure: the Surfel Stripping. This

approach enables the large repository of polygonal rendering techniques without performing full surface reconstruction, and comes as an alternative to surface splatting and point-based surface raytracing. Moreover, when the input is too large, we have proposed an appearance-preserving enhancement of our technique, which captures, in a streaming process, the essential part of the visually richness present in the large sampled surfaces, and expresses it in a set of high-resolution normal and color maps for rendering.

Last, we have focused on geometry synthesis for interactive applications. We have introduced a generic adaptive mesh refinement kernel which runs on the GPU and allows to refine arbitrary meshes, with arbitrary displacement, offering real-time high-resolution mesh synthesis. This kernel allows to deal directly with high-level representations, like subdivision domain meshes, on the CPU side, letting the GPU creating a refined surface in single rendering pass, at vertex shading level. With this kernel, we have then been able to propose an evolution of the Curved PN Triangle refinement, by taking into account surface features described by scalar tags. Finally, we have addressed the problem of real-time subdivision surface rendering, and proposed a plausible approximation that avoids recursion and enables deep adaptive refinement at high framerate.

This last contribution opens a way to data driven geometry processing. As shown all along this thesis, the acquired models need to trade efficiency for quality in order to use them in computer graphics applications. We have developed three kinds of analysis:

- a *hierarchical volume-surface decomposition* which improves hierarchical processing of 3D surfaces and allows lower dimensional methods

- a *sampling-reconstruction in streaming* which structures out-of-core editing

- a *synthesis by instancing* which offers flexible geometric refinement for real-time applications.

These three general approaches, combined with the genericity of point-based techniques, can be applied to various other problems, including surface compression, large object topology editing, aggressive visibility computation and data-driven geometry synthesis. This last topic represents probably the most promising research direction, as the growing size of 3D content data will soon impose the development of suitable higher-level representation that both allows highly dynamic structure for rich editing and accurate on-demand sampling for real-time geometry synthesis. Finally, we believe that one major open problem remains the gap that exists between raw data sampling and procedural representations: the former is the only information we can get from the real world, and the latter is the native language of computers. We will try to build the missing bridge in future work.

Bibliography

[AA03]   ADAMSON A., ALEXA M.: Ray tracing point set surfaces. In *Proceedings of Shape Modelling International* (2003). 65

[ABCO*01]   ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Point set surfaces. In *IEEE Visualization* (2001). 17, 38, 47, 65, 82

[ACK01]   AMENTA N., CHOI S., KOLLURI R. K.: The power crust. In *Symposium on Solid Modeling and Applications* (2001). 16

[Ado06]   ADOBE: Photoshop, 2006. 58

[AGP*04]   ALEXA M., GROSS M., PAULY M., PFISTER H., STAMMINGER M., ZWICKER M.: Point-based computer graphics. *ACM SIGGRAPH 2004 Course Notes* (2004). 42, 46, 63

[AH05]   ASIRVATHAM A., HOPPE H.: *GPU Gems 2*. Addison-Wesley, 2005, ch. Terrain rendering using GPU-based geometry clipmaps. 108, 120

[AK04a]   AMENTA N., KIL Y. J.: Defining point-set surfaces. *ACM SIGGRAPH* (2004). 17

[AK04b]   AMENTA N., KIL Y. J.: The domain of a point set surfaces. *Eurographics Symposium on Point-based Graphics* (2004). 17

[AKP*05]   ADAMS B., KEISER R., PAULY M., GUIBAS L. J., GROSS M., DUTRE P.: Efficient raytracing of deforming point-sampled surfaces. In *Proceedings of Eurographics* (2005). 65, 83

[Ali06]   ALIASWAVEFRONT: Maya, 2006. 41

[AWD*04]   ADAMS B., WICKE M., DUTR P., GROSS M., PAULY M., TESCHNER M.: Interactive 3d painting on point-sampled objects. In *Point-Based Graphics* (2004). 40

[BB83]   BARSKY B., BEATTY J.: Local control of bias and tension in beta-splines. *ACM SIGGRAPH* (1983). 130

[BC00]   BOISSONNAT J.-D., CAZALS F.: Smooth surface reconstruction via natural neighbour interpolation of distance functions. In *Symposium on Computational geometry* (2000). 16

[BD02]   BENSON D., DAVIS J.: Octree textures. In *ACM Siggraph* (2002). 40

[Bec94]   BECHMANN D.: Space deformation models survey. *Computer and Graphics* (1994). 41

[Ben75]   BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18* (1975). 25

[BK03]   BOTSCH M., KOBBELT L.: High-quality point-based rendering on modern GPUs. *Pacific Graphics 2003* (2003). 64

[BK04]   BOTSCH M., KOBBELT L.: An intuitive framework for real-time freeform modeling. *ACM SIGGRAPH* (2004). 41

[BK05]   BOTSCH M., KOBBELT L.: Real-time shape editing using radial basis functions. *Eurographics* (2005). 41, 57

[BKS00]   BISHOFF S., KOBBELT L., SEIDEL H.-P.: Towards hardware implementation of loop subdivision. *ACM SIGGRAPH//Eurographics Graphics Hardware* (2000). 108

[Blo88]   BLOOMENTHAL J.: Polygonization of implicit surfaces. *Computer Aided Geometric Design 5* (1988), 341–355. 16

[Blo94]   BLOOMENTHAL J.: An implicit surface polygonizer. In *Graphics Gems IV*. Academic Press, 1994. 26, 33

[Bly06]   BLYTHE D.: The direct3d 10 system. *ACM Siggraph* (2006). 107, 109, 123, 124, 138, 140

[BM92]   BESL P. J., MCKAY N. D.: A method for registration of 3-d shapes. *IEEE Trans. Pat. Anal. and Mach. Intel. 14*, 2 (1992), 239–256. 16

[BMZB01]   BIERMANN H., MARTIN I., ZORIN D., BERNARDINI F.: Sharp features on multiresolution subdivision surfaces. *Pacific Graphics* (2001). 109

[Bot05]   BOTSCH M.: *High Quality Surface Generation and Efficient Multiresolution Editing Based on Triangle Meshes*. PhD thesis, RWTH Aachen, 2005. 26

[BPGK06]   BOTSCH M., PAULY M., GROSS M., KOBBELT L.: PriMo: Coupled prisms for intuitive surface modeling. *Eurographics Symp. on Geom. Processing* (2006). 41

[BPK*07]   BOTSCH M., PAULY M., KOBBELT L., ALLIEZ P., LEVY B., BISCHOFF S., RSSL C.: Geometric modeling based on polygonal meshes. *ACM SIGGRAPH Course Notes* (2007). 17

[BR02]   BERNARDINI F., RUSHMEIER H.: The 3d model acquisition pipeline. *Computer Graphics Forum 21*, 2 (2002). 14

[BS95]   BLANC C., SCHLICK C.: X-splines: A spline model designed for the end-user. *ACM SIGGRAPH* (1995). 109

[BS02]   BOLZ J., SCHRODER P.: Rapid evaluation of catmull-clark subdivision surfaces. *3D Web Technology* (2002). 108, 137

[BS03]   BOLZ J., SCHRODER P.: Evaluation of subdivision surfaces on programmable graphics hardware, 2003. 108, 137, 141

[BS07]   BOTSCH M., SORKINE O.: On linear variational surface deformation methods. *IEEE TVCG* (2007). 41

[BSK04]   BOTSCH M., SPERNAT M., KOBBELT L.: Phong splatting. In *Symposium on Point Based Graphics 2004* (2004). 64, 67, 77

[BSK05]   BOTSCH M., SPERNAT M., KOBBELT L.: High quality splatting on today's gpu. In *Symposium on Point Based Graphics 2005* (2005). 64, 77, 78, 96

[Bun05]   BUNNELL M.: *Adaptive Tesselation of Subdivision Surfaces w/ Displacement Mapping*. nVidia, 2005, ch. GPU Gems 2. 109, 137

[BW06]   BOKELOH M., WAND M.: Hardware accelerated multi-resolution geometry synthesis. *ACM I3D* (2006). 108

[BWG03]   BALA K., WALTER B., GREENBERG D. P.: Combining edges and points for interactive high-quality rendering. *ACM SIGGRAPH 22*, 3 (2003), 631–640. 90

[BWK02]   BOTSCH M., WIRATANAYA A., KOBBELT L.: Efficient high quality rendering of point sampled geometry. In *Eurographics Workshop on Rendering* (2002). 85

[CAZ01]   COHEN J. D., ALIAGA D. G., ZHANG W.: Hybrid simplification: Combining multi-resolution polygon and point rendering. *IEEE Visualization* (2001). 64

[CB04]   CHRISTENSEN P. H., BATALI D.: An irradiance atlas for global illumination in complex production scenes. *Eurographics Symposium on Rendering* (2004). 41

[CBC*01]   CARR J. C., BEATSON R. K., CHERRIE J. B., MITCHELL T. J., FRIGHT W. R., MC-CALLUM B. C., EVANS T. R.: Reconstruction and representation of 3D objects with radial basis functions. In *ACM SIGGRAPH* (2001). 16

[CC78]   CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological surfaces. *Computer-Aided Design 10*, 6 (1978). 136

[CGG*04]   CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive TetraPuzzles – efficient out-of-core construction of gigantic polygonal models. *ACM SIGGRAPH 2004* (2004). 20, 85

[CH02]   COCONU L., HEGE H.-C.: Hardware-accelerated point-based rendering of complex scenes. *Eurographics Workshop on Rendering 2002* (2002). 64

[CK03a]   CHUNG K., KIM L.-S.: Adaptive tessellation of pn triangle with modified bresenham algorithm. *SOC Design Conference* (2003). 128

[CK03b]   CHUNG K., KIM L.-S.: A pn triangle generation unit for fast and simple tesselation hardware. *IEEE International Symposium on Circuits and Systems* (2003). 128

[CL96]   CURLESS B., LEVOY M.: A volumetric method for building complex models from range images. *ACM SIGGRAPH* (1996). 16

[CMRS98]   CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: A general method for preserving attribute values on simplified meshes. In *IEEE Visualization* (1998). 86

[CN01]   CHEN B., NGUYEN M. X.: POP: a Hybrid Point and Polygon Rendering System for Large Data. *IEEE Visualization 2001* (2001). 64

[COM98]   COHEN J., OLANO M., MANOCHA D.: Appearance-preserving simplfication. *ACM SIGGRAPH 98* (1998). 86

[Coq90]   COQUILLART S.: Extended freeform deformation: a sculpturing tool for 3D geometric modeling. *ACM SIGGRAPH* (1990). 41

[CS00]   CURLESS B., SEITZ S.: 3d photography. *ACM SIGGRAPH Course* (2000). 14

[CS07]   C.LOOP, SCHAEFER S.: *Approximating Catmull-Clark Subdivision Surfaces with Bicubic Patches*. Tech. rep., Microsoft Research MSR-TR-2007-44, 2007. 142

[CSAD04] COHEN-STEINER D., ALLIEZ P., DESBRUN M.: Variational shape approximation. In *ACM SIGGRAPH* (2004). 25, 28, 29, 45

[CSD02] COHEN-STEINER D., DA F.: *A Greedy Delaunay Based Surface Reconstruction Algorithm*. Tech. rep., INRIA Sophia Antipolis, 2002. 16, 70

[DD04] DUGUET F., DRETTAKIS G.: Flexible point-based rendering on mobile devices. *IEEE Computer Graphics and Applications 24*, 4 (2004). 85

[Dev98] DEVILLERS O.: Improved incremental randomized delaunay triangulation. In *ACM Symposium Computational Geometry 1998* (1998). 68

[DGH01] DEY T. K., GIESEN J., HUDSON J.: Delaunay based shape reconstruction from large data. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2001). 16

[DH02] DEY T. K., HUDSON J.: PMR: Point to mesh rendering, a feature-based approach. *IEEE Visualization* (2002). 64

[DQ01] DUAN Y., QIN H.: Intelligent balloon. In *ACM Symposium on Solid Modeling and Applications* (2001). 16

[dRBAB02] DEL RIO A., BOO M., AMOR M., BUGUERA J.: Hardware implementation of the subdivision loop algorithm. *ACM SIGGRAPH/Eurographics Graphics Hardware* (2002). 108

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. *ACM SIGGRAPH 2003* (2003). 57, 64, 74, 86

[DYQS04] DUAN Y., YANG L., QIN H., SAMARAS D.: Shape reconstruction from 3d and 2d data using pde-based deformable surfaces. In *ECCV* (2004). 34

[EDD*95] ECK M., DEROSE T., DUCHAMP T., HOPPE H., LOUNSBERY M., STUETZLE W.: Multiresolution analysis of arbitrary meshes. In *ACM SIGGRAPH* (1995). 26

[ESV96] EVANS F., SKIENA S. S., VARSHNEY A.: Optimizing triangle strips for fast rendering. *IEEE Visualization 1996* (1996). 66, 71, 115

[Far02] FARIN G.: *Curves and Surfaces for CAGD (Fifth Edition)*. Morgan Kaufman Inc., 2002. 130, 132

[FCOS05] FLEISHMAN S., COHEN-OR D., SILVA C.: Robust moving least-squares fitting with sharp features. *ACM SIGGRAPH* (2005). 17

[Fer05] FERNANDO R.: Shader model 3. nVidia, 2005. 119

[FKN80] FUCHS H., KEDES Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. In *ACM SIGGRAPH* (1980). 25

[Flo03] FLOATER M. S.: Mean value coordinates. *Comp. Aided Geom. Design 20*, 1 (2003). 49

[For87] FORTUNE S.: A sweepline algorithm for vorono diagrams. *Algorithmica 2* (1987), 153–174. 78

[GBK05] GUTHE M., BALZS ., KLEIN R.: Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Transactions on Graphics 24*, 3 (2005). 109

[GBK06] GUTHE M., BALZS ., KLEIN R.: Gpu-based appearance preserving trimmed nurbs rendering. *Journal of WSCG 14* (2006). 109

[GBP04] GUENNEBAUD G., BARTHE L., PAULIN M.: Deferred Splatting. *Eurographics* (2004). 64

[GBP05] GUENNEBAUD G., BARTHE L., PAULIN M.: Interpolatory refinement for real-time processing of point-based geometry. *Eurographics* (2005). 49

[GBP06] GUENNEBAUD G., BARTHE L., PAULIN M.: Splat-mesh blending, perspective rasterization and transparency for point-based rendering. In *Point-Based Graphics* (2006). 64

[GD98] GROSSMAN J. P., DALLY W. J.: Point sample rendering. *Eurographics Workshop on Rendering 1998* (1998). 63

[gDGPR02] (GRUE) DEBRY D., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. In *ACM Siggraph* (2002). 40, 57

[GG07] GUENNEBAUD G., GROSS M.: Algebraic point set surfaces. *ACM SIGGRAPH* (2007). 47

[GGSC96] GORTLER S., GRZESZCZUK R., SZELISKI R., COHEN M.: The lumigraph. In *ACM SIGGRAPH* (1996). 90

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *ACM SIGGRAPH* (1997). 25, 29, 31, 43

[GKS00] GOPI M., KRISHNAN S., SILVA C.: Surface reconstruction based on lower dimensional localized delaunay triangulation. In *Eurographics* (2000). 16, 27, 28, 35, 65

[GM04] GOBBETTI E., MARTON F.: Layered point clouds. In *Eurographics Symposium on Point Based Graphics* (2004). 85, 95

[GM05] GOBBETTI E., MARTON F.: Far voxels. *ACM SIGGRAPH* (2005). 57, 64, 85

[Goe04] GOESELE M.: *New Acquisition Techniques for Real Objects and Light Sources in Computer Graphics*. PhD thesis, Max-Planck-Institut fr Informatik (MPII), Saarbrcken, Germany, 2004. 15, 40

[GP03] GUENNEBAUD G., PAULIN M.: Efficient Screen Space Approach for Hardware Accelerated Surfel Rendering. *Vision, Modeling and Visualization* (2003). 64

[GPG06] GPGPU: General-purpose computation using graphics hardware. http://www.gpgpu.org, 2006. 107

[Gre06] GREEN S.: Next generation games with direct3d 10. *Game Developer Conference* (2006). 124

[Gro06] GROSS M.: Getting to the point. *IEEE Computer Graphics and Applications 26*, 5 (2006). 16

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *ACM SIGGRAPH* (1992). 16, 17, 25, 27, 28, 32, 45

[HDD*93] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Mesh optimization. In *ACM SIGGRAPH* (1993). 25

[HDD*94] HOPPE H., DEROSE T., DUCHAMP T., HALSTEAD M., JIN H., MCDONALD J., SCHWEITZER J., STUETZLE W.: Piecewise smooth surface reconstruction. In *ACM SIGGRAPH* (1994). 16, 25

[Hec86a] HECKBERT P.: Survey of texture mapping. *IEEE Computer Graphics and Applications* (1986). 40

[Hec86b] HECKBERT P. S.: Survey of texture mapping. *IEEE Computer Graphics and Applications 6*, 11 (1986). 64

[Hei05] HEIDRICH W.: Computing the barycentric coordinates of a projected point. *Journal of Graphics Tools 10*, 3 (2005). 49

[HH90] HANRAHAN P., HAEBERLI P.: Direct wysiwyg painting and texturing on 3d shapes. In *ACM Siggraph* (1990). 40

[Hop96] HOPPE H.: Progressive meshes. *ACM SIGGRAPH* (1996). 86, 109

[HSRG07] HAN C., SUN B., RAMAMOORTHI R., GRINSPUN E.: Frequency domain normal map filtering. In *ACM SIGGRAPH* (2007). 89, 91

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. *IEEE Visualization* (2005). 42

[ILSS06] ISENBURG M., LIU Y., SHEWCHUK J., SNOEYINK J.: Streaming computation of delaunay triangulations. *ACM SIGGRAPH* (2006). 42, 44

[Jen96] JENSEN H. W.: Global illumination using photon maps. *Rendering Techniques* (1996). 40

[JK02] JEONG W., KIM C.: Direct reconstruction of displaced subdivision surface from unorganized points, 2002. 16

[JT80] JACKINS C., TANIMOTO S.: Oct-trees and their use in representing three-dimensional objects. *CGIP 14* (1980). 25

[JZH07] JU T., ZHOU Q.-Y., HU S.-M.: Editing the topology of 3d models by sketching. *ACM SIGGRAPH* (2007). 58

[KAG*05] KEISER R., ADAMS B., GASSER D., BAZZI P., DUTR P., GROSS M.: A unified lagrangian approach to solid-fluid animation. In *IEEE/Eurographics Symposium on Point-Based Graphics* (2005). 83

[Kaz05] KAZHDAN M.: Reconstruction of solid models from oriented point sets. In *Symposium on Geometry Processing* (2005). 16

[KB04] KOBBELT L., BOTSCH M.: A survey of point-based techniques in computer graphics. *Computers and Graphics, v28, n6* (2004). 63

[KBH06] KAZHDAN M., BOLITHO M., , HOPPE H.: Poisson surface reconstruction. In *Symposium on Geometry Processing* (2006). 16

[KBR04] KESSENICH J., BALDWIN D., ROST R.: The opengl shading language. http://www.opengl.org, 2004. 117

[KHS03] KÄHLER K., HABER J., SEIDEL H.-P.: Dynamically refining animated triangle meshes for rendering. *The Visual Computer* (2003). 109

[KKDH07] KAZHDAN M., KLEIN A., DALAL K., HOPPE H.: Unconstrained isosurface extraction on arbitrary octrees. In *Symposium on Geometry Processing* (2007). 33

[Kob00] KOBBELT L.: Sqrt(3) subdivision. *ACM SIGGRAPH* (2000). 109, 114

[KS06] KRAEVOY V., SHEFFER A.: Mean-value geometry encoding. *International Journal of Shape Modeling 12*, 1 (2006). 48, 49

[KSW05] KRÜGER J., SCHNEIDER J., WESTERMANN R.: Duodecim - a structure for point scan compression and rendering. In *Eurographics Symposium on Point Based Graphics* (2005). 85, 96

[KV03] KALAIAH A., VARSHNEY A.: Modeling and rendering of points with local geometry. *IEEE Trans. Visualization 9*, 1 (2003). 63

[KZB03] KRIVANEK J., ZARA J., BOUATOUCH K.: Fast depth of field rendering with surface splatting. *Computer Graphics International 2003* (2003). 64

[LC87] LORENSEN W., CLINE H.: Marching cubes : a high resolution 3d surfaceconstruction algorithm. *Computer Graphics 21* (1987). 16

[LC03] LARSEN B. D., CHRISTENSEN N. J.: Real-time terrain rendering using smooth hardware optimized level of detail. *Journal of WSCG* (2003). 120

[Lev98a] LEVIN D.: The approximation power of moving least square. *Mathematics of Computation 67*, 224 (1998). 17

[Lev98b] LEVIN D.: Mesh-independent surface interpolation. *Geometric Modeling for Scientific Visualization* (1998). 17

[Lev06] LEVIN A.: Modified subdivision surfaces with continuous curvature. *ACM SIGGRAPH* (2006). 142

[LH04] LOSASSO F., HOPPE. H.: Geometry clipmaps: Terrain rendering using nested regular grids. *ACM SIGGRAPH* (2004). 41

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: *GPU Gem's 2: Octree Textures on the GPU*. 2005. 40, 57

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *ACM SIGGRAPH* (2000). 20, 25, 43, 87

[Lin03] LINDSTROM P.: Out-of-core construction and visualization of multiresolution surfaces. In *Symposium on Interactive 3D graphics* (2003), pp. 93–102. 85

[LKS*06] LEFOHN A. E., KNISS J., STRZODKA R., SENGUPTA S., OWENS J. D.: Glift: Generic, efficient, random-access gpu data structure. *ACM Transaction on Graphics* (2006). 57

[LMH00] LEE A., MORETON H., HOPPE H.: Displaced subdivision surfaces. *ACM SIGGRAPH* (2000). 41, 109, 119

[Loo87] LOOP C.: *Smooth subdivisions surfaces based on triangles*. Master's thesis, Department of Mathematica, University of Utah, August 1987. 33, 136, 137, 138

[LP03] LINSEN L., PRAUTZSCH H.: Fan clouds - an alternative to meshes. *Workshop on Theoretical Foundations of Computer Vision* (2003). 49, 65, 82

[LPC*00]   LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital michelangelo project : 3d scanning of large statues. In *ACM SIGGRAPH* (2000). 20, 64, 84

[LSLCO05] LIPMAN Y., SORKINE O., LEVIN D., COHEN-OR D.: Linear rotation-invariant coordinates for meshes. *ACM SIGGRAPH* (2005). 48, 49

[LSS*98]   LEE A. W. F., SWELDENS W., SCHROEDER P., COWSAR L., DOBKIN D.: MAPS: Multiresolution adaptive parameterization of surfaces. In *ACM SIGGRAPH* (1998). 26

[LW85]   LEVOY M., WHITTED T.: The use of points as display primitive. *TR 82-022, Univ. of North Carolina at Chapel Hill* (1985). 63

[MAMS06]   MUNKBERG J., AKENINE-MOLLER T., STROM J.: High quality normal map compression. In *EUROGRAPHICS/SIGGRAPH Graphics Hardware* (2006). 86, 90

[MJ96]   MACCRACKEN R., JOY K. I.: Free-form deformations with lattices of arbitrary topology. *ACM SIGGRAPH* (1996). 41

[MK04]   MARINOV M., KOBBELT L.: Optimization techniques for approximation with subdivision surfaces. In *ACM Symposium on Solid Modeling and Applications* (2004). 26, 34

[MOSAM07] MUNKBERG J., OLSSON O., STROM J., AKENINE-MOLLER T.: Tight frame normal map compression. In *EUROGRAPHICS/SIGGRAPH Graphics Hardware* (2007). 86, 90

[NRDR05]   NEHAB D., RUSINKIEWICZ S., DAVIS J., RAMAMOORTHI R.: Efficiently combining positions and normals for precise 3d geometry. In *ACM SIGGRAPH* (2005). 17, 32

[OBA*03]   OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.: Multi-level partition of unity implicits. In *ACM SIGGRAPH* (2003). 16, 26, 32, 35, 36

[PG01]   PAULY M., GROSS M.: Spectral processing of point-sampled geometry. In *ACM SIGGRAPH* (2001). 28, 32, 67, 90, 91

[PGB03]   PEREZ P., GANGNET M., BLAKE A.: Poisson image editing. *ACM SIGGRAPH 22*, 3 (2003), 313–318. 90

[PGK02]   PAULY M., GROSS M., KOBBELT L.: Efficient simplification of point-sampled surfaces. In *IEEE Visualization* (2002). 25, 32, 46, 74, 95

[Pha06]   PHARR M.: Interactive rendering in the post-gpu era. *Keynote at the 2006 Eurographics/SIGGRAPH Conference on Graphics Hardware* (September 2006). 125

[Pix06]   PIXOLOGIC: Z brush, 2006. 41

[PKKG03]   PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM SIGGRAPH* (2003). 51, 52, 54, 75, 76

[PS96]   PULLI K., SEGAL M.: Fast rendering of subdivision surfaces. *Eurographics Workshop on Rendering* (1996). 108, 137

[Pul99]   PULLI K.: Multiview registration for large data sets. In *Proceedings of 3DIM* (1999). 16

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: Surface elements as rendering primitives. *ACM SIGGRAPH* (2000). 40, 63, 64

[RB93]   ROSSIGNAC J., BORREL P.: Multi-resolution 3d approximation for rendering complex scenes. *Modeling in Computer Graphics* (1993). 25, 43, 87

[RBA05]   REUTER P., BEHR J., ALEXA M.: An improved adjacency data structure for fast triangle stripping. *Journal of Graphics Tools 10*, 2 (2005). 71, 115

[Rig06]   RIGHTHEMISPHERE: Deep paint 3d, 2006. 41

[RJT*05]   REUTER P., JOYOT P., TRUNZLER J., BOUBEKEUR T., SCHLICK C.: Surface reconstruction with enriched reproducing kernel particle approximation. In *IEEE/Eurographics Symposium on Point-Based Graphics* (2005), pp. 79–87. 17

[RL00]   RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. *ACM SIGGRAPH 2000* (2000). 20, 57, 64, 73, 74, 85, 95

[RPZ02]   REN L., PFISTER H., ZWICKER M.: Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. *Eurographics* (2002). 64

[Rus04]   RUSINKIEWICZ S.: Estimating curvatures and their derivatives on triangle meshes. *Symposium on 3D Data Processing, Visualization, and Transmission* (2004). 111

[Sam89]   SAMET H.: *Quadtree, Octrees, and Other Hierarchical Methods*. Addison Wesley, 1989. 25

[SFS05]   SCHEIDEGGER C. E., FLEISHMAN S., SILVA C. T.: Triangulating point set surfaces with bounded error. In *Eurographics Symposium on Geometry Processing* (2005). 32

[SG01]   SHAFFER E., GARLAND M.: Efficient adaptive simplification of messive meshes. In *IEEE Visualization* (2001). 25

[SJ00]   SCHAUFLER G., JENSEN H. W.: Ray tracing point sampled geometry. *Rendering Techniques* (2000). 65

[SJP05]   SHIUE L.-J., JONES I., PETERS J.: A realtime gpu subdivision kernel. *ACM Siggraph* (2005). 109, 112, 137, 141

[SJW07]   SCHAEFER S., JU T., WARREN J.: Manifold dual contouring. *IEEE Transactions on Visualization and Computer Graphics* (2007). 33

[SLCO*04] SORKINE O., LIPMAN Y., COHEN-OR D., ALEXA M., RÖSSL C., SEIDEL H.-P.: Laplacian surface editing. *Symp. on Geometry Processing* (2004). 41, 49

[Slo06]   SLOAN P.-P.: Direct3d 10 and beyond. *Keynote at the 2006 Eurographics/SIGGRAPH Conference on Graphics Hardware* (September 2006). 125

[SLS*06]   SHARF A., LEWINER T., SHAMIR A., KOBBELT L., COHEN-OR D.: Competing fronts for coarse tofine surface reconstruction. In *EUROGRAPHICS* (2006). 16

[Sor06]   SORKINE O.: Differential representations for mesh processing. *Computer Graphics Forum 25*, 4 (2006). 41

[SP86]   SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. *ACM SIGGRAPH* (1986). 41

[SPOK95]   SAVCHENKO V. V., PASKO A. A., OKUNEV O. G., KUNII T. L.: Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum 14*, 4 (1995). 16

[SSGH01] SANDER P. V., SNYDER J., GORTLER S. J., HOPPE H.: Texture mapping progressive meshes. In *ACM SIGGRAPH '01* (2001). 86

[Sta98] STAM J.: Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. ACM SIGGRAPH, 1998. 137

[Sta99] STAM J.: Evaluation of loop subdivision surfaces. ACM SIGGRAPH Course Notes, 1999. 125, 137

[STKK99] SUZUKI H., TAKEUCHI S., KIMURA F., KANAI T.: Subdivision surface fitting to a range of points. In *Pacific Graphics* (1999). 16

[SW03] SCHAEFER S., WARREN J.: Adaptive vertex clustering using octrees. In *Geometric Design and Computing* (2003). 25, 43, 57

[SWND05] SHREINER D., WOO M., NEIDER J., DAVIS T.: *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2*. Addison-Wesley Professional, 2005. 122

[TBH06] TRIFONOV B., BRADLEY D., HEIDRICH W.: Tomographic reconstruction of transparent objects. In *Eurographics Symposium on Rendering* (2006). 15

[TCS03] TARINI M., CIGNONI P., SCOPIGNO R.: Visibility based methods and assessment for detail-recovery. In *IEEE Visualization* (2003). 86

[TM91] TERZOPOULOS D., METAXAS D.: Dynamic 3d models with local and global deformations: Deformable superquadrics. *IEEE Trans. Pattern Anal. Mach. Intell. 13*, 7 (1991). 16

[TO02] TURK G., O'BRIEN J. F.: Modeling with implicit surfaces that interpolate. vol. 21. 16, 46

[Tol99] TOLEDO S.: A survey of out-of-core algorithms in numerical linear algebra. *External memory algorithms* (1999), 161–179. 85

[TRS04] TOBOR I., REUTER P., SCHLICK C.: Multiresolution reconstruction of implicit surfaces with attributes from large unorganized point sets. In *Shape Modeling International* (2004). 16, 26, 36

[Tur91] TURK G.: Generating textures on arbitrary surfaces using reaction-diffusion. In *ACM Siggraph* (1991). 40, 46

[VK03] VANECCEK P., KOLINGEROVA; I.: Fast delaunay stripification. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics* (2003). 71

[VPBM01] VLACHOS A., PETERS J., BOYD C., MITCHELL J. L.: Curved pn triangles. In *ACM I3D* (2001). 33, 108, 117, 118, 119, 128, 130, 131, 132, 138, 139, 141, 142

[Wal05] WALD I.: Interactive ray tracing of point based models. In *Proceedings of the Eurographics Symposium on Point Based Graphics* (2005). 65

[WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of the Eurographics Symposium on Rendering* (2004). 85

[WHA*07] WEYRICH T., HEINZLE S., AILA T., FASNACHT D. B., OETIKER S., BOTSCH M., FLAIG C., MALL S., ROHRER K., FELBER N., KAESLIN H., GROSS M.: A hardware

155

architecture for surface splatting. *ACM Transactions on Graphics (Proc. SIGGRAPH) 26*, 3 (2007). 64

[WOG05] WICKE M., OLIBET S., GROSS M.: Conversion of point-sampled models to textured meshes. In *Symposium on Point-Based Graphics* (2005). 65

[WPK*04] WEYRICH T., PAULY M., KEISER R., HEINZLE S., SCANDELLA S., GROSS M.: Post-processing of scanned 3d surface data. *Eurographics Symposium on Point-Based Graphics* (2004). 16

[WS04] WARREN J., SCHAEFER S.: A factored approach to subdivision surfaces. *Computer Graphics and Applications 24*, 3 (2004). 109

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points. In *Symposium on Point-Based Graphics 2006* (2006). 86

[XP98] XU C., PRINCE J. L.: Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing 7*, 3 (1998). 90

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-lods: Fast lod-based ray tracing of massive models the visual computer. In *Pacific Graphics* (2006). 85

[YZ06] YANCI ZHANG R. P.: Single-pass point rendering and transparent shading. In *Point-Based Graphics* (2006). 64

[YZX*04] YU Y., ZHOU K., XU D., SHI X., BAO H., GUO B., SHUM H.-Y.: Mesh editing with poisson-based gradient field manipulation. *ACM SIGGRAPH* (2004). 41

[ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3d: an interactive system for point-based surface editing. *ACM SIGGRAPH 2002* (2002). 40, 52, 54

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: Surface splatting. *ACM SIGGRAPH* (2001). 64, 67

[ZPvBG02] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: EWA splatting. *IEEE Trans. Visualization 8*, 3 (2002). 77

[ZRB*04] ZWICKER M., RASSNEN J., BOTSCH M., DACHSBACHER C., PAULY M.: Perspective accurate splatting. *Graphics Interface 2004* (2004). 64

[ZS00] ZORIN D., SCHRODER P.: Subdivision for modeling and animation. *ACM SIGGRAPH Courses Notes* (2000). 41, 108, 128, 134, 136

[ZSS96] ZORIN D., SCHROEDER P., SWELDENS W.: Interpolating subdivision for meshes with arbitrary topology. *ACM SIGGRAPH* (1996). 138

[ZSS97] ZORIN D., SCHROEDER P., SWELDENS W.: Interactive multiresolution mesh editing. *ACM SIGGRAPH* (1997). 26, 34, 41

Annex: Scalability and Antialiasing of Point-Sampled Textures

Texture Antialiasing   The up-scaling of the PST does not exhibit artifacts thanks to the smooth filtering provided by the kernel function. However, in the case of ray-tracing, when the texture is directly used for evaluating the color of a pixel, the down-scaling of the PST may lead to aliasing. Using cone tracing instead of ray tracing is a common (but expensive) solution to prevent such aliasing. In our case, cone intersection can be speeded up by replacing the evaluation of $f_S(p)$ by the average of the samples falling in the sphere Φ, centered at the intersection point. The diameter of Φ is chosen as the object-space size of the pixel at the intersection point. This special evaluation is performed as soon as more than one ray sample intersects Φ.

Full Scalability   At any time, if the in-core model becomes itself too large for maintaining an interactive frame-rate, a down-sampling is performed, again on a per-1-node basis, by replacing the *Least Recently Used* (LRU) area by a unique sample, and storing the edited piece of surface on the disk. Latter, if the user comes back to this part of the object, the area is reloaded, and an LRU down-sampling is again performed until reaching interactivity. In practice, it can be useful to maintain a ring of 1-nodes at current resolution around the currently edited piece of surface (i.e. 1-neighborhood safe, whatever the LRU selection). This simple LRU down-sampling rule makes the PST itself *scalable*.

About models and software

We would like to thank the following organizations for providing the various models used to test the algorithms of this thesis:

- Stanford University and the Digital Michelangelo Project

- Aim@Shape Network

- Cyberware

- Ausonius

- EDF and L'Ecole de France D'Athénes.

We also would like to thanks the following organizations for their free 3D tools:

- Computer Graphics Group of ETH Zurich for PointShop 3D

- Blender Foundation for Blender 3D

- Stanford University for QSplat

All along this thesis we have developed 3 software tools:

- OSIRIS for processing and rendering sampled surfaces

- SIMOD for size-insensitive editing

- ARK/GLRK for real-time adaptive refinement.

Translation in French

Introduction et Contexte

La *modélisation numérique* représente les objets et phénomènes du monde réelle par une suite de valeurs numériques dérivant leurs propriétés. Parmis ces propriétés, la *forme* a une importance fondamentale dans toutes les applications mettant en oeuvre une simulation, la plus simple étant la synthèse d'une image capturant une approximation de l'éclairage subit par l'objet, en d'autres mots : le *rendu*. En amont de ce procédé, la *modélisation géométrique* d'un objet en 3 dimensions utilise une grande variété de fonctions pour décrire cette forme : elle sont structurées par des relations spatiales, spectrales ou sémantiques, et diffèrent selon l'application, les contraintes de temps de calcul et de mémoire, le niveau de précision souhaité, voire même les règles artistiques imposées.

Le champ d'application de la modélisation géométrique et de la synthèse d'images s'étends de la simulation aux applications de divertiseement, en passant par la conception, les effets spéciaux, l'étude archéologique, les jeux vidéo, l'apprentissage et l'animation. Avec le dévelopement récent des technologies numériques, motivé en grande partie par l'essor d'Internet, toutes ces applications doivent désormais affronter une demande grandissante sur une plage de temps réduite, imposant de complexes systèmes multimédia, tels que par exemple les *modeleurs CAO*, les *moteurs des rendu temps-réel* ou bien encore les *simulateurs de vol*. Cependant, bien que de nombreuses technologies existent aujourd'hui pour gérer ces opérations, un cruel manque demeure concernant la création de ce qui compte réellement : le contenu.

Depuis des décennies, les modèles 3D sont créés par des inforgraphistes, utilisants de complexes outils informatiques afin de reproduire les objets du monde réel et d'en inventer de nouveaux. Bien que certains domaines, en particuliers dans l'industrie du divertissement, tirent partie de leur compétences artistiques, les infographiste ne peuvent répondre rapidement "à la main" aux demandes de modélisation précise de surfaces réelles.

Récemment, une nouvelle technique de modélisation a émergé: la *modélisation automatique*, ou comment générer une surface 3D à l'aide d'un scanner, exactement comme l'on génére une image avec un appareil photo. A l'aide de ces nouvelles machines, générer des millions de polygons échantillonnant un visage humain ne demande que quelques secondes et générer la géométrie d'un batiment à une précision infra-millimétrique se fait en quelques heures.

Malheureusement, cette nouvelle source de données amène de nouveaux problèmes, liés à deux caractéristiques essentielles de l'acquisition:

- l'acquisition est un processus discret qui ne produit qu'un ensemble d'échantillons. Ainsi, la notion de surface, intrinsèquement continue, doit être reconstruite, induisant un ensemble de décisions plus ou moins arbitraires pour établir la connexion entre les échantillons ;

- la haute précision des feuilles de scan implique des masses de données non triviales, difficiles à manipuler même sur les machines les plus puissantes, et particulièrement lorsque un comportement intéractif est nécessaire (traitement, édition et synthèse).

Dans cette thèse, nous proposons de nouvelles structures de données et de nouveaux algorithmes pour les grands objets, architecturés pour être efficicace en temps et en espace, et capables d'appréhender les formes complexes provenant directement du pipeline d'acquisition 3D.

Descriptions des Contributions

Tout au long de ce travail de recherche, nous nous sommes attaqué à divers problèmes liés au traitement géométrique et à la synthèse à partir de géométries numérisées. Nous présentons plusieurs contribtuions originales dans les domaines du traitement rapide, de l'édition intéractive et du rendu de surface échantillonnées. Voici la liste des principales contributions:

Traitement

- une nouvelle structure hiérarchique de partitionnement spatial, l'Arbre Volume-Surface, se substituant avantageusement à l'octree pour un partitionnement efficace, et offrant une meilleur découpe par l'erreur.

- un nouvel algorithme de simplification rapide de surfaces basé sur l'Arbre Volume-Surface ;

- un nouvel algorithme de reconstruction rapide de surfaces basé sur l'Arbre Volume-Surface ;

- un noyau générique pour la simplification hors-mémoire.

Edition

- un système insensible à la taille pour l'édition intéractive de grands objets;

- deux algorithmes en flux pour le transfert de l'apparence et de la deformation entre différents échantillonnages d'un même objet.

Synthèse

- une structure multi-résolution offrant un rendu polygonal de surfaces de points ;

- un enrichissement de cette structure, préservant l'apparence des grands objets à l'aide de textures de normales ;

- un noyau générique pour le raffinement de maillages en temps-réel avec un déplacement géométrique arbitraire ;

- un contrôle des singularités de surfaces dans le raffinement ;

- une approximation temps-réel des surfaces de subdivision.

Organisation

Parmis les divers sujets abordés dans cette thèse, une grande partie des travaux précédents sont discutés en préambule. Afin de préserver une certaine clarté, nous ne présentons pas l'ensembles des travaux précédents en un seul chapitre, et les répartissons, selon le contexte, aux débuts des différents chapitres.

Ce manuscrit est organisé en 3 parties :

Part I : nouveaux traitement géométriques et méthodes d'éditions pour les grands objets.

Chapter 3 : introduction de l'Arbre Volume-Surface et de ses applications à la simplification et à la reconstruction de surfaces.

Chapter 4 : description d'un système insensible à la taille pour l'édition intéractive de grands objets.

Part II : une nouvelle structure multirésolution pour le rendu polygonal de surfaces de points.

Chapter 5 : génération et rendu en-mémoire de la structure à partir d'un nuage de points.

Chapter 6 : enrichissement hors-mémoire de cette structure pour la préservation de l'apparence des grands objets.

Part III : synthèse géométrique temps-réel par raffinement pour les applications intéractives.

Chapter 7 : description d'un nouveau noyau pour le raffinement adaptatif et le displacement en une seule passe de rendu.

Chapter 8 : contrôle des singularités de surface pour la représentation des arêtes vives, de la tension et du biais au cours du raffinement.

Chapter 9 : approximation des surfaces de subdivision pour la synthèse géométrique temps-réel.

Chaque chapitre débute par une description du contexte et de l'état de l'art du sous-domaine relatif. Chaque contribution est systématique accompagnée d'une discussion sur les résultats, les performances, les limitations et les perspectives d'avancées liées au domaine.

Résultats et Conclusion

Au cours de cette thèse, nous avons proposé de nouvelles techniques pour le traitement rapide, l'édition et le rendu de géométrie numérisées. Ces techniques offrent des solutions efficaces en mémoire et en temps à de nombreux problèmes apparaissant lors de l'acquisition de surfaces pour les applications graphiques. Elles sont toutes construites sur un ensembles de nouveaux concepts et structures de données qui sont suffisamment génériques pour être utilisées dans d'autres contextes.

Premièrement, nous avons introduit l'Arbre Volume-Surface en montrant qu'une structure hiérarchique de partitionnement spatiale peut mieux prendre en compte la géométrie d'une surface 3D en utilisant un schéma hybride de partitionnement, mélangeant décomposition 3D et décomposition 2D. Le partitionnement hierarchique étant au coeur de nombreux algorithmes de traitement géométrique, nous avons ensuite appliqué cette structure aux problèmes de la simplification de surface et de la reconstruction de surface. Les algorithmes obtenus améliorent le compromis temps-qualité de l'état de l'art et illustrent clairement le fait que les quantités et mesures géométriques ne sont pas seules en charge de la qualité du résultat d'un traitement, et que la structure utilisée pour localiser les calculs a également une grande influence.

Deuxièmement, nous avons présenter le premier système d'édition intéractive pour grands objets. Ce système autorise une manipulation de haut niveau de l'apparence et de la formes de grandes surfaces, sans nécessiter d'information topologique sur les modèles. Bien entendu, ce système trouve sa première utilité dans le cadre des objets scannés, pour lesquels on souhaite préserver toute la richesse de l'information géométrique. Ce système est basé sur un principe d'*échantillonnage-reconstruction*, où deux algorithmes en flux assurent le "dialogue" entre un programme et les masses de données en stockage externe. Le premier algorithme effectue une simplification adaptive en flux. L'apparence et la forme du modèle simplifié peuvent ensuite être éditées. Enfin, le second algorithme transfert ces modifications au grand objet original. Les deux algorithmes fonctionnent en flux sont basés-points, ce qui permet de manipuler de grandes masses de données non structurées. De plus, le système reste actif pendant l'édition, offrant la possibilité de raffiner à la demande la géométrie manipulée à partir du grands modèles, et fournissant un système multi-échelles hors-mémoire générique. On notera que la représentation simplifiée, basée-points a offert toute la flexibilité requise en terme d'édition durant les nombreuses expérimentations menées : les modèles de points prouvent ainsi leur supériorité aux textures régulières, et plus encore dans le cadre des grands objets.

Troisièmement, nous avons abordé le problème de la synthèse d'images à partir de surfaces de points, une étape nécessaire pour visualiser les modèles numérisés avant reconstruction. En fait, nous avons montré qu'une surface de points peut être interfacée à faible coût avec le système de rendu polygonal matériel (GPU), en générant localement un maillage en deux dimensions organisé dans une structure hiérarchique: le *Surfel Stripping*. Cette approche ouvre la voie à l'utilisation du large répertoire de techniques de rendu polygonal aux surfaces de points et se présente comme une alternative au *splatting* et au *ray tracing*. De plus, dès lors que l'objet en entrée est trop grand, nous avons proposé un algorithme en flux capable de capturer la plus grande partie de la richesse visuelle de l'objet, en extrayant directement un ensemble de cartes de normales (et couleurs) en haute définition à partir du modèle de points, utilisés ensuite sur les *surfels strips*.

Finalement, nous nous sommes orienté vers la snythèse géométrique pour les applications intéractives. Nous avons introduit un noyau générique de raffinement de maillage sur GPU qui autorise le raffinement de maillages arbitraires avec un *déplacement* arbitraire, offrant des performances temps-réel pour une synthèse géométrique haute définition. Ce noyau permet de traiter directement les représentations de haut niveau, telles que les surfaces de subdivision, au niveau application/CPU, laissant le GPU créer une surface raffinée en une unique passe de rendu, au niveau du *vertex shader*. A l'aide de ce noyau, nous avons été en mesure de proposer un contrôl local des singularités de surfaces, en les décrivant à haut niveau par des facteurs de formes scalaires. Enfin, nous avons abordé le problème du rendu temps-réel de surfaces de subdivision en proposant une approximation qui évite la récursion tout en offrant un rendu adaptatif temps-réel.

Cette dernière contribution ouvre la voie à la synthèse géométrique dirigée par les données. Comme montré tout au long de cette thèse, les modèles numérisés doivent équilibrer qualité et efficacité afin d'être implantés dans les applications infographiques. Plus précisément, nous avons développé trois types d'analyse:

- une *décomposition hiérarchique volume-surface* qui améliore le traitement géométrique des surfaces 3D, tout en offrant la possibilité de mettre en oeuvre des algorithmes de dimension inférieur

- un schéma d'échantillonnage-reconstruction en flux qui structure l'édition hors-mémoire

- une synthèse *par l'instance*, qui offre un raffinement géométrique flexible pour les applications temps-réel.

Ces trois approches générales, combinées à la générictié des méthodes par points, peuvent être ap-

pliquées à divers autres problèmes, notamment la compression, l'édition de topologie, le calcul de visibilité et la synthèse géométrique par les données. Ce dernier sujet représente probablement la direction de recherche la plus prometteuse, puisque la taille grandissante des contenus 3D nécessitera rapidement le développement de représentations de plus haut niveau permettant à la fois une structuration dynamique pour l'édition et un échantillonnage précis à la demande pour la synthèse temps-réel.

Enfin, nous pensons que le lien entre données échantillonnées et procédurales demeure un problème majeur : les premières sont les seules informations capturées du monde réel, et les secondes sont le langage natif des machines. Nous aborderons ce problème dans les travaux à venir.

Author's Publications

- **Publications in 2007:**
  - **A Flexible Kernel for Adaptive Mesh Refinement on GPU**, *Tamy Boubekeur, Christophe Schlick*, Computer Graphics Forum, Volume to appear
  - **QAS: Real-time Quadratic Approximation of Subdivision Surfaces on GPU**, *Tamy Boubekeur, Christophe Schlick*, Proceedings of Pacific Graphics 2007 to appear
  - **SIMOD: Making Freeform Deformation Size-Insensitive**, *Tamy Boubekeur, Olga Sorkine, Christophe Schlick*, IEEE/Eurographics Symposium on Point-Based Graphics 2007 to appear
  - **GPU Gems 3: Generic Adaptive Mesh Refinement**, *Tamy Boubekeur, Christophe Schlick*, Addison-Wesley, page 93–104 2007 to appear
  - **Approximation of Subdivision Surfaces for Interactive Applications**, *Tamy Boubekeur, Christophe Schlick*, ACM Siggraph 2007 - Sketch Program to appear
  - **Scalable Freeform Deformation**, *Tamy Boubekeur, Olga Sorkine, Christophe Schlick*, ACM Siggraph 2007 - Sketch Program to appear
  - **On-the-fly Appearance Quantization on GPU for 3D Broadcasting**, Julien Hadim, *Tamy Boubekeur, Mickal Raynaud, Xavier Granier, Christophe Schlick*, ACM SIGGRAPH Web3D 2007
  - **Informatique Graphique et Rendu - Rendu par Points** , *Christophe Schlick, Patrick Reuter, Tamy Boubekeur*, Hermes Science Publication, 2006
- **Publications in 2006:**
  - **Volume-Surface Trees**, *Tamy Boubekeur, Wolfgang Heidrich, Xavier Granier, Christophe Schlick*, Computer Graphics Forum - EUROGRAPHICS 2006, Volume 25, Number 3, page 399–406
  - **Interactive Out-Of-Core Texturing**, *Tamy Boubekeur, Christophe Schlick*, ACM SIGGRAPH 2006 - Sketch Program
  - **Interactive Out-Of-Core Texturing Using Point-Sampled Textures**, *Tamy Boubekeur, Christophe Schlick*, IEEE/Eurographics Point-Based Graphics 2006
  - **Local Reconstruction and Visualization of Point-Based Surfaces Using Subdivision Surfaces** , *Tamy Boubekeur, Patrick Reuter, Christophe Schlick*, Computer Graphics & Geometry, Volume 8, Number 1, page 22-40, 2006
  - **Quantification la vole de l'apparence sur GPU**, Julien Hadim, *Tamy Boubekeur, Xavier Granier, Christophe Schlick*, AFIG '06
- **Publications in 2005:**
  - **Generic Mesh Refinement On GPU**, *Tamy Boubekeur, Christophe Schlick*, ACM SIGGRAPH/Eurographics Graphics Hardware 2005
  - **Rapid Visualization of Large Point-Based Surfaces**, *Tamy Boubekeur, Florent Duguet, Christophe Schlick*, EUROGRAPHICS VAST 2005
  - **Surfel Stripping**, *Tamy Boubekeur, Patrick Reuter, Christophe Schlick*, ACM Graphite 2005
  - **Visualization of Point-Based Surfaces with Locally Reconstructed Subdivision Surfaces**, *Tamy Boubekeur, Patrick Reuter, Christophe Schlick*, Shape Modeling International 2005
  - **Scalar Tagged PN Triangles**, *Tamy Boubekeur, Patrick Reuter, Christophe Schlick*, EUROGRAPHICS 2005 - Short Papers
  - **Surface Reconstruction with Enriched Reproducing Kernel Particle Approximation**, *Patrick Reuter, Pierre Joyot, Jean Trunzler, Tamy Boubekeur, Christophe Schlick*, IEEE/Eurographics Symposium on Point-Based Graphics 2005, page 79–87
  - **Multiresolution in Geometric Modeling - Techniques and Trends** , *Patrick Reuter, Tamy Boubekeur*, Virtual Concept 2005 - Invited Session