# Compression de maillages de grande taille

Clément Courbet

CENTRALE
PARIS
Ecole Centrale des Arts
et Manufactures
Ecole Centrale Paris

***THÈSE***

présentée par

**Clément COURBET**

pour l'obtention du

GRADE de DOCTEUR

Spécialité: Informatique

Laboratoire: Mathématiques Appliquées aux Systèmes (MAS)

# Compression de Maillages de Grande Taille

## *Efficient Compression of Large Meshes*

| | | | | |
|---|---|---|---|---|
| **Jury:** | MM. | Bruno | Lévy | Président |
| | | Pierre | Alliez | Rapporteur |
| | | Peter | Lindstrom | Rapporteur |
| | | Guillaume | Lavoué | Examinateur |
| | | Jean-Philippe | Nominé | Examinateur |
| | | Sébastien | Valette | Examinateur |
| | | Marc | Aiguier | Directeur |
| | | Céline | Hudelot | Encadrante |

# Abstract

A decade ago, 3D content was restricted to a few applications – mainly games, 3D graphics and scientific simulations. Nowadays, thanks to the development cheap and efficient specialized rendering devices, 3D objects are ubiquitous. Virtually all devices with a display – from a large visualization clusters to smart phones – now integrate 3D rendering capabilities. Therefore, 3D applications are now far more diverse than a few years ago, and include for example real-time virtual and augmented reality, as well as 3D virtual worlds. In this context, there is an ever increasing need for efficient tools to transmit and visualize 3D content.

In addition, the size of 3D meshes always increases with accuracy of representation. On one hand, recent 3D scanners are able to digitize real-world objects with a precision of a few micrometers, and generate meshes with several hundred million elements. On the other hand, numerical simulations always require finer meshes for better accuracy, and massively parallel simulation methods now generate meshes with billions of elements. In this context, 3D data compression – in particular 3D mesh compression – services are of strategic importance.

The previous decade has seen the development of many efficient methods for encoding polygonal meshes. However, these techniques are no longer adapted to the current context, because they suppose that encoding and decoding are symmetric processes that take place on the same kind of hardware. In contrast, remote 3D content will typically be created, compressed and served by high-performance machines, while exploitation (e.g. visualization) will be carried out remotely on smaller – possibly hand held – devices that cannot handle large meshes as a whole. This makes mesh compression an intrinsically *asymmetric* process.

Our objective in this dissertation is to address the compression of these large meshes. In particular we study random-accessible compression schemes, that consider mesh compression as an *asymmetric* problem where the compressor is an *off-line* process and has access to a large amount of resources, while decompression is a *time-critical* process with limited resources. We design such a compression scheme and apply it to interactive visualization.

In addition, we propose a *streaming* compression algorithm that targets the very large hexahedral meshes that are common in the context of scientific numerical simulation. Using this scheme, we are able to compress meshes of 50 million hexahedra in less than two minutes using a few megabytes of memory.

Independently from these two specific algorithms, we develop a generic theoretical framework to address mesh geometry compression. This framework can be used to derive geometry compression schemes for any mesh compression algorithm based on a predictive paradigm – which is the case of the large majority of compression schemes. Using this framework, we derive new geometry compression schemes that are compatible with existing mesh compression algorithms but improve compression ratios – by approximately 9% on average. We also prove the optimality of some other schemes under usual smoothness assumptions.

# Résumé

Il y a une décennie, le contenu numérique virtuel était limité à quelques applications – majoritairement les jeux vidéos, les films en 3D et la simulation numérique. Aujourd'hui, grâce à l'apparition de cartes graphiques performantes et bon marché, les objets 3D sont utilisés dans de nombreuses applications. A peu près tous les terminaux possédant des capacités d'affichage – des *clusters* de visualisation haute-performance jusqu'aux *smart phones* – intègrent maintenant une puce graphique qui leur permet de faire du rendu 3D. Ainsi, les applications 3D sont bien plus variées qu'il y a quelques années. On citera par exemple la réalité virtuelle et augmentée en temps réel ou les mondes virtuels 3D. Dans ce contexte, le besoin de méthodes efficaces pour la transmission et la visualisation des données 3D est toujours plus pressant.

De plus, la taille des maillages 3D ne cesse de s'accroître avec la précision de la représentation. Par exemple, les scanners 3D actuels sont capables de numériser des objets du monde réel avec une précision de seulement quelques micromètres, et génèrent des maillages contenant plusieurs centaines de millions d'éléments. D'un autre côté, une précision accrue en simulation numérique requiert des maillages plus fins, et les méthodes massivement parallèles actuelles sont capables de travailler avec des milliards de mailles. Dans ce contexte, la compression de ces données – en particulier la compression de maillages – est un enjeu important.

Durant la décennie passée, de nombreuses méthodes ont été développées pour coder les maillages polygonaux. Néanmoins, ces techniques ne sont plus adaptées au contexte actuel, car elles supposent que la compression et la décompression sont des processus symétriques qui ont lieu sur un matériel similaire. Dans le cadre actuel, au contraire, le contenu 3D se trouve créé, compressé et distribué par des machines de hautes performances, tandis que l'exploitation des données – par exemple, la visualisation – est effectuée à distance sur des périphériques de capacité plus modeste – éventuellement mobiles – qui ne peuvent traiter les maillages de grande taille dans leur intégralité. Ceci fait de la compression de maillage un processus intrinsèquement *asymétrique*.

Dans cette thèse, notre objectif est d'étudier et de proposer des méthodes pour la compression de maillages de grande taille. Nous nous intéressons plus particulièrement aux méthodes d'accès aléatoire, qui voient la compression comme un problème intrinsèquement *asymétrique*. Dans ce modèle, le codeur a accès à des ressources informatiques importantes, tandis que la décompression est un processus temps réel (souple) qui se fait avec du matériel de plus faible puissance. Nous décrivons un algorithme de ce type et l'appliquons au cas de la visualisation interactive.

Nous proposons aussi un algorithme *streaming* pour compresser des maillages hexaèdriques de très grande taille utilisés dans le contexte de la simulation numérique. Nous sommes ainsi capables de compresser des maillages comportant de l'ordre de 50 millions de mailles en moins de deux minutes, et en n'utilisant que quelques mégaoctets de mémoire vive.

Enfin, nous proposons, indépendamment de ces deux algorithmes, un cadre théorique général pour améliorer la compression de géométrie. Cet algorithme peut être utilisé pour n'importe quel algorithme basé sur un paradigme prédictif – ce qui est la cas de la majorité des méthodes existantes. Nous dérivons ainsi des schémas de prédictions compatibles avec plusieurs méthodes de la littérature. Ces schémas augmentent les taux de compression de 9% en moyenne. Sous des hypothèses usuelles, nous utilisons aussi ces résultats pour prouver l'optimalité de certains algorithmes existants.

# Acknowledgments

I would like to begin by thanking Jean-Philippe Nominé, by whom I was introduced to the field of visualization. Without him, I would never have though about doing this Ph.D in the first place.

Thanks to Marc and Céline for having advised me during my PhD. They have constantly driven me forward and given me the freedom to choose the particular aspects of my research on which I wanted to focus. They also have provided me with a number of opportunities to travel and meet people.

I am thankful to Pierre Alliez and Peter Lindstrom who have accepted to review the manuscript and whose comments have helped in improving my dissertation, as well as all the members of the jury,thanks to whom my defense was particularly enjoyable. I am also grateful to all the people who have written high quality papers that helped me get started in the field of geometry processing, and later develop my knowledge in that domain.

I would like to thank everyone at Lawrence Livermore National Laboratory – in particular Dan, Martin, Ming and Peter – for having made my stay here a very nice and comfortable one. The time I spent in the United States was a great opportunity for me to discover a complete new world and meet very interesting people. The Cottons provided me with a room and innumerable occasions of arguing about religion. Sophie and Vamsi drove me around San Francisco and helped me discover the city and its surroundings. I am most grateful to Martin, who took all the arrangements for my visit at LLNL. He provided me with both academic and personal advice, and introduced me to streaming compression as well as chicken farming and gave me the secret location of hot springs in the *Sierra Nevada*. He also was a great partner to bike to the lab every morning. Martin, I am *really really* happy that you were able to welcome me in Livermore.

I also thank Pascal Laurent, who teaches Mathematics at *Ecole Centrale* and introduced me to applied – and less applied – mathematics. His lectures and handouts were the best I have had during my studies, and contributed a lot to my interest for applied mathematics.

A huge thank you goes to Laetitia who loves me so much that she could actually undergo my oh-so-boring 'autistic' – and sometimes thesis-related – discussions with geeky friends.

I would like to thank Maryannick and Xavier for inviting me *à l'Ile de Ré*. This was an idyllic place to write part of my dissertation, rewarding hard work in the *syndicat d'initiative* – who were kind enough to offer me a nice and quiet shelter – with beach sessions and *caramel au beurre salé*.

I would like to thank my parents and my brother for their love and support during my studies and Ph.D, and for having – almost – force-fed the jury and attendants after my defense.

Finally, I thank my colleagues and friends – Adrien, Amel, Bilal, Hichem, Julie, Konstantin, Laurent, Matthieu, Nicolas, Sofiane, ...– for making work in the Lab so enjoyable. I am glad that the lab has many visitors from abroad, which provides endless possibilities to learn from other cultures.

# Contents

# List of Figures

# List of Tables

# Introduction

A few years ago, multimedia content was mainly limited to audio (music) and video (movies). In the same way records and movies have progressively provided better and better alternatives to traditional concerts and theatre representations, virtual 3D content now credibly emulates real-world objects in various contexts. 3D content is used in virtually every application that needs to provide a user with an intuitive experience of a real-world or synthetic object. With the development of the Internet, this content has been widely diffused to the public, with the result that most users of personal computers can navigate within a 3D scene comfortably using common input devices such as mouse and keyboard. One of the mainstream applications of virtual 3D content is entertainment, in particular the game industry, and the global spendings in this field will shortly exceed that of other traditional media like music [1].

Various Internet applications are progressively developing 3D services. Google augments geographic maps by rendering the terrain in 3D instead of using traditional isolevel curves. They also integrate 3D models of important buildings directly on the map[2]. Individual users can even upload their own models that are integrated into the database to be accessible to the community[3]. All these services are accessible with any Internet browser. This shows that 3D content is really becoming mainstream.

To provide even better immersion into the virtual world, more complex visualization devices such as *caves* have been developed, where the user is surrounded by display walls. In this setting, *haptic* devices replace the traditional mouse and keyboard, and provide *force feedback* to give virtual objects mass and momentum. The user is then able to manipulate virtual objects as if they were real. Because these 3D immersion systems are expensive, they mainly find applications in professional contexts like Computer Assisted Design[4] [Berta 1999] or Medicine (e.g. treatment of phobia [Garcia-Palacios et al. 2002]).

In the last decades, the various communities interested in 3D content – numerical simulation, computer graphics, geometry processing, ... – have come to a consensus where any surface or volume dataset is represented as a piecewise linear approximation, or *mesh*. This representation has the advantage of being *discrete*, and uses simple elements that computers can easily process. This representation is so ubiquitous that specialized hardware with adapted architecture, called *Graphic Processing Units* (GPUs), have been developed with the only goal of enabling the efficient visualization of surface meshes. Due to the importance of virtual or digitized 3D content in today's world, virtually every recent electronic device with display capabilities is now equipped with GPUs, from large visualization clusters to smart phones, also including commodity personal computers and game consoles.

A few years ago, most digital 3D content used to be synthetic, the market being driven mainly by applications like games and the film industry, and to a lesser extent numerical simulation. With the

[1] http://www.reuters.com/article/idUSN2132172920070621
[2] http://earth.google.com/plugin/
[3] http://sketchup.google.com/yourworldin3d/index.html
[4] Various firms now propose virtual reality services for engineering firms, e.g. http://www.virtalis.com/

development of accurate acquisition hardware, 3D objects have found several new applications. In particular, the growth of the Internet selling market has naturally led on-line stores to propose virtual 3D replicas of the products being sold, so that the customer can make a choice in complete confidence, as if in a store. A recent trend is the use of *augmented reality*: a typical application uses an image of the environment of the user – captured e.g. by a web cam – and integrates the virtual content inside the scene. This way, the user can see the behaviour of the object in its environment. For example, a company named FittingBox[5] proposes a software solution that enables users to virtually try pairs of glasses (see Figure 1).

Figure 1: The virtual mirror of FittingBox enables virtual try-on of glasses (Source: FittingBox[5]).

For larger adoption, such applications would ideally be able to run on consumer devices such as smart phones, that are usually equipped with a camera and some (small) 3D rendering capabilities. We use these devices as example to introduce several practical limits to what can be done in the field of 3D visualization. First, storage on these devices is limited. It would be totally impossible to store the complete collection of virtual objects that a store has to provide directly on the disk – usually a FLASH memory – of the device. On the other hand, the network bandwidth is limited. Therefore, keeping the collection on the remote server and downloading a new 3D model each time the user requests a product leads to a large latency.

3D applications have traditionally dealt with these problems using *compression*. The first mesh compression algorithms were designed to provide mesh representations that were as compact as possible, for efficient static storage. However, being able to store 3D models on a disk is not the ultimate goal. The actual application has to be taken into account. In the previous example of augmented reality, a nice feature of the compression algorithm would be *progressiveness*. A progressive decompression algorithm would start displaying a rough approximation of the mesh as soon as the first bytes of the model are received, somehow hiding the network latency to the user. In addition, in the concept of a globalized world where several different actors will process the information, possibly in a collaborative setting, compression/decompression algorithms must be able to handle the heterogeneous capabilities of the hardware of each user (see Table 1). For example, professional users like research laboratories dealing with large-scale numerical simulations will usually possess supercomputers with tremendous amounts of memory and computing power, and visualization clusters capable of rendering millions of triangles per second on displays with several megapixels. On the other hand of the spectrum, typical consumers will visualize 3D content with commodity hardware such as laptop computers or smart phones. In an ideal setting, the same tools could be used to handle these various systems. Therefore, compression/decompression algorithms should *adapt* to the various hardware limitations, in terms of network bandwidth, storage capability, available memory, display size, etc.

There is another fundamental difference between mesh compression for consumer applications and other professional contexts. In the first case, the only important thing is the visual aspect of the model. The actual goal is not to visualize the *mesh itself*, but the *surface* of the underlying object. In particular, the image must be devoid of *discretization artefacts*. Therefore, typical algorithms

[5]FittingBox, http://www.fittingbox.com/

| **Device** | **Storage** (size/bandwidth) | **Core memory** (size/bandwidth) | **graphics memory** (size/bandwidth) | **GPU** FLOPS | **Display resolution** (Megapixels) |
|---|---|---|---|---|---|
| iPhone | 8GB / ? | 256MB / ? | 24MB / 2GBps | 6M | 0.15 |
| 700 USD laptop (2010) | 300GB / 300MBps | 4GB / 1GBps | 512MB / 3GBps | 400G | 1 |
| Visualization cluster | virtually unlimited / 5GBps | 24MB per node / 18GBps | 6GB / 12GBps | 1T | 13 |

Table 1: Typical capabilities of various devices in 2010.

used in consumer graphics (e.g. games) make heavy use of perceptual techniques like normal interpolation, bump maps, smoothing, making lossy compression and remeshing acceptable (see Figure 2). In the second case, the correctness of the representation matters more than a pleasing aspect. In particular, in Computer Assisted Design (CAD), Medical or Numerical simulation applications, compression artefacts may hide interesting phenomena or features, lead to an incorrect diagnosis, or simply modify the results of a simulation. Therefore, most of these professional applications need *lossless* mesh compression, in contrast to games where lossy compression is acceptable as long as there is no perceptual modification.

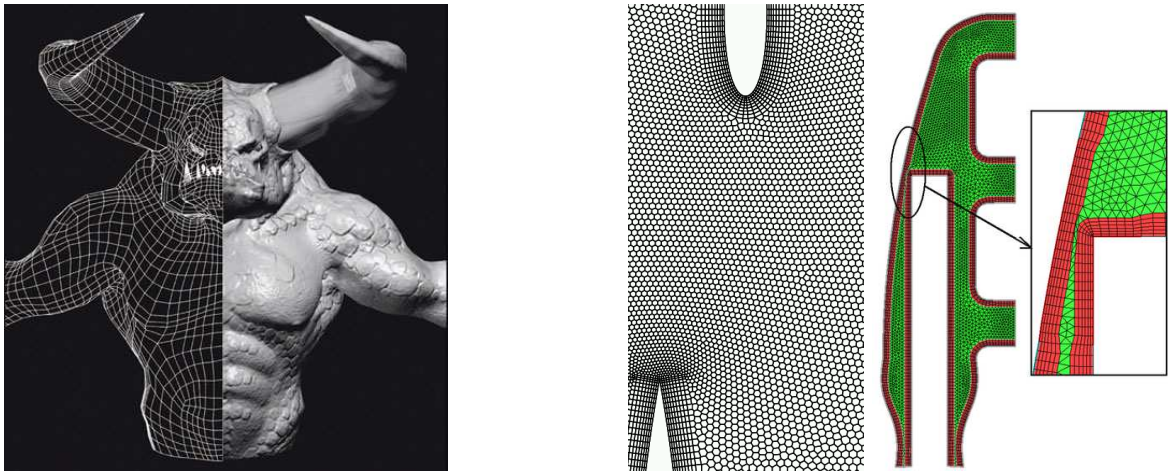

Figure 2: Typical meshes used by consumer and professional applications. The picture on the left shows the mesh used to represent a character of a video game. The mesh is very coarse and regular (left part), and rendering makes heavy use of texturing and normal mapping to recreate the details of the skin texture (right part). On the right, we show two simulation meshes, respectively a Computational Fluid Dynamics mesh for flow simulation around a wing, and a Finite Elements mechanical simulation. Structural details are captured by finer mesh elements adapted to the geometry. This disparity in the size and type of the elements must be conserved by the compression process.

The work presented in this dissertation specifically targets compression and decompression of meshes for *interactive visualization*. We do not consider mesh generation itself: the datasets can be synthetic (simulation, CAD, games, ...) or result from acquisition of real-world objects (medical data, heritage scanned objects[6], terrain data, ...). We aim at developing algorithms that read the input mesh, convert it to a compressed, *read-only* representation specifically designed so that it can be efficiently visualized. In particular, the comfort of the user during visualization should not be hindered by the size of the mesh. Frame rates should remain as high as possible, so that the user is able to interact smoothly with the model. This is different in spirit to the first compression algorithms,

[6]See e.g. the Digital Michelangelo Project, http://graphics.stanford.edu/projects/mich/

that targeted only efficient *storage* of the datasets, with no regard to their exploitation. Some of the constraints of visualization have been addressed in the past, by providing compression services with decompression *progressiveness*, in order to reduce latency. However, this approach is not suitable for *large meshes*. The following section examines the specific problems posed by this type of meshes.

## 0.1 Large Meshes

Recent technological developments in various fields have given birth to various techniques that generate very large meshes:

- Improvements in optical telemetry using lasers have permitted the digitization of real-world objects with a very fine precision. In particular, two applications have drawn the attention of the graphics community: Digitized Heritage Objects such as the Digital Michelangelo[7], and terrain data acquired via LIDAR-equipped vehicles (see Figure 3).
- Thanks to algorithmic advances, it is now possible to generate triangle meshes from the very large point clouds acquired with the previous techniques. For example, Isenburg *et al.* [Isenburg et al. 2006b] generate meshes with 9 billion triangles in 5 hours using $11MB$ of memory.
- The rise in processing power of simulation clusters has enabled more accurate simulations that require increasingly larger meshes.

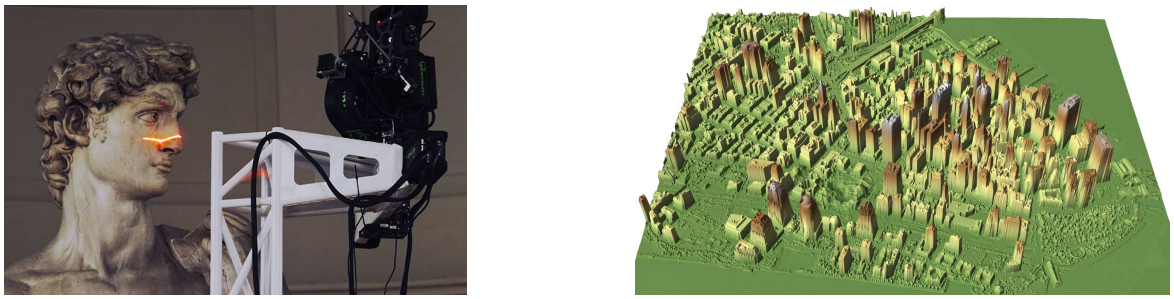

Figure 3: Digitizing real-world objects with lasers: A laser scanner used to digitize heritage man-made sculptures, here David's Michelangelo (left), and terrain data acquired with the LIDAR technology (right).

Most mesh representations require an amount of memory and processing power that is *linear* in terms of number of elements. Therefore, it is important to note the a mesh is only large *with respect to the capabilities of the device that must handle it*. In particular, the recent development of small hand held devices with limited capabilities has introduced a fundamental *asymmetry* in the compression/decompression process. On one hand, compression is usually carried out off-line on regular machines using mains power, and thus benefiting from the full processing power of recent performance-aimed technologies. On the other hand, mesh visualization can be made on any client terminal, up to mobile devices where performance is measured more in terms of energy efficiency than FLOPS. In this context, meshes that are *small* for the machine running the compressor may be very large for the client terminal running the decompression – and visualization – software. For example, a mesh of several million vertices will fit in-core on a large supercomputer with 16GB, however a mesh as small as tens of thousands of vertices will seem large to a smart phone user. This situation has renewed the interest for mesh compression given the importance of the consumer market.

[7] http://graphics.stanford.edu/projects/mich/

The two different problems discussed above show that the stakes brought by large mesh compression are twofold:

1. *Symmetric* algorithms must be developed to address very large meshes with respect to *state of the art* hardware in terms of performance. Here, both compressor and decompressor will have difficulties in dealing with the size of the mesh.

2. *Asymmetric* algorithms will enable off-line compression of moderately sized models, but real-time decompression for interactive visualization on lower-end devices.

In the following sections, we detail the various limitations brought by the hardware, and we show what features compression/decompression algorithms can propose to deal with them.

### 0.1.1 Storage

The first works on mesh compression did not consider speed nor memory requirements. The problem was to represent a mesh using the smallest possible amount of bits, for efficient storage on disk. Before visualization, the mesh had to be decompressed off-line and stored inside a temporary in-core memory structure, and only then be interactively visualized.

All these compression algorithms first load the whole mesh from the disk to the core memory, and convert it to a mesh structure that enables traversing the mesh in any desired order. The elements of the mesh are coded in a deterministic order *chosen* by the coder. During decompression, the mesh is rebuilt from the compressed stream *in the same order* to another in-core mesh structure, that can then be exploited for further processing. These algorithms are similar in spirit to popular general compression schemes like *ZIP*, except that they are capable of exploiting the specific redundancy found in meshes to achieve better compression rates. This helped reduce the amount of hard drive space to store the meshes, and thus decrease the associated cost. This problem is less important now that the price of hard drives has fallen.

### 0.1.2 Bandwidth

The previous algorithms designed for storage can also address the problem of efficient *transmission* through the network. Visualization often takes place in different places than mesh generation. For example, the results of a simulation may be visualized in an engineering center very remote from the cluster where the simulation was run. As network bandwidth is limited, compressing the mesh before transmission enables faster transmission. This approach is efficient as long as compression/decompression speed is competitive with transmission, which is still –and even more – the case now since processing power evolves faster than network bandwidth. The processing pipeline of this technique is illustrated on Figure 4.

In the specific case of transmission, these algorithms are able to decrease the effects of the limitation in network bandwidth. However, the user still has to wait for the end of the transmission to be able to begin visualization. For example, transmitting the 128 million vertices "Lucy"[8] model through a typical 10 Mbps DSL link would take several minutes even in compressed form. *Progressive* approaches represent the mesh as a *coarse* mesh approximating the original, and a sequence of refinements that enable recovering the full resolution model. That way, the user can begin visualizing the model before the transmission is complete, effectively hiding the transmission latency. The corresponding processing pipeline is shown in Figure 5.

[8] http://graphics.stanford.edu/data/3Dscanrep/

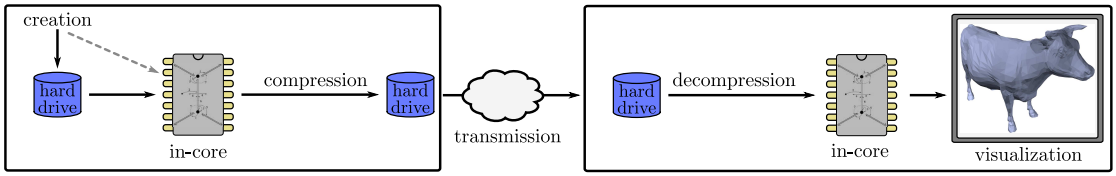


Figure 4: The single-rate transmission pipeline: The mesh is created. A complete in-core representation is built, either from a file on disk or directly after creation. The mesh is squeezed to a compressed file which is transmitted over the network. Once the file is received, it is decompressed to memory and further processed (e.g. visualized).

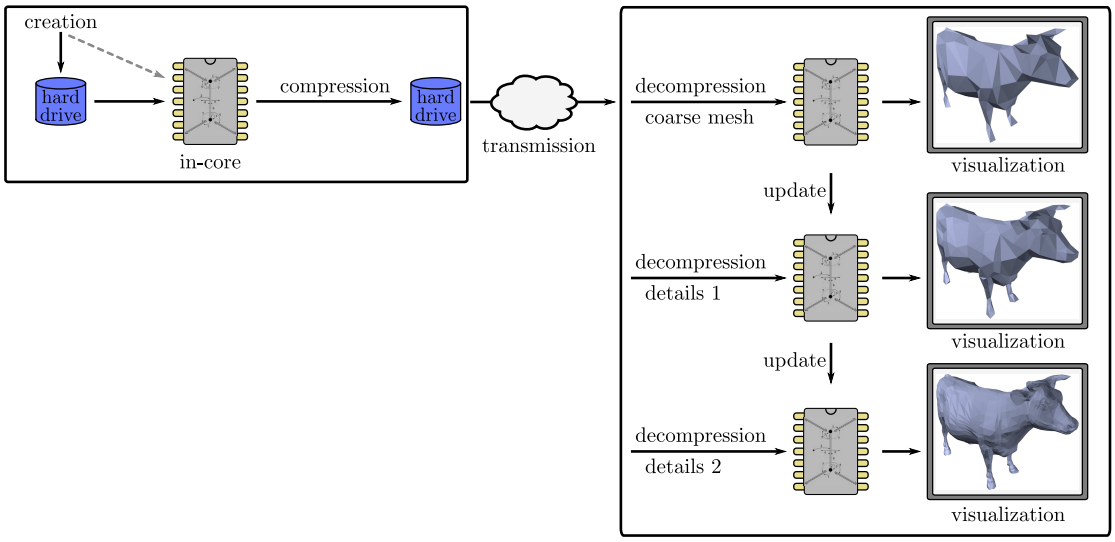


Figure 5: The progressive transmission pipeline: The mesh is created. A complete in-core representation is built, and the compressor writes it to a compressed file, as a coarse mesh and a sequence of details. The file is transmitted, and decompressed to memory as soon as the coarse mesh (i.e. the beginning of the file) is received. At this point, the visualization process can begin, hiding the latency to the user. The in-core mesh representation is then progressively refined as more details are transmitted.

This paradigm has another advantage: It enables *adapting* the mesh resolution to the capabilities of the client. The user – or the program itself – can decide to ignore further refinements if the desired quality has been reached or if the model becomes too complex for the hardware.

### 0.1.3 Memory

For very large meshes (or lower-end devices), the compression or decompression processes can be hindered by still another limitation. Both approaches begin by building an in-core representation of the mesh. As the size of datasets increase, this representation eventually becomes larger than the available memory. The same problem appears at the decompression side. Letting the in-core representation reside on the hard drive is not an option. Access times would be increased beyond practical rates – the difference between hard drives and in-core memory are about $10^6$ for latency and 10 for bandwidth. This would render the access with element granularity that is required by typical compression algorithms several orders of magnitude slower.

## 0.2 Contributions

This dissertation presents our work on mesh compression during the three years at Ecole Centrale Paris. The contributions are twofold:

- The first part of the thesis concentrates on geometry coding. As all compression algorithms spend most of the bit budget on compressing the *geometry* of a mesh, i.e. the position of the vertices, we tried to improve this aspect exclusively, without imposing any connectivity compression constraint. This work was motivated by the fact that typical compression algorithms do not derive geometry compression techniques from a formal and consistent approach, but the coding rules are usually determined *experimentally*. We designed a generic approach to linear geometry compression using prediction, based on smoothness assumptions.
- The second part of the thesis concentrates on handling large meshes. The most successful previous method, called *streaming*, enables compression and decompression of arbitrary large meshes, by reducing the amount of memory needed to handle them, and rendering processing I/O efficient. However it imposes various constraints on the decoding process, in particular the processing order. Therefore, it is not adapted to the case of visualization, where the user (and therefore the decoder) chooses where to look – i.e. the region to decompress. We designed an alternative approach that enables *random-accessible* decompression, where the decoder chooses the part to decompress.

## 0.3 Overview

These two contributions are totally independent. The first one is more general, and can be applied to traditional compression algorithms. Being local, it can also be used in either streaming or random-accessible algorithms. Therefore, it is presented separately. The dissertation is organized as follows:

In **Chapter 1**, the various notions used throughout the dissertation are introduced. Then, we give an overview of the various mesh compression approaches.

**Chapter 2** presents our contributions on geometry prediction. This is a very general approach that can be applied to any mesh compression methods, either the traditional algorithms of Chapter 1 or the specialized algorithms presented in the rest of the dissertation. Therefore, this section is completely independent from the rest. In this chapter, we begin by analysing the most widely used

*linear prediction* rules, and how they are determined. In particular, we present the *spectral* approach used to derive some predictors. In a second part, we show that this approach cannot be used in all generality in the context of irregular meshes. As an alternative, we propose a formalism that takes mesh smoothness as a starting assumption to automatically derive coding rules that have a theoretical background and work well in practice. This approach does not improve nor worsen speed or memory usage. It only tries to further shrink the mesh representation, following traditional approaches.

Chapters 3 to 5 specifically address the compression of large meshes:

**Chapter 3** details the specific problems posed by large meshes. We present the three major paradigms used to deal with this problem and the associated algorithms. We also show the relationships between these approaches.

**Chapter 4** presents our contributions in the field of *streaming* compression. The ideas of this chapter are not new, and are derived from the work of Isenburg and Lindstrom on streaming meshes [Isenburg and Lindstrom 2005]. All previous work in streaming compression dealt with simplicial complexes (triangle and tetrahedral meshes). We extend their ideas to meshes made of elements of higher degree, that pose specific problems. We target hexahedral meshes, that are interesting for scientific simulation, but the approach would be similar in the case of quad meshes. This work was conducted with Martin Isenburg at the *Lawrence Livermore National Laboratory*.

**Chapter 5** is about *random-accessible* compression. We present a totally novel compression algorithm that enables the decompressor to query random parts of the mesh in a hierarchical fashion. In contrast to previous random-accessible approaches that use traditional compression algorithms as a basis, the compression algorithm presented here is new. We prove the concept by presenting an interactive visualization application that is competitive with previous approaches although the implementation is not optimized.

# Chapter 1

# State of the art in mesh compression

In this chapter, we present the various concepts used throughout the manuscript. In particular, we describe in details the notion of a *mesh* which is the ubiquitous way of describing 3D objects. We detail the specific aspects in which mesh processing differs from traditional structured signal processing like image and sound, in particular the dissociation of *connectivity* and *geometry* information. Then, we present the various methods used to represent meshes in a more compact way, either in a lossless or lossy way. To keep the discussion general, we limit ourselves to the case of smaller meshes. The specific problems raised by larger meshes are detailed later in Chapter 3.

Traditional media like sound or images have an intrinsically regular structure that naturally guides the representation of the data. Most representations *uniformly* sample the signal values on a grid – 1D temporal grid for audio, 2D spatial grid for images. Therefore, the positions of the samples are *implicit* – for example, audio is sampled at time $t_0, t_1 = t_0 + T, t_2 = t_0 + 2T, \ldots$ – and only the *value* of the signal at each sample has to be stored. For 1D signals, this provides a natural *ordering* to process the data, inherited from the ordering of the set of natural numbers (the set of samples of a signal of size $S$ is in bijection with $[\![1; S]\!]$ ). For 2D signals, there is a large number of orderings to choose from, but some orderings are more natural, for example *scanline* or *z-order* [Morton 1966].

3D objects are far more complex to describe than the aforementioned media types. First, complex topologies prevent using a grid for sampling, therefore the positions of the samples must be specified *explicitly*. Second, there is no more a natural processing order, since the set of samples is arbitrary.

## 1.1 Preliminaries

A *volume mesh* is a simple representation of a 3D object by a set of polyhedra, as on Figure 1.1. It is composed of various elements, which are named *cells*, *faces*, *edges*, and *vertices* depending on their dimension. A *surface mesh* is a volume mesh that has only one cell, and thus only represents the surface of an object. Most computer graphics applications (animated movies, games) use surface meshes, since the insides of an object are rarely visible, while scientific simulations may use surface or volume meshes depending on the setting.

A mesh is described with two different components:

- The *Geometry* intuitively specifies the shape of a mesh. To each vertex is associated a position $(x, y, z)$ in 3D space. The function that maps the set of vertices $V$ to $\mathbb{R}^3$ and gives the position

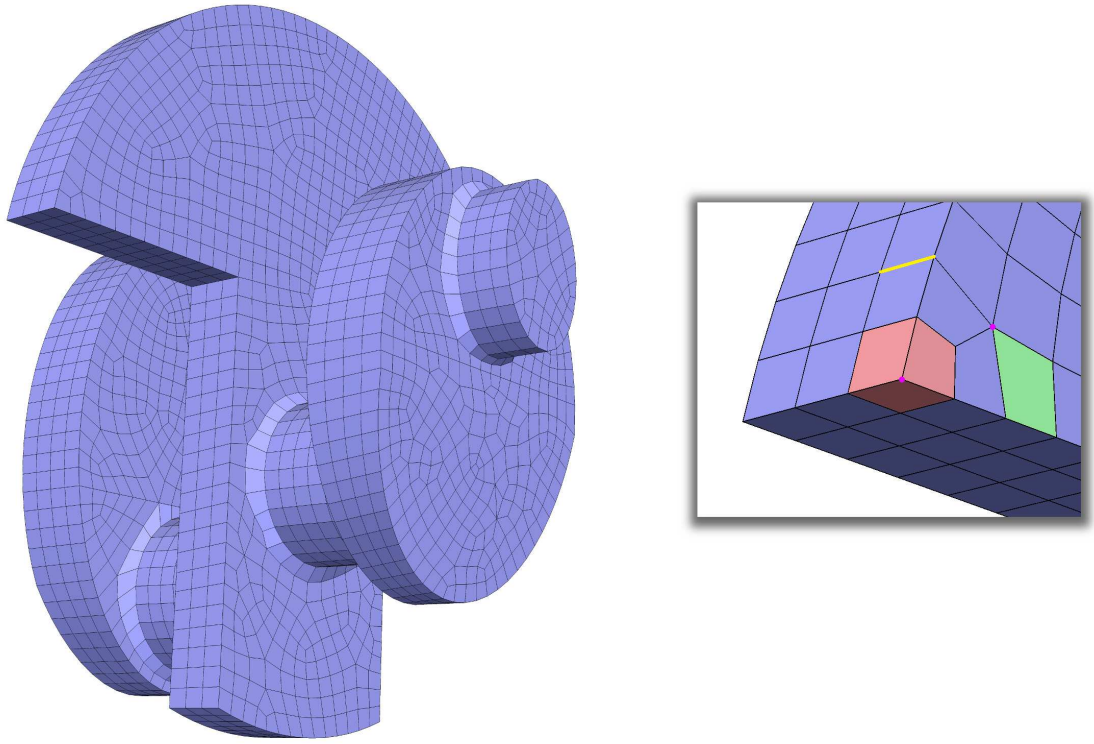

Figure 1.1: Example of a volume mesh with 6883 cells and 9218 vertices. The figure on the right shows a cell (red), a face (green), an edge (yellow), and two vertices (purple). In this model, all the cells are hexahedra, and all the faces are quads (degree 4). The vertices shown in purple have a degree of 3 (left) and 6 (right, with an edge that goes inside the volume).

of each vertex is called the *geometry function*. A mesh may also have additional properties such as vertex normals or physical quantities that are attached to the vertices. The concept of geometry function extends naturally to these situations by using a co-domain of higher dimension. Therefore, we usually use the term *coordinate* for both the positions and the properties attached to the vertices. In addition, there may be properties attached to edges, faces or cells (for example pressure or density attached to cells in a computational fluid dynamics simulation).

- The *Connectivity* describes the *structure* of the mesh, i.e. how the mesh elements are connected to each other. The connectivity of a mesh can be seen as a *graph*. Two vertices are *adjacent* (or *neighbours*) if they are connected by an edge. Two faces (resp. cells) are adjacent/neighbours if they share an edge (resp. face). The *degree* (or *valence*) of a vertex is the number of adjacent vertices. The *degree* (or *valence*) of a face (resp. cell) is the number of vertices it contains. Some authors choose to use the term valence only for vertices, and reserve degree to faces and cells. However, using the same term for both concepts enables more genericity when describing algorithms (e.g. independence to dimension). The *one-ring* of an element is the set of all adjacent elements of the same dimension.

The Figure 1.2 illustrates the difference between geometry and connectivity.

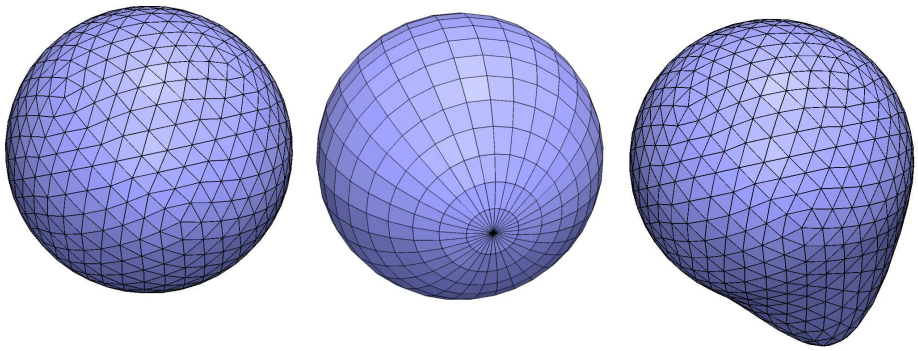

Figure 1.2: The two components of a mesh: connectivity and geometry. The leftmost image shows a triangle mesh of a sphere. The center picture shows a surface mesh made of quadrilaterals that has a different connectivity, but the same underlying geometry, while the right picture shows a mesh with the same connectivity but a different geometry than the leftmost picture.

Most of the meshes found in typical applications use only one type of element. The vast majority of surface meshes use triangular or quadrangular elements, while volume meshes usually employ tetrahedral or hexahedral elements (see Figure 1.3). These meshes are called respectively triangle (or triangular), quad, tetrahedron (tet) and hexahedron (hex) meshes. To designate a surface mesh with mixed elements, the term *polygon mesh* is used.

A surface mesh is *manifold* when each edge is shared by one or two adjacent faces (one at boundaries) and the neighbourhood of each vertex is homeomorphic to a disk. A volume mesh is *manifold* if the neighbourhood of each vertex is homeomorphic to a disk and the neighbourhood of each edge is homeomorphic to a cylinder. Figure 1.4 shows typical non-manifold situations. A connected surface manifold (or connected 2-manifold) mesh has *genus* $g$ if it is possible to remove $g$ closed loops without loosing connectivity. For a volume mesh, this translates to making $g$ cuts through closed loops in the boundary. Intuitively, $g$ is the number of *handles* that a mesh contains (see Figure 1.5). In a 2-manifold mesh with genus $g$, whose connectivity graph is planar in a surface of genus $g$, Euler's formula links the number of faces ($N_f$), edges ($N_e$) and vertices ($N_v$):

$$N_f - N_e + N_v = 2 - 2g \tag{1.1}$$

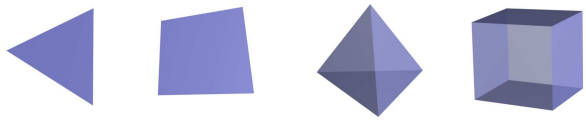

Figure 1.3: The most common mesh elements. From left to right: triangle, quad, tetrahedron, and hexahedron.

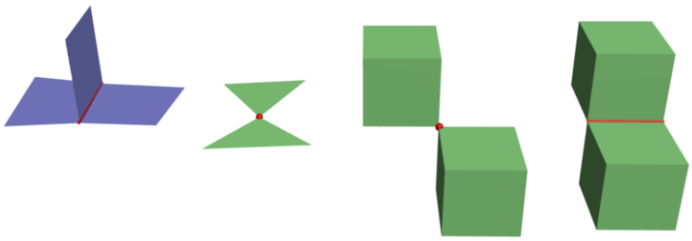

Figure 1.4: Typical non-manifold situations: On the left, in blue, three faces of a surface mesh share an edge. On the right, in green, the vertex (resp. edge) neighbourhood is not homeomorphic to a ball (resp. cylinder). The non-manifold elements are shown in red.

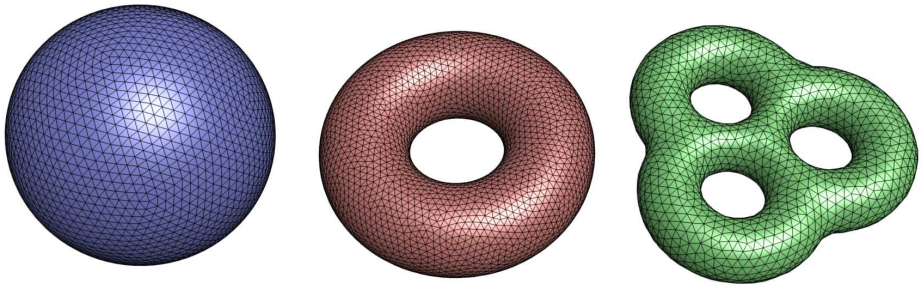

Figure 1.5: Manifold meshes with genus 0,1,3 (from left to right).

**Regularity:** One of the characteristics of a mesh which is of particular importance for mesh compression is its *regularity*. The regularity of a mesh is a notion that is based on connectivity only, and is measured by the deviation of the degrees of its vertices. A mesh with all its vertex degrees equal is called *regular*. This would be the case of a triangle mesh with vertices of degree 6, a hexagon mesh with vertices of degree 3, or a quad mesh with vertices of degree 4 (the latter mesh is said to have a *grid connectivity*). A mesh that is not regular is *irregular*. These notions are illustrated on Figure 1.6.

**Indexed Format:** The most widely used format for storing a mesh in memory or on disk is the *indexed format*. The geometry of the mesh is stored as an array of floating-point coordinates representing the position of each vertex. This introduces a natural numbering of the vertices: The index of each vertex is its rank in the array of coordinates. The connectivity of the mesh is stored as an array of integers representing the indices of the vertices of each face (resp cell) (see Figure 1.7). This representation is very simple to manipulate and implement, and numerous interchange formats such

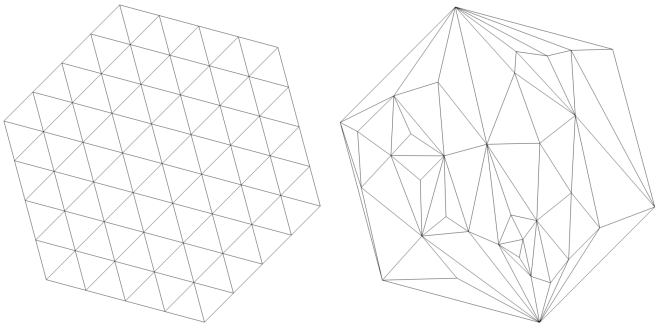

Figure 1.6: Difference between regular (left) and very irregular (right) meshes. Note that the interior vertices of a regular triangle mesh have degree 6.

as *OBJ*, *PLY*, *VRML*, *OFF*,... use it. On the other hand, it has numerous disadvantages, among which the fact that it provides no means for adjacency queries. It is also not adapted to the representation of large meshes, because the amount of memory used is tremendous. If each coordinate of a triangle mesh is coded with a float (32 bits), then the number of bits required to store the geometry is $32 \times 3 \times N_v$. To code the index of each vertex, $log_2(N_v)$ bits are needed. On the whole, as there are twice as many faces as vertices, the total number of bits required is $3N_v(2log_2(N_v) + 32)$. That means that a computer with 4GB of memory will not be able to work with meshes larger than 50 million vertices.

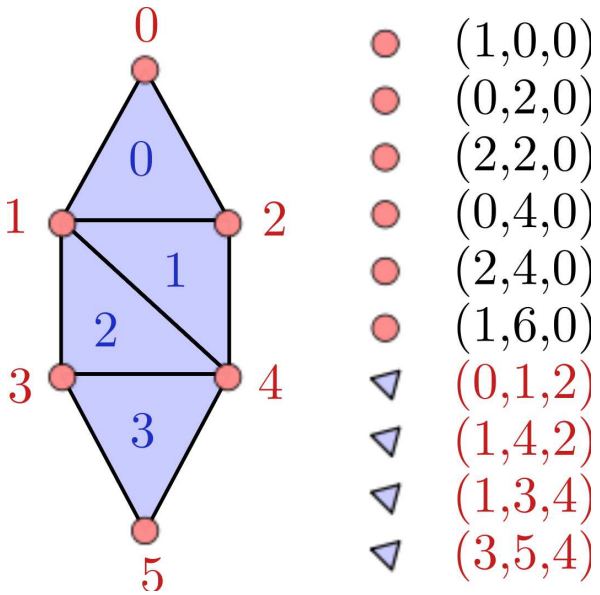


Figure 1.7: Indexed mesh format: First, the position of each vertex is specified as a 3-tuple of coordinates. This introduces a natural implicit vertex order (numbers in red on the left). Then, each face is specified by referencing the index of each of its vertices. The corresponding implicit order is given by the numbers in blue on the left.

## 1.2 Coding Data with Few Bits

Information theory, and data compression in particular, are mature areas of computer science. Their roots were set by Shannon in the middle of the last century, who first defined the notion of *entropy* as the information content of a signal [Shannon 1948]. Our goal here is not to give a complete overview of the domain, but just to quickly describe some basic notions that are needed to understand this work.

The *entropy* of a source of symbols measures how many bits one needs to losslessly encode each input symbol, and is defined by:

$$H = -\sum_{i=1}^{N} p_i log_2(p_i) \tag{1.2}$$

where $N$ is the number of different symbols, and $p_i$ is the probability of occurrence of symbol $i$.

Usually, the probabilities $p_i$ associated with the symbols output by the source are not known. In that case, one may substitute the *frequencies* of each symbol in the message $M$ to encode. The entropy is then:

$$H(M) = -\sum_{i=1}^{N} \frac{n_i}{N_s} log_2(\frac{n_i}{N_s}) \tag{1.3}$$

where $N_s$ is the total number of symbols in $M$, and $n_i$ is the number of occurrences of the symbol $i$.

### 1.2.1 Transforming the input

It is important to note that entropy considers the symbols individually and is only a measure of the information content in the *raw* message seen as a sequence of occurrences of a random variable. If the successive symbols of the message are correlated, it may be possible to decrease the entropy of the message by first applying an *invertible transformation* to reduce (or remove) redundancy.

To illustrate this idea, let us consider a simple source with two symbols $a$ and $b$. After outputting $a$, the source has 0.99 probability of outputting $b$, and 0.01 probability of outputting $a$. After outputting $b$, the source has 0.99 probability of outputting $a$, and 0.01 probability of outputting $b$. Because of symmetry, the source has an entropy of 0.5 bits per symbol. However, let us apply to the list of symbols the transformation $T : \{a, b\} \to \{0, 1\}$ defined by:

$$\begin{cases} a \mapsto 0 & \text{if the previous symbol is } a \\ a \mapsto 1 & \text{if the previous symbol is } b \\ b \mapsto 0 & \text{if the previous symbol is } b \\ b \mapsto 1 & \text{if the previous symbol is } a \end{cases} \tag{1.4}$$

Then, the symbols 0 and 1 have respectively the probability 0.99 and 0.01 in $T(M)$. This means that $T(M)$ has an entropy of 0.08 bits per symbol. For $T$ to be invertible, only the first symbol has to be stored, which uses 0.5 bits.

This simple example shows two things:

1. Transforming the input before coding may result in a large decrease in bitrate.

2. The efficiency of the transformation depends strongly on the existence of a model for a typical message. In the previous example, the knowledge of the conditional probabilities $p(S_n = s|S_{n-1} = \bar{s})$ helped in defining the transformation $T$. A transformation that is very efficient for a set of input messages may behave poorly for a broader or smaller set of input messages because of the change in the probability distribution of messages within the set.

Most mesh compression algorithms that we describe in the following chapters use this *transform coding* paradigm, to represent both connectivity (e.g. degree-based approaches, Section 1.3.1.1) and geometry (e.g. predictive or spectral approaches, Sections 1.3.1.3 and 2).

### 1.2.2 Coding

**Huffman Coding:** One of the most widely used methods to code a message was developed by Huffman [Huffman 1952]. It is a statistical approach that maps symbols with higher probability of occurrence to shorter codes and symbols that appear less frequently to longer symbols. This way, on average, the shorter symbols are used more and the size of the message decreases. The only constraint on the codes is that the decoder must be able to know the length of each code when decoding the message. The use of *prefix-free codes* guarantees that a code cannot be a prefix of another, so that there is no ambiguity during decoding. In addition to that, this method is easy to implement and can benefit from optimizations that make decoding very fast [Moffat and Turpin 1997]. On the other hand, as codes consist of an integer number of bits, the lower bound for entropy can not always be attained depending on the symbol probability distribution.

**Arithmetic Coding [Rissanen and Langdon 1979; Witten et al. 1987]:** Instead of operating in a per-symbol basis as Huffman's algorithm, arithmetic coding compresses the entire message as a single real number in the range $[0, 1[$. This enables to overcome the integer length code limitation of Huffman coding. Starting from $[0, 1[$, the current interval $I$ is progressively subdivided into several subintervals $D_i$ each associated with a symbol $s_i$. The width of each interval $D_i$ is proportional to the probability $p(S = s_i)$. At each step, the subinterval corresponding to the actual symbol becomes the current interval. At the end of a process, the current interval represents the message. This method performs very well in a bitrate point of view, with compression ratios usually close to the lower bound.

Arithmetic coding may seem superior to Huffman coding because of the better compression ratios. However, the latter is still used for various applications. In particular, it is a lot faster than arithmetic coding, and will thus be preferred for compression algorithms with emphasis on speed. Furthermore, the bit-aligned codes are not always a limitation. We will see in Chapter 5 how we can use this property as a *feature* to enable random access.

### 1.2.3 Coding meshes

These previous coding methods have found a wealth of applications in audio and image coding. By taking advantage of the *natural ordering* of the samples, typical compression algorithms *scan* the signal, transform it to a stream of compressible symbols that are encoded.

We have seen that in the case of a mesh, the set of samples lacks the simple structure found in audio and image signals. In particular, the positions of the samples themselves are explicit. In the case of the previous signals, the ordering itself is sufficient to encode the positions of the samples. The position of the next sample can be deduced from the current one at no cost. For example, if an image is processed in scanline order, the next sample is situated to the right of the current sample if the row is not full, and at the beginning of the next row otherwise. Only the redundancy in the *signal values* has to be removed. The fact that successive samples in processing order are *neighbours*, and thus the signal value at these samples is usually correlated, makes this process easy.

In the case of a mesh, there is no natural processing order, and the adjacency information is found in connectivity alone, and not in the order in which the vertices are given[1]. Therefore, the processing order has to be *invented* by the compressor so that there is a strong correlation between successive

[1]This is not exactly true. In some cases, there is some adjacency information hidden in the vertex numbering. The *Hexzip* [Lindstrom and Isenburg 2008] coder uses this information.

samples. Thus, mesh compression algorithms are more complex than their counterparts for uniformly sampled signals. The two following sections present the various classes of methods that are used for mesh compression. The first one regroups *connectivity-preserving algorithms*, that use different ideas than traditional signal coding schemes – although they largely borrow from them. The second one contains *connectivity-oblivious algorithms*, that transform the connectivity of the mesh in order to apply traditional compression algorithms.

## 1.3 Connectivity Preserving Compression

Most scientific applications require *lossless coding*. They cannot tolerate a loss in connectivity, since this could destroy interesting features. In particular, lossy compression methods based on remeshing (see Section 1.4) usually result in *smoother* meshes, effectively removing hard features in the model. In general, lossless compression is the safe bet to ensure that further mesh processing will not be hindered by compression artifacts.

We classify connectivity preserving compression algorithms in two groups, depending on whether they are driven by the *connectivity* or *geometry* components of the mesh. Both kinds of approaches yield similar results in term of bit rate.

### 1.3.1 Connectivity-driven algorithms

#### 1.3.1.1 Single-Rate Mesh Compression

The first works that address succinct planar graph representation are due to Tutte. The goal was mainly theoretical since this work was conducted before meshes found a large field of applications in computer graphics and numerical simulation. Tutte enumerates all the possible distinct genus 0 surface manifolds (or *planar graphs*) [Tutte 1962; 1963], and he deduces that the *connectivity* of any such mesh can be represented with 3.24 bits per vertex. However, the proof is enumerative rather than constructive, so determining explicit algorithms to generate efficient encodings has remained elusive until the work of Turan [Turan 1984], which was the first to propose an algorithm to encode the connectivity of a planar graph using a constant number (12) instead of $6log_2(N_v)$ bits per vertex as in the indexed representation (see Section 1.1).

The importance of succinct mesh representation was introduced to the graphics community by Deering [Deering 1995], when mesh compression became of practical importance due to the growth of 3D models. The early work of Turan was brought to the attention of the computer graphics community by Taubin and Rossignac [Taubin and Rossignac 1998], and these results were gradually improved using various methods until Poulahon and Shaeffer [Poulalhon and Schaeffer 2003] proposed an optimal method – in the worst case, but not very efficient on typical meshes – effectively reaching Tutte's bound.

In the following, we describe several classes of methods that were proposed to compress the connectivity of surface and volume meshes. Although different methods have been proposed for mesh compression [Bajaj et al. 1999a; Lindstrom and Isenburg 2008], most of them are strongly related to Turan's approach. Therefore, we first detail this method as a starting point. Then, most of the other algorithms can be classified into the following categories:

- *Label-based* approaches which represent the mesh with a sequence of *labels* derived from a depth-first traversal of the tree, and encode it in an efficient manner.
- *Degree coders* which represent the mesh as a sequence of vertex/edge/face degrees which are entropy coded, plus a usually small number of split offsets.

The algorithms for volume mesh compression are usually adapted from their earlier surface counterparts. Therefore, we emphasize the description of surface mesh coders, and then explain how the algorithms were extended to volume meshes. Most of the algorithms in this category target manifold meshes by design, with the exception of *Hexzip*[Lindstrom and Isenburg 2008], which is a totally lossless and very fast coder, but can only compress quad (or hex) meshes, and *TFAN* [Mamou et al. 2009] which is a generalization of degree coders to non-manifold meshes.

**Coding planar graphs using spanning trees** The early work of Turan [Turan 1984] is based on spanning trees of the connectivity graph. The algorithm first builds a vertex spanning tree (VST) and its dual face spanning tree (FST). Turan remarks that coding both these trees is sufficient to completely describe the connectivity of the mesh. To describe the trees in a succinct manner, the vertex spanning tree is traversed in a depth-first manner. Two symbols are used to code the structure of the VST: "+" and "–" respectively mean walk down and up the tree. The faces of the mesh are coded in parallel by the symbols "(" and ")" that respectively indicate that a face is to be opened and closed. These four symbols are sufficient to code both the VST and FST in an interleaved manner. The Figure 1.8 illustrates this algorithm. The VST has $N_v - 1$ edges, each of which is walked once down and once up, so there are $N_v - 1$ "+" symbols and $N_v - 1$ "–" symbols. There is also one pair of symbols "(" ")" for each edge that is not in the VST, i.e. $2N_e - 2N_v + 2$ pairs of parentheses. Turan codes each symbol using 2 bits, therefore the algorithm uses $4N_e$ bits to code the mesh, or 12 bits per vertex. Isenburg remarks that for triangle meshes, a slight modification of the algorithm halves the bit rate [Isenburg and Snoeyink 2004].

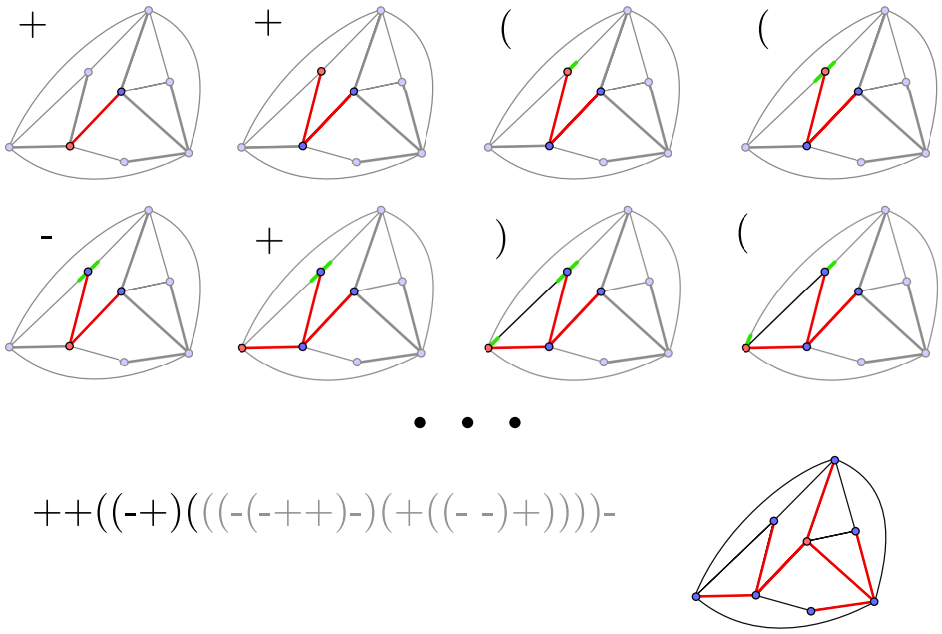


Figure 1.8: Decoding a planar graph with Turan's algorithm [Turan 1984]: The bold edges are the edges of the VST. Decoded edges are red or black depending on whether they belong to the VST or not. The edges that are not yet decoded are in light gray. The red vertex is the current vertex, and green slots denote the open faces ("(" symbol) that are not yet closed (by a ")" symbol). The bottom row shows the complete sequence to code the model, the darker symbols being the 8 stages detailed on the first 2 rows.

**Label-based methods:** Several other algorithms improve this approach by using a deterministic approach to build the VST and FST [Keeler and Westbrook 1995; Rossignac 1999; Gumhold and Strasser 1998; Isenburg and Snoeyink 2000; Li and Kuo 1998a; Ivrissimtzis et al. 2002]. In these schemes, the spanning trees are not explicitly constructed. Instead, the algorithms progressively grow a *processed region* from an initial seed face in the mesh – we call these methods *conquest-based* or *region-growing* algorithms. The processed region is separated from the *unprocessed region* by one or several closed *boundary loops* (see Figure 1.9). The boundary loops are placed on a stack and processed one at a time. Each of these boundary loops has a *focus element*, which indicates where the processing occurs. The focus element can be either a face ( [Gumhold and Strasser 1998; Rossignac 1999]) or an edge ( [Li and Kuo 1998a; Isenburg and Snoeyink 2000]). At each step of the algorithm, the processed region is grown from the focus element towards the unprocessed region so that the focus element passes inside the processed region. A *label code* is output that denotes how the boundary was modified.

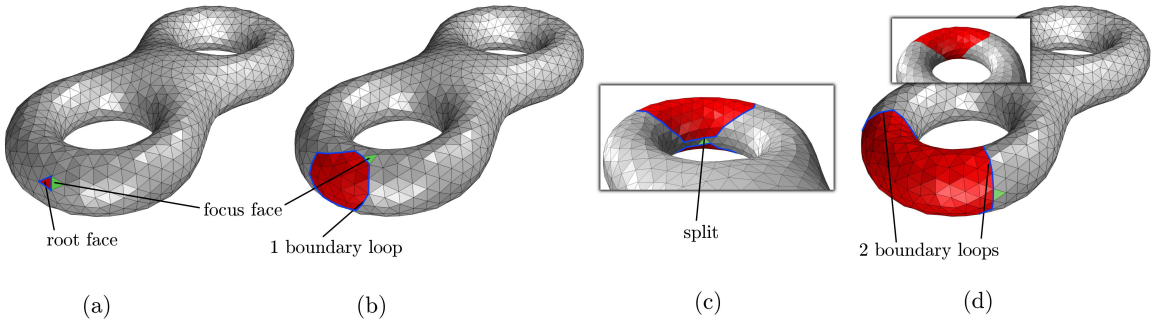


Figure 1.9: Illustration of a conquest-based mesh traversal using a face as focus element, at 4 different steps of the algorithm. Starting from a root face (a, step 0), the processed region (in red) is grown through the focus face (green). The processed region is separated from the unprocessed region by a closed boundary loops (in blue). At some point in the conquest, the processed region folds on itself (c), splitting the boundary in two (d).

As an example, we detail Rossignac's EdgeBreaker [Rossignac 1999]. This algorithm works on triangle meshes and uses faces as focus elements. Starting from an arbitrary root triangle, it grows the processed region by recursively applying one of the 5 operations shown in Figure 1.10. The choice of the operation to use depends on the processed state of the elements in the neighbourhood of the focus triangle, as shown in Figure 1.10. This process traverses the mesh in a *depth first*, spiraling manner, and codes a symbol for each operation ($C$,$L$,$E$,$R$,$S$). At each step, the processed region is augmented with the focus triangle, and one of the adjacent faces is chosen as the new focus triangle. It may happen that the processed region folds on itself (see Figure 1.9, c). In that case, a *split* symbol ($S$) is issued, the boundary is split in two and one of the resulting parts is pushed on the stack. Processing will resume by popping this boundary when an *end* ($E$) symbol is encountered. As vertices are only introduced by the $C$ operation, there are exactly $N_v$ $C$ symbols in the stream, i.e. half the symbols are $C$. Therefore, the authors code the $C$ symbol on one bit, and the remaining 4 symbols on 3 bits. Thus, any mesh is guaranteed to be encoded using 2 bits per triangle (or 4 bits per vertex). The symbol to code mapping and the algorithm itself were later modified to provide better compression rates [King and Rossignac 1999a; Rossignac and Szymczak 1999]. More importantly, this method was extended to handle meshes with an arbitrary number of holes and handles [Lopes et al. 2003] as well as quad meshes [King and Rossignac 1999b] and tet volume meshes [Szymczak and Rossignac 1999].

The edge-based FaceFixer algorithm [Isenburg and Snoeyink 2000] works in a similar manner, but uses an edge focus instead of a face. A more recent approach by Kälberer *et al.* [Kälberer et al.

2005] uses a vertex focus that is picked among all possible focus vertices in the boundary according to some local geometry criterion. This scheme greatly improves the compression rates of the previous approaches on average and performs very well on regular meshes. Although this scheme looks like a *degree coder* (next section), in the sense that it uses symbols representing vertex degrees, it is actually a label-based method.

Two other algorithms [Gumhold and Strasser 1998; Li and Kuo 1998a], that respectively use face and edge focus, use explicit *split offsets* to code where the split occurs in the boundary. These offsets, which are redundant (i.e. provide no more information than the stream of symbols) use $log_2(N_b)$ bits, where $N_b$ is the size of the boundary. Therefore, they are a bit less efficient than their split offset-free counterparts EdgeBreaker and FaceFixer. However, this enables one-pass encoding and decoding of meshes, which can be an interesting feature.

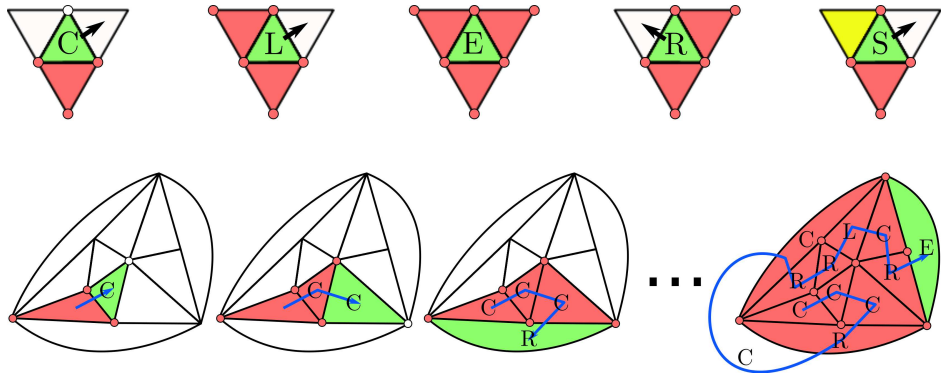


Figure 1.10: Top: The five operations used by the EdgeBreaker algorithm [Rossignac 1999]. The green triangle is the focus triangle. Red and white respectively denote visited and not visited triangles/vertices. The arrow shows which triangle will be processed next. For the $S$ operation, the yellow triangle is pushed on a stack and will be popped after each $E$ operation (that does not have a next triangle). Bottom: a simple run of the algorithm, where the triangle spanning tree (blue) is a chain.

These approaches have later been extended to handle volume meshes, but only in the case of tetrahedrals [Szymczak and Rossignac 1999].

**Degree Coders** The other major class of approaches for connectivity compression is called *degree coders*. The original algorithm of Touma and Gotsman[Touma and Gotsman 1998] codes a triangle mesh by using a region-growing paradigm like the other algorithms presented above. In this case, the focus element is a vertex. Each vertex of the boundary stores the number of adjacent vertices that are either unprocessed or in the boundary. Let $v_f$ be the focus vertex, $e_f$ the next boundary edge adjacent to the focus vertex, and $f_f = e_f \uplus v_o$ the face adjacent to $e_f$ that is not yet processed. Three cases may arise, that are shown in Figure 1.11:

- (*add*) the last vertex $v_o$ of $f_f$ is not on the boundary. Then the processed region is grown to include $f_f$, and $v_f$ is now inside the boundary (Figure 1.11, $(a)$). To maintain the count of adjacent vertices for $v_o$, only its degree needs to be coded. The count of unprocessed adjacent vertices of $v_f$ is decreased.

- (*full*) $v_o$ is full, i.e. its count of adjacent vertices is 0. This case can be solved without ambiguity by adding $f_f$ to the processed region (Figure 1.11, $(b)$). No information needs to be coded. The next vertex in the boundary becomes the focus vertex.

- (*split*) the last vertex $v_o$ of $f_f$ is on the boundary. Then the processed region is grown to include $f_f$, and a split symbol is issued, and the boundary is split in two and one of the parts is pushed on a stack (Figure 1.11, (*c*)). A *split offset* codes where the split occurs in the boundary.

When there are no vertices in the current boundary, the next boundary is popped from the stack. When there are no more boundaries on the stack, the algorithm terminates.

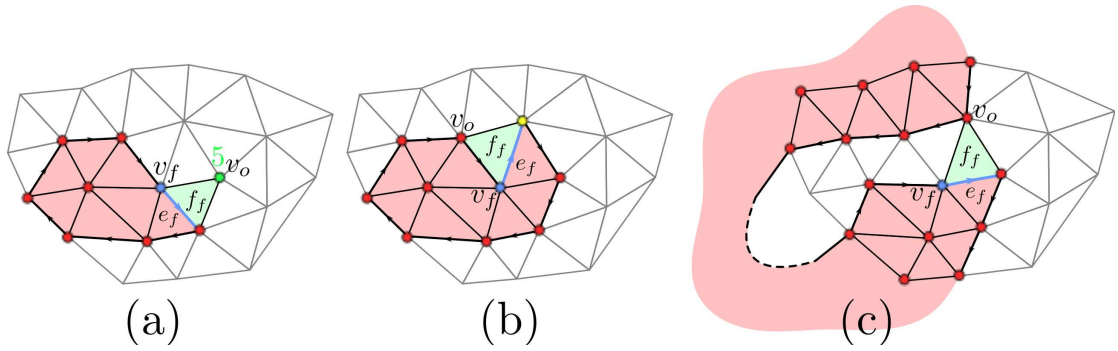


Figure 1.11: Typical cases encountered in the traversal of a mesh by the algorithm of Touma-Gotsman [Touma and Gotsman 1998]: vertex addition (a), full vertex (b), and split (c). The blue vertex/edge are the focus vertex/edge, the yellow vertex will become the new focus vertex. The red faces belong to the processed region.

Using this method, the mesh is represented by an ordered sequence of *vertex degrees*, along with the usually small information required for splitting. Thus, most of the information is carried by the list of degrees. For triangle meshes, the vast majority of vertices have degree 6, the degree of the other vertices being tightly spread around this value (Figure 1.12). This enables entropy compression of the sequence of vertices, with bitrates that depend directly on the regularity of the mesh, and can amount to a fraction of a bit par vertex for very regular meshes.

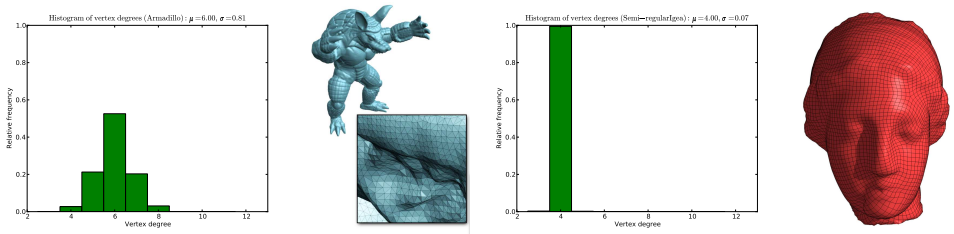


Figure 1.12: Histogram of vertex degrees for the *Armadillo* model (entropy 1.74) and the very regular remeshed *Igea* model (entropy 0.05). $\mu$ and $\sigma$ respectively give the expectancy and standard deviation of the distributions.

Alliez and Desbrun [Alliez and Desbrun 2001b] have used an *adaptative* mesh traversal method to decrease the number of splits and thus decrease the bitrate. To do that, the focus vertex is chosen so that it minimizes the split likelihood. In addition, the split offset is given in the ordered list of closest potential split candidates. This decreases the average number of bits to specify the offset. The authors also study the optimality of their approach. They show that if the number of splits is negligibly small, then degree coding is optimal is the sense that the entropy of vertex degrees for meshes that respect Euler's formula reaches the theoretical bound of Tutte. However, Gotsman later found [Gotsman 2003] that their lower bound could be improved by adding constraints to the

distribution of vertex degrees, falling *strictly below* Tutte's bound if splits are negligible. Thus, there *must be* some additional information needed to code a triangle mesh in addition to the sequence of vertex degrees, and the question of the optimality of degree coding is still open.

The approach of Touma and Gotsman has later been generalized to polygon meshes by Khodakovsky *et al.* [Khodakovsky et al. 2002] and Isenburg *et al.* [Isenburg 2002] by coding the degree of faces along with that of vertices. For meshes with only triangles, this is essentially free since the sequence of face degrees has zero entropy. For volume meshes, this approach has been extended to code hex meshes respectively with edge degrees by Isenburg and Alliez [Isenburg and Alliez 2002a] and vertex degrees by Krivograd *et al.* [Krivograd et al. 2008].

All the techniques presented here deal only with orientable manifold meshes. Mamou *et al.* have extended the degree coding approach to non-manifold triangle meshes [Mamou et al. 2009]. Their TFAN algorithm grows a region by adding a new *triangle fan* to the current region at each step of the algorithm. They identify that only 10 different triangle fan configurations can happen (Figure 1.13). Depending on the configuration, they need to transmit additional information. The first two configurations (Figure 1.13, $(a)$ and $(b)$) are sufficient to compress oriented manifold meshes, and correspond respectively to an *add* and *split* command in the algorithm of Touma and Gotsman. The first configuration $(a)$ requires coding the central vertex degree, while the second $(b)$ requires the central vertex degree and a split offset. The 8 other configurations enable the compression of non-oriented and/or non manifold meshes. In addition to being generic, this method is also very fast. However, this approach can only deal with triangle meshes and cannot easily be generalized to polygon meshes since it relies on an extensive listing of all possible configurations. If the faces have arbitrary degrees, there is no longer a small fixed number of configurations to choose from.

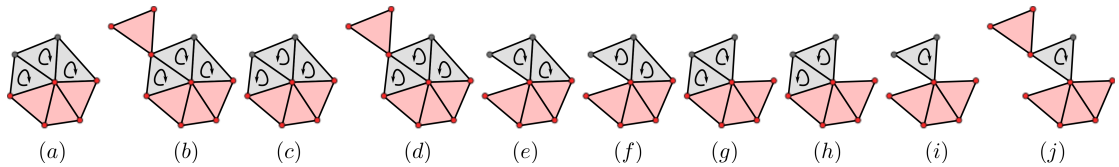


Figure 1.13: The 10 TFAN [Mamou et al. 2009] configurations. Grey vertices and triangles are being added, and red triangles belong to the processed region. The first configuration corresponds to a simple vertex addition in a degree coder, while the second corresponds to a *split* command. The other enable handling of non oriented/manifold meshes.

**Hexzip** In all the previous approaches, the coder chooses the order in which it traverses the mesh. This has two advantages: First, it greatly reduces the amount of information stored, since it destroys vertex ordering. Second, it enables greater decorrelation of signal values, since the coder is free to choose to have adjacent vertices appear contiguously in the compressed stream.

Recently, Lindstrom and Isenburg have introduced a completely new approach to quad/hex mesh compression [Lindstrom and Isenburg 2008]. Their *Hexzip* algorithm departs from the traditional spanning-tree approach and works directly on the *indexed* representation. They take advantage of the fact that quad/hex meshes usually have a coherent numbering of vertices. They read elements one at a time and predict their *vertex indices* using hash tables. Thus, they are able to transform the indexed representation into a very redundant, byte-aligned list of symbols that can be compressed very efficiently using conventional compressors (e.g. gzip). The geometry of vertices is predicted using *spectral prediction* [Ibarria et al. 2007] (see Section 2.1). This approach is the most efficient in terms of speed, and has the advantage that it is *completely lossless*, because it keeps both the vertex and quad/hex order. Furthermore, it is manifoldness-oblivious, and compression rates can be very

high when the input order is very coherent. The main drawback of this approach is that it is unable to compress triangle and mixed meshes.

#### 1.3.1.2 Progressive Mesh Compression

The single-rate methods described in the previous section target *low bitrates* (with the exception of Hexzip that also optimizes speed). While this is very important for storage, these approaches are not always adapted for transmission, because the user must load the whole mesh to be able to have a global view of the model. On the other hand, progressive methods build successive approximations of the mesh, that begin with a *base* coarse mesh that has few vertices and end with the mesh in full resolution (Figure 1.14). The finer meshes are built from the coarser representations using algorithm-specific refinement operations (usually either a *vertex split* [Hoppe 1996] or *addition of a center vertex* [Alliez and Desbrun 2001a]). This way, the mesh can be globally visualized even if the full compressed model has not yet been totally downloaded.

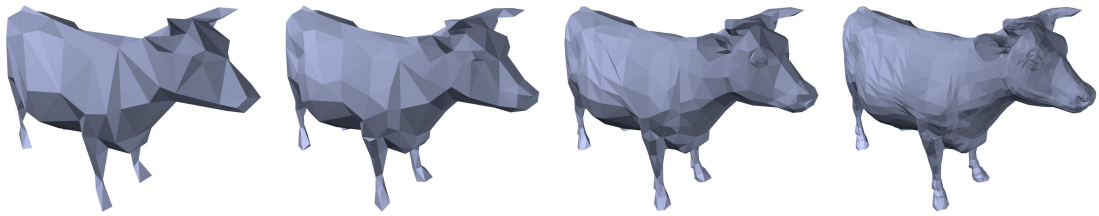

Figure 1.14: Progressive decompression of the cow model.

As a detailed knowledge of the inner workings of progressive compression schemes is not necessary to understand the work presented in the following sections, we will not detail here the various algorithms that exist for progressive mesh compression. There already exist detailed surveys on the subject [Alliez and Gotsman 2005; Peng et al. 2005].

The compression performance of progressive algorithms is not as good as single-rate algorithms. However, the total bitrate to encode a mesh is not the most relevant measure of the efficiency of a progressive compression algorithm. What is desired is that the coarser approximations of the mesh faithfully represent the surface, i.e. that decoding only the beginning of the bit stream still provides a good idea of the shape of the model. Therefore, the performance is often given in the form of a of Rate/Distortion (R/D) curve, which gives a measure of the error between the original model and the model obtained by truncating the bitstream at a certain position. Usually, progressive algorithms that have a fine granularity [Hoppe 1996; Popovic and Hoppe 1997; Li and Kuo 1998b] will perform better at intermediate bitrates but will be less efficient at the full resolution than algorithms that have larger granularity, e.g. those who make refinements in batches [Taubin et al. 1998; Bajaj et al. 1999b; Cohen-Or et al. 1999; Pajarola and Rossignac 2000a;b; Alliez and Desbrun 2001a; Valette and Prost 2004; Lee et al. 2010a;b]. Ultimately, single-rate methods compress the mesh in a single batch.

#### 1.3.1.3 Coding the geometry

The algorithms described above concentrate on connectivity coding, and consider geometry coding as a subordinate task. These algorithms – both single rate and progressive – impose a natural traversal order on the vertices, each vertex being added at a specific step of the algorithm. Most of them choose to specify the position of a vertex at this moment. Directly specifying the vertex position as a triplet $(x, y, z)$ of floats costs 96 bits. This figure can be drastically reduced using two complementary techniques:

- **Prediction:** The position of the vertex can be predicted using the already decoded vertices of the neighbourhood. If the mesh is *smooth* enough, then the *prediction residuals*, i.e. the differences between the predicted and actual vertex positions, are small, and they are narrowly spread around zero. This makes them good candidates for entropy coding. We review existing predictive approaches in detail in Section 2.1.
- **Quantization:** The vertex positions are usually quantized to integers using a fixed number of bits (usually 12). While this destroys the full floating point precision, it is usually sufficient for applications that can tolerate slight loss (e.g. visualization), and enables entropy coding, leading to good compression rates. Truly lossless compression is harder, since direct entropy coding of the residuals is impossible. However, the representation of floating-point numbers is actually discrete. Isenburg *et al.* have taken advantage of this property to propose a method suitable for the compression of floating-point residuals [Lindstrom and Isenburg 2006b]. However, the results are not as good as when a slight loss is tolerated. Another approach to quantization is due to Sorkine *et al.* [Sorkine et al. 2003]. They remark that usual quantization introduces *high frequency* errors that are visually very noticeable. To alleviate this problem, they propose to apply the quantization step in a transformed $\delta$-*coordinates* space obtained by applying an invertible linear transformation to the vertices. They show that quantizing the $\delta$-coordinates concentrates the errors in the low frequencies, which enables aggressive quantization with less visually noticeable artifacts (see Figure 1.15).

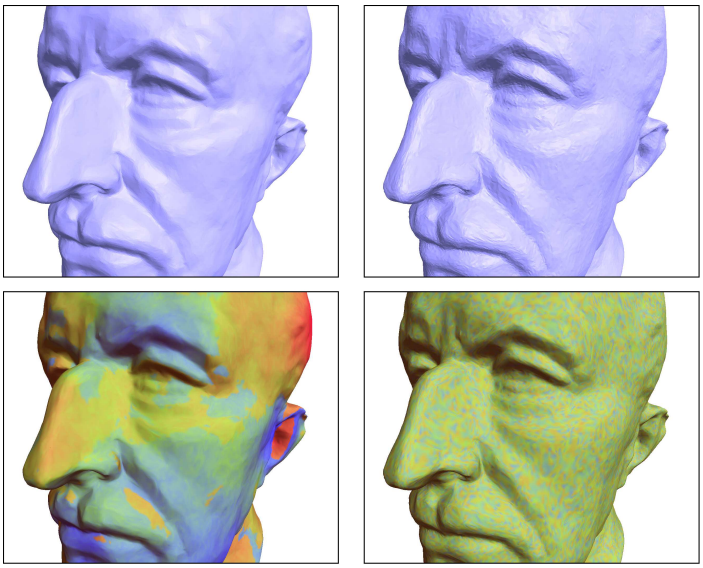

Figure 1.15: The $\delta$-coordinates quantization to 5 bits/coordinate (left) introduces low-frequency errors, whereas Cartesian quantization to 11 bits/coordinate (right) introduces noticeable errors. The upper row shows the quantized model, and the bottom figures use color to visualize the corresponding quantization errors. (Figure from [Sorkine et al. 2003])

**Spectral coding: Mesh-JPEG** Karni and Gotsman have successfully extended the very popular JPEG compression scheme used for images to handle irregular meshes [Karni and Gotsman 2000]. Similarly to JPEG, they transform the geometry function that is defined on the vertices from the *spatial* to the *frequency* (*spectral*) domain using a *linear transformation* (that can also be seen as a change of basis). However, the Discrete Cosine Transform (DCT) used by JPEG cannot be directly transposed to the irregular setting.

Karni and Gotsman define their spectral transform by analogy with the DCT. As the basis vectors of the DCT are the eigenvectors of the Laplacian of a 2D image, they use as spectral basis vectors the eigenvectors of the mesh Laplacian $L$ defined by:

$$L_{i,j} = \begin{cases} degree(v_i) & \text{if i=j} \\ -1 & \text{if } v_i \text{ and } v_j \text{ are adjacent} \\ 0 & \text{else} \end{cases} \quad (1.5)$$

The *frequencies* correspond to the eigenvalues associated with the eigenvectors. This basis has the same properties as the DCT: for typical meshes, most of the energy is localized in the low frequencies, which enables aggressive quantization and/or zeroing of the higher frequency coefficients with low distortion (see Figure 1.16). Because extracting the eigenvectors of the $N_v \times N_v$ Laplacian matrix is a costly operation (in $O({N_v}^3)$), the mesh is split in smaller independent patches which are compressed independently. To further improve decompression speed, they later employed fixed spectral bases [Karni and Gotsman 2001]. While being less efficient in terms of compression, this approach avoids having to compute eigenvectors, which enables faster decompression.

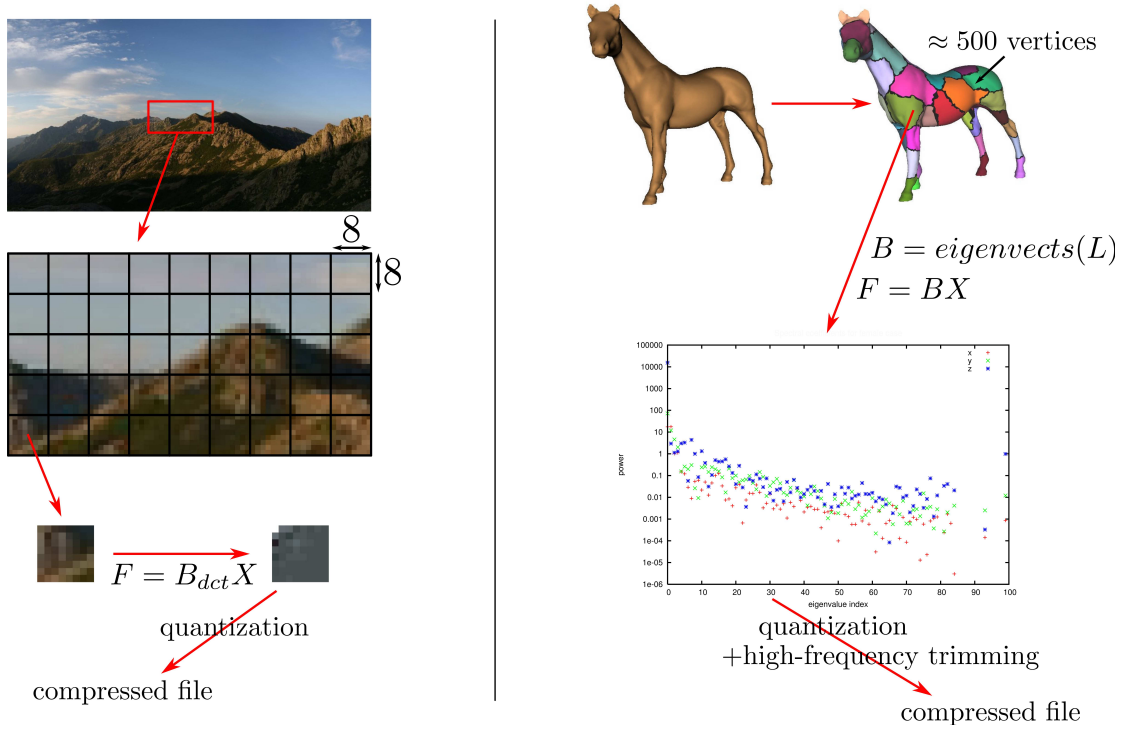


Figure 1.16: Parallel between JPEG (left) and Spectral Geometry [Karni and Gotsman 2000] (right) compression processes. JPEG cuts an image into blocks of 8×8 pixels, then uses the DCT to transform the blocks in the spectral domain, and quantizes and entropy codes the resulting coefficients. Karni and Gotsman cut the mesh into patches of around 500 vertices. For each patch, they compute the associated spectral basis $B$ from its connectivity, and transform the geometry to the spectral domain, then quantize and trim the high frequencies. The Figure on the right gives the amplitude of coefficients versus the *frequency* (log scale) for the $x$, $y$ and $z$ components of the geometry. For both methods, most of the energy is localized in the low frequencies.

The spectral approach gives the best geometry compression results if a slight degradation of the geometry is permitted. Ben-Chen and Gotsman even proved the approach to be optimal for a certain class of meshes with a natural geometry distribution [Ben-Chen and Gotsman 2005].

### 1.3.2 Geometry-driven algorithms

Purely connectivity-based approaches are combinatorial in nature, which is a property that some find more elegant and attractive. However, this has two major drawbacks:

1. Mesh compression remains limited by geometry compression rates, which are limited by the order in which the mesh is traversed. The geometry to connectivity bit rate ratio is usually around 10. This is a good reason to try to design algorithms specifically to compress the geometry of a mesh without constraints arising from connectivity coding, hopefully with results that surpass those of connectivity-constrained geometry coding. The connectivity compression is then subordinate to the geometry coding method.

2. The connectivity-based algorithms usually deal with meshes that are *manifold* and *connected*. To deal with non-manifold elements, these elements usually have to be duplicated, which adds compression overhead [Gueziec et al. 1998; 1999; Isenburg and Gumhold 2003]. In some cases, the mesh to compress will be a *triangle soup*, i.e. a mesh highly non-manifold and non-connected. Most connectivity-based algorithms do not deal efficiently with these situations. Geometry-based algorithms, however, that do not traverse the mesh in the same way, usually handle this situation nicely with no difference with a manifold mesh.

As in the case of connectivity-driven compression, both single-rate and progressive methods have been proposed to deal with geometry-driven compression. On the one hand, constrained remeshing single-rate methods start form a point cloud and succinctly encode the information that is needed to constrain a meshing algorithm so that it outputs the original mesh. On the other hand, progressive tree-based methods interleave point cloud and connectivity coding. In all cases, the basis is an algorithm to encode a point cloud.

#### 1.3.2.1 Coding point clouds

Interest in point clouds goes largely beyond mesh compression. Point clouds appear in various areas of science, e.g. astrophysics, molecular dynamics, medical imaging,... In the field of visualization we are interested in, a recent trend consists in directly visualizing point clouds instead of meshes (see e.g. [Hopf and Ertl 2003; Rusinkiewicz and Levoy 2000]). Different algorithms for point cloud compression have been designed to tackle the problems inherent to these domains [Gumhold et al. 2005; Schnabel and Klein 2006; Huang et al. 2006; 2008]. In the following, our goal is not to review all these different methods. Instead, we introduce a particular kind of approaches that most geometry-driven mesh compression algorithms and a lot of point cloud coding algorithms take as starting point [Lewiner et al. 2005; Gandoin and Devillers 2002; Peng and Kuo 2005; Schnabel and Klein 2006; Huang et al. 2006; 2008; Marais and Gain 2007; Chaine et al. 2007]. Note, however, that constrained remeshing algorithms do not postulate a particular point cloud compression algorithm, and therefore any of the aforementioned algorithms can be used in conjunction.

Most approaches for point cloud compression use some kind of hierarchical subdivision of the space, either kd-trees [Gandoin and Devillers 2002; Lewiner et al. 2005] or octrees [Peng and Kuo 2005; Schnabel and Klein 2006; Huang et al. 2006; 2008]. The tree is built by progressively subdividing the space. Each non-empty node $N$ in the tree with associated spatial domain $D_N$ has two children $N_L$ and $N_R$ whose associated domains are obtained by splitting $D_N$ in two. The children nodes are labeled with the number $P_{N_L}$ and $P_{N_R}$ of points that fall in $D_{N_L}$ and $D_{N_R}$ respectively (see Figure 1.17). The process is stopped whenever the size of $D_N$ reaches the desired precision. Then the tree exactly represents the point set. This approach is inherently progressive as on each level of the tree, a coarse approximation of the dataset can be obtained by placing the $P_N$ points at the center of the domain $D_N$.

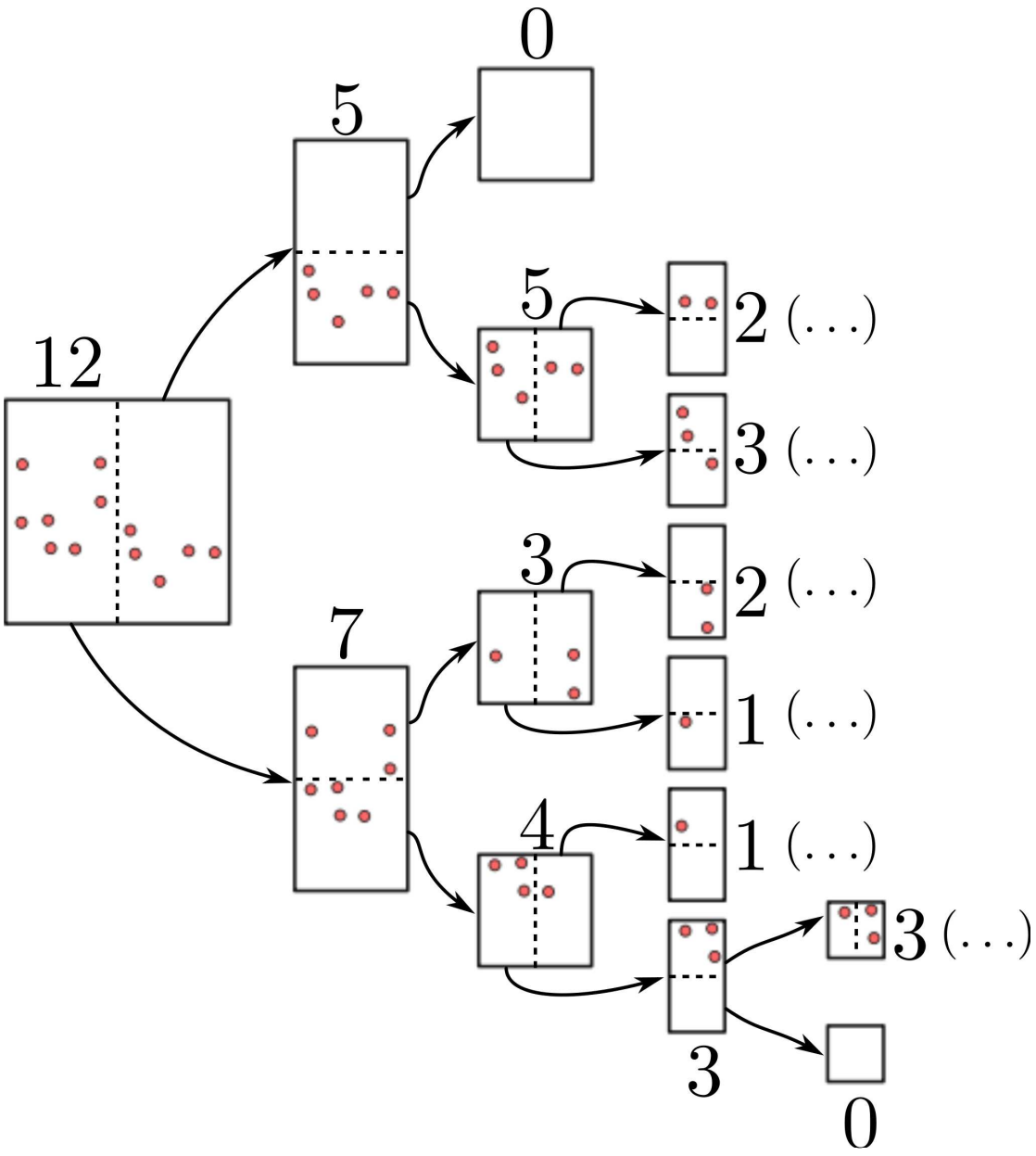


Figure 1.17: Coding point clouds: The point cloud in red is represented using a spatial kd-tree. Using the method of [Gandoin and Devillers 2002], the associated sequence of codes is $\{12, 5, 0, 5, 2, \dots\}$.

Coding the point set is then a matter of efficiently coding the tree. In their kd-tree approach, Gandoin and Devillers [Gandoin and Devillers 2002] code the tree in depth-first order. To code the label of a node $N_L$ with parent $N$, only $log_2(P_N+1)$ bits are required, since $P_{N_L} \leq P_N$. In addition, the label $P_{N_R}$ of the other child of $N$ can be deduced from that of the parent domain ($P_N = P_{N_L} + P_{N_R}$), therefore there is no need to code it explicitly. Another choice consists in coding an *occupancy* bit for each child specifying whether it is *occupied* or *empty* [Lewiner et al. 2005; Schnabel and Klein 2006; Huang et al. 2006; 2008]. Some methods decrease the bitrate further by using information from the occupancy of the neighbourhood of a each node to predict occupancy for its children [Schnabel and Klein 2006; Huang et al. 2006], based on the fact that the finer levels usually introduce changes of small amplitude.

These schemes have a compression performance that depends on the distribution of the points in space, with very non-uniform distributions resulting in the best compression ratios. For typical point clouds representing surfaces, these methods achieve bit rates of around 10 to 16 bits per vertex depending on the point density, which is more efficient that connectivity-driven geometry prediction.

#### 1.3.2.2 Single-rate: Constrained Remeshing

Meshes representing scanned surfaces are usually generated from the sampled surface point cloud using a deterministic method. Therefore, if both the point cloud representing a surface and the meshing algorithm used to generate the connectivity are known, it is possible to code the mesh connectivity using 0 bits. Constrained remeshing approaches use this idea to encode mesh connectivity. It is obviously not possible to encode the algorithm used to generate the mesh, for several reasons. First, a description of the algorithm may cost more than directly specifying connectivity – i.e. the Kolmogorov complexity of the mesh connectivity may be higher that the connectivity compressed using a known coder. Then, the meshing process may not be totally deterministic, for example if the connectivity has been manually modified after automatic meshing, or if the mesh was built by a human modeler. However, as the goal is usually to generate meshes with triangles of high quality, all the algorithms that generate a mesh from a given point cloud will output meshes that are similar to a certain extent. Constrained remeshing approaches exploit that property by coding how the connectivity of the mesh to compress differs from that of a mesh generated from the same point cloud, but using an arbitrary deterministic meshing algorithm. The efficiency of the compression will then depend on how accurately this meshing algorithm is able to predict the connectivity of the mesh from the point cloud. In the limit, if the connectivity of the input mesh corresponds exactly to the *guess* made by the decompressor, then coding the connectivity will essentially be free.

There exist three constrained meshing compressors. All start from a point cloud $V$ representing the vertices of the mesh, but they use different remeshing algorithms, and specify differences between the guessed and actual mesh connectivity in a different manner.

- GEncode [Lewiner et al. 2005] maintains a list of *active edges* which are the boundary edges of a *processed region*. At each step, the algorithm will either *attach a triangle* to one of the active edges, thus growing the processed region, or *remove an active edge*. First, the decoder picks a *focus edge* $e$ in the list of active edges in a deterministic way (e.g. the longest edge). Then, it ranks the vertices $v \in V$ in a list $L$, according to a certain geometric criterion $G(e, v)$ (e.g. circumradius, see Figure 1.18, top). The third vertex $w$ of the triangle $e + w$ to add to the processed region can be determined by its rank in $L$. By a careful choice of the geometric criterion, the distribution of ranks can be made to be heavily biased towards 0, resulting in a very low entropy. To avoid having to rank all the vertices in $V$, Lewiner *et al.* transmit along with the rank of $w$ a range of admissible values for $G(e, w)$. All vertices for which $G(e, v)$ is outside this range are not ranked. Connectivity compression ratios vary a lot depending on the models, from nearly 0 to 6 bpv, with an average of 1.7 bpv. Because there are no specific assumptions on the mesh, GEncode can deal with non-manifold meshes.

- Marais and Gain [Marais and Gain 2007] use a similar approach, but they rank the vertices by their distance to a prediction of $w$ obtained from the triangle adjacent to $e$ in the processed region. The performance is similar to GEncode, with respectively 0.6 and 1.8 bpv on average for very regular and irregular meshes.

- Convection reconstruction [Chaine et al. 2007] uses a different approach. Instead of growing a processed region from an initial triangle, the algorithm maintains a surface $S$ that is initially the convex hull of the point set and evolves towards the input mesh. The process can be seen as a progressive carving of the volume $V_S$ delimited by the surface $S$. The volume $V_S$ is tetrahedrized using the Delaunay tetrahedrization of the point cloud. At each step, the algorithm picks a triangle of $S$ and removes the associated tetrahedron if and only if the associated Gabriel half-sphere contains the fourth vertex of the tetrahedron (see Figure 1.18, bottom, for an illustration in 2D). To make sure that $S$ evolves towards the input mesh $M$, this algorithm must be guided to sometimes force the removal of a tetrahedron that would not have been removed by the convection process, or avoid removing a tetrahedron that would have been removed but whose facet belongs to $S$. To do that, the coder encodes the steps at which such exceptions occur. As this happens very rarely during the convection process, this represents a small amount of data. As this process only works for manifold Delaunay meshes, the algorithm must be modified to handle non-Delaunay and non-manifold meshes. The convection reconstruction algorithm can achieve very low bitrates of around 0.1 bpv for very finely sampled meshes, and still performs quite well in the general case (around 2 bpv). However, the algorithm remains slow (around 1000 vertices per second for both compression and decompression) and uses a large amount of memory.

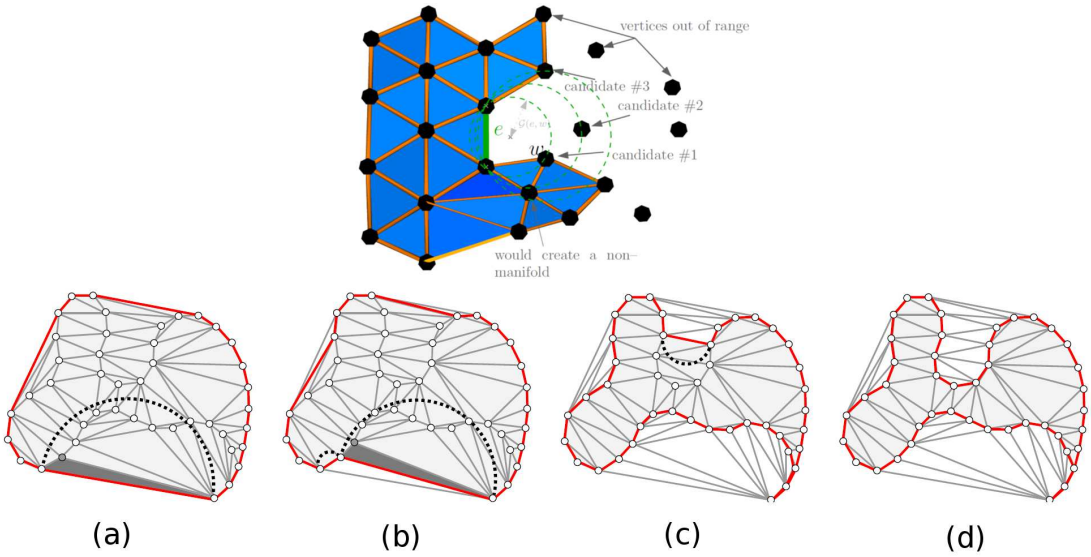


Figure 1.18: The two different kinds of constrained remeshing approaches: Gencode [Lewiner et al. 2005] (top) grows the boundary of a processed region by specifying which vertex forms a triangle with the focus edge from a ranked list of best candidates. Convection reconstruction [Chaine et al. 2007] (bottom) carves the convex hull of the vertices to recover the original mesh (here with a curve in 2D): (a) initial convex hull, (b) the curve locally evolves at the level of an edge iff the half-circle associated with this edge is not empty, (c) result of the initial convection process, (d) the convection process is locally enhanced to hollow a pocket out. Figures from [Lewiner et al. 2005] and [Chaine et al. 2007].

#### 1.3.2.3 Tree-based Progressive coding

We have seen that most point cloud compression approaches use a tree-based hierarchical subdivision of the space, and are thus progressive in nature. Gandoin and Devillers [Gandoin and Devillers 2002] have taken advantage of the inherent progressiveness of geometry coding and propose to add connectivity information inside the tree used to represent geometry. They consider the subdivision of a parent cell in two subcells as a vertex split. In contrast with traditional progressive approaches based on vertex splits, where the position of the split vertices $v_a$ and $v_b$ must be specified to the decoder, here their position is known to the decoder (it is simply the center of each subcell). Hence, only the *connectivity refinement* codes need to be transmitted to the decoder. For the typical *edge expansion* operation (the inverse of the edge collapse operation), the connectivity information only consists in the two split edges $e_1$ and $e_2$ that need to be expanded into faces. Because the position of split vertices is known to the decoder, this information can be used to predict which of the edges adjacent to the parent vertex $v$ are the split edges. Using this scheme, an edge collapse operation is coded using less that 3 bits per vertex.

As the *edge collapse* operation only enables compression of manifold meshes, they use a second operation, *generalized vertex split* (inverse operation of *vertex unification*), to deal with non-manifold situation. This is a more expensive operation that costs approximately 8 bits per vertex, but enables the compression of any mesh – even triangle soups.

Peng and Kuo [Peng and Kuo 2005] have later improved this approach by using octrees, leading to better compression rates for both connectivity and geometry, with a global improvement of 10 to 60% depending on meshes.

These methods have compression rates that are usually better than connectivity-driven progressive approaches for both connectivity and geometry. However, meshes of intermediate level exhibit highly visible quantization artifacts (Figure 1.19, left). The main advantage of these methods is that they can deal with highly non-manifold meshes.

#### 1.3.2.4 Incremental Parametric Refinement

The main problem with tree-based methods is that the hierarchical decomposition of the space induces high quantization artifacts at lower bitrates, as seen in the previous section. Recently, Valette *et al.* [Valette et al. 2009] remarked that the connectivity information of intermediate levels is meaningless to the user, who is mainly interested in the *geometric error* compared to the original model. However, for low bitrates, tree-based approaches spend most of the bit budget on *connectivity*. Thus, they propose to transmit strictly *no connectivity information* until the last level, and let the decoder itself decide where to refine the mesh based on some geometric criterion. The criterion that they use is *edge length*: At each step, they pick the longest edge, split it in two, adding one vertex and two triangles. Then, edges are locally flipped to satisfy a Delaunay connectivity. This way, the meshes of intermediate level are always of very good quality. Only two pieces of information need to be transmitted: the geometric information corresponding to the position of the newly introduced vertices, and a *split confirmation flag* to indicate to the decoder whether the longest edge must be split or not. At the beginning of the algorithm, the longest edge will be split nearly all the time, so the entropy of the split confirmation flag is very low. At the end, a split is very rare, also leading to a small entropy.

When all vertices have been introduced, the algorithm rebuilds the original connectivity by directing a deterministic edge-flipping algorithm so that it converges to the correct connectivity. This process uses around 3 bits per vertex.

On average, this approach is approximately as efficient as other progressive approaches. However, rate/distortion is drastically better than other methods (Figure 1.20), and visual quality is excellent at low bitrates (Figure 1.19, right). The main drawback of this approach is that there is no guaranteed convergence of the final edge-flipping algorithm. In addition, it can only deal with manifold meshes.

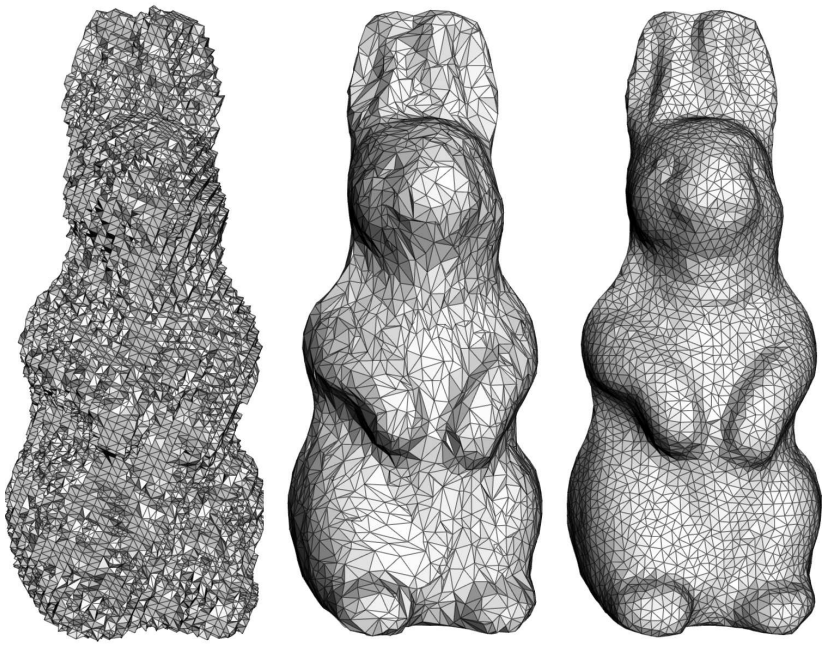

Figure 1.19: Visual comparison of progressive compression artifacts at low bitrates. The left and right images show the rabbit model compressed down to 1 bit per original vertex respectively with Octree [Peng and Kuo 2005] and Incremental parametric Refinement [Valette et al. 2009]. The middle model was compressed using wavemesh [Valette and Prost 2004] at 1.4 bpv. (Image from [Valette et al. 2009])

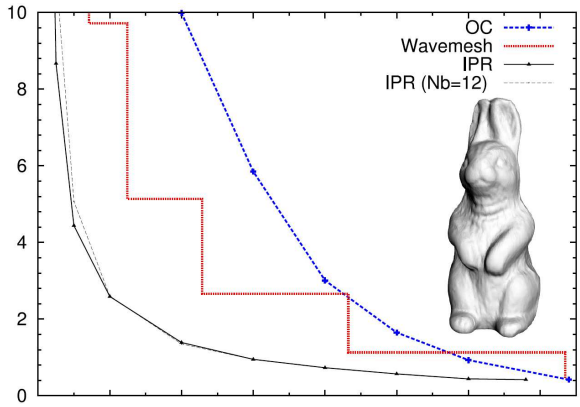


Figure 1.20: Rate/Distortion curves (RMS error vs bitrate) for the *rabbit* model compressed with Octree [Peng and Kuo 2005], Incremental parametric Refinement [Valette et al. 2009] and wavemesh [Valette and Prost 2004]. (Figure from [Valette et al. 2009])

## 1.4 Connectivity-Oblivious Compression

The connectivity-preserving schemes of the previous section are well adapted to scientific visualization, where in addition to the actual signal value, the *mesh connectivity* itself is of interest. For some other computer graphics applications, the mesh is only a necessary *means* for representing a surface, and the connectivity has no particular interest to the user. Only the geometry matters. In games, for example, the mesh is used by the 3D artist to define a character, but for actual rendering, any mesh can be used as long as there are no visual artifacts. In this situation, it is natural to choose a mesh connectivity that is best suited to the envisioned application. The process of modifying the mesh connectivity while keeping a good approximation of its geometry is called *remeshing* [Alliez et al. 2007].

Because mesh compression applications try to shrink the size of a mesh as much as possible, they naturally choose to remesh surfaces with connectivities that contain the smallest possible amount of information. The ideal case is a *totally regular* remeshing. Then, no connectivity at all needs to be transmitted. However, in most cases having only regular vertices is impossible – as can be shown by applying Euler's formula – and connectivity-oblivious compression algorithms strive to introduce the least possible irregularity during remeshing.

Note that connectivity-oblivious algorithms are not only *lossy* in terms of connectivity, but also in geometry. Indeed, in order to achieve a high regularity, remeshing algorithms usually also *resample* the signal. Therefore, the geometry of the mesh is also modified, in the sense that vertex positions are modified. The performance of lossy compression algorithms is measured with respect to the *approximation error* compared to the original. However, one must keep in mind that the original mesh is already an *approximation* of the surface. Therefore, this approximation error is not an accurate measure of the error with respect to the original continuous surface. It is only used for lack of a better tool, since the actual geometry of the original surface is not known.

There exist several connectivity-oblivious algorithms, that employ different remeshing methods to remove connectivity information. The following sections detail these techniques, roughly from most to least regular.

### 1.4.1 Geometry images

The most obvious type of connectivity that is completely implicit is a *regular* connectivity, i.e. a mesh where all (interior) vertices have the same valence. Gu *et al.* remesh the input mesh using a regular quadrilateral grid [Gu et al. 2002]. First, the input mesh is cut to make it homeomorphic to a disk and enable a "good parametrization" of the resulting mesh. To determine a good parametrization that will result in an as-uniform-as-possible sampling, the authors try to minimize the *geometric stretch* [Sander et al. 2002]. The mesh geometry and the normals are then sampled on the grid to form a *geometry image* (see Figure 1.21) and a *normal map* that are compressed using standard image compression techniques (Gu *et al.* use wavelets, but any image compression scheme could be used). To ensure seamless stitching at the cuts, the cut topology is stored alongside the geometry image.

This technique has very good compression ratios and enables efficient rendering because of its inherent cache coherence. In addition, compression is straightforward and efficient using highly optimized image compression algorithms. However, it introduces numerous visual artifacts due to the discontinuity of the information at the cuts and the possibly non-uniform sampling if the parametrization is too stretched. To overcome this problem, various approaches have proposed to decompose the mesh into several charts that are independently mapped and compressed using geometry images, in a uniform [Sander et al. 2003] or adaptative [Yao and Lee 2008] way. These approaches typically improve PSNR by 10 dB for low bitrates compared to the original geometry images. The mapping has also been improved to reduce error for genus 0 meshes [Praun and Hoppe 2003]. This approach was

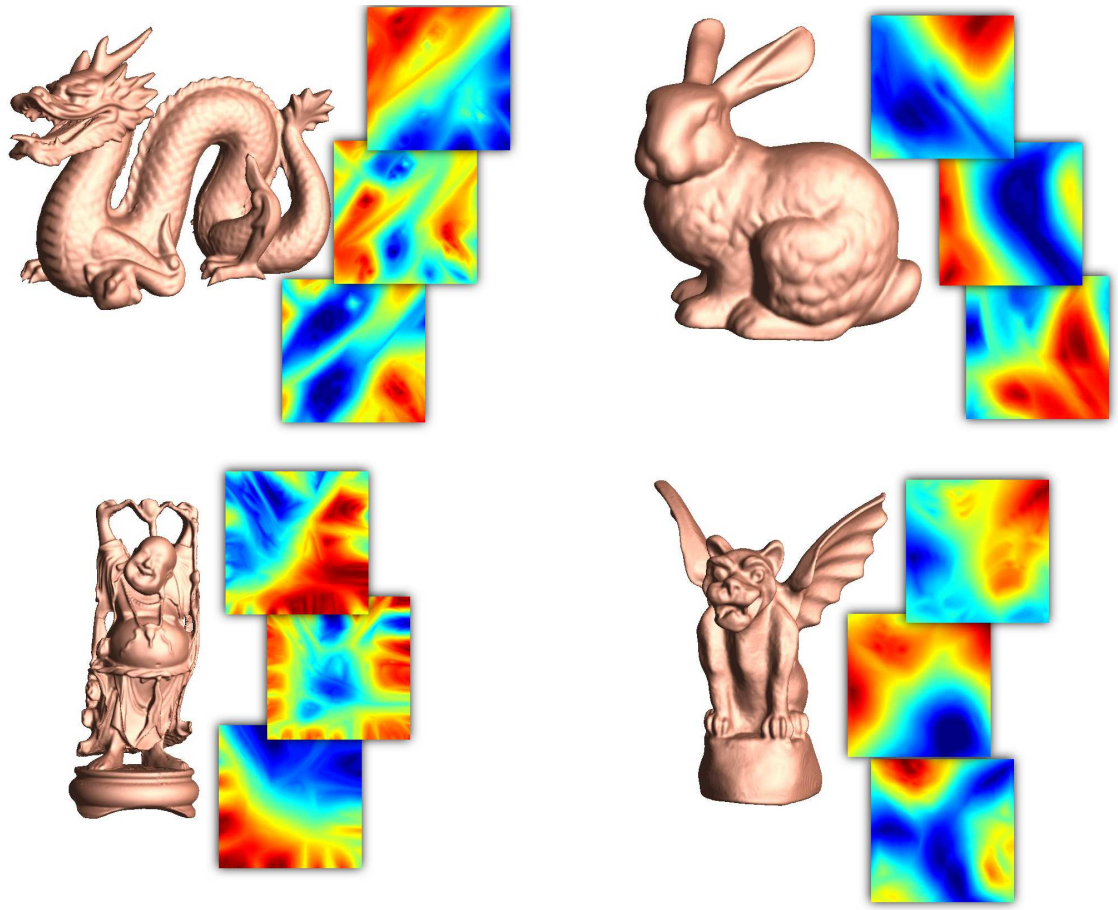

Figure 1.21: Geometry images: the three images alongside each model respectively represent the $x$, $y$ and $z$ components of the geometry, regularly sampled onto a grid.

further extended to produce smooth genus 0 surfaces by taking advantage of the regular quadrilateral parametrization to apply b-spline subdivision to the resulting mesh [Losasso et al. 2003].

### 1.4.2 Subdivision Wavelets

The previous geometry image approach takes advantage of the existence of efficient mathematical tools for the analysis of tensor product spaces (e.g. Fourier/wavelet transforms). The main disadvantage of this approach is that it requires remeshing the surface with a connectivity which may not be adapted to its geometry. Therefore, there has been some effort to extend these tools to handle irregular connectivities. We have already seen in section 1.3.1.3 the extension of the Fourier transform to irregular meshes, that is quite efficient but computationally expensive. The efficiency of wavelets to decorrelate signals defined on grids has motivated their extension to meshes. The mesh wavelet schemes that have been developed deal with *semi-regular* meshes, where nearly all vertices are regular. This means that the original surface has to be remeshed with a mesh having *subdivision connectivity*.

A *semi-regular* or *subdivision* mesh is a mesh that is constructed by recursively subdividing the faces of an irregular *base mesh* $M^0$. There exists several subdivision schemes, the most popular being 4-to-1 subdivision. At each level, a finer mesh $M^{i+1}$ is constructed from $M^i$ by adding new vertices at the middle of each edge, and triangulating the resulting hexagonal faces as in figure 1.22. The newly added vertices will have degree 6. Therefore, in a mesh $M^i$, all the vertices will be regular, except the vertices of $M^0$. The mesh on the right of Figure 1.22 has a semi-regular connectivity.

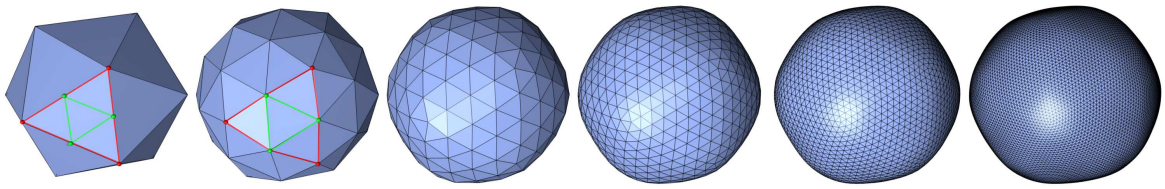

Figure 1.22: Recursive 4-to-1 subdivision of a coarse irregular mesh (left) to obtain a semi-regular mesh (right). The interpolation scheme used here is the *modified butterfly scheme* [Zorin et al. 1996a], and all the wavelet coefficients of this mesh are 0.

As in the tensor product case, there exist both classical filter-bank [Strang and Nguyen 1996] and lifted [Sweldens 1998] wavelets. In the first case, the position of the vertices $\mathbf{p^{i+1}}$ of $M^{i+1}$ is computed from the vertices $\mathbf{p^i}$ of $M^i$ and the wavelet coefficients $\mathbf{d^i}$ using two synthesis filters:

$$\mathbf{p^i} = [P\ Q] \begin{bmatrix} \mathbf{p^i} \\ \mathbf{d^i} \end{bmatrix} \tag{1.6}$$

where $P$ is dependent on the subdivision scheme and $Q$ can be derived from $P$. The analysis phase of the wavelet transform requires the inversion of the previous system, which can be computationally expensive. A more efficient approach uses the lifting scheme [Sweldens 1998]. During synthesis, the position of each new vertex is predicted from the vertices of a given neighbourhood in $M^i$. This step constitutes the *predict* step of the wavelet scheme, and the wavelet coefficients are the differences between the actual and predicted positions. The wavelet is then *lifted* by updating the positions of the vertices of $M^i$ using the vertices of $M^{i+1} - M^i$. This reduces aliasing and ensures that each of the intermediate meshes $M^i$ provides a good approximation of the final mesh. The analysis process simply reverses the two steps. Therefore, no linear system inversion is necessary and the analysis as well as the synthesis can be made in linear time.

Khodakovsky *et al.* have applied these approaches to surface compression [Khodakovsky et al. 2000]. They have used both lifted wavelets built from the *butterfly scheme* [Dyn et al. 1990; Zorin et al. 1996a] and classical wavelets derived from the *Loop* subdivision scheme [Loop 1987]. They found that both approaches yielded approximately the same rate/distortion performance. The Loop scheme has the advantage of better visual results, but this comes at the cost of a longer analysis phase because it uses classical filter wavelets, and thus requires linear system inversion.

The new prediction method exposed in Section 2.3 can be used to derive subdivision schemes. While we obtain the same weights as the subdivision schemes used here, the approach is different and gives another insight on these schemes. Also, we found that some of the subdivision schemes [Labsik and Greiner 2000] are not optimal with respect to the support of the prediction filter, and we use our method to derive a subdivision scheme with the same smoothness of the limit surface, but smaller support (see Section 2.3.5.2).

### 1.4.3 Normal Meshes

In the wavelet schemes presented above, wavelet coefficients consist in a three-dimensional vector. It may be expressed in a global or local frame, but the coder needs to transmit a normal and tangential component for each vertex. Guskov *et al.* have proposed another approach, that minimizes the tangential component of the wavelet coefficients [Guskov et al. 2000; Friedel et al. 2004]. They dub their representation Normal Mesh, because most detail coefficients only lie in the *normal* direction. This reduces the information to just *one* coefficient per vertex, drastically reducing the number of bits required to represent the surface [Khodakovsky and Guskov 2003; Payan and Antonini 2005]. However, using normal coefficients prevents the use of lifting: If an update step modifies the position of neighbouring vertices, then the normal to the surface will be modified, and the wavelet coefficients will no longer lie in the normal direction. Thus, the most efficient scheme uses the *unlifted butterfly wavelet*, which is an *interpolating scheme*: As there is no update step, $M^{i+1}$ interpolates $M^i$.

### 1.4.4 Remeshing to optimize connectivity-aware compression

We have seen in Section 1.3.1.1 that most compression methods represent connectivity with a bit rate that directly depends on the regularity of the mesh. Based on this property, a number of methods remesh the surface so that the resulting mesh is as regular as possible, either globally [Surazhsky and Gotsman 2003; Alliez et al. 2005; 2002] or in a piecewise manner [Attene et al. 2003; Szymczak et al. 2002]. In contrast to previous methods, the connectivity is not implicit, and has to be specified to the decoder. However, this may be done very efficiently is the mesh is regular enough. In practice, global remeshing using [Surazhsky and Gotsman 2003] enables bit rates of around $15-17$ bpv with an error that is less than 0.1% (with less than 1 bpv for connectivity). For piecewise remeshing, Szymczak *et al.* report bitrates of 4 bpv with 0.02% error on average [Szymczak et al. 2002], while SwingWrapper [Attene et al. 2003] yields the best results, that are competitive with wavelet approaches.

### 1.4.5 Discussion

The above schemes can be classified into three groups, depending on which information they code:

- Wavelet schemes [Khodakovsky et al. 2000; Guskov et al. 2000] only transmit the connectivity of the base mesh. Because the connectivity of the successive subdivided meshes is implicit, these schemes can save some bit budget by not transmitting it. However, they still transmit geometry *and* parameter information because the position of each vertex is represented by three coefficients.

- Single-rate remeshing approaches try to minimize connectivity information, however they still transmit connectivity in a global, single rate manner, in contrast with wavelet methods. Therefore, they are usually less effective than the latter. SwingWrapper [Attene et al. 2003] is an exception, because it does not code *parametric* (or tangential) information, but only transmits one coefficient per vertex. This enables it to beat wavelet schemes (Figure 1.24). However, although this method has better rate/distortion performance than subdivision wavelet methods, it fails to provide the attractive progressivity which enables the *decoder* to choose the quality of approximation.

- Geometry Images [Gu et al. 2002] and Normal Meshes [Guskov et al. 2000] both save on connectivity and parametric information. Normal meshes can be specified with only the connectivity of the base mesh, and no parameter information since nearly all coefficients are in the normal direction. Geometry images have both implicit connectivity and parametric information, and thus only transmit geometry. Both approaches are progressive (by mipmapping geometry images), but normal meshes have the advantage of adaptivity and more uniform error, as well as slightly better rate/distortion (figure 1.23). On the other hand, geometry images can take advantage of the structure of graphics devices to enable efficient rendering.

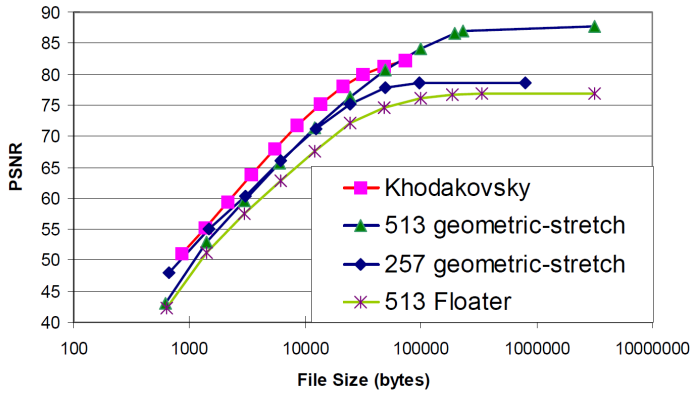


Figure 1.23: Comparison of the rate/distortion performance of Geometry Images [Gu et al. 2002] and Subdivision Wavelets [Khodakovsky et al. 2000]. (Figure from [Gu et al. 2002])

## 1.5 Discussion

**Connectivity vs. Geometry:** In this chapter, we have seen that in the last fifteen years, mesh compression has been the target of a lot of research effort. Various approaches have been proposed to represent meshes in a very succinct way. Most of the work was oriented towards reducing the size of the connectivity information, that represents the highest cost in a traditional indexed structure. Pure connectivity compression algorithms have reached a mature state where virtually all meshes – from very regular subdivision meshes to triangle soups – can be compressed using a very small number of bits per vertex. Geometry compression schemes are drastically less efficient, and most of the bit budget is now dedicated to the representation of geometry – and other data associated with vertices. The ratio between geometry and connectivity in a typical compressed mesh is usually about ten to one. Obviously, geometry should now be the target of further effort to decrease bit rates. In this dissertation, we propose two different approaches to compress mesh geometry. In Chapter 2, we remain within the classical linear prediction paradigm that nearly all the compression methods presented in this Chapter use, but we give a rationale for choosing prediction weights. When designing

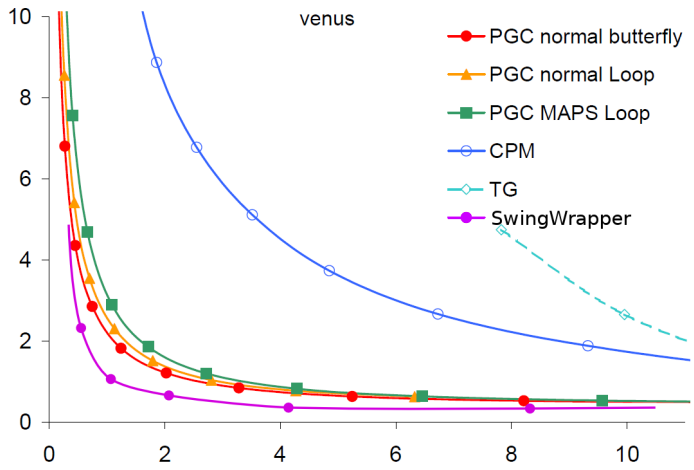


Figure 1.24: Comparison of the rate/distortion performance of SwingWrapper [Attene et al. 2003] and Normal Meshes [Khodakovsky et al. 2000]. (data from [Attene et al. 2003] and [Khodakovsky and Guskov 2003])

the random-accessible scheme presented in Chapter 5, we have used a geometry coding method that departs from the approaches presented here, since we use prediction within chains of vertices. This approach had only been used for point cloud compression [Gumhold et al. 2005]. However, although these methods are quite efficient, geometry remains the most important part of the information.

**Lossless vs lossy:** By remeshing, some algorithms are able to provide far better compression rates. They increase the regularity of the mesh – and therefore decrease the coding cost – by removing the connectivity information. But remeshing has another very important side effect on geometry: It removes *parameter* information (i.e. the components of a vertex position that lie in the tangent direction of the surface). That also reduces the cost for geometry coding – but still this component remains predominant in the bit budget.

Although its results in terms of compression rates are more appealing, lossy compression does not fit every application. There is no doubt that both lossy and lossless compression approaches will subsist in the future, in the same way as the efficiency of lossy image compression schemes like *JPEG* did not kill lossless coding. The introduction of compression artifacts is the main reason – for example, lossy wavelet compression has the tendency to smooth out models, which is a problem in the case of CAD models with sharp features. But there are several other cases where lossless instead of lossy compression will be chosen. In particular, lossy compression poses the problem of *generation loss* – i.e. how meshes behave when they are compressed and decompressed a large number of times, as is the case when an artist designs a computer graphics mesh. In that case, lossless compression is the option to choose. In the work presented here, we choose to address only lossless coding, as this is the safest and most general approach.

**Triangle vs Polygon compression:** A lot of mesh compression approaches only deal with triangle (resp. tetrahedral) meshes. Their argument is that any polygon (resp. polyhedral) mesh can be triangulated (resp. tetrahedralized) and thus these compression methods are general. However, Isenburg has shown [Isenburg 2005] that compressing polygon meshes in their original form can lead to higher compression ratios, for two different reasons. First, no information needs to be stored to code how to retrieve the polygon mesh from the triangulated version. Second, polygon flatness

and regularity can be exploited to improve geometry compression. This is particularly true for hexahedral (volume) meshes, that can sometimes be compressed using only a few bits per vertex *including geometry*, as we will see in Chapter 4. In addition, we would say that being able to directly handle polygon meshes without having to go through a triangularization step is a very interesting feature as far as software engineering is concerned. Therefore, we have striven to design algorithms that are as generic as possible in the way in which they handle polygon meshes. Apart from our streaming compressor (Chapter 4), that only handles hexahedra, the two other algorithms that we propose for geometry prediction (Chapter 2) and random-accessible compression (Chapter 5) handle general meshes.

**Large meshes:** The main focus of this dissertation is *large meshes*. In this Chapter, we have deliberately chosen to ignore the problems posed by mesh size, and we have supposed that meshes could be fully loaded in memory. This way, the various problems posed by mesh compression could be presented in a simple manner. When this assumption no longer holds, the algorithms described in this chapter cannot be used. We dedicate the Chapter 3 to the study of the schemes that handle large meshes. In addition, we propose two schemes for the compression of large meshes. Chapter 4 presents a hexahedral mesh compressor capable of compressing and decompressing meshes on the order of several million hexahedra in a few minutes using only a few megabytes of memory. On the other hand, we present in Chapter 5 a surface mesh compression method that addresses efficient decompression of meshes too large to fit in the memory of the client device.

# Chapter 2

# Improved Prediction for Geometry Compression

In an indexed representation of a mesh, the connectivity accounts for a bigger part of the bit budget than the geometry information. However, typical compression algorithms do a far better work at compressing connectivity, resulting in connectivity sizes that are usually a fraction of geometry bitrates. For example, degree coders offer a compression performance that is around $1.5-3$ bits per vertex for connectivity, but raises to roughly 16 bits per vertex to represent the geometry for the same models. Therefore, compressing the geometry of meshes remains at stake for mesh compression.

We have seen in Section 1.3.1.3 that most connectivity-driven compression methods use prediction to compress the geometry of the mesh. Among the great classes of methods for geometry compression, the only ones that do not use the predict/correct paradigm are the spectral methods based on the work of Karni and Gotsman ([Karni and Gotsman 2000], Section 1.3.1.3) and the tree-based point cloud compression techniques ([Gandoin and Devillers 2002], Section 1.3.2.

The vast majority of the predictors used are *linear*, which means that the predicted position of the new vertex is a linear combination of the positions of neighbour vertices. This includes virtually all region-growing (single-rate), as well as progressive methods based on both edge collapse/vertex split and vertex decimation. Wavelet methods also use linear prediction. Some approaches use more sophisticated non-linear prediction schemes (e.g. [Marais and Gain 2007; Lee 2002; Gumhold and Amjoun 2003]), but this comes at the cost of increased complexity with no clear advantage [Lee 2002].

Linear schemes are attractive for several reasons:

- **Speed:** Linear prediction requires few arithmetic operations, namely $n$ multiplications and $n-1$ additions if $n$ is the number of neighbour vertices used for prediction. In comparison, non-linear predictive methods are a lot more expensive.
- **Simplicity:** For the same reasons, these schemes are very easy to implement.
- **Inversion:** A linear prediction scheme can be easily inverted by solving a linear system. For classical (non-lifting-scheme) wavelet schemes, this is a very important property, because the analysis phase is based on an inversion of the prediction scheme used during the synthesis phase [Khodakovsky et al. 2000].

In the following sections, we first detail the various approaches used to predict a vertex from its known neighbours. Then, we extend to the irregular case the *spectral* approach of Ibarria *et al.* [Ibarria et al. 2007] that has been used to derive very efficient predictors in the regular grid and

polygonal case. We show that this extension can perform well in some specific cases, but generally fails to provide good extrapolation results. Then, we introduce a new formalism based on the local Taylor expansion of the geometry of the mesh. We show that this method can be used to derive various existing linear predictors in a generic way, and we improve upon some of them.

## 2.1 Linear Prediction

### 2.1.1 Parallelogram Rule and Extensions

For single-rate compression, the *Parallelogram Prediction (PP)* of Touma and Gotsman [Touma and Gotsman 1998] is the most widely used prediction. The position of a vertex is predicted as completing the parallelogram containing the three vertices of an already decoded adjacent triangle. A residual vector (i.e. the difference between the actual and predicted position) is stored, that can be expressed either in a global [Touma and Gotsman 1998; Alliez and Desbrun 2001b] or local [Alliez and Desbrun 2001a; Lee et al. 2002; 2010a] reference frame. This residual is usually small and can be compressed using an entropy coder. This method typically results in compression ratios of about $16-18$ bits per vertex at 12 bits quantification. It has the advantages of simplicity and speed (it only requires one addition and one subtraction), which makes it and its derivatives the most widely used methods for single-rate geometry compression.

The parallelogram rule can be seen as a linear predictor which assigns the weights $\{-1,1,1\}$ to three already visited vertices of the neighbourhood which form a triangle (Figure 2.1, top left). It is easy to verify that these prediction weights are optimal (in the *least square* sense): finding the weights that minimize the 2-norm of the prediction residuals is a simple linear problem. In our experiments, the optimal weights were constantly $\{-1,1,1\}$ for all our test meshes.

The prediction error of the parallelogram rule is highly dependent on the order in which the mesh is traversed. Various methods [Kronrod and Gotsman 2002; Chen et al. 2005] have applied the parallelogram rule along a geometry-driven traversal of the mesh to reduce this error. They obtain 32% improvement on average.

Some linear predictors use different neighbourhoods. Sim *et al.* [Sim et al. 2003] introduce the *dual parallelogram prediction (DPP)*. When possible, they predict the position of a vertex as the average of two parallelogram predictions (Figure 2.1, top center). Approximately 75% of the vertices of their test meshes can be predicted this way. This improves compression ratios by 3% over simple PP.

In a later work, Kälberer *et al.* use the same neighbourhood but average *three* parallelogram rules [Kälberer et al. 2005], the first two being the previous parallelogram rule, and the third being applied across a virtual edge joining the two outer vertices (dashed line on Figure 2.1, top right). We call this rule *Freelence Dual Parallelogram Prediction* (*fDPP*). They report improved compression rates compared to applying a simple parallelogram rule for prediction, but they only compare with the parallelogram rule (we will see later in this chapter that their scheme also performs better than the DPP). They also discuss another stencil which would take into account the last vertex of the one ring of the center vertex of the DPP neighbourhood if that vertex is of degree 6. They experimentally determine weights for this neighbourhood, and conclude that these weights depend on the type of model (irregular, subdivision, CAD). They do not give any method to derive these weights in a theoretical, consistent manner.

Cohen-Or *et al.* [Cohen-or et al. 2002] take the parallelogram averaging idea one step further by using *all* possible parallelogram predictions around a vertex. They proceed in two steps. The first one starts with the connectivity of the mesh and the position of a small number of vertices. The mesh is processed using several passes. Each pass predicts each vertex using the average of *all* the possible parallelogram predictions around this vertex (*Average Parallelogram Prediction* or *APP*), assuming

that the displacement is zero. This predictor is shown on the bottom left of Figure 2.1. A smooth approximation to the mesh is thus obtained. In a second step, they apply a scheme similar to [Sim et al. 2003] that uses a dual parallelogram rule to correct the position of the vertices. This scheme gives the best result for lossless single-rate geometry compression.

Sorkine *et al.* use a global optimization approach to reduce the visibility of coordinates quantization by applying quantization in the *prediction residual* space (delta coordinates) instead of the original coordinates space [Cohen-or et al. 2002]. This results in quantization errors that are mostly in the low frequencies (see Section 1.3.1.3). They predict the position of a vertex as the average of the positions of the vertices in the one-ring. However, they also study the reduction in prediction error when using the average of all parallelogram predictions and reach the same conclusions as [Cohen-or et al. 2002].

For progressive compression, Pajarola and Rossignac remarked that the APP did not give the best results [Pajarola and Rossignac 2000a]. They use a variation of it (that we call CPM after the name of their compressor) where they use the weights $-\frac{\alpha}{K}$ and $\frac{1+\alpha}{K}$ instead of simply $-\frac{1}{K}$ and $\frac{2}{K}$ (where $K$ is the number of parallelogram predictions). They experimentally determined that $\alpha = 0.15$ performed well on average (Figure 2.1, center right).

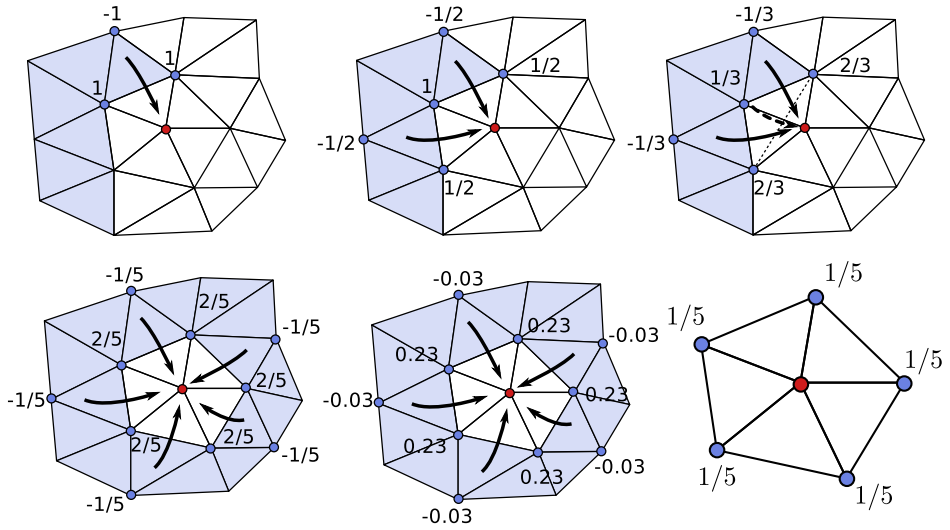


Figure 2.1: Linear prediction rules, in reading order: PP [Touma and Gotsman 1998] , DPP [Sim et al. 2003], FreelenceDPP [Kälberer et al. 2005], APP [Cohen-or et al. 2002], CPM [Pajarola and Rossignac 2000a], and Barycentric [Alliez and Desbrun 2001a] . The vertices of the triangles in blue are known to the decoder.

### 2.1.2 Polygon meshes

Isenburg and Alliez [Isenburg and Alliez 2002b] have successfully extended the use of the parallelogram rule to compress quad meshes. They even improve the compression ratios (by about 10 to 40%) compared to triangulated versions of the same quad meshes by applying the parallelogram rule *within* rather than *across* quads. This works because quads are generally flat.

In a later work, Isenburg *et al.* extend this idea to predict the geometry of polygons of higher degree [Isenburg et al. 2005a]. They suppose that in a polygon of degree $K$, there are $N$ vertices whose positions are already known $(v_1, \ldots, v_N)$, and want to predict the position of the $(N+1)$-th vertex as a linear combination of these $N$ vertices. They first introduce the *fourier basis* $(\mathbf{b^i})_{i\in[\![1;K]\!]}$

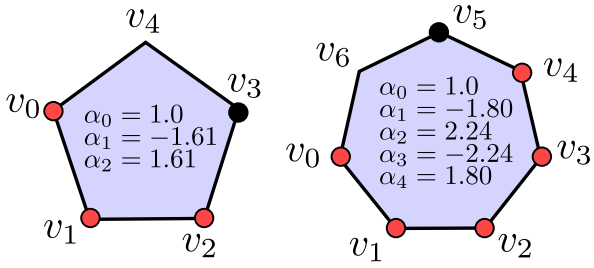


Figure 2.2: Example of linear prediction rules for high degree polygons with the method of [Isenburg et al. 2005a]. The vertices in red are known to the decoder, and the vertex in black is being predicted. The weights are given in the center of each polygon.

of the polygon, and represent the vertex geometry $\mathbf{v}$ as a linear combination of the Fourier basis vectors:

$$\mathbf{v} = \sum_{i=1}^{K} c_i \mathbf{b^i} \tag{2.1}$$

Then, they determine the coefficients $(c_i)$ such that:

1. Equation (2.1) is verified.

2. The energy of the geometry is localized in the low frequencies.

As there are $N$ fixed vertex positions, the first $N$ lowest frequency coefficients $c_1, \ldots, c_N$ are fixed, and the rest are 0. The non-null, low frequency coefficients can be determined using the equation (2.1):

$$\forall j \in [\![1; N]\!], \quad v_j = \sum_{i=1}^{K} c_i b_j^i = \sum_{i=1}^{N} c_i b_j^i \tag{2.2}$$

This is a linear system of size $N \times N$, and we let $c_i = \sum_{j=1}^{N} \kappa_{i,j} v_j$ its solution. Then, the $(N+1)$-th line of equation (2.1) can be expanded as:

$$\begin{array}{rcl} v_{N+1} & = & \sum_{i=1}^{K} c_i b_{N+1}^i \\ & = & \sum_{i=1}^{N} c_i b_{N+1}^i \\ & = & \sum_{i=1}^{N} \sum_{j=1}^{N} \kappa_{i,j} v_j b_{N+1}^i \\ & = & \sum_{j=1}^{N} \sum_{i=1}^{N} \kappa_{i,j} b_{N+1}^i v_j \end{array} \tag{2.3}$$

The above equation gives the prediction weights $\alpha_j = \sum_{i=1}^{N} \kappa_{i,j} b_{N+1}^i$. We did not consider here the case when basis vectors $b_N$ and $b_{N+1}$ are associated with the same frequency, which, for symmetry reasons, prevents discarding either one of the two vectors. Isenburg *et al.* deal with this problem by determining prediction rules when $N-1$ vertices are known, and then using a linear combination of the two prediction rules when either vertices $(2, \ldots, N)$ or $(1, \ldots, N-1)$ are known. The coefficient $\lambda$ of this linear combination is chosen such that the norm $\|\lambda\alpha^{\mathbf{1}} + (1-\lambda)\alpha^{\mathbf{2}}\|$ of the resulting prediction weights is smallest.

This method results in small prediction residuals. However, we will see in section 2.3.4.4 that our method can result in even smaller prediction error.

### 2.1.3 Prediction for Regular Grids

Mesh and image compression traditionally have a lot of ideas in common. Prediction is widely used for image compression, and the parallelogram rule finds its regular counterpart in the JPEG-LS predictor [Weinberger et al. 2000]. It is thus very tempting to adapt other predictors used for image compression.

For prediction on regular grids (e.g. images), a wealth of predictors exist, because the neighbourhood of a specific sample always has the same structure. Recently, Ibarria *et al.* [Ibarria et al. 2007] have introduced a spectral interpolation method which enables prediction when an irregular set of the 3x3 stencil of neighbouring samples is known. This is the case, for example, during scanline or progressive transmission. Similarly to [Isenburg et al. 2005a], they decompose the stencil on the 9 vectors of the Discrete Cosine Transform basis. If there are $k$ unknown samples, they set the $k$ highest frequencies to 0, and solve the resulting system analytically. Thus, they determine the weights to predict one unknown sample from the known samples.

Their approach finds the weights that give the *smoothest* prediction (in terms of frequency) that fits the known samples. They have computed a lookup table that gives the weights for a given stencil configuration. However, this approach is not directly applicable in the case of mesh compression, because the connectivity itself of the neighbourhood changes from vertex to vertex, and not just the number of known and unknown samples. We will see in Section 2.2 that it can be extended to deal with general meshes, but with limited success.

### 2.1.4 Prediction for Subdivision Meshes: Building Wavelets

Wavelet-based approaches intrinsically use linear prediction to define their filters. To reconstruct the final mesh, a base (usually irregular) coarse mesh $M^0$ is progressively subdivided. At each subdivision level, a finer mesh $M^{i+1}$ is constructed from $M^i$ by adding new vertices, either at the middle of each edge [Dyn et al. 1990; Zorin et al. 1996a; Loop 1987; Kobbelt 1996] or at the center of each face [Kobbelt 2000; Labsik and Greiner 2000]. The position of each new vertex is predicted linearly from the vertices of a given neighbourhood in $M^i$. Among these techniques, *interpolating wavelets* [Zorin et al. 1996a; Labsik and Greiner 2000] are of particular interest, because they enable building non-lifted wavelets suitable for the very efficient *normal mesh* representation (see Section 1.4.3).

A necessary condition for $C^2$ smoothness of the limit surface of interpolating subdivision process is that the wavelet prediction step reproduces polynomials of degree 2 [Warren 1995], i.e. if the points used for prediction are samples of a polynomial of degree 2, then the predicted point will also be a sample of the same polynomial. A detailed explanation on how this property can be used to derive prediction weights is given in [Zorin et al. 1996b].

Most of the tools used to prove results in the subdivision field are based on the concept of *subdivision matrix*, therefore we briefly define it here. For more details, the reader can refer to [Zorin et al. 1996b]. A *$K$-regular* mesh is a triangulation $M^i$ of the plane where all vertices are regular (in valence) except one (we will call that vertex $v_0$), that has valence $K$, and all the triangles are similar. In the following, we will take $K = 5$ (Figure 2.3), but the construction is similar for other values of $K$. The result $M_{i+1}$ of applying one step of the subdivision process of the *modified butterfly scheme* [Zorin et al. 1996a] is shown in the center of Figure 2.3. Because the prediction of each new vertex is linear, there exists a linear operator $S_{\text{inf}}$ (of infinite dimension) such that the points $P^{i+1}$ of $M^{i+1}$ can be expressed as a function of the points $P^i$ of the original mesh $M^i$ as:

$$P^{i+1} = S_{\text{inf}} P^i \tag{2.4}$$

Note that the new vertices are regular, and the new mesh is a simple shrinking of the original mesh by a factor 2. We call *invariant neighbourhood* the smallest symmetric neighbourhood (non reduced

to $\{v_0\}$) $N^i(v_0)$ of $v_0$ in $M^i$ such that there exists a matrix $S$ verifying

$$N^{i+1}(v_0) = SN^i(v_0) \tag{2.5}$$

In the case of the modified butterfly scheme, $N^i(v_0)$ is the 2-neighbourhood of $v_0$ (in green and red on Figure 2.3), and

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ a_0 & a_1 & a_2 & a_3 & a_3 & a_2 \\ a_0 & a_2 & a_1 & a_2 & a_3 & a_3 \\ a_0 & a_3 & a_2 & a_1 & a_2 & a_3 \\ a_0 & a_3 & a_3 & a_2 & a_1 & a_2 \\ a_0 & a_2 & a_3 & a_3 & a_2 & a_1 \end{bmatrix} \tag{2.6}$$

where $a_0$, $a_1$, $a_2$ and $a_3$ are the prediction weights (because of symmetry, there are only 4 weights). These weights are determined by ensuring the necessary polynomial-reproducing condition, which yields $a_0 = \frac{3}{4}$ and $a_j = \frac{1}{5}\left(\frac{1}{4} + cos\left(\frac{2\pi j}{5}\right) + \frac{1}{2}cos\left(\frac{4\pi j}{5}\right)\right)$ in the case of a 5-regular neighbourhood.

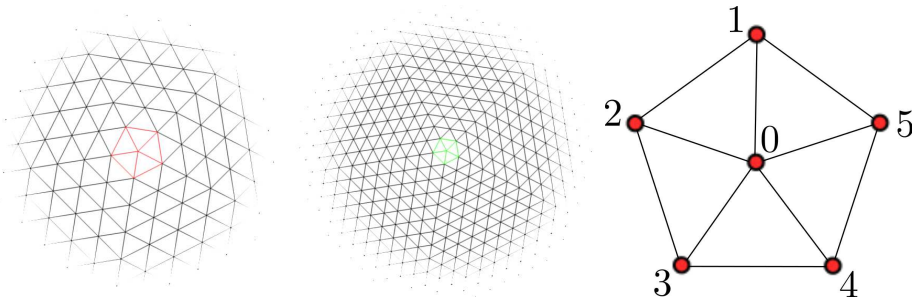


Figure 2.3: A 5-regular mesh (left) and the same mesh after one subdivision step (center). Superimposed are the corresponding invariant neighbourhoods $N^i(v_0)$ (red) and $N^{i+1}(v_0)$ (green). On the right, the numbering used in equation 2.6.

Note that this is only a necessary condition, and the prediction weights thus determined must be subjected to the following sufficient condition [Reif 1995]:

If $\lambda_1, \ldots, \lambda_{K+1}$ are the eigenvalues of the subdivision matrix, sorted in by decreasing magnitude, then the limit surface is $C^1$ if:

$$\begin{cases} \lambda_1 = 1 > |\lambda_2| > |\lambda_4| \\ \lambda_2 = \lambda_3 \end{cases} \tag{2.7}$$

### 2.1.5 Determining Prediction Weights

The linear predictors described above mainly use two different approaches to determine weights. On one hand, the *spatial* approach considers prediction as a geometric problem. For example, the parallelogram rule predicts that adjacent triangles are geometrically similar. Average Parallelogram Prediction uses the point closest to all different parallelogram predictions. On the other hand, *spectral* methods work in the frequency domain. This is the idea of [Isenburg et al. 2005a] and [Ibarria et al. 2007], but is also true in the case of wavelets. The wavelet coefficients used by Khodakovsky and Guskov [Khodakovsky and Guskov 2003] are determined using a spectral approach very similar to [Ibarria et al. 2007] (this approach had already been used, and is presented in more details, in [Zorin et al. 1996b]). While the spatial and spectral methods use different arguments, we will see in sections 2.3.5.1 and 2.3.4 that they are sometimes equivalent.

## 2.2 Local Spectral Prediction

Ibarria *et al.* use a spectral approach [Ibarria et al. 2007] to derive prediction weights on grids, i.e. regular quad meshes. We have extended this approach to the case of irregular polygon meshes, with variable success. The predictors that we obtain perform very well for interpolation, but extrapolation remains problematic. In the following sections, we first describe the algorithm that we use to determine prediction weights. Then, we evaluate the resulting predictor and discuss its strengths and weaknesses.

### 2.2.1 Determining Prediction Weights

The idea of our approach is to predict the geometry of a vertex as the position which makes its neighbourhood as *smooth* as possible with respect to its connectivity. By smooth, we mean that the high-frequency content, in the sense of [Ibarria et al. 2007], of the neighbourhood is as small as possible.

Let $v_0$ be the vertex whose geometry is to be predicted, $v_1, \ldots, v_n$ the vertices of the neighbourhood, and $({v_i}^x, {v_i}^y, {v_i}^z)$ the position of $v_i$. In the following, because all coordinates are similar, we only consider ${v_i}^x$. We suppose that $k$ vertices of the neighbourhood are known $(v_1, \ldots, v_k)$, and we define the *local Laplacian* $L^0$ of the mesh around $v_0$ as the Laplacian of the neighbourhood considered as a separate mesh, that is:

$$L^0_{i,j} = \begin{cases} |\{v_k | v_k \text{ and } v_i \text{ are adjacent}\}|, & \text{if } i = j \\ -1, & \text{if } v_i \text{ and } v_j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

The eigenvectors of $L^0$ constitute the *local spectral basis* of the neighbourhood. If $E$ is the orthogonal matrix of eigenvectors (one per column) ordered by ascending eigenvalue magnitude, then the vector $S$ defined by:

$$ES = \begin{pmatrix} {v_0}^x \\ \vdots \\ {v_n}^x \end{pmatrix} = V^x \tag{2.9}$$

is the local spectral decomposition of the neighbourhood.

As we want the prediction to result in the smoothest possible neighbourhood geometry, we look for $S$ such that the $n-k$ highest frequencies $(S_{k+1}, \ldots, S_n)$ are 0. This is equivalent to finding the predictor that reproduces the Fourier basis vectors of lower frequency. The predictor $p(v_0) = \sum_{i=1}^{k} \alpha_i {v_i}^x$ reproduces the basis vector $E_{:,m}$ if:

$${E_{1:k,m}}^T . \alpha_i = E_{0,m} \tag{2.10}$$

Progressively adding equations arising from the reproduction of basis vectors of increasing frequency allows us to build a matrix $A$ and a vector $B$ such that the condition $A\alpha = B$ is equivalent to reproducing the maximum number of basis vectors of low frequency. Each time a new basis vector $E_{:,m}$ is to be added to the current set of $l < k$ equations, two situations can arise:

1. $E_{:,m}$ is associated with a simple eigenvalue. Then if the vector $E_{1:k,m}$ is linearly independent from $A_i, i \in [\![1, l]\!]$, we set $A_{l+1} = {E_{1:k,m}}^T$ and $B_{l+1} = E_{0,m}$. Else $E_{:,m}$ is dropped.
2. $E_{:,m}$ and $E_{:,m+1}$ are associated with a double eigenvalue. The subspace $Span\,(E_{1:k,m}, E_{1:k,m+1})$ associated with this eigenvalue is either totally, partially or not linearly independent from $A_i, i \in [\![1, l]\!]$, which will result in the addition of respectively 2, 1 or 0 equations to $A$ and $B$, as in [Ibarria et al. 2007].

| K | tip | crease |
|---|---|---|
| 3 | -.470 | .800 |
| 4 | -.250 | .500 |
| 5 | -.146 | .346 |
| 6 | -.094 | .260 |
| 7 | -.064 | .207 |
| 8 | -.047 | .172 |

Table 2.1: Spectral prediction weights for the $K$-star case. *Crease* and *tip* respectively denote the weights for the inner ring (one-ring) and outer ring of the star-shaped stencil.

Finding the weights is then a simple matter of solving a linear system of size $k \times k$. It is interesting to note that the first eigenvector $[1, \ldots, 1]$ of the Laplacian has an eigenvalue of 0, which is always simple. Therefore, it is always added to the set of equations. This ensures the condition $\sum \alpha_i = 1$ and makes the scheme affine invariant.

### 2.2.2 Compression

To evaluate our new spectral prediction method, we have computed weights for several different configurations classically used for compression, in particular the *Parallelogram Rule* of Touma and Gotsman. The spectral method yields the same weights $\{-1, 1, 1\}$.

We have seen in Section 2.1.1 that more sophisticated approaches [Pajarola and Rossignac 2000a; Cohen-or et al. 2002; Sim et al. 2003] have used an average of several parallelogram predictions. This results in a predictor that is also linear, and that led to the expected decrease in prediction error and residual entropy. However, in some cases the results achieved by the spectral approach are better than a simple average of parallelogram predictions.

The weights derived using our spectral approach are given in Figure 2.4 and Table 2.1. In the case of the $K$-star neighbourhood, that is used in the first step of [Cohen-or et al. 2002] and [Pajarola and Rossignac 2000a], the predictor is interpolating (it knows a complete neighbourhood *around* the vertex to predict). In that case, the spectral method is very efficient, and reduces the error from 13 to 48%, as seen in Table 2.2.

However, the spectral weights for the DPP stencil (Figure 2.4) are far worse than an average of two parallelogram rules. The predictor even makes a systematic prediction error if the mesh is regular, by predicting the missing vertex at the center vertex of the configuration. This is a major problem with the spectral approach, that can often be found when the predictor is extrapolating. We will discuss this in further details in a following section (2.2.3).

**Least square weights** A naturally good set of weights for a linear predictor is the set of weights that minimizes the norm of the residuals for a given mesh. We call this predictor the *Least Squares Predictor (LSP)*. By definition, the LSP has the best prediction performance in terms of mean square error.

As all predictors that are based on a varying neighbourhood, a set of weights has to be computed for each given vertex neighbourhood. For irregular meshes, the neighbourhood information can vary a lot between two vertices. As the weights of the LSP depend on the mesh, that means that to use it, an algorithm must store along with the mesh data the sets of weights for all possible neighbourhood configurations.

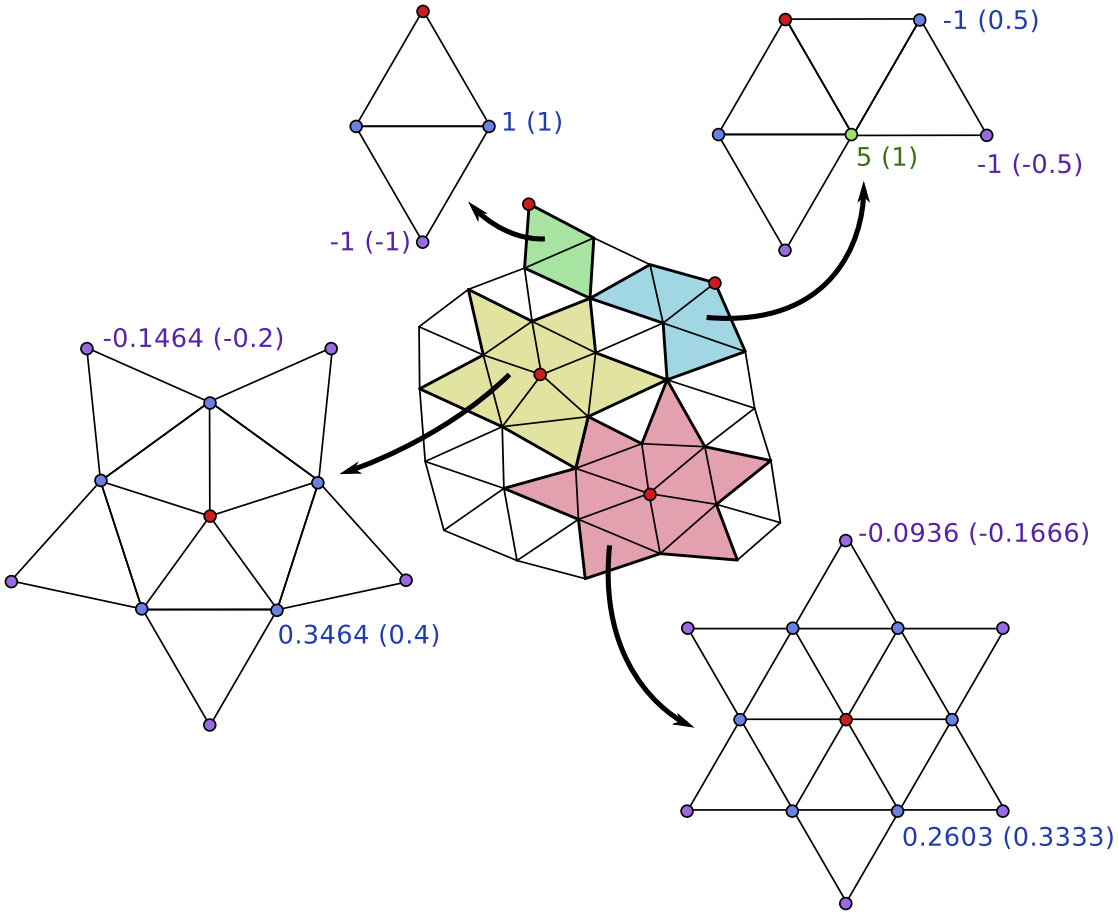


Figure 2.4: Some examples of prediction weights determined by our spectral approach, with several prediction stencils: Starting from the top left, in clockwise order: Parallelogram, Dual Parallelogram, 6-star (APP) and 5-star (APP). The numbers between parentheses are the Average Parallelogram weights. For symmetry reasons, the vertices with the same colors have the same weights. Note that the parallelogram predictor is a particular case of the algorithm.

| Model | vs. Vertex Degree | | | | | | Total |
|---|---|---|---|---|---|---|---|
| | 3 | 4 | 5 | 6 | 7 | 8 | |
| armadillo | (138) | 100 | 93 | 84 | 87 | 75 | 87 |
| guardian | 128 | 100 | 87 | 84 | 84 | 88 | 86 |
| gipshand | * | (100) | 90 | 60 | 83 | 99 | 64 |
| max planck | (136) | 100 | 90 | 79 | 90 | 96 | 86 |
| neptune (uniform) | * | (100) | 58 | 57 | 72 | * | 59 |
| guardian (uniform) | * | * | 56 | 51 | * | * | 52 |

Table 2.2: Prediction error of our spectral approach relative to Average Parallelogram Prediction for $K$-star neighbourhoods (in percents, i.e. $100 \times \frac{E_{spectral}}{E_{APP}}$). We only give details for $K \leq 8$, but the total in the last column takes into account all possible $K$, weighted by their respective importance (there are far more 6-stars than other degrees). Parentheses denote statistically doubtable results (for which there were less than 50 occurrences). The asterisk denotes absence of data.

In comparison, the weights computed by the spectral approach are independent of the mesh, and thus do not require any other information to be stored along with the mesh data. However, the experimental results show that for the $K$-star configuration, the spectral weights are close to the LSP weights on average. This can explain the good performance of the spectral predictor for these prediction stencils. Table 2.3 gives the LSP weights for the most common 5- and 6-star neighbourhoods.

| **Model** | **6-Star** | **5-Star** |
|---|---|---|
| eight | -.084, .251 | - |
| venus | -.054, .221 | -.046, .246 |
| cow | -.094, .261 | -.109, .309 |
| femur | -.056, .223 | -.062, .2630 |
| shark | -.129, .297 | -.120, .320 |
| sphere | -.111, .278 | - |
| horse | -.058, .225 | -.049, .249 |
| dino | -.043, .210 | -.044, .244 |
| armadillo | -.061, .227 | -.108, .308 |
| **Average LS** | -.077, .243 | -.077, .277 |
| **Spectral** | -.094, .260 | -.146, .346 |
| **APP** | -.166, .333 | -.200, .400 |

Table 2.3: Best weights in terms of least squares error for the most common 5- and 6-star neighbourhood, spectral weights, and APP weights. The weights are respectively those of the tips and creases of the stars.

### 2.2.3 Discussion

**Systematic Extrapolation Error:** The spectral method is often subject to a systematic prediction error when extrapolating, i.e. when the vertex to predict is at the boundary of the neighbourhood. This problem is visible in the case of the *Dual Parallelogram Stencil* weights derived via the spectral method: The vertex is predicted systematically and erroneously at the center vertex of the configuration. This problem is related to the properties of the local Laplacian, that tends to shrink a mesh at

boundaries. To illustrate it, let us consider a triangle mesh with an *ear* configuration (i.e. a degree 2 boundary vertex). Smoothing the mesh (i.e. applying the Laplacian to the mesh) collapses the ear, shrinking the mesh. Another way of seeing the same problem is that the Laplacian of a mesh measures how much a vertex fails to be at the center of gravity of its one-ring neighbours. Therefore, it works when there exists a complete one-ring around a vertex, which is not the case at boundaries.

To alleviate this problem, it is possible to add *virtual vertices* to the neighbourhood, such that the virtual neighbourhood is rotation-invariant (Figure 2.5). These vertices will not take part in the prediction but their connectivity will modify the local Laplacian. This avoids the harmful boundary effects and yields new weights that are exempt of systematic error (Figure 2.5). However, these new weights are usually large, which is bad for prediction: If we consider a triangle mesh $M$ that is regular in both connectivity and geometry, and a predictor $p$ with weights $(\alpha_1, \ldots, \alpha_N)$ that has no systematic error. If $M$ is perturbed by adding a white noise $\epsilon$ with variance ${\sigma_\epsilon}^2$, then the variance of the residual is:

$$\begin{array}{rcl} Var\left(\mathbf{v_0^x} - \mathbf{p}(v_0)\right) & = & Var\left(\mathbf{v} - \sum_{v_i \in N(v_0)} \alpha_i \left(\mathbf{v_i^x} + \epsilon(v_i)\right)\right) \\ & = & Var\left(\sum_{v_i \in N(v_0)} \alpha_i \epsilon(v_i)\right) \\ & = & \sum {\alpha_i}^2 Var\left(\epsilon(v_i)\right) \\ & = & \|\alpha\| {\sigma_\epsilon}^2 \end{array} \tag{2.11}$$

This shows that although adding virtual vertices prevents the systematic prediction error, it is not very efficient because it also decreases robustness to noise. Therefore, extrapolation remains a problem with this approach.

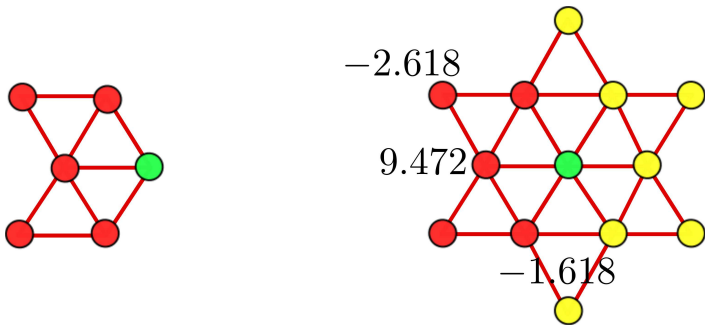


Figure 2.5: Adding virtual vertices avoids systematic prediction error: the original stencil (left), and the new stencil with virtual vertices in yellow (right). The green vertex is being predicted.

**Deriving subdivision schemes:** Interestingly, if we try to use spectral prediction to derive predictors for subdivision surface wavelets, it is possible to obtain the same predictors as the usual subdivision wavelets for regular settings (linear, butterfly, loop, and $\sqrt{3}$). One has to take care, however, of the boundary effects. Because of that, we embed the prediction stencil inside a regular mesh made of virtual vertices, far from boundaries. The neighbourhoods that we used are shown in Figure 2.6. We have verified experimentally that the approximation of the weights drew closer to the actual wavelet prediction weights as the virtual mesh grew. Currently, we cannot explain why this behaviour is observed, but we find that it was interesting enough to be mentioned here. Also, the butterfly weights for the irregular case cannot be obtained using this method.

**Numerical cost:** The symbolic computation of eigenvectors for neighbourhoods with more than a few vertices is too computationally expensive to be practical. Therefore, all computations are numerical. This may pose some problems when determining linear dependance. However, in our

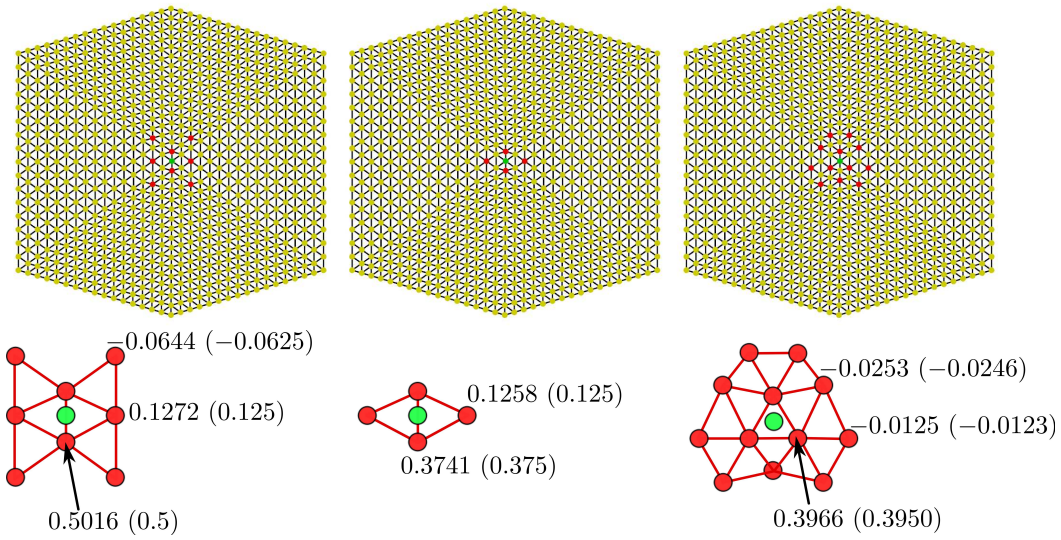


Figure 2.6: Deriving an approximation of typical wavelet prediction schemes using irregular spectral prediction: Butterfly (left), Loop (middle), and $\sqrt{3}$ (right). The top row shows the neighbourhood the we used. The yellow vertices are virtual vertices, the red vertices are the known vertices, and the green vertex is being predicted. The bottom row shows the prediction stencils (coarser mesh) and associated weights. The actual weights are given between parentheses for reference.

tests, we have never encountered such a situation, even for the larger neighbourhoods used in the previous paragraph, where numerical errors could accumulate.

**Computational cost:** While this approach is appealing for interpolation, there remains a major practical problem: computation time. Computing the eigenvectors of the neighbourhood Laplacian is a very costly operation. While determining weights for small neighbourhoods (tens of vertices) will be very fast (hundreds of milliseconds), the larger neighbourhoods used for example to derive subdivision weights result in several seconds of computation. This problem is emphasized by the fact that the method needs larger neighbourhoods to yield best results, for example when adding virtual vertices.

## 2.3 Taylor Prediction

The major drawback of the local spectral approach presented in the previous section is its inability for *extrapolation*. The *dummy stencil extension* method that we suggested to alleviate this problem is not very satisfying because it drastically increases the computational complexity. The inability of the spectral prediction to extrapolate originates in the fact that the Laplacian operator uses a *combinatorial* description of the neighbourhood: The link between the positions of two vertices is based on their *adjacency* relationship. We tried to alleviate this problem by using a *parametric* description of the neighbourhood. In this description, the links between the position of two vertices are expressed in terms of their relative position in the *local parameter space* of a canonical geometric configuration of the neighbourhood. The main difference between the two approaches is that the latter considers the mesh as an approximation of a surface, while the former works exclusively on the mesh connectivity, especially on the adjacency between vertices. This makes the spectral approach very sensitive to the way a surface is meshed. In particular, the spectral method is sensitive to the

presence of vertices even if they do not participate in the prediction (this is the case, for example, of virtual vertices). On the other hand, the Taylor prediction method we present here only considers the vertices that will be used for prediction, and is somehow oblivious to their adjacency relationships.

### 2.3.1 Setup

In the following, $M$ is a differentiable 2-manifold defined by the vector-valued function:

$$\mathbf{f}(u,v) = (x(u,v), y(u,v), z(u,v))$$

where $(u,v)$ spans a certain subdomain $D$ of $\mathbb{R}^2$. $x$, $y$ and $z$ are the *coordinates* of the points of $M$. For the sake of clarity, we only consider surfaces, but the method generalizes seamlessly to the case of volumes defined by three parameters $(u,v,w) \in \mathbb{R}^3$.

**Definition 1.** *$M$ is* 2-smooth *if $\mathbf{f}$ is $C^2$-differentiable on all but a finite number of points $p_1, \dots p_N$.*

Around these points, $\mathbf{f}$ can be expanded using the second-order Taylor approximation:

$$\begin{array}{l} \forall \mathbf{u} \in D - \{p_1, \dots p_N\}, \\ \mathbf{f}(\mathbf{u}+\mathbf{du}) = \mathbf{f}(\mathbf{u}) + \mathbf{du}^T.(\nabla \mathbf{f})(\mathbf{u}) + \frac{1}{2}\mathbf{du}^T.(\nabla^2 \mathbf{f})(\mathbf{u}).\mathbf{du} + o(\mathbf{du}^T.\mathbf{du}) \end{array} \tag{2.12}$$

where $\nabla f$ and $\nabla^2 f$ are respectively the Gradient and Hessian of $f$.

The mesh $G$ that we are studying can be seen as an piecewise linear approximation of such a surface $M$. Let us suppose that the positions of some vertices of $G$ are known. This means that we know the value of $\mathbf{f}$ at the parameter values corresponding to these vertices. In the following, we show how to derive a linear predictor to compute the position of an unknown vertex $v \in V(G)$ at parameter $(u,v)$ from $k$ vertices of the neighbourhood. We denote by $\alpha_i$ the coefficients of the prediction:

**Definition 2.** *Let $\mathbf{u_1}, \dots, \mathbf{u_k} \in D^k$. Then a linear prediction of $\mathbf{f}(\mathbf{u})$ is :*

$$\mathbf{f}(\mathbf{u}) = \sum_{i=1}^{k} \alpha_i \mathbf{f}(\mathbf{u_i}). \tag{2.13}$$

### 2.3.2 Prediction

If $f$ is 2-smooth, then we can approximate the value of $f$ at $\mathbf{u_1}, \dots, \mathbf{u_k}$ in the prediction definition (2.13) using equation (2.12):

$$\begin{array}{rl} 0 = & \left[\left(\sum_{i=1}^k \alpha_i\right) - 1\right].\mathbf{f} \\ + & \left[\sum_{i=1}^k \alpha_i (u_i - u)\right] \frac{\partial \mathbf{f}}{\partial u} + \left[\sum_{i=1}^k \alpha_i (v_i - v)\right] \frac{\partial \mathbf{f}}{\partial v} \\ + & \frac{1}{2}\left[\sum_{i=1}^k \alpha_i (u_i - u)^2\right] \frac{\partial^2 \mathbf{f}}{\partial u^2} \\ + & \frac{1}{2}\left[\sum_{i=1}^k \alpha_i (v_i - v)^2\right] \frac{\partial^2 \mathbf{f}}{\partial v^2} \\ + & \left[\sum_{i=1}^k \alpha_i (u_i - u)(v_i - v)\right] \frac{\partial^2 \mathbf{f}}{\partial u \partial v} \\ + & o\left((u_i - u)^2 + (v_i - v)^2 + (u_i - u)(v_i - v)\right) \end{array} . \tag{2.14}$$

We can see that if the predictor is to generate constant surfaces ($\mathbf{f}$ is not zero, but $\nabla \mathbf{f}$ and $\nabla^2 \mathbf{f}$ are), then the condition $\left(\sum_{i=1}^k \alpha_i\right) - 1 = 0$ must be verified. This property also makes the scheme affine invariant.

If surfaces with local zero curvature ($\nabla^2 \mathbf{f} = 0$) are to be predicted exactly, then the coefficients of $\frac{\partial \mathbf{f}}{\partial u}$ and $\frac{\partial \mathbf{f}}{\partial v}$ must be zero. By repeating this process, the predictor weights can be determined by solving the linear system arising from the reproduction of polynomials of increasing degree, until we have a sufficient number of equations to determine all the weights.

**Polynomial basis point of view:** This approach is similar to the spectral approach. The polynomials of increasing degree can be seen as a local basis on which we decompose the geometry of the neighbourhood of $v$ in $M$. In the spectral case, we zeroed the high frequency coefficients. In the spatial case presented here, the position of $v$ is predicted as that which zeroes the *high-degree* coefficients (i.e. $\frac{\partial^{(i+j)}\mathbf{f}}{\partial u^i \partial u^j}$ for larger $i+j$). Therefore, *degree* in our spatial approach plays the same role as *frequency* in spectral schemes. We will see in Section 2.3.4.5 that for some neighbourhoods, the spectral basis and the basis of polynomials sampled at locations $\{\mathbf{u_i}\}_{1\leq i\leq k} \cup \{\mathbf{u}\}$ are actually the same, leading to an equivalence between the spectral and spatial approaches.

**Dealing with over/under-constrained systems:** In the spectral case, multiple eigenvectors of the spectral basis were sometimes associated with the same eigenvalue. In that case, removing the highest frequency could lead to an under-constrained system, while selecting both eigenvectors would over-constrained the system. This is also true in the case of the current spatial approach. We choose to deal with it in the spatial domain by making assumptions on the geometry of $M$. This was not possible with the spectral approach because the description of the neighbourhood was combinatorial and not parametric. This approach is also more intuitive because those assumptions can be directly interpreted as local symmetries of $\mathbf{f}$.

To fix ideas, let us consider the case where there are not enough equations to uniquely determine the weights at expansion order $d$. Determining the weights by expanding the function to order $d+1$ might be impossible for two reasons: First, one may not be willing to make the assumption that $\mathbf{f}$ is $(d+1)$-smooth. Then, the equations at order $d+1$ may be incompatible with the ones from orders 0 to $d$. We devised two different ways of dealing with an over-constrained system, depending on the assumptions one is willing to make about the geometry of $M$.

1. **Weights Norm Minimization:** In the case where $\mathbf{f}$ is not $(d+1)$-smooth, then the Taylor expansion will obviously not yield additional constraints. In that case we use a different heuristic. It consists in choosing, among the weights that verify the under-constrained set of equations, the weights of smaller amplitude (i.e. for which the norm of the weights vector $\|\alpha\| = \sum_{i=1}^{k} {\alpha_i}^2$ is smallest). Isenburg *et al.* [Isenburg et al. 2005a] already noted that in the absence of other assumptions, this heuristic helps spread the dependency of the prediction on all the points by avoiding giving too much importance to a single one. This heuristic is also efficient because it is the most robust choice of weights with respect to *noise*. Consider a function $\mathbf{f}$ for which a given predictor with weights $\alpha$ is exact. If $\mathbf{f}$ is perturbed by adding a white noise $\epsilon$ with variance ${\sigma_\epsilon}^2$, then the variance of the residual $\mathbf{f}(u,v) - \mathbf{p}_\alpha(u,v)$ is:

$$\begin{aligned} Var\left(\mathbf{f}(u,v) - \mathbf{p}_\alpha(u,v)\right) &= Var\left(\mathbf{f}(u,v) - \sum \alpha_i \left(f(u_i,v_i) + \epsilon_i\right)\right) \\ &= Var\left(\sum \alpha_i \epsilon_i\right) \\ &= \sum {\alpha_i}^2 Var\left(\epsilon_i\right) \\ &= \|\alpha\| {\sigma_\epsilon}^2 \end{aligned} \tag{2.15}$$

   It is clear that the heuristic minimizes the prediction error in the presence of noise, and is thus optimally robust to white noise. This heuristic is actually used (although not in this form) to determine the weights of nearly all subdivision schemes. This is discussed in the next sections.

2. **Variance Minimization:** This applies to the case where $\mathbf{f}$ is $(d+1)$-smooth, but the set of equations from order $d+1$ are not compatible with those from orders $\leq d$. Although it may be impossible to zero the $(d+1)$-order term in the Taylor expansion, it is possible to find the weights $\alpha$ that minimize its variance $E(\alpha)$ over the set of possible functions:

$$E(\alpha) = Var\left(\sum_{m=0}^{d+1} \binom{d+1}{m} \left[\sum \alpha_i . {du_i}^m . {dv_i}^{d+1-m}\right] \frac{\partial^{d+1} F}{\partial u^m \partial v^{d+1-m}}\right) \tag{2.16}$$

If we set

$$\left\{ \begin{array}{lcl} b_{m+1}(\alpha) & = & \binom{d+1}{m}\left[\sum \alpha_i . {du_i}^m . {dv_i}^{d+1-m}\right] \\ \left(\partial^{d+1} f\right)_{m+1} & = & \frac{\partial^{d+1} f}{\partial u^m \partial v^{d+1-m}} \end{array} \right. \tag{2.17}$$

then the equation above can be written:

$$E(\alpha) = \mathbf{b}(\alpha)^T Var\left(\partial^{\mathbf{d+1}}\mathbf{F}\right)\mathbf{b}(\alpha), \tag{2.18}$$

and the optimal weights are:

$$\alpha = \arg\min \quad \mathbf{b}(\alpha)^T Var\left(\partial^{\mathbf{d+1}}\mathbf{F}\right)\mathbf{b}(\alpha). \tag{2.19}$$

It is easy to see that the optimal weights depend on the covariance matrix $Var\left(\partial^{\mathbf{d+1}}\mathbf{F}\right)$ of the partial derivatives vector. To avoid breaking the line of thoughts, we delay the study of this matrix to a later section (2.3.3). The only important thing to note is that since the partial derivatives vector $\partial^{\mathbf{d+1}}\mathbf{F}$ is independent of $\alpha$ and the covariance matrix is positive semi-definite, the function $\alpha \mapsto E(\alpha)$ is a convex bilinear form. Therefore, it has a global minimum and we can derive new linear equations to determine $\alpha$ from the first-order conditions:

$$\begin{array}{rcl} 0 & = & \frac{\partial E}{\partial \alpha_i} \\ & = & \frac{\partial}{\partial \alpha_i}\mathbf{b}(\alpha)^T Var\left(\partial^{\mathbf{d+1}}\mathbf{F}\right)\mathbf{b}(\alpha) \end{array} \tag{2.20}$$

The covariance matrix $Var\left(\partial^{\mathbf{d+1}}\mathbf{F}\right)$ can be derived from a simple statistical model of possible functions (Section 2.3.3). This provides as many equations as needed to correctly constrain the system.

### 2.3.3 A study on the variance of $\frac{\partial^{i+j} F}{\partial u^i \partial v^j}$

We have seen that Variance Minimization is one of the useful approaches to deal with an overdetermined system. However, the evaluation of the variance of the partial derivatives of $f$ is crucial since the equations derived from Variance Minimization depend on the covariance matrix of the partial derivatives. In this section, we introduce a simple statistical model for possible geometry functions which enables us to estimate the value of $Var\left(\partial^{\mathbf{d}}\mathbf{F}\right)$.

In fact, the value of the matrix need only be determined up to a multiplicative constant, since only the ratio of its elements appear in the Variance Minimization process. We choose to scale the matrix so that the top-left element is 1, by setting:

$$\Sigma^d_{i,j} = Var\left(\partial^{\mathbf{d}}\mathbf{F}\right)/Var\left(\frac{\partial^d \mathbf{F}}{\partial u^d}\right) = Cov\left(\frac{\partial^d \mathbf{F}}{\partial u^{(i-1)}\partial v^{(d+1-i)}}, \frac{\partial^d \mathbf{F}}{\partial u^{(j-1)}\partial v^{(d+1-j)}}\right)/Var\left(\frac{\partial^d \mathbf{F}}{\partial u^d}\right) \tag{2.21}$$

To compute this value, we consider the geometry of a mesh as an outcome of a random variable $\mathbf{F}(u,v) = (X(u,v), Y(u,v), Z(u,v))$. To simplify the demonstration, we work in the Frenet frame of the mesh $(\mathbf{e_u}, \mathbf{e_v}, \mathbf{e_n} = \mathbf{e_u} \times \mathbf{e_v})$, and we consider only the normal component of the geometry: $N(u,v) = \mathbf{F}(u,v).\mathbf{e_n}$. Also, we only consider derivatives up to the second order ($d \leq 2$), because a 2-smoothness is usually sufficient for most surfaces.

Locally, the geometry can be approximated by a second-order polynomial $\bar{N}$ in $(u,v)$:

$$\bar{N}_{\mathbf{a}}(u,v) = a_1 u^2 + a_2 uv + a_3 v^2 + a_4 u + a_5 v + a_6 \tag{2.22}$$

where $(a_1, a_2, \dots)$ are approximations of $(\frac{\partial^2 \mathbf{N}}{\partial u^2}, \frac{\partial^2 \mathbf{N}}{\partial u \partial v}, \dots)$.

If the mesh is sufficiently densely sampled, then a good approximation of the parameters $(a_1, \dots, a_6)$ can be obtained by using *only the one-ring neighbourhood* of a vertex. In the following, we consider

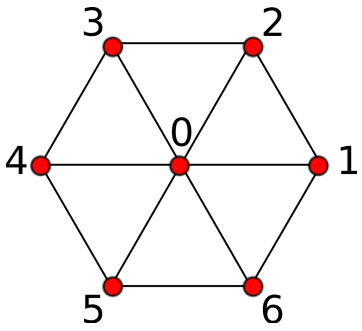


Figure 2.7: The numbering of the one-ring used in the proof.

a vertex of degree 6 and its neighbours, numbered as shown in Figure 2.7. The best approximation of $N$ by $\bar{N}$ in the least squares sense is found by minimizing:

$$H(\mathbf{a}) = \sum_{k=0}^{6} \left[\bar{N}(u_k, v_k) - N(u_k, v_k)\right]^2 \tag{2.23}$$

Minimizing $H(\mathbf{a})$ is a linear problem $M.\mathbf{a} = \mathbf{b}$ with:

$$\begin{cases} M_{i,j} &= \sum_{k=0}^{6} \frac{\partial \bar{N}_{\mathbf{a}}}{\partial a_i}(u_k, v_k).\frac{\partial \bar{N}_{\mathbf{a}}}{\partial a_j}(u_k, v_k) \\ b_i &= \sum_{k=0}^{6} \frac{\partial \bar{N}_{\mathbf{a}}}{\partial a_i}(u_k, v_k).N(u_k, v_k) \end{cases} \tag{2.24}$$

Using the neighbourhood of Figure 2.7 yields

$$M = \begin{bmatrix} \frac{9}{4} & 0 & \frac{3}{4} & 0 & 0 & 3 \\ 0 & \frac{3}{4} & 0 & 0 & 0 & 0 \\ \frac{3}{4} & 0 & \frac{9}{4} & 0 & 0 & 3 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 3 & 0 & 3 & 0 & 0 & 7 \end{bmatrix} \tag{2.25}$$

and therefore

$$\mathbf{a} = M^{-1}\mathbf{b} = \begin{bmatrix} \frac{N_1+N_4}{2} - N_0 \\ \frac{\sqrt{3}}{3}(N_6 - N_5 + N_3 - N_2) \\ -\frac{N_1+N_4}{6} + \frac{1}{3}(N_6 + N_5 + N_3 + N_2) - N_0 \\ \frac{N_1-N_4}{3} + \frac{1}{6}(N_6 - N_5 - N_3 + N_2) \\ \frac{\sqrt{3}}{6}(-N_6 - N_5 + N_3 + N_2) \\ N_0 \end{bmatrix} \tag{2.26}$$

The covariance of the partial derivatives of $N$ can be derived from this expression:

$$\begin{cases} Var\left(\frac{\partial^2 \mathbf{N}}{\partial u^2}\right) = Var\left(a_1\right) = Var\left(\frac{N_1+N_4}{2} - N_0\right) \\ Var\left(\frac{\partial^2 \mathbf{N}}{\partial v^2}\right) = Var\left(a_3\right) \\ \dots \end{cases} \tag{2.27}$$

Thus, the covariance of partial derivatives can be expressed as linear combinations of terms $Var(N_i)$ and $Cov(N_i, N_j)$. To simplify the problem, we make an additional assumption on the mesh, that has been introduced and discussed by Ben-Chen and Gotsman[Ben-Chen and Gotsman 2005]: We suppose that the covariance of two vertices that are not adjacent is 0, or more formally:

$$Cov\left(N_i, N_j | N_k = n_k, k \notin \{i, j\}, (i, j) \notin E\right) = 0. \tag{2.28}$$

| Model | $\Sigma^2_{1,2}$ | $\Sigma^2_{1,3}$ | $\Sigma^2_{1,1}$ |
|---|---|---|---|
| armadillo | 0.00 | 0.38 | 1.27 |
| guardian | -0.66 | -0.02 | 1.68 |
| gipshand | 0.00 | 0.42 | 1.15 |
| Max Planck | 0.00 | 0.12 | 0.57 |
| Neptune (uniform) | -0.01 | 0.36 | 1.30 |
| guardian (uniform) | 0.00 | 0.34 | 1.35 |
| our theoretical model | 0 | 0.33 | 1.33 |

Table 2.4: Covariances of partial derivatives measured on our test meshes.

Furthermore, because of symmetry, all variances are equal, and we note $\sigma_s^2 = Var(N_i)$. For the same reason, the covariances of the positions of adjacent vertices are equal, and we note $\sigma_n^2 = Cov(N_i, N_j)$ for $(i,j) \in E$.

This yields:

$$\begin{cases} Var(a_1,a_1) = Var(a_3,a_3) = \frac{3}{2}\sigma_s^2 - 2\sigma_n^2 \\ Var(a_2,a_2) = \frac{4}{3}\left(\sigma_s^2 - \sigma_n^2\right) \\ Cov(a_1,a_3) = \frac{5}{6}\sigma_s^2 - \frac{4}{3}\sigma_n^2 \\ Cov(a_1,a_2) = Cov(a_2,a_3) = 0 \end{cases} \tag{2.29}$$

On the other hand, since the value of the derivatives of $N$ should not be dependent on a global translation of all the vertices $(u_k, v_k)$, we can choose $a_6 = 0$, i.e. $Cov(N_0, N_i) = 0 \quad \forall i \in [0,6]$. This yields:

$$\begin{cases} Var(a_1,a_1) = Var(a_3,a_3) = \frac{1}{2}\sigma_s^2 \\ Var(a_2,a_2) = \frac{4}{3}\left(\sigma_s^2 - \sigma_n^2\right) \\ Cov(a_1,a_3) = -\frac{1}{6}\sigma_s^2 + \frac{2}{3}\sigma_n^2 \\ Cov(a_1,a_2) = Cov(a_2,a_3) = 0 \end{cases} \tag{2.30}$$

Combining equations (2.29) and (2.30), we can deduce that $\sigma_s^2 = 2\sigma_n^2$. Putting back this value in (2.29) or (2.30) shows that the normalized variance/covariance matrix of the partial derivatives is:

$$\Sigma^2 = \begin{bmatrix} 1 & 0 & \frac{1}{3} \\ 0 & \frac{4}{3} & 0 \\ \frac{1}{3} & 0 & 1 \end{bmatrix} \tag{2.31}$$

Using the same process, we can compute:

$$\Sigma^1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.32}$$

We confronted these theoretical values with experiments. Table 2.4 gives the measured covariances [1] on our set of test meshes. We can see that the results agree very well with the theoretical results, except for the *guardian* and *Max Planck* models. We cannot currently explain why our model fails on these meshes, however we conjecture that this may come from the fact that they have a high number of very distorted triangles (see Figure 2.10).

[1] To estimate the value of the partial derivatives, we used a finite difference scheme on the one-ring of each vertex with arbitrary $\mathbf{e_u}$, $\mathbf{e_v}$.

### 2.3.4 Predictors for connectivity-driven compression

We now apply the theoretical framework presented above to the derivation of prediction weights in various settings.

#### 2.3.4.1 Parallelogram Rule:

The parallelogram rule, used by nearly all predictors, is a special case of our approach. For symmetry reasons, it has only two different weights shown in Figure 2.8 (a). Because the number of weights is small, expanding equation (2.14) to the first order is sufficient to constrain them:

$$\begin{aligned} 0 = [2\alpha_1 + \alpha_3 - 1]\mathbf{f} - \sqrt{3}[\alpha_1 + \alpha_3]\tfrac{\partial \mathbf{f}}{\partial u} dl + o(dl) \\ \Rightarrow \begin{cases} \alpha_1 = 1 \\ \alpha_3 = -1 \end{cases} \end{aligned} \tag{2.33}$$

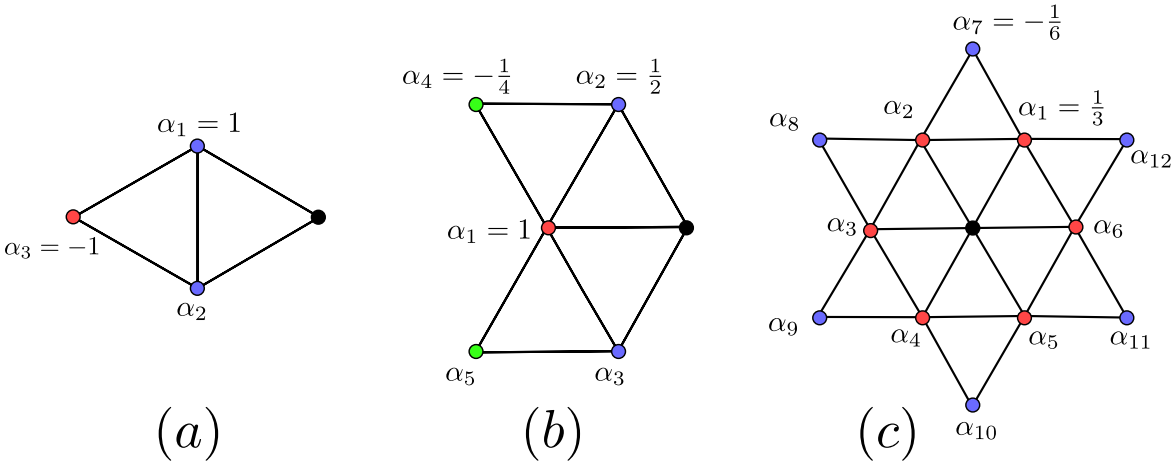


Figure 2.8: Prediction stencils traditionally used for compression: Parallelogram (left), Dual Parallelogram (center) and $K$-star (right, $K = 6$).

#### 2.3.4.2 Dual Parallelogram Prediction:

We have seen in previous sections that it was possible to average several applications of the parallelogram rule to decrease the prediction error, and that the weights derived from the spectral approach performed even better. However, it is usually possible to further reduce the error by using other prediction weights derived from our spatial approach. Applying (2.14) to the DPP stencil yields the following equation:

$$\begin{aligned} 0 = & \quad [\alpha_1 + 2\alpha_2 + 2\alpha_4 - 1]\,\mathbf{f} \\ & - [\alpha_1 + \alpha_2 + 3\alpha_4]\,\tfrac{\partial \mathbf{f}}{\partial u} \\ & + \left[\tfrac{\alpha_1}{2} + \tfrac{\alpha_2}{4} + \tfrac{9}{4}\alpha_4\right]\tfrac{\partial^2 \mathbf{f}}{\partial u^2} + \tfrac{3}{4}[\alpha_2 + \alpha_4]\,\tfrac{\partial^2 \mathbf{f}}{\partial v^2} \end{aligned} \tag{2.34}$$

The first order expansion imposes that:

$$\begin{cases} \alpha_1 = -1 - 4\alpha_4 \\ \alpha_2 = 1 + \alpha_4 \end{cases} \tag{2.35}$$

| Model | Classical | Optimal | $\Delta R/R$ |
|---|---|---|---|
| armadillo | 7.40 | 6.67 | 10 % |
| guardian | 6.85 | 5.95 | 13 % |
| gipshand | 5.54 | 4.66 | 16 % |
| Max Planck | 4.54 | 3.46 | 24 % |
| Neptune (uniform) | 4.43 | 3.55 | 20 % |
| guardian (uniform) | 6.39 | 5.65 | 12 % |

Table 2.5: Comparison of the original DPP prediction scheme of [Cohen-or et al. 2002; Sim et al. 2003] and the optimal Freelence approach [Kälberer et al. 2005]. The two first columns are the entropy of residuals, in bits per vertex, for an original quantization of 12 bpv (14 bpv for the finer *Neptune* mesh). The last column shows the improvement using the optimal weights.

We can easily verify that the DPP weights $\alpha_1 = 1$, $\alpha_2 = \frac{1}{2}$, $\alpha_4 = -\frac{1}{2}$ fulfill these requirements. However, substituting these weights into (2.34) yields:

$$0 = -\frac{1}{2}\frac{\partial^2 \mathbf{f}}{\partial u^2} dl^2 + o(dl^2). \tag{2.36}$$

This means that the DPP weights force a zero curvature along the $u$ direction. This makes sense if the mesh is aligned with the features of the surface it represents. This is the case for some CAD models when the designer has taken into account the anisotropy of the model. However, for most triangle meshes, there is no reason why the mesh should have such properties, so it is more natural to avoid privileging a direction. For this reason, under the usual assumption of a 2-smooth surface, we use *Variance Minimization* to determine the weights. The energy $E(\alpha) = b(\alpha)^T \Sigma^2 b(\alpha)$ to minimize is defined by:

$$b = \begin{bmatrix} \frac{1}{2}\alpha_1 + \frac{1}{4}\alpha_2 + \frac{9}{4}\alpha_4 \\ 0 \\ \frac{3}{4}[\alpha_2 + \alpha_4] \end{bmatrix} = \begin{bmatrix} -\frac{1}{4} + \frac{1}{2}\alpha_4 \\ 0 \\ \frac{3}{4} + \frac{3}{2}\alpha_4 \end{bmatrix} \tag{2.37}$$

from which we can derive the following first order condition:

$$\frac{\partial E}{\partial \alpha_4} = 2 + \alpha_4 \left(5 + 3\sigma_{0,2}^2\right) = 0 \tag{2.38}$$

And the corresponding weights are:

$$\begin{cases} \alpha_1 = \frac{3(1-\sigma_{0,2}^2)}{5+3\sigma_{0,2}^2} = \frac{1}{3} \\ \alpha_2 = \frac{3(1+\sigma_{0,2}^2)}{5+3\sigma_{0,2}^2} = \frac{2}{3} \\ \alpha_4 = -\frac{2}{5+3\sigma_{0,2}^2} = -\frac{1}{3} \end{cases} \tag{2.39}$$

This shows that the Freelence weights are optimal for smooth meshes, in the sense that they provide the smallest prediction error for all possible meshes.

To show the difference between the DPP and optimal weights in terms of compression rate, we implemented a Dual Parallelogram Compressor similar to Freelence [Kälberer et al. 2005]. In our experiments, we used both irregular and uniform meshes. They are illustrated in Figure 2.10. The table 2.5 compares the entropy of residuals for the DPP and Freelence predictors. The experiments show a constant improvement of around 16% using the optimal weights.

#### 2.3.4.3 Average Parallelogram Prediction

We also computed the weights for $K$-*star*. Because of symmetry, there are only two different weights, one for the creases ($\alpha_c$) of the star and the other for the tips ($\alpha_t$). There is another degree of freedom regarding how far the tip vertices (respectively the crease vertices) are from the center. Let $R_T(K)$ be the ratio between the distance from a tip vertex to the center and that from a crease vertex to the center for a given vertex degree $K$. Then, the neighbourhood has the following geometry:

$$\begin{cases} \mathbf{u_{i+1}} & = & \left(u + cos\left(\frac{2\pi i}{K}\right)dl, v + sin\left(\frac{2\pi i}{K}\right)dl\right) \\ \mathbf{u_{K+i+1}} & = & \left(u + R_T(K)cos\left(\frac{2\pi i}{K} + \frac{\pi}{K}\right)dl, v + R_T(K)sin\left(\frac{2\pi i}{K} + \frac{\pi}{K}\right)dl\right) \end{cases} \tag{2.40}$$

Using this neighbourhood, equation (2.14) at order 2 yields the following equations:

$$\begin{cases} \alpha_c + \alpha_t & = & \frac{1}{K} \\ \alpha_c \sum_{i=0}^{K-1} cos^2\left(\frac{2\pi i}{K}\right) + [R_T(K)]^2 \alpha_t \sum_{i=0}^{K-1} cos^2\left(\frac{2\pi i}{K} + \frac{\pi}{K}\right) & = & 0 \\ \alpha_c \sum_{i=0}^{K-1} sin^2\left(\frac{2\pi i}{K}\right) + [R_T(K)]^2 \alpha_t \sum_{i=0}^{K-1} sin^2\left(\frac{2\pi i}{K} + \frac{\pi}{K}\right) & = & 0 \end{cases} \tag{2.41}$$

and the $K$-star weights are:

$$\begin{cases} \alpha_t & = & \frac{1}{K}\frac{1}{1-[R_T(K)]^2} \\ \alpha_c & = & \frac{1}{K}\left(1 - \frac{1}{1-[R_T(K)]^2}\right) \end{cases} \tag{2.42}$$

Depending on the assumptions on the mesh, several canonical values exist for $R_T(K)$, which determine the shape of the canonical neighbourhood. We experimented with 2 configurations, that are shown in Figure 2.9:

- *Symmetric triangles*: All the triangles of the neighbourhood are the same. Or equivalently, the tip vertices are mirror images of the center vertex with respect to the edges joining two crease vertices. In that case, $R_T(K) = 2cos\left(\frac{\pi}{K}\right)$.
- *Equilateral tip triangles*: The tip triangles are equilateral, and $R_T(K) = cos\left(\frac{\pi}{K}\right) + \sqrt{3}sin\left(\frac{\pi}{K}\right)$. This approach is usually better than the previous one, and (expectedly) a lot better for uniform meshes.

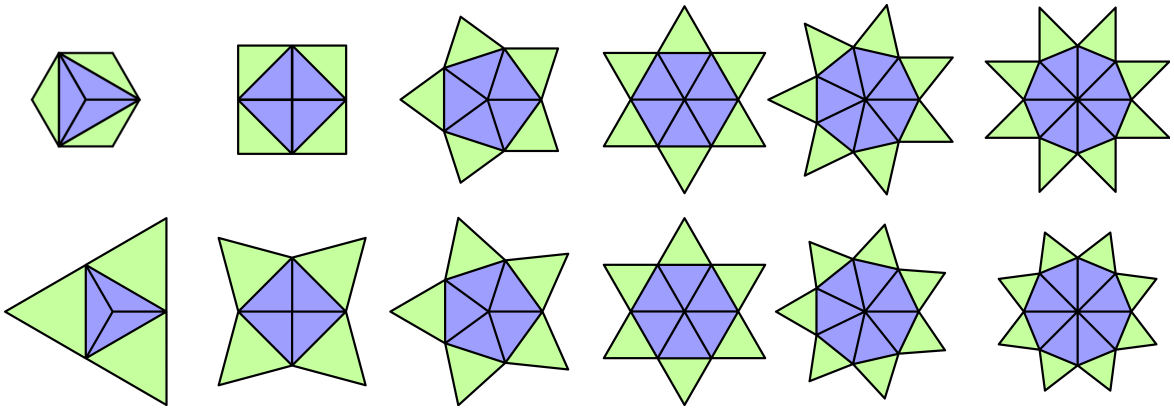

Figure 2.9: Geometry of canonical $K$-star neighbourhood configurations: *symmetric triangles* (top) and *equilateral tip triangles* (bottom).

In our experiments, we used both irregular and uniform meshes. They are illustrated in Figure 2.10.

| Model | vs. Vertex Degree | | | | | | Total |
|---|---|---|---|---|---|---|---|
| | 3 | 4 | 5 | 6 | 7 | 8 | |
| armadillo | ( 46) | 66 | 92 | 83 | 88 | 83 | 86 |
| guardian | 81 | 78 | 81 | 82 | 86 | 90 | 83 |
| gipshand | * | ( 98) | 93 | 57 | 84 | 87 | 61 |
| max Planck | ( 49) | 77 | 85 | 77 | 92 | 97 | 83 |
| Neptune (uniform) | * | ( 81) | 50 | 62 | 58 | * | 60 |
| guardian (uniform) | * | * | 51 | 56 | * | * | 55 |

Table 2.6: Prediction error of our method (with equilateral tip triangles) relative to Average Parallelogram Prediction for $K$-star neighbourhoods (in percents, i.e. $100 \times \frac{E_{new}}{E_{APP}}$). We only give details for $K \leq 8$, but the total if for all $K$. Parentheses denote statistically doubtable results (for which there were less than 50 occurrences). The asterisk denotes absence of data. Our method constantly diminishes the prediction error, especially for uniform meshes.

We first evaluate prediction error. Tables 2.6 and 2.7 summarize the results. We can see that the new predictors are far better than the traditional Average Parallelogram Predictor for all $K$. Compared to the approach of [Pajarola and Rossignac 2000a], the new predictors are roughly as efficient for general meshes, but win by a large margin for uniform meshes. If the mesh to compress is known to be mostly irregular, then the *topological distance* approach may be more efficient. If nothing is known about the mesh *a priori*, then the *equilateral tip triangles* strategy is the best approach. It is interesting to note that for $K$-star configurations, the Taylor approach also yields weights that are close to the least square weights (Table 2.3), even closer than the spectral weights.

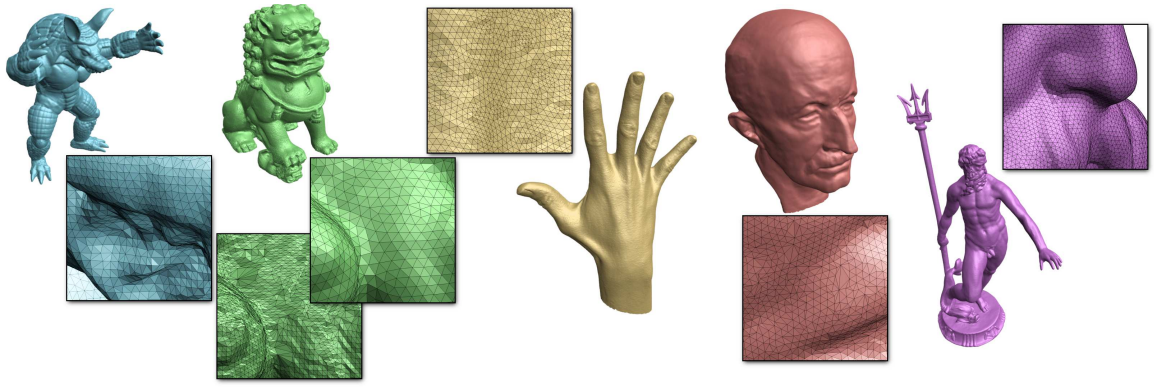

Figure 2.10: Some of the meshes used in our experiments, with close-up views to show their regularity. From left to right: *armadillo* [173 kv], *guardian* [656 kv], *guardian (uniform)* [153 kv], *gipshand* [137 kv], *max Planck* [199 kv], *Neptune (uniform)* [1.6 Mv].

We also compared the efficiency of the prediction schemes in terms of actual entropy, which gives a better measure of the efficiency of prediction weights. To compare the three different schemes (APP, CPM and ours), we implemented a geometry compression scheme similar to that of [Pajarola and Rossignac 2000a], where the original CPM predictor is presented. We only slightly modified the quantization scheme: Instead of a global quantization, we apply quantization in a local frame, as in Freelence [Kälberer et al. 2005]. The results are given in Table 2.8. We can see that even if Taylor and CPM weights were approximately as efficient in terms of prediction error, our Taylor predictor

| Model | vs. Vertex Degree | | | | | | Total |
|---|---|---|---|---|---|---|---|
| | 3 | 4 | 5 | 6 | 7 | 8 | |
| armadillo | (121) | (105) | 94 | 98 | 98 | 113 | 97 |
| guardian | 96 | 100 | 96 | 102 | 100 | 101 | 101 |
| gipshand | * | ( 96) | 87 | 106 | 97 | 73 | 102 |
| Max Planck | (118) | 103 | 98 | 106 | 101 | 100 | 102 |
| Neptune (uniform) | * | (103) | 57 | 53 | 45 | * | 52 |
| guardian (uniform) | * | * | 54 | 49 | * | * | 49 |

Table 2.7: Prediction error of our method relative to CPM prediction [Pajarola and Rossignac 2000a], for the equilateral triangles canonical neighbourhood geometry.

| Model | APP | CPM | Taylor | $\Delta R/R$ |
|---|---|---|---|---|
| armadillo | 15.83 | 13.10 | 10.67 | 18% |
| guardian | 13.28 | 12.31 | 11.94 | 3% |
| gipshand | 11.16 | 8.86 | 7.61 | 14% |
| Max Planck | 10.59 | 8.32 | 7.76 | 7% |
| Neptune (uniform) | 8.06 | 6.32 | 5.74 | 9% |
| guardian (uniform) | 14.27 | 12.87 | 12.25 | 5% |

Table 2.8: Comparison of the APP, CPM and Taylor (equilateral) prediction schemes, when applied to the geometry compression method described in [Pajarola and Rossignac 2000a], modified to use local quantization. The three first columns are the entropy of residuals, in bits per vertex, using a quantization of 14 bpv. The last column shows the improvement using the Taylor weights instead of CPM.

consistently improves compression compared to CPM, which is in turn superior to APP.

#### 2.3.4.4 High Degree Polygon Prediction:

The Taylor approach can also be used to derive weights to predict polygon geometry as in [Isenburg et al. 2005a]. To determine their prediction weights, Isenburg *et al.* also have to deal with an over-constrained system. Their heuristic is to use *weights norm minimization*. However, we found that making the assumption that $f$ was smooth and using *variance minimization* instead usually resulted in better prediction.

Figure 2.11 gives the prediction weights for the spectral method of Isenburg *et al.* [Isenburg et al. 2005a] and our spatial approach. In most situations, the weights are the same for the two methods. Such polygons are shown in blue on Figure 2.11. In some cases, however, the weights are different (polygons in green and red). For odd number of known points and for cases where only one vertex is missing, the system is simply constrained and there is no need for *variance* or *weights norm* minimization.

We have repeated the experiment conducted in [Isenburg et al. 2005a], and compared the results with our method. For a variety of models, we have computed the prediction error given by the weights from [Isenburg et al. 2005a] and the weights using our method. As these weights are the same as those of Isenburg *et al.* for pentagons, table 2.9 only report results for hexa-, hepta- and octagons. We can see that in nearly all cases, the prediction error was reduced. In the case where there are 4 known points of the octagon, results are heterogeneous. Although the error reduction is usually

biggest for higher degree polygons, the improvements in weights are most useful for the hexagon: For the meshes we use, approximately 70% of the vertices are predicted using hexagon rules, and 25% of the total number of vertices are predicted using the rule hexagon/v4. The problem noted by Isenburg *et al.* [Isenburg et al. 2005a] remains: It is sometimes better to use the octagon/v3 rule instead of octagon/v5 and octagon/v6 rules.

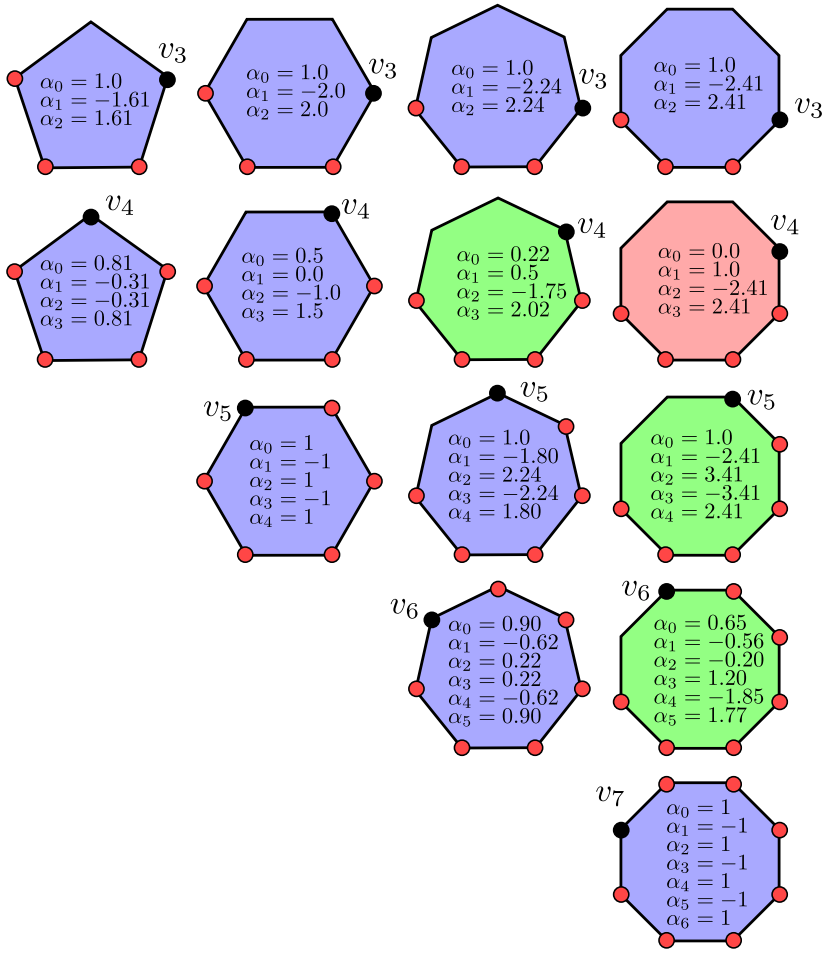


Figure 2.11: Polygon prediction weights. The black vertex is being predicted using the geometry of the red vertices. Blue polygons have the same prediction weights as [Isenburg et al. 2005a]. Other colours denote cases where the new weights perform always better (green) or sometimes better and sometimes worse (red) than [Isenburg et al. 2005a].

#### 2.3.4.5 Equivalence with Spectral Prediction for 3x3 Stencils:

Although developed for meshes in the irregular setting, our method can also be used to determine prediction weights on regular grids. In the following, we show that our method is actually equivalent to that of Ibarria *et al.* [Ibarria et al. 2007] on $3 \times 3$ stencils.

The spectral method of Ibarria *et al.* consists in finding the prediction that zeroes the high frequency content of the discrete cosine transform of $f$. This is equivalent to finding the prediction that reproduces the basis vectors $B_i$ of the DCT for low frequencies, i.e. the prediction weights $\alpha_i$,

| Model | | hexa | | | hepta | | | | octa | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | v3 | v4 | v5 | v3 | v4 | v5 | v6 | v3 | v4 | v5 | v6 | v7 |
| armadillo | I | 630 | 577 | 1 | 1249 | 1266 | 810 | 351 | 3335 | **3054** | 4923 | 2879 | 3 |
| | VM | | **545** | | | **1173** | | | | 3335 | **3939** | **2795** | |
| guardian | I | 9 | 8 | 1 | 66 | 67 | 43 | 19 | 151 | 158 | 177 | 107 | 6 |
| | VM | | 8 | | | 62 | | | | **151** | **146** | **104** | |
| fertility | I | 753 | 690 | 1 | 1522 | 1528 | 1025 | 445 | 2827 | **2677** | 3621 | 2347 | 3 |
| | VM | | **652** | | | **1425** | | | | 2827 | **3212** | **2279** | |
| gipshand | I | 371 | 340 | 1 | 1294 | 1359 | 694 | 301 | 2608 | 2994 | 6537 | 1385 | 9 |
| | VM | | **321** | | | **1230** | | | | **2608** | **1895** | **1344** | |
| max Planck | I | 126 | 116 | 1 | 1081 | 1060 | 789 | 342 | 3187 | **3156** | 4326 | 2485 | 8 |
| | VM | | **109** | | | **1004** | | | | 3187 | **3400** | **2412** | |

Table 2.9: Average prediction error when predicting the missing vertices of polygons. For each polygon of degree $K$, we computed the error when 3 to $K-1$ vertices were known. The figures have been scaled per model, 1 corresponding to the smallest error. For every model, each row respectively gives the results of [Isenburg et al. 2005a] (**I**), our method (**VM**). As the weights differ only for hexa-, hepta- and octagons, we only give the results for these polygons. (Note: The *Max Planck* model we used is finer than that of [Isenburg et al. 2005a], so the numerical results differ.)

$i \in [1,8]$ are such that:

$$\overline{B_i}.\alpha = B_{i,u} \tag{2.43}$$

where $u$ is the index of the sample to predict, and $\overline{A}$ is the matrix $A$ without its $u$-th row.

In the case of the $3 \times 3$ stencil, the basis vectors of the DCT are the same as the 'basis vectors' of our method, which are the polynomials in $du$ and $dv$. The Figure 2.12 shows the correspondence between the DCT basis and the values of $f$ that generate the same vectors. This shows that for the $3 \times 3$ neighbourhood, spatial smoothness (small high-order terms in the Taylor expansion) and spectral smoothness (small high-frequency content) are equivalent.

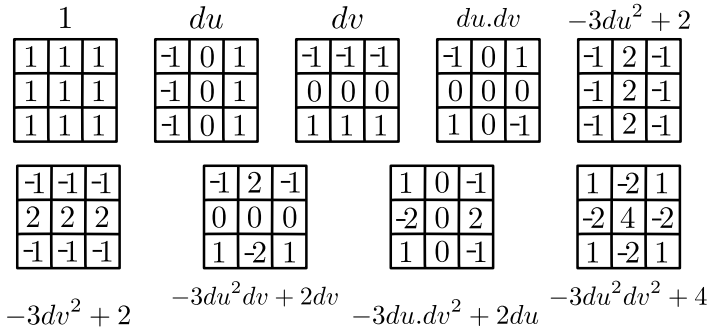


Figure 2.12: The nine basis vectors of the DCT used in [Ibarria et al. 2007] and the corresponding values of $f$.

### 2.3.5 Subdivision

Most subdivision schemes use linear prediction to define refinement operations. Therefore, our method can be used to compute refinement operators for various subdivision schemes [Zorin et al. 1996a; Loop 1987; Kobbelt 1996; Dyn et al. 1987; Labsik and Greiner 2000]. Although our approach does not give new predictors to be used in place of existing ones, it gives an alternate, *spatial* way of determining prediction weights, whereas other methods transfer the problem in the *spectral* setting. As the derivation is similar for all these methods, we only show how to derive the *modified butterfly scheme*

using our approach. In addition, we propose a new predictor for one of these methods [Labsik and Greiner 2000], that has smaller support but the same smoothness of limit surfaces.

#### 2.3.5.1 Modified Butterfly Scheme

The modified butterfly scheme [Zorin et al. 1996a] presented in Section 2.1.4 uses two different stencils depending on the regularity of the mesh around each newly added vertex (Figure 2.13), the regular case being the same as in the original butterfly scheme [Dyn et al. 1990]. The authors note that polynomial reproduction up to degree $n$ is a necessary condition for $n$-smoothness of the limit surface. They use the fact that the neighbourhood is symmetric to use the Discrete Fourier Transform (DFT). They translate the polynomial reproduction condition to the spectral domain, which, for polynomials of degree 2, constrains five of the eigenvalues of the subdivision matrix (see Section 2.1.4). The weights are determined by taking the inverse DFT of the vector of eigenvalues of the subdivision matrix, where these constraints have been met. They are given on Figure 2.13.

It is to be noted that these constraints are sometimes not sufficient to completely determine the weights, and sometimes too strong. In the case of an extraordinary vertex of degree $K$ (Figure 2.13, right), the polynomial reproduction condition fixes the second and $K$-th eigenvalues to $1/2$ and the first, third and $(K-1)$-th eigenvalues to $1/4$. In the case when $K=3$, the constraints for the first and second eigenvalues are incompatible. When $K>5$, there are eigenvalues that are not fixed. Both these situations need additional assumptions to uniquely determine the weights. We discuss them later in this section.

Because both spectral and Taylor approaches rely on polynomial reproduction to determine the weights, the Taylor prediction approach is equivalent to the modified butterfly scheme. However we do not transfer the problem from the spatial to the spectral domain. While we do not derive new weights for this subdivision scheme, the spatial domain approach can help justify in a more intuitive manner some of the choices made when all constraints cannot be met (e.g. at extraordinary vertices when $K=3$) or when there remain some degrees of freedom ($K \geq 7$). Therefore, we show below how to compute the modified butterfly weights using the spatial approach, in the ordinary and extraordinary case.

For ordinary vertices, the modified butterfly scheme uses the first stencil of Figure 2.13. Note that because of symmetry, $\alpha_1=\alpha_2$, $\alpha_3=\alpha_4$ and $\alpha_5=\alpha_6=\alpha_7=\alpha_8$. In the regular setting, let $dl$ be the length of edges. Affine invariance enables us to choose $\mathbf{u}=(0,0)$ and $\{\mathbf{u_1},\ldots,\mathbf{u_k}\}$ equal to:

$$\begin{array}{lcl}
\mathbf{u_1} & = & (u+\frac{dl}{2} \quad , \quad v \qquad\quad) \\
\mathbf{u_2} & = & (u-\frac{dl}{2} \quad , \quad v \qquad\quad) \\
\mathbf{u_3} & = & (u \qquad\quad , \quad v+\frac{\sqrt{3}}{2}dl) \\
\mathbf{u_4} & = & (u \qquad\quad , \quad v-\frac{\sqrt{3}}{2}dl) \\
\mathbf{u_5} & = & (u+dl \quad , \quad v+\frac{\sqrt{3}}{2}dl) \\
\mathbf{u_6} & = & (u-dl \quad , \quad v+\frac{\sqrt{3}}{2}dl) \\
\mathbf{u_7} & = & (u+dl \quad , \quad v-\frac{\sqrt{3}}{2}dl) \\
\mathbf{u_8} & = & (u-dl \quad , \quad v-\frac{\sqrt{3}}{2}dl)
\end{array} \tag{2.44}$$

Then by applying equation 2.14, we have:

$$\begin{array}{rl}
0 = & [2\alpha_1+2\alpha_3+4\alpha_5-1]\mathbf{f} \\
+ & 0\times\frac{\partial\mathbf{f}}{\partial u}dl+0\times\frac{\partial\mathbf{f}}{\partial v}dl \\
+ & 0\times\frac{\partial^2\mathbf{f}}{\partial u\partial v}dl^2+\frac{1}{4}[\alpha_1+8\alpha_5]\times\frac{\partial^2\mathbf{f}}{\partial u^2}dl^2 \\
+ & \frac{3}{4}[\alpha_3+2\alpha_5]\times\frac{\partial^2\mathbf{f}}{\partial v^2}dl^2+o(dl^2)
\end{array} \tag{2.45}$$

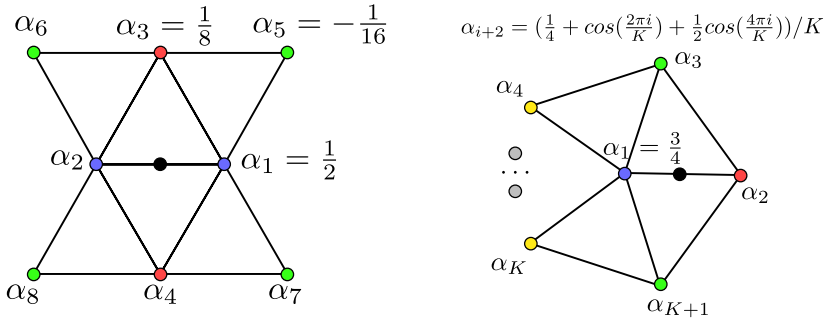


Figure 2.13: Prediction stencils of the modified butterfly scheme, around ordinary vertices (left), and extraordinary vertices (right). The vertex to predict is drawn in black, and identical colours indicate similar weights. $K$ is the valence of the extraordinary vertex.

From this equation, we can immediately see that to reproduce all polynomials of degree 2, the following conditions must be met:

$$\left\{\begin{array}{rcl} 2\alpha_1 + 2\alpha_3 + 4\alpha_5 &=& 1 \\ \alpha_1 + 8\alpha_5 &=& 0 \\ \alpha_3 + 2\alpha_5 &=& 0 \end{array}\right. \Rightarrow \left\{\begin{array}{rcl} \alpha_1 &=& +\frac{1}{2} \\ \alpha_3 &=& +\frac{1}{8} \\ \alpha_5 &=& -\frac{1}{16} \end{array}\right. \tag{2.46}$$

The two approaches being equivalent, these weights are the same as those of the butterfly scheme.

Around extraordinary points, the modified butterfly scheme uses the vertices of the 1-ring of the irregular vertex of degree $K$. The corresponding stencil is shown in Figure 2.13. This time, the neighbourhood is defined by:

$$\left\{\begin{array}{rcl} \mathbf{u_1} &=& \left(u - \frac{dl}{2}, v\right) \\ \mathbf{u_{i+2}} &=& \left(u + \left(-\frac{1}{2} + cos\left(\frac{2\pi i}{K}\right)\right) dl, v + sin\left(\frac{2\pi i}{K}\right) dl\right) \end{array}\right. \tag{2.47}$$

Expanding to the second order gives the following conditions:

$$\left\{\begin{array}{rcl} \alpha_1 + \sum_{k\geq 2} \alpha_k &=& 1 \\ \sum_{k\geq 0} \alpha_{k+2}.cos\left(\frac{2k\pi}{K}\right) &=& \frac{1}{2} \\ \sum_{k\geq 0} \alpha_{k+2}.sin\left(\frac{2k\pi}{K}\right) &=& 0 \\ \sum_{k\geq 0} \alpha_{k+2}.sin^2\left(\frac{2k\pi}{K}\right) &=& 0 \\ \alpha_1 &=& \frac{3}{4} \\ \sum_{k\geq 0} \alpha_{k+2}.sin\left(\frac{4k\pi}{K}\right) &=& 0 \end{array}\right. \tag{2.48}$$

These conditions are met by the weights of the modified butterfly scheme, except for $K = 3$. In this particular case, the resulting system is under-constrained for a first order Taylor expansion, but over-constrained at order 2:

$$\begin{array}{rcl} 0 = & & \left[\alpha_1 + \alpha_2 + 2\alpha_3 - 1\right]\mathbf{f} \\ & + & \left[-\frac{\alpha_1}{2} + \frac{\alpha_2}{2} - 2\alpha_3\right]\frac{\partial \mathbf{f}}{\partial u} dl \\ & + & \left[\frac{\alpha_1}{8} + \frac{\alpha_2}{8} + \alpha_3\right]\frac{\partial^2 \mathbf{f}}{\partial u^2} dl^2 + \frac{3}{4}\alpha_3 \frac{\partial^2 \mathbf{f}}{\partial v^2} dl^2 \\ & + & o(dl^2) \end{array} \tag{2.49}$$

The first order expansion imposes that:

$$\left\{\begin{array}{rcl} \alpha_1 &=& \frac{1}{2} - 3\alpha_3 \\ \alpha_2 &=& \frac{1}{2} + \alpha_3 \end{array}\right. \tag{2.50}$$

To determine $\alpha_3$ in the system above, an additional constraint is needed, that is derived from the second order reproduction constraint. In their scheme, Zorin *et al.* [Zorin et al. 1996a] manually pick the constraints that are ignored. In the spatial domain, it is possible to better justify the choice that we make. While it is not possible to zero the second order term in the expansion, it is possible to minimize it by using the *Variance Minimization* approach presented before. The quantity to be minimized is $E(\alpha) = b(\alpha)^T \Sigma^2 b(\alpha)$, where

$$b = \begin{bmatrix} \frac{1}{8}\alpha_1 + \frac{1}{8}\alpha_2 + \alpha_3 \\ 0 \\ \frac{3}{4}\alpha_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{8} + \frac{3}{4}\alpha_3 \\ 0 \\ \frac{3}{4}\alpha_3 \end{bmatrix} \tag{2.51}$$

In that particular case, we have

$$E(\alpha) = \left(\frac{1}{8} + \frac{3}{4}\alpha_3\right)^2 + \frac{9}{16}{\alpha_3}^2 + \frac{3}{32}\sigma_{0,2}^2\alpha_3\left(1 + 12\alpha_3\right) \tag{2.52}$$

The first order condition is $\frac{\partial E}{\partial \alpha_3} = \frac{3}{16}(1 + 12\alpha_3)(1 + \sigma_{0,2}^2) = 0$. The optimal weights do not depend on $\sigma_{0,2}^2$, and $\alpha_3 = -\frac{1}{12}$, $\alpha_1 = 3/4$, $\alpha_2 = 5/12$, which are the same weights as in the scheme of Zorin *et al.*.

In the cases $K = 4$ and $K = 5$, the system obtained via the spatial approach is consistent, and yields once again the same weights as the modified butterfly scheme.

For $K >= 7$, the system is under-constrained for polynomial reproduction at order 2. To fix the remaining weights, Zorin *et al.* choose to take the non-constrained eigenvalues equal to zero. They justify the choice by the *simplicity* of the approach. In the spatial domain, however, it is easier to find a rationale for this choice. As the desired smoothness (order 2) is already reached, Variance Minimization does not help here, but Weights Norm Minimization can be used. This technique, that is initially valid only in the spatial domain, can be used retroactively to justify the choice made by Zorin *et al.*. Because the (inverse) DFT is a *unitary* transformation, the norm of the weights vector is the same as the norm of the vector of eigenvalues. Making the remaining eigenvalues zero actually minimizes the norm of the vector of eigenvalues, thus also minimizing the norm of the weights. In this case, it the norm is:

$$\|\alpha\|^2 = \left(\frac{3}{4}\right)^2 + \frac{1}{K}\|\frac{1}{4}, \frac{1}{2}, \frac{1}{4}, 0, \ldots, 0, \frac{1}{4}, \frac{1}{2}\|^2 = \frac{1}{16}\left(9 + \frac{11}{K}\right). \tag{2.53}$$

Although we presented here only the example of modified butterfly subdivision, the process that we use to derive subdivision weights is general. Since a lot of subdivision schemes use the polynomial reproduction condition to derive their weights (e.g. [Zorin et al. 1996a; Loop 1987; Kobbelt 1996; Dyn et al. 1987]), our method can also be used to derive their weights.

#### 2.3.5.2 Interpolating $\sqrt{3}$ subdivision

The previous subdivision schemes (modified butterfly, loop, b-spline) split each edge in two and each face in 4 subfaces. In contrast to this approach, Kobbelt [Kobbelt 2000] proposed a subdivision method that converged *slower* than traditional 1-to-4 subdivision. At each subdivision step, a vertex is added at the center of each face, and the face is split in 3 subfaces. Then, edges are flipped to rebuild a regular triangulation. His method is factored into a prediction step which is a simple average rule with weights $\{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, and an update step which uses the one-ring neighbourhood $p^{n+1} = (1 - \alpha_K)p^n + \frac{\alpha_K}{K}\sum_{q \epsilon OneRing(p)} q$. $\alpha_K$ is determined so that the subdivision matrix has the correct *sufficient* eigenstructure that guarantees $C^1$ smoothness of the limit surface according to the criterion of [Reif 1995].

Labsik and Greiner [Labsik and Greiner 2000] later proposed an *interpolating* (non-lifted) scheme based on the same subdivision process. They determined the weights using the same approach as [Zorin et al. 1996a] that is discussed above. They use the predictors of Figure 2.14. The weights for these predictors are derived from the usual polynomial reproducing necessary condition, and thus our method gives the same results. Labsik and Greiner also show that their scheme has $C^1$ limit using the subdivision matrix eigenstructure.

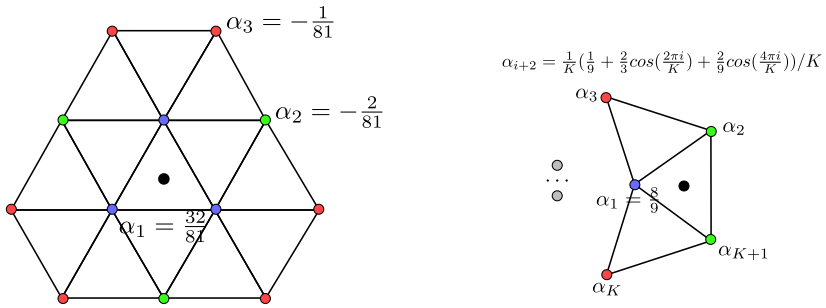


Figure 2.14: Prediction stencils of the interpolating $\sqrt{3}$ scheme of Labsik and Greiner [Labsik and Greiner 2000], around ordinary vertices (left), and extraordinary vertices (right). The vertex to predict is drawn in black, and identical colours indicate similar weights. $K$ is the valence of the extraordinary vertex.

However, we found that it was possible to derive another interpolating prediction scheme, which is a lot simpler, but nevertheless also guarantees convergence to $C^1$ limit surfaces. This scheme has a smaller support (the prediction neighbourhood has 6 vertices instead of 12) and thus enables faster iteration with barely noticeable difference (see Figure 2.16). The prediction scheme uses the stencil of Figure 2.15. The convergence analysis can be found in Appendix 7.6.

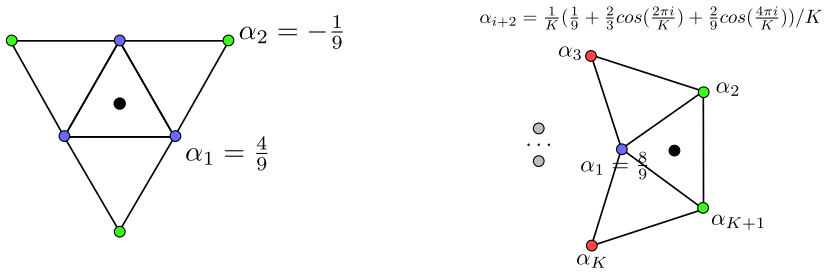


Figure 2.15: Prediction stencils of our interpolating $\sqrt{3}$ scheme, around ordinary vertices (left), and extraordinary vertices (right). The stencil is smaller than for the scheme of Labsik and Greiner [Labsik and Greiner 2000] (Figure 2.14), but also guarantees $C^1$ smoothness.

## 2.4 Conclusion

In this chapter, we have presented two approaches to improve geometry prediction. Both remain within the *predict and correct* paradigm used by nearly all mesh compression methods. This paradigm has the advantage of simplicity and computational efficiency (only a few elementary operations are needed), and will therefore probably remain a widely made choice when implementing compression algorithms.

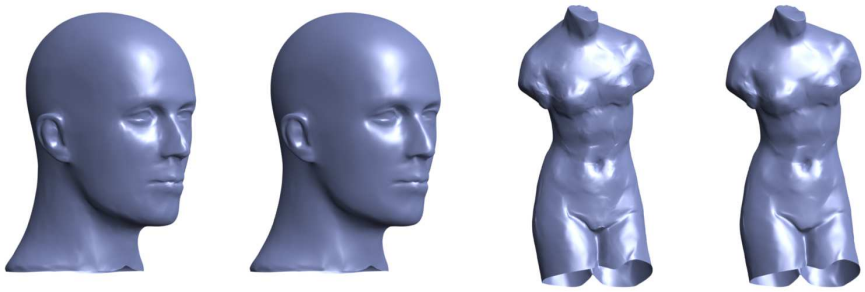

Figure 2.16: Subdivision results for our approach (left) and that of Labsik and Greiner (right). The left subdivision scheme has a twice smaller support.

Spectral approaches had already been used to derive prediction weights in the case of grid settings (i.e. images) [Ibarria et al. 2007]. We have extended this approach to irregular settings found in meshes and shown that although this technique performed well in some cases, it could not be reliably used for extrapolation. In particular, it results in a systematic prediction error in the widely used average parallelogram predictor. This failure shows that a method for deriving prediction weights should take into account the geometry of the neighbourhood instead of being purely based on its combinatorial properties.

Then, we have proposed another completely different approach. Instead of considering spectral smoothness (i.e. predicting small high frequency content) like previous methods [Isenburg et al. 2005a; Ibarria et al. 2007], we have considered smoothness in a spatial context, by using the Taylor expansion of the local geometry function, and predicting this time small high order content. We have shown that this method did not have the drawbacks of the spectral approach and could handle extrapolation very well. We have provided three examples – Dual and Average Parallelogram and High Degree prediction – where the Taylor weights were better than usual weights derived from experience of spectral approaches.

It is important to keep in mind that this approach is limited by the well known *Runge phenomenon* [Runge 1901]. For larger neighbourhoods, increasing the interpolation order by further expanding the Taylor approximation can lead to large oscillations. In these cases, a better approach consists in stopping the expansion at a lower order, which constrains prediction weights, but leaves an under-determined system. A unique solution can then be determined by minimizing the norm of the prediction weights.

The spatial approach is appealing since it provides a completely generic way of deriving linear prediction weights for a large spectrum of mesh processing methods. In addition, the assumptions are very simple, and theoretical results are therefore quite easy to obtain. The genericity of the technique could find several applications in mesh processing. As an example, we have used our method to derive wavelet coefficients, but we think that it can be used in various other environments.

# Chapter 3

# Handling Large Meshes: State of the Art

Most of the mesh compression algorithms presented in Sections 1.3 and 1.4 assume that the entire mesh fits in main memory. This poses a problem for very large meshes, since typical representations for these meshes are very large, eventually becoming bigger than the available main memory. For example, using a halfedge [Campagna et al. 1998] mesh representation uses 480 bits per vertex. The 186 million vertices "Michelangelo's St. Matthew"[1] model uses 11 GB of memory, which is beyond the reach of typical desktop computers.

In some cases, *compression* can be carried out with large computing resources. However, the end user – who will need to *decompress* the mesh – usually has smaller available resources, namely a desktop workstation. Therefore, for meshes of intermediate size, the problem is asymmetric, i.e. algorithms may use larger resources for compression than for decompression. For very large meshes, however, it is desirable to compress a mesh that resides on a disk rather than inside the core memory. Algorithms that have this property are named *Out-Of-Core* (OOC).

Three solutions have been proposed to compress large meshes. The following sections detail each of these approaches.

- Some methods cut the mesh into smaller pieces and compress them one by one [Ho et al. 2001] or employ external memory structures that page mesh parts from disk when needed [Isenburg and Gumhold 2003]. These methods are typical examples of *Out-Of-Core* processing.

- The streaming paradigm is a kind of OOC method that represents the mesh as a *stream* of interleaved vertices, elements, and finalization tags. A few compression methods have been designed that can encode such *streaming meshes* [Isenburg and Lindstrom 2005] on-the-fly as they stream in [Isenburg et al. 2005b; 2006a].

- *Random-accessible* methods enable efficient *decompression*, by providing a compressed mesh data structure from which the decompressor can query only the parts it is interested in [Choe et al. 2009; Yoon and Lindstrom 2007; Kim et al. 2010; 2006; Du et al. 2009; Jamin et al. 2009].

[1] http://graphics.stanford.edu/data/mich/

## 3.1 Out Of Core

The earliest attempts at the compression of large meshes used the OOC paradigm. Because the available amount of core memory was limited, these algorithms simply chose to keep the whole mesh on the disk and work on smaller pieces that could be loaded from disk and would fit in memory (see Figure 3.2).

Ho *et al.* [Ho et al. 2001] split a large mesh $M$ into $N$ pieces of approximately the same size, using a graph cutting approach. They make an initial pass on the mesh to compute the mesh bounding box. Then, they build a *coarse weighted graph* $G$ in one pass over the mesh. They divide the bounding box into a grid of size $D \times D \times D$, with $D = 64$ or $D = 128$. Then, they traverse the mesh triangles. For each vertex of a triangle $t$, they compute in which grid cell $C$ it falls, and create a node for $G$ at the center of $C$ if there is no such node yet (the node has a weight of 0). If the three vertices of $t$ fall within the same cell, they increase the cell weight by 1. If $t$ has two vertices in one cell $C_A$ and one vertex in cell $C_B$, then the weight of edge $(C_A, C_B)$ is increased by 1 (the edge is created if it does not exist). This process results in a graph $G$, that is hopefully small enough to fit in-core. Each node (resp. edge) of $G$ has a weight that represents how many triangles of the original mesh are mapped to it.

Ho *et al.* then use METIS [Karypis and Kumar 1998] to find a *balanced partition* (in terms of weights) $(P_i)_{i=1...N}$ of $G$ with *small edge cuts*. This partition of $G$ induces a partition $(P'_i)_{i=1...N}$ of $M$ that has the following properties:

- $P'$ is *balanced* – in terms of number of triangles per piece – since the sum of the weights of the vertices in each $P_i$ represents exactly the number of triangles in $P'_i$.
- $P'$ has a simple boundary structure, i.e. there are few triangles that span two mesh pieces. This results from the small edge cuts of $P$.
- The pieces $P'_i$ are spatially coherent, since they result from snapping vertices onto a grid.

After the mesh has been partitioned, the compression process uses $N$ further passes over the mesh. For each pass $i$, the method extracts the triangles of $P_i$ by repeating the above vertex grid snapping process, keeping triangles that snap to grid cells associated with the nodes of $P_i$. This way, it builds an in-core mesh that is compressed using a single-rate in-core algorithm [Touma and Gotsman 1998].

To deal with the boundary of the pieces, they propose two alternatives:

- The first one is to code the pieces in a totally independent manner, by duplicating the vertices shared by several pieces. This has the advantage of being very easy to implement, but the obvious drawback is that there can be a large overhead associated with the duplicated vertices – up to 50% of the vertices can be coded twice. However, a careful choice of the partitioning algorithm results in less than 1% of duplicated vertices in most cases.
- When the exact connectivity of the original mesh must be recovered – i.e. duplicating vertices is not an option – they provide a *gluing* process that stitches vertices across adjacent partitions. At any time in the decoding process, the algorithm maintains a list $C_L$ of boundary vertices that have not yet been glued to. This list is organized in strips of consecutive boundary vertices. When the connectivity of a piece $S$ of the mesh is fully decoded, its boundary vertices can be glued to the vertices of $C_L$. The algorithm first identifies one vertex $v$ in $C_L$ that corresponds to the first vertex in the boundary $B_S$ of $S$. This is done by explicitly specifying the index of $v$ in $C_L$ and $B_S$. Then, the correspondence between the other vertices of $B_S$ and those of $C_L$ can be specified at minimal cost by *zippering* the vertices of $B_S$ with those of $C_L$ starting from $v$ (see Figure 3.1). Only the zippering direction and the number of vertices to be glued need to be given. When all the gluing is done, the geometry of the remaining vertices is decoded.

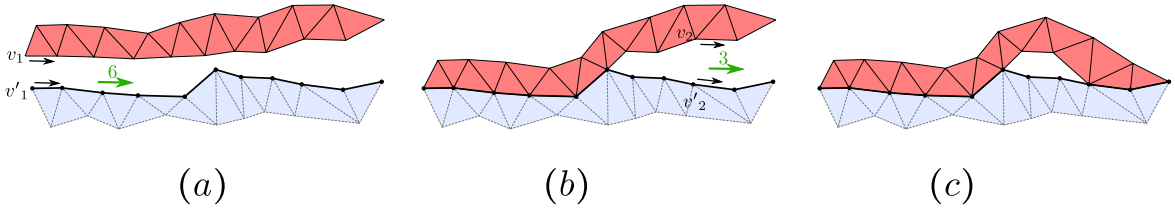


Figure 3.1: The boundary gluing process using zippering. $C_L$ is drawn in bold, $S$ is the red part, and the previously decoded – and released – part is in light blue. $(a)$ The correspondence between $v_1$ and $v'_1$ is given by specifying the index of $v_1$ (resp. $v'_1$) within $C_L$ (resp. $B_S$) . Then, the zippering directions are given (arrows). Finally, the number of vertices to glue are specified (green arrows). (b) This is enough information to glue a strip of vertices. (c) This step is repeated to glue a second part.

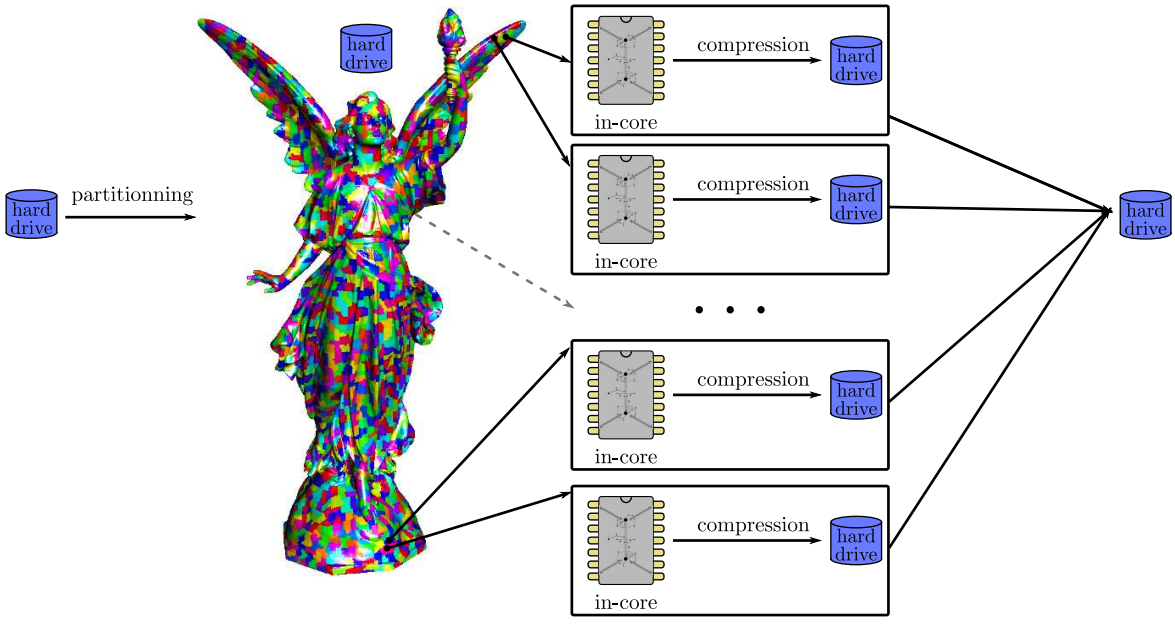


Figure 3.2: The Out-Of-Core compression pipeline: The mesh is split into smaller pieces that can be handled by a traditional compression algorithm.

Later, Isenburg and Gumhold presented another method also based on a partition of the input mesh in a set of *clusters* [Isenburg and Gumhold 2003] using a graph partitioning method similar to that of the previous approach. Then, instead of making $N$ passes over the mesh that each builds an in-core cluster and compresses it on the fly, they do *only two* passes over the mesh and sort the vertices and faces of the mesh in several files, each of which represents a single cluster *in an out-of-core way*. Then, they apply two additional passes to ensure that each of these files can be loaded as an in-core mesh, by creating the necessary adjacency relationships between vertices and faces in each cluster. Thus, they have a complete out-of-core representation of the mesh that provides transparent access to its elements.

In a second phase – and only *after all* the clusters have been fully created – they apply a region-growing compression process similar to that of Touma and Gotsman [Touma and Gotsman 1998]. Only the clusters that are being traversed by the active region boundary are kept in-core (see Figure 3.3). In contrast to [Ho et al. 2001], there is no need to explicitly handle cluster boundaries, since everything functions as if clusters did not really exist. However, great care must be taken in order to minimize cache misses, that occur whenever an element is processed that is not in one of the clusters that are stored in-core. When the latter happens, a new cluster is loaded from the disk, which is an expensive operation. Isenburg and Gumhold use two complementary methods to ensure this:

- When a specific gate of the boundary has been processed, the next gate is chosen so that it minimizes the chance of triggering a cache miss. In practice, the more remote along the boundary the next gate is, the greater the chance of it being within another cluster, and therefore the higher the probability of a cache miss. Therefore, where optimizing the compression bit rate would drive them to choose as next gate a *zero-slot* gate [Isenburg 2002], the authors choose to limit the search to a distance of +/- 10 gates along the boundary (starting from the current gate). If there is no such gate, they stay at the current position, lest choosing a better gate costs a cache miss.

- When a split operation occurs (see Section 1.3), they process the shorter of the two resulting boundary loops first. This improves the locality of references and reduces the memory footprint.

Using this technique enables making compression in only one pass once the OOC representation has been built, as in the original algorithm of Touma and Gotsman [Touma and Gotsman 1998]. This enables 100 times faster decompression than the algorithm of Ho *et al.* [Ho et al. 2001]. In addition, the bit rates are approximately 25% better.

The latter algorithm prefigures the *streaming paradigm* that we will describe in the next section. During the compression process, once the active boundary has left a cluster, this cluster will not be used again (except if another boundary of the stack uses it). Everything works as if the cluster had been *finalized*. The streaming paradigm takes this idea one step further by *explicitly* specifying finalization and using actual faces instead of face clusters.

## 3.2 Streaming

In the streaming paradigm [Isenburg and Lindstrom 2005], the mesh is represented as an interleaved sequence of vertices, triangles, and *finalization tags* (Figure 3.5). A vertex is introduced just before the first triangle that references it, and is released just after the last face that references it (see Figure 3.4 for an example). Finalization tags specify when a vertex is referenced for the last time. This information enables the compressor to continuously release and reuse data structures, making it possible to compress gigantic meshes which cannot be handled by non-streaming algorithms. For example, the streaming triangle mesh compressor of Isenburg *et al.* [Isenburg et al. 2005b] encodes the 186 million vertices "St. Matthew" model using less than 5 MB of main memory. Note that in

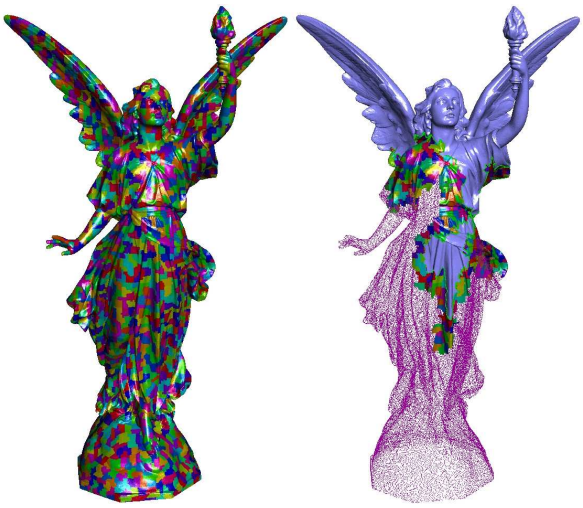

Figure 3.3: Out-of-core compression of the "Lucy" model with the approach of [Isenburg and Gumhold 2003]. The left picture shows the partition of the mesh. The right picture illustrates the compression process. The already compressed clusters are rendered as points, while the rest is rendered as triangles. The cached clusters are coloured. The boundary of the active region is drawn in green. (Figure taken from [Isenburg and Gumhold 2003]).

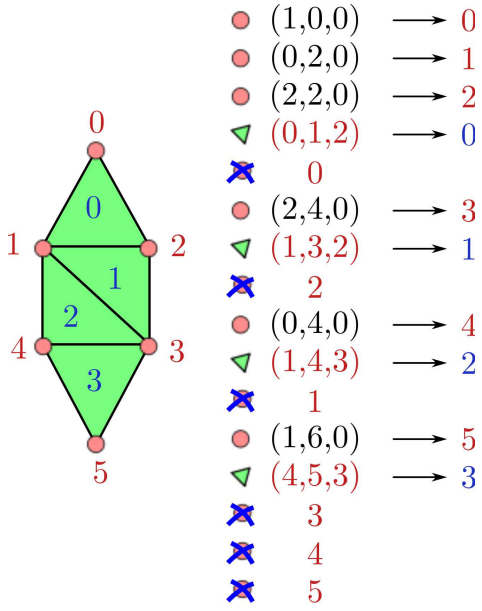


Figure 3.4: The simple example mesh of Figure 1.7, specified as a streaming mesh. The crossed vertices denote finalization tags.

order to have a *streaming decompression*, the compressor must transmit the finalization tags to the decompressor (else, the decompressor cannot release finalized elements). This adds some (usually small) overhead compared to non-streaming approaches.

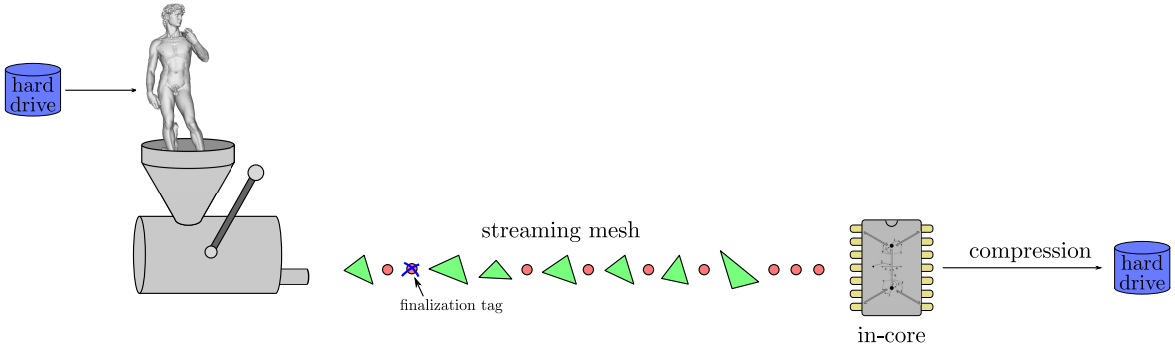


Figure 3.5: The streaming compression pipeline: The mesh is fed to the compressor as an interleaved sequence of vertices, triangles and finalization tags.

The compression process is simple: The coder maintains an *active boundary*, that consists in the vertices and edges that have been introduced but not yet finalized. Each time a new face is read, the coder specifies to which of the active vertices or edges this face is adjacent. The face may be adjacent in 8 different ways to the active boundary (see Figure 3.6). These 8 situations correspond to the 8 connectivity compression symbols that are entropy coded. If the face introduces new vertices, the position of these vertices is predicted and residual coded. The coder then writes finalization information and releases finalized vertices.

The memory footprint of the compressor/decompressor is directly linked to how long the vertices remain in memory. A mesh with local references will use less memory than one where vertices remain active during a long time. To illustrate the quality of a streaming mesh, Isenburg and Lindstrom use two-dimensional *layout diagrams* [Isenburg and Lindstrom 2005]. The vertices are indexed along the vertical axis, in the order in which they appear in the input file. The cells are indexed on the horizontal axis, in the same order. The vertices of the $i$-th cell are drawn as points on the $i$-th column, and a vertical line connects them. Intuitively, the length of this line indicates the locality of vertex references (see Figure 3.7). Similarly, a horizontal line is drawn for each $j$-th vertex which connects all cells that reference it. Thus, the layout diagram gives a visual understanding of the coherence of a mesh: The closer the points and lines group around the diagonal, the more coherent the layout is, and the more efficient compression is – in terms of both memory consumption and bit rates.

Several methods naturally generate streaming meshes [Lorensen and Cline 1987; Silva and Mitchell 1998; Bernardini et al. 1999; Ju et al. 2002; Mascarenhas et al. 2004; Isenburg et al. 2006b]. They have either been designed to provide this feature [Mascarenhas et al. 2004; Isenburg et al. 2006b], or just happen to generate coherently organized elements [Lorensen and Cline 1987; Silva and Mitchell 1998; Bernardini et al. 1999; Ju et al. 2002]. In the latter case, algorithms can be modified to interleave elements and provide finalization information. On the other hand, existing models must be converted to the streaming format. If the mesh is coherent enough, this can be done efficiently via *vertex compaction* [Isenburg and Lindstrom 2005]. When this is not the case, Isenburg and Lindstrom have designed several methods to reorder meshes and provide more coherent layouts [Isenburg and Lindstrom 2005].

Because the faces of the mesh are compressed as they stream in, the order in which they appear is preserved. This means that in addition to the adjacency relationships stored by traditional compression algorithms, streaming compressors compress an additional information: face order. Therefore,

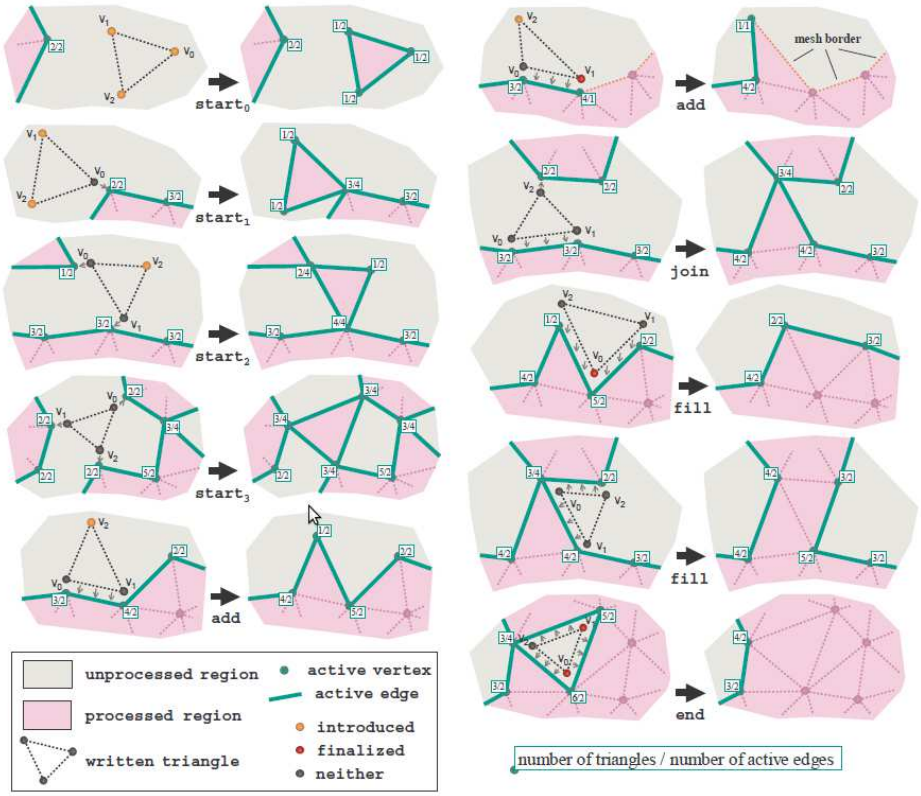


Figure 3.6: The eight possible adjacency configurations between the written triangle and the active vertices and half-edges maintained by the compressor of [Isenburg et al. 2005b]: a $start_x$ triangle is not adjacent to any active half-edge, but may be adjacent to zero, one, two, or even three active vertices; an *add* triangle is only adjacent to one active half-edge, with the third vertex being newly introduced; for the similar *join* configuration this third vertex is already active; a *fill* triangle is adjacent to two half-edges and an *end* triangle is adjacent to three half-edges. The small boxes show the triangle count and number of active half-edges. (Figure from [Isenburg et al. 2005b]).

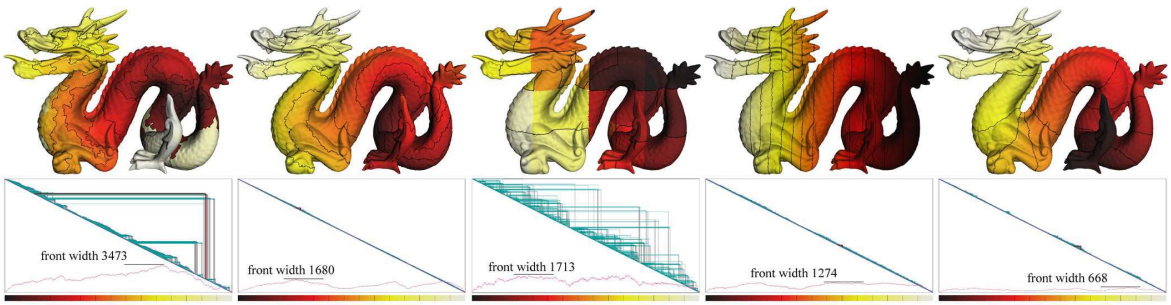


Figure 3.7: Streaming layout diagrams for various reorderings of the "dragon" model. From left to right: depth-first, breath-first, z-order, spatial sort, spectral sequencing. (Figure from [Isenburg and Lindstrom 2005]).

streaming compressors usually have a worse compression ratio than traditional methods. To alleviate this problem, Isenburg *et al.* have proposed to use a *delay buffer* [Isenburg et al. 2005b] to cache a fixed number of faces. By reordering the faces within this buffer, they are able to increase compression rate by loosening the ordering constraint to the cost of increased memory consumption during encoding.

In addition to using drastically less memory than traditional compression methods, streaming compressors are also very fast because they can interleave I/O and computations. They also benefit from better memory locality. The authors have later generalized their streaming triangle compressor to tetrahedral volume meshes [Isenburg et al. 2006a]. In Chapter 4, we propose two methods for streaming compression of hexahedral meshes. One is inspired from the triangle mesh compressor of Isenburg *et al.* [Isenburg et al. 2005b], and the other is a simple modification of the non-streaming Hexzip [Lindstrom and Isenburg 2008] compressor to enable streaming compression.

## 3.3 Random Access

In the previous section, we have presented the streaming mesh paradigm, which enables handling large meshes by algorithms that process *the whole mesh*. This approach is useful for algorithms that traverse the mesh and use a small neighbourhood around the current element. In the context of mesh compression, this is the case as long as the order in which the decompressor traverses the mesh can be fixed by the coder, and this is what most mesh compression methods in the literature have assumed. However, there are various situations in which this is not the case. A program may only need to access a certain part of a mesh, either because the algorithm is designed that way, or because the user himself is only interested in a small part of the mesh.

In the context of interactive visualization of large meshes, being able to decode only a part of a mesh is a required feature. Indeed, visualization is inherently a *time-critical* process. Each frame must be rendered in an appropriate time frame. As the amount of available processing power is limited, there is a limit to the number of primitives that can be processed during the time available to compute a frame. If a model is too large to fit in memory, the rendering must be done *from the compressed version*. The streaming approach described in the previous section makes it *possible* to realize on-the-fly decompression and rendering, however the whole mesh would have to transit through the decompressor. For very large meshes, this will be too long to render the frame in *timely* manner. Therefore, interactive visualization algorithms require individual access to only parts of a mesh.

Data structures that have the property of enabling individual access to their elements without decoding the rest of the structure (there may exist a small sub-linear overhead) are said to be *random-accessible*. For example, the simple indexed triangle mesh representation of Section 1.1 enables random access to any element by its index: triangles being stored contiguously in memory on a constant number of bytes, decoding the $i$-th triangle of the mesh can be done in $O(1)$ by reading the 3 vertex indices at position $3i$ in the array of faces, and then reading the coordinates of the three associated vertices in the vertex table. Note that the random accessibility property that holds for triangle meshes does not extend to polygon meshes with mixed face elements, since the offset is no longer a multiple of the face index. To be able to determine the offset of a face of index $i$, one must know the degree of all faces of index $j < i$, and the overhead is no longer sub-linear. Therefore the indexed polygon mesh structure is not random accessible.

Although interest for random access in meshes is quite recent [Choe et al. 2004], random access is a common feature found in many other signal processing applications. For example, the MPEG standards for sound and video coding enable random access by the means of a *index table* that specifies memory offsets that serve as entry points in the compressed stream. This enables the user to skip to any part of the file without having to wait for a full decoding of the stream. This is

particularly important for video decoding: Decompression performance for h.264 High Definition is typically around 60 frames per second – a few times real time performance. This means that without random accessibility, a user who wants to skip to the middle of a two hours movie would have to wait for several minutes, as if he was using a video tape ! Note that most random accessible decompression algorithms only deal with *one-dimensional* data. In the case of video, only the *time* dimension is random accessible. In contrast, providing random accessibility in the case of multi-dimensional data, as images or volumes, is a harder problem that has found only a few solutions [Ihm and Park 1999; Rodler 2001; Lefebvre and Hoppe 2007].

Random accessibility is defined relatively to an access means. In the indexed structure above, the elements are accessed through their index. Mesh processing algorithms usually do not rely on indices to access elements. Generally, they will randomly pick elements through *adjacency* or *geometric* criteria. Data structures that enable random access with respect to one means of access will generally not provide this service for other means. For example, indexed meshes enable random access in $O(1)$ through indices, but they do not provide any mechanism to randomly query adjacent elements given an element, or request all elements at a specified position – both are in $O(n)$.

We classify random accessible mesh structures in three categories (that are not exclusive), depending on the type of random accessibility that they provide:

- **Index-based Queries:** Like the indexed triangle mesh data structure presented above, elements are randomly accessed through the use of their indices. As this information cannot generally be linked to the connectivity or geometry of the mesh, this property is generally of limited interest.

- **Connectivity Queries:** These structures enable randomly querying the adjacent elements of an element. This type of random accessibility is required by plenty of mesh processing algorithms (e.g. Laplacian smoothing). In most mesh processing libraries (*VTK*, *CGAL*, *OpenMesh*, ...), this feature is implemented through the use of pointers to adjacent elements, which is very efficient in terms of speed, but makes the representation redundant [Weiler 1985; Guibas and Stolfi 1985; Campagna et al. 1998; Weiler 1988; Kettner 1998]. A compact data structure with this property was designed by Yoon and Lindstrom [Yoon and Lindstrom 2007], that enables around $20:1$ compression with $O(1)$ adjacency queries. They even demonstrate improved speed for some algorithms because of the reduced bandwidth use associated with data compaction.

- **Geometric Queries:** Some algorithms require a random access to the primitives lying in a given geometric domain. For example, ray tracing algorithms need to determine which primitives cross the path of a ray. This is usually achieved using a *bounding volume hierarchy* (BVH) [Rubin and Whitted 1980; Lin and Manocha 2003; Teschner et al. 2005], using bounding spheres [Hubbard 1993], axis-aligned boxes [Bergen 1997], or other domains [Gottschalk et al. 1996; Klosowski et al. 1998]. However, the additional data used to provide random access adds a large overhead to the raw mesh data. It is generally possible to trade granularity for size: the two limit cases would be on one hand a bounding volume hierarchy with only one bounding domain containing the whole mesh, that has virtually no overhead for storing the bounding volume hierarchy but poor random access performance; and on the other hand a hierarchy where each leaf domain contains only one element, with minimum granularity but a large memory overhead.

Designing compression algorithms that result in random accessible structures is not an easy task. We have seen above that to provide random accessibility other than index-based queries, additional information must be stored that inflates data size and introduces redundancy. In addition, all compression methods use some kind of code length reduction process which results in symbols being coded on a variable (integer or rational) number of bits. Therefore, the position of a specific element in memory does not only depend on its index, but also on the compressed size of all previous elements in

memory. Hence, the only possible options are (1) to decode all the previous elements, which is what traditional compression algorithms do; or (2) to store additional information to determine where in memory the compressed data for the element resides. This overhead information must be *small* – else compression will not be efficient – and *random accessible* – else the problem has just been displaced.

All existing random accessible mesh compression algorithms (with the exception of [Piperakis and Kumazawa 2001], that uses a radically different approach that can hardly be considered mesh compression) choose to provide *block* random accessibility. As for traditional compression, there exist both *single-rate* and *progressive* approaches, that we present in the following sections.

### 3.3.1 Single-rate

The simplest way of providing random-accessible compression is to split the mesh into several parts that are compressed independently. Within each part, a traditional compression algorithm is used, that does not provide random accessibility. The overhead information, stored in the file header, consists in storing in which part the elements of interest are to be found.

Choe *et al.* have applied this idea to the compression of triangle meshes [Choe et al. 2004; 2009]. They split the mesh in relatively flat *charts* that are coded independently. To generate the charts, the authors use a method that departs from that used by out-of-core approaches (see Section 3.1). Starting from several *seed faces* uniformly distributed on the mesh, each defining a chart, they use a *region growing* process using a cost function that favours as-flat-as-possible, balanced charts. Once growing is finished, they reposition the seed faces towards the centroids of the charts and iterate the growing process until the chartification converges. At the end of this process, they obtain a partition with reasonably flat and balanced charts.

As in the approach of Ho *et al.* [Ho et al. 2001] (see Section 3.1), naively compressing charts in an independent manner would result in multiple coding of the vertices of the chart boundaries. Therefore, the authors encode these vertices independently in the file header, using the concept of *wire-net mesh* (see Figure 3.8). In the wire-net mesh, each chart $C$ is represented by a polygonal face $f_C$. The vertices of the original mesh that are shared by more than two charts are the vertices of the wire-net mesh. The sequence of vertices of the original mesh that are shared by exactly two charts $C_A$ and $C_B$ are assigned to the edge adjacent to faces $f_{C_A}$ and $f_{C_B}$. This sequence of vertices is called a *wire*.

Using these concepts, a mesh is represented by three things:

1. The wire-net mesh, which is compressed using a single-rate coder. As the faces of the wire-net mesh are polygons, possibly with high degree, a polygon mesh compressor must be used. The authors have chosen to use that of Khodakovsky *et al.* [Khodakovsky et al. 2002]. We would like to point out a minor improvement that could be made to their geometry compression scheme. To code the vertex positions of the wire-net mesh, Choe *et al.* predict the position $\bar{b}$ of the barycentre of each face using parallelogram prediction (by mirroring the barycentre of an adjacent face). Then, they predict the position of each vertex of the face as $\bar{b}$, and code the residual. This suffers from a systematic prediction error, as already noted in [Isenburg et al. 2005a], and may certainly be improved by using the approach of [Isenburg et al. 2005a] or by using Taylor prediction (Section 2.3).

2. The *wires* of vertices corresponding to the edges of the wire-net mesh. Storing the connectivity of the wires is *free* since they are a simple sequence of connected vertices. Therefore, only the geometry of the wires has to be stored.

3. The *charts* corresponding to the faces of the wire-net mesh. The authors note that the only primitives that cannot be shared among charts are *faces*. Using a face-based connectivity compression algorithm thus enables coding charts in a completely independent manner. Therefore,

Choe *et al.* use the Angle Analyzer algorithm [Lee et al. 2002] to code the connectivity of the charts. They also enhance the coding efficiency using two complementary methods:

- Angle Analyzer is a region-growing method, and is usually initialized by choosing a seed triangle as conquered region, and by using its three edges as active boundary. Choe *et al.* note that using the *chart boundary* as starting active boundary (and considering that the chart *exterior* is the conquered region) has two benefits. First, the boundary is already decoded, hence no information needs to be stored to initialize the traversal. Then, they show that traversing the chart *inwards* leads to less symbol dispersion than *outwards* traversal, and thus smaller file size.
- The coder makes use of geometry information to predict the next connectivity symbol, as in the Freelence approach [Kälberer et al. 2005]. The authors report an improvement of about 5-10% in compression ratio using this technique.

This technique is very efficient as far as compression rates are concerned, since it uses a single-rate coder and has a small overhead as long as the number of charts is small. However, the overhead increases with granularity, which limits the number of charts for efficient compression. Since the charts are entirely decoded, and the wire-net mesh stores the adjacency between charts, the algorithm enables **random adjacency** queries. There is also a **geometric random accessibility**, which is linear in the number of charts: All the polygons of the wire-net mesh are tested for intersection with the requested domain, and the intersecting charts are decoded. However, this approach can fail in certain cases (see Figure 3.9). For example, if the query domain does not intersect any face of the coarse mesh (Figure 3.9, center), then there is no way to determine whether or not the domain is empty. Although the search is linear in the number of charts, the algorithm enables *random access* in the sense that geometric random access uses only $K$ operations, where $K$ is the number of charts and is small (usually around 100).

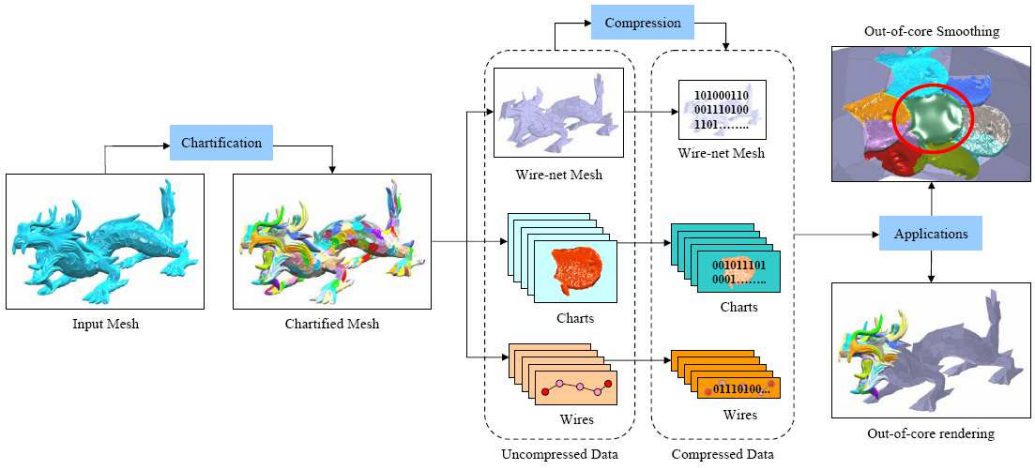


Figure 3.8: The random accessible framework of Choe *et al.* (Figure taken from [Choe et al. 2009]).

The compression algorithm of Yoon and Lindstrom [Yoon and Lindstrom 2007] combines the benefits of a streaming coder (Section 3.2) with that of block-wise random access. They compress the mesh as it streams in using the compressor of Isenburg *et al.* [Isenburg et al. 2005b], but split the mesh in clusters having the same number $S_c$ of triangles each. In contrast to the previous chart-based approach of Choe *et al.*, they cannot choose the shape of the clusters, since they pack the mesh elements together as they stream in. Therefore, the shape of the clusters depends on the layout of the streaming mesh, and can be totally different from one layout to another (see Figure 3.10). In

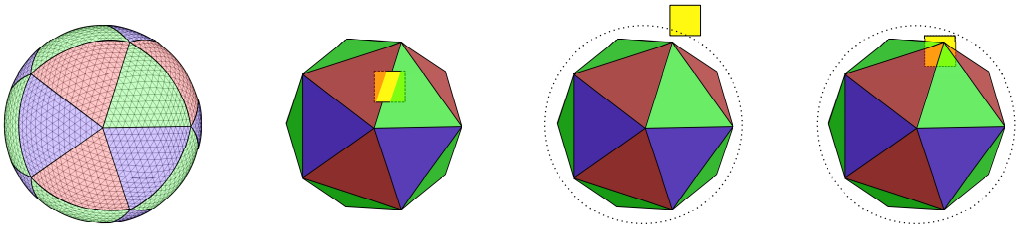

Figure 3.9: Geometric random accessibility for chart-based methods [Choe et al. 2004; 2009]. The mesh is a sphere, and the left picture shows the associated chartification (colours). The query domain is the yellow box, and the dotted circle shows the silhouette of the original sphere. From left to right, we show a successful query (the green and red charts are decoded), a negative failure (the blue chart is not decoded since the wire-net mesh does not intersect the domain, but the sphere intersects it), and a positive failure (the blue chart is decoded since it intersects the domain, however the original sphere does not intersect it). Note that a positive failure is acceptable, since it only occurs at the cost of additional processing time: By decompressing the chart, the algorithm can see that the mesh does not really intersect the domain. On the other hand, the negative failure **is** a problem: The chart is not decoded, therefore the algorithm cannot correct the error, and the results may become incorrect.

particular, compared to the previous approach, there is no guarantee that the resulting clusters will be geometrically compact.

Figure 3.10: Clusters of [Yoon and Lindstrom 2007] for different layouts. From left to right: Cache-oblivious, z-order, spectral, depth-first and breadth-first layouts. This figure highlights clusters of 8K consecutive triangles for different layouts of the Puget Sound terrain simplified to 512K triangles. The cluster colours smoothly vary with the sequential layout from red to yellow to green, and the brightness alternates between each consecutive pair of clusters. The cache-oblivious mesh layout has high spatial coherence, leading to well-shaped clusters with short boundaries and few inter-cluster references. As a result, it yields the highest compression ratio and best runtime performance. Figure from [Yoon and Lindstrom 2007].

To enable random access, a header stores the position of each cluster in memory. Note that because they use the streaming compressor of Isenburg *et al.* [Isenburg et al. 2005b], they preserve the original ordering of the mesh. This enables providing **index-based random accessibility**. Is is easy to see that the $i$-th triangle can be found in the $\lfloor \frac{i}{S_c} \rfloor$-th cluster, whose position can be found in the header. Hence, only this cluster must be decompressed, and the requested triangle is the $(i \bmod S_c)$-th in the cluster. The algorithm also enables **random adjacency queries**. Each time a triangle is read from a cluster, the whole cluster is kept in memory in a halfedge structure, enabling $O(1)$ access to adjacent elements. The only difficulty is at boundaries, because the vertices at a cluster boundary can be stored in another cluster (in they are adjacent to a triangle with lower index). To deal with this problem, Yoon and Lindstrom also store in the header the index of the first vertex of each cluster. Therefore, when a boundary vertex is queried that does not reside in the

current cluster, the decompressor can locate and decompress the cluster that contains this vertex. Order-preserving compression has another advantage: if the mesh has a cache-coherent layout, the compressed structure is also cache-coherent. This enables algorithms to run from the compressed version efficiently. Yoon and Lindstrom apply their technique to isocontouring of terrain data and obtain speed-ups of around 2.5 compared to the uncompressed version.

The approach of Yoon and Lindstrom has very recently been extended to provide geometric random access [Kim et al. 2010]. First, the mesh is stored using the original mesh compression scheme. In addition, a hierarchy of axis-aligned bounding boxes (AABB) is computed, such that each box has two child boxes, and its leaves are individual triangles. Each box is defined by its extents in the $x$, $y$ and $z$ direction. To compactly encode them, the extents of the box in each direction are quantized with respect to the extents of the parent box. The hierarchy of AABBs is then compressed using the same cluster-based approach originally used to compress the mesh [Yoon and Lindstrom 2007], except that they cluster the nodes of the AABB tree instead of vertices and faces. The position (in memory) of each AABB tree node cluster is indexed in the header. Coding the structure and geometry of the AABB hierarchy uses 50 to 100 bits per vertex, roughly multiplying the total data size needed to represent the mesh by 3 compared to the original approach ([Yoon and Lindstrom 2007]). Even if it is not very efficient as far are compression rates are concerned, it is the only scheme that enables all three random access queries (indexed, adjacency and geometric). However, the efficiency of geometric random access will be greatly conditioned by the original mesh layout. Because clusters may not be localized in the spatial domain (e.g. in the case of a *spectral*, *depth-first* or *breadth-first* layouts), decoding a few spatially close elements may result in a lot of thrashing.

### 3.3.2 Progressive Meshes

In the previous section, we presented several algorithms which tackled the problem of random access by splitting the mesh in blocks. By using traditional single-rate compression algorithms, they reduced the difficulty to handling chart/cluster boundaries. Providing *progressive* random accessibility is a harder problem.

Progressive meshes based on the edge collapse/vertex split algorithm are not *locally refinable*: At decoding time, splitting a given vertex may require a large number of parent splits to recover the correct neighbourhood to make the split. In the worst case, the progressive mesh may even have to be refined to the original mesh [Kim and Lee 2001]. This goes against random access. Ideally, any vertex of a random accessible progressive mesh would be splittable without having to split any other vertex. Kim and Lee have designed a selective refinement scheme that has the property that splitting an arbitrary vertex of the mesh can be done without triggering a full chain of vertex splits [Kim and Lee 2001]. Additionally, the vertex split only modifies the mesh in the vicinity of the split vertex. Therefore, the levels of detail of the mesh can change abruptly between two regions of the mesh (see Figure 3.11).

They later proposed a compression approach based on this refinement scheme [Kim et al. 2006], that enables connectivity random access (any vertex can be split at any time) and a sort of geometric random access (the mesh can be refined within a certain domain until the original mesh is reached). They code the hierarchy of vertex splits in a concise manner that we will not detail here. We only note that they also resort to *blockwise coding* to represent the hierarchy of vertices. Each sub-block is coded using arithmetic coding and a header stores the position of the sub-blocks, as in the single-rate approaches of the previous sections. As in previous approaches, the random access granularity is the size of a block. They obtain compression rates of around 11 bpv for connectivity and 21 bpv for geometry. Although very interesting from a theoretical point of view, this scheme cannot achieve interactive performance for large models. It also suffers from the same drawback as the single-rate scheme of Choe *et al.* [Choe et al. 2004] for geometric random access, but on an extended scale. Starting from the base mesh, and given a space domain, only refining the intersecting faces

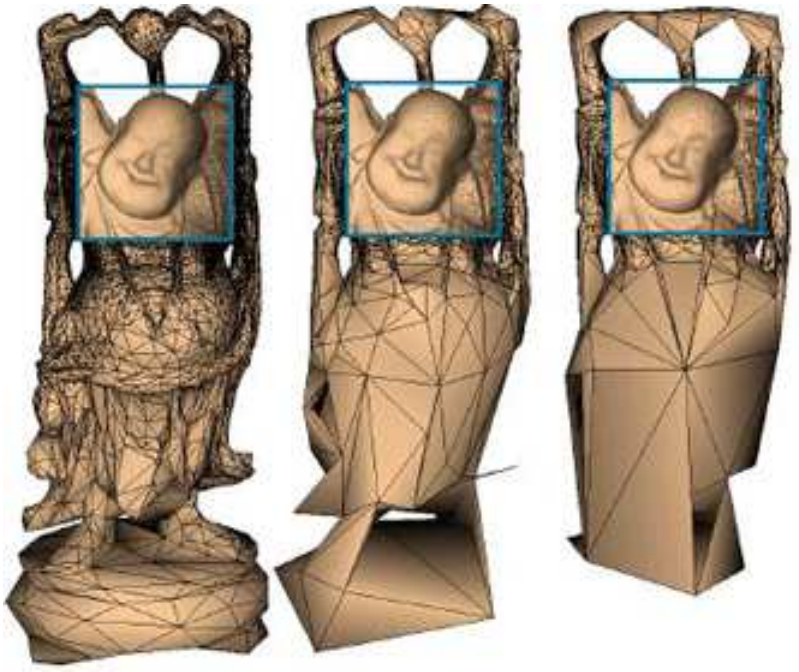

Figure 3.11: Selective refinement of the Buddha model using the methods of [Xia and Varshney 1996], [Hoppe 1997] and [Kim and Lee 2001] (from left to right). The latter algorithm enables faster transition between levels of detail. (Figure from http://home.kookmin.ac.kr/~junho/).

of the base mesh does not guarantee that all the faces that fall within the query domain will be decompressed, because refining other faces may result in new vertices that fall within the viewport. Also, if the query domain does not intersect any face of the coarse mesh, there is no way to determine whether or not the domain is empty, and if it is not the case, which vertices must be split to recover the queried elements.

Recently, Du *et al.* have proposed a geometry-driven random-accessible scheme [Du et al. 2009] based on the work of Gandoin and Devillers [Gandoin and Devillers 2002]. They split the kd-tree in two *layers*. The first layer consists of the top $A$ of the tree, from the root to a certain level $L$. The second layer consists in the $2^{L+1}$ subtrees rooted in the leaves of $A$. Each of these trees is coded independently as a block, and the boundaries between blocks are also coded independently. As in other approaches, a header indexes the position of each block. During decoding, the first layer $A$ is *always* decoded, providing a coarse approximation to the mesh. Once the coarse mesh is decompressed to its finest level, each block of the second layer can be decoded to any level, enabling different levels of detail in different regions of the space. To resolve dependency at the boundary between adjacent blocks, a boundary is decoded if one of the blocks that use it is decoded *at any level*. In this case, the boundary is always *fully decoded*. The authors apply this technique to view-dependent visualization (Figure 3.12), to refine only the part that falls within the viewport. Note that this approach can compress non-manifold meshes, since it uses the compressor of [Gandoin and Devillers 2002].

Another independent scheme, named *CHuMI viewer*, uses the approach of Gandoin and Devillers to design a random-accessible compressed mesh structure [Jamin et al. 2009]. In contrast to the previous approach, they do not divide the kd-tree in two layers. Instead, they build a more complete hierarchy of so-called $n$SP-blocks that are the individual random-accessible elements in the compressed file. The hierarchy is a tree where each node ($n$SP-block) is split $n$ times in all three dimensions ($x$, $y$, $z$), and its children are the resulting $n^3$ $n$SP-block. The authors choose $n$ as a power of 2 ($n = 2^{p_l}$).

Figure 3.12: Geometry-driven, random-accessible progressive compression using the approach of Du *et al.*. Only the head of the "David" model is decompressed fully, while the rest of the model remains at the coarse resolution level of $A$. (Figure from [Du et al. 2009]).

This way, $p_l$ bits of precision are gained at each level of the tree. The value of $n$ for the root node is chosen separately, using a larger number of precision bits: $n_r = 2^{p_r}$. Typical parameter values are $p_l = 4$ and $p_r = 7$, which means that a precision of $p = 15 = 7 + 4 \times 2$ bits will be reached with a random access tree having 5 levels. Within each block, the mesh is coded using the algorithm of Gandoin and Devillers [Gandoin and Devillers 2002], and the binary data for all the blocks is aggregated in breadth-first order in the compressed file. Using this scheme alone would result in a complete tree. As the density of primitives per cell of the kd-tree usually varies with the position on space, Jamin *et al.* only split a $n$SP-block if the number of primitives it contains is above a threshold $N_{min}$ (whose value is typically on the order of a few thousands). This enables more granularity in areas where there are a lot of mesh elements, and also decreases the random access overhead in regions with smaller primitive density – by disabling random access where it is least useful. Note that this results in a non-complete tree.

In order to enable random access, the algorithm stores in the file header the structure of the $n$SP-block tree, where each node stores a pointer to the beginning of the associated $n$SP-block data. That way, the decompressor can decompress any part of the mesh with the granularity of an $n$SP-block (see Figure 3.13). The boundaries are dealt with by duplicating the data, which leads to a compression rate overhead, but greatly simplifies the compressor and decompressor. The latter can just ignore boundary problems until the selected part of the mesh is decompressed. The discrepancy in precision between adjacent blocks is removed by simply deleting the duplicate vertices with lower geometric precision. Compared to [Du et al. 2009], this approach enables a more efficient decompression in terms of speed. In addition, the algorithm provides an implicit bounding volume hierarchy with the granularity of the $n$SP tree, which enables random geometric queries without the drawbacks of failed refinement, as the single-rate approach of Choe *et al.* [Choe et al. 2004; 2009] and the progressive one of Kim and Lee [Kim et al. 2006].

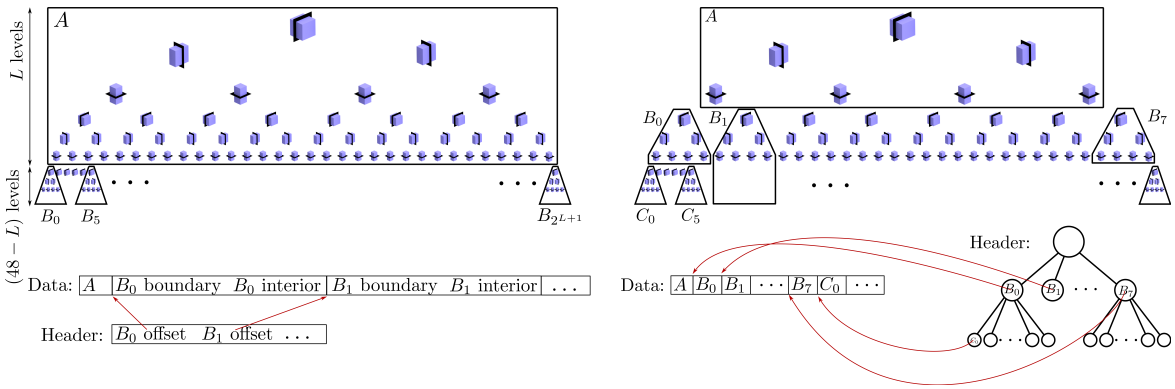


Figure 3.13: Comparison of the block layouts of the random-accessible progressive compression methods of [Du et al. 2009] (left) and [Jamin et al. 2009] (right). In the former approach, the structure of the random access tree is fixed to a two-level complete tree, and stores boundary elements at the beginning of each block. The latter has a more general random access tree, and blocks are totally independent since shared (boundary) elements are duplicated. On the right, we show a simple example where $n = 2$, $p = 3$, $p_r = 1$, and $p_l = 1$.

### 3.3.3 Discussion

All the previous approaches to random accessible compression, both single-rate and progressive, use block-based coding to enable random access. This limits granularity to the size of a block. Also, as they all use some kind of header to index the position of the blocks in memory, there is a strong

dependency between the number of blocks and the size of the resulting compressed data structure. Therefore, there is always a trade-off between granularity and file size.

In addition, we note that only one of the single-rate schemes for mesh compression explicitly uses a bounding volume hierarchy [Kim et al. 2010]. In the other algorithms, geometric random access is always performed by linearly traversing the blocks. As long as the number of blocks remains small, this is not a problem since the linear search will remain very fast. However, as models get bigger, number of blocks will increase if granularity is to remain the same. Therefore, we think that designing more efficient, sub-linear schemes is interesting. We show in Section 5 that increasing the number of charts in the previous approaches results in a granularity and random access times that are both in $O(\sqrt{N_v})$. However, this choice may drastically increase file size. As an alternative, we present a hierarchical scheme that uses a bounding volume hierarchy and has inherently $\sqrt{N_v}$ granularity and random access time.

The major problem when dealing with random-accessible mesh compression is the handling of *boundaries* between blocks/charts/clusters/pieces. Algorithms deal with this problem either by replicating the data (which decreases compression efficiency), or independently compressing the boundaries and specifying which block makes use of which boundary. This poses another problem, concerning dimensional scaling of random accessible compression algorithms. For a mesh of dimension $d$, the amount of data on the boundary is proportional to ${N_v}^{1-\frac{1}{d}}$. As $d$ increases, the amount of data on the boundary grows, making boundary handling an even more important problem.

Evaluating progressive random accessible compression methods is difficult. The usual rate/ distortion approach used in the case of classical compression is of no use here, since the decompression process can no longer be seen as simply decoding a stream up to a certain point. Random access exactly strives to avoid sending data as a *stream*, but provides ways to directly skip to the interesting data. Any evaluation of a random accessible scheme will necessarily be *subjective*. In particular, metrics must integrate how much the user cares for the region of interest compared to the rest of the mesh. If only the requested part is of interest, then a measure of the efficiency may be the ratio between the number of elements of the requested part and the total number of decoded mesh elements. However, having a coarse representation of the mesh, as in progressive approaches [Kim et al. 2006; Du et al. 2009; Jamin et al. 2009] (and to a lesser extent [Choe et al. 2009]), is useful in some applications. Therefore, performance metrics may want to take this feature into account.

We also want to mention a radically different approach, that uses neural networks to compress oriented manifolds without boundary [Piperakis and Kumazawa 2001]. This method has nothing to do with all other methods: it compresses neither connectivity nor geometry, but only represents an approximation to a surface. It provide answers to a single type of geometric query: Given a point in 3D space, the answer is a value between −1 and 1 that indicates whether this point is inside (1) or outside (−1) the volume delimited by the mesh. This type of compression provides extremely compact representation of the mesh (only the neural net structure and its weights have to be stored) – usually on the order of hundreds of bytes. In addition, the access is truly random: Every query is answered in constant time. The only drawback of this approach is the training time of the neural network: the authors report compression times ranging from days to weeks.

# Chapter 4

# Streaming Compression of Hexahedral Meshes

Numerical scientific simulation aims at describing the physical processes of the real world with as much precision as possible. To achieve this goal, and converge to a result closer to the actual solution, the space is discretized using finer and finer elements, leading to increasingly larger datasets. Most simulated spatial domains are either 2D domains embedded in $\mathbb{R}^2$, or 3D domains within $\mathbb{R}^3$; surfaces in $\mathbb{R}^3$ are seldom used, since they do not represent general real-world objects. In this section, we address the more challenging case of volume meshes (i.e. 3D domains within $\mathbb{R}^3$). Indeed, the number of elements of these meshes roughly increases as $\frac{1}{p^3}$, where $\frac{1}{p}$ is the required precision, whereas surface meshes have approximately $\frac{1}{p^2}$ elements. Because of this property, it is no longer uncommon to see meshes with tens to thousands million elements in 3D scientific simulations (see e.g. Figure 4.15). Processing these meshes is a challenge even with powerful computers.

Although mixed element meshes are sometimes used, most meshes only have a single cell type: For 3D problems these are either tetrahedra or hexahedra, because this simplifies the implementation of numerical solvers. Hexahedral meshes have received a lot of attention because of several desirable properties: they usually enable building meshes with fewer elements and exhibit better numerical behavior in various problems [Benzley et al. 1995]. Although the automatic generation of hexahedral meshes tends to be more difficult than that of tetrahedral meshes, there are now a number of algorithms [Blacker 1996; Muller-Hannemann 2001; Staten et al. 2005] that can generate high-quality hexahedral meshes (see Figure 4.1) with little or no user intervention.

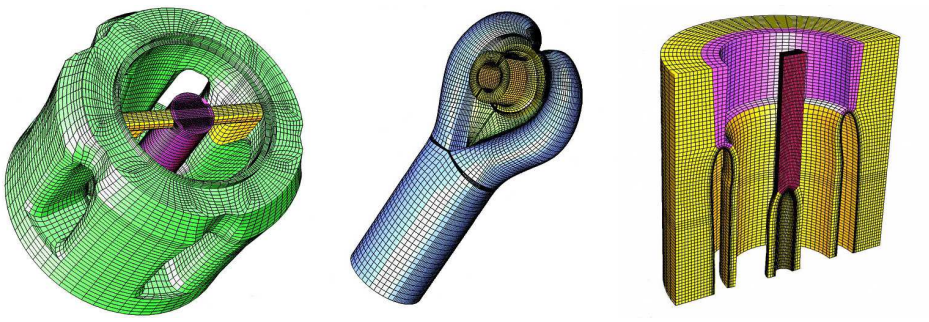

Figure 4.1: Some examples of high-quality hex meshes generated with the *TrueGrid* software (http://www.truegrid.com).

The main field of application of unstructured hexahedral meshes is scientific simulations – the graphics community often resorts to ray-traced structured voxel grids or "onion peeled" transparent surfaces when dealing with volume data. Because of that, there are often mesh *attributes* (e.g. pressure, temperature or velocity values) attached to the vertices or cells of the dataset. The attributes attached to the vertices can be compressed using the same prediction methods used for vertex positions; however there is currently no way of dealing with *cell* attributes, for reasons that will be made clear in Section 4.2.3. Yet as there may be several such attributes (our test dataset "Cedre" contains 12 quantities attached to the cells) the total bit budget dedicated to them can be very high. Therefore, we devised a compression scheme specifically targeted towards cell data, specifically designed so that it does not hinder streamability.

This chapter introduces two streaming methods and compares them to existing methods for the compression of hexahedral meshes:

- In Section 4.1, we review existing techniques: *Degree coders* choose how they traverse the mesh and code the sequence of vertex or edge degrees. They inherently destroy vertex and cell ordering. *Hexzip* [Lindstrom and Isenburg 2008] is a completely lossless coder that preserves both the vertex and cell order.
- Then, Section 4.2 describes how to design a streaming hexahedral mesh coder. This coder is an intermediate between the two previous methods (i.e. degree coding and Hexzip). Its preserves the global cell order but not that of the vertices.
- We also explain (Section 4.3) how to modify the *Hexzip* coder to trade compression speed for memory efficiency by introducing streamability.
- In a fourth section (4.4), we detail the implementation of a streaming hex mesh generator based on the *transfinite* method used by the open-source mesher GMSH [1]. This enables the generation of *already compressed* meshes that have never transited in uncompressed form through a temporary (e.g. Hard Drive) storage.
- Finally, we show the advantages of the streaming approaches when compressing very large meshes with several million hexahedra.

## 4.1 Existing Hexahedral Mesh Compression Schemes

Only three algorithms have been published specifically on the subject of hexahedral mesh compression [Isenburg and Alliez 2002a; Krivograd et al. 2008; Lindstrom and Isenburg 2008]. Isenburg and Alliez [Isenburg and Alliez 2002a] extend the concept of degree coding (see Section 1.3.1.1) to compress the connectivity of hexahedral meshes. Their algorithm uses a region-growing approach that traverses the connectivity graph one hexahedron at a time. They record the degree of all new edges (and a few special symbols) which allows the decoder to replay this traversal. Since hexahedral meshes are very regular, this method is extremely efficient, and entropy coding of edge degrees results in connectivity compression rates that range from 1.55 bph (bits per hexahedron) down to 0.18 bph on our test set.

The algorithm of Krivograd et al. [Krivograd et al. 2008] is based on the same idea, but uses vertex instead of edge degrees. They first compress the quadrilateral boundary surface of the volume mesh which becomes the initial hull. This hull is then grown as in Isenburg and Alliez's method [Isenburg and Alliez 2002a], but *inwards*, until there are no more cells left. They obtain rates similar to [Isenburg and Alliez 2002a] on regular grids but are worse on irregular models. Also, their method is more complex to implement because they have to deal with numerous special cases.

[1] http://www.geuz.org/gmsh

Lindstrom and Isenburg [Lindstrom and Isenburg 2008] recently proposed the *Hexzip* compressor that we evoked in Section 1.3.1.1. This compressor uses a radically different approach that neither reorders vertices nor hexahedra and is therefore *completely* lossless. It also handles non-manifold meshes or degenerate elements. They compress connectivity directly in its indexed form by predicting the eight indices of a hexahedron from preceding ones. This works because hexahedral meshes found in practice tend to have regular strides between indices of subsequent hexahedra. Their algorithm is an order of magnitude faster and has lower memory consumption than [Isenburg and Alliez 2002a] because it does not reconstruct and traverse the mesh connectivity. Instead, the cells are compressed as soon as they are read, and they are destroyed immediately after they have been output to the compressed stream. This enables very efficient input/output and local memory access. The connectivity compression rates of this method strongly depend on regularities in the indexing and are as high as 20.4 bph on our test set.

In addition to these three schemes, that are specialized for the treatment of hex meshes, Prat *et al.* [Prat et al. 2005] have developed an algorithm to compress arbitrary manifolds – including hexahedral meshes. The genericity of their method negatively impacts compression rates (+400% on average compared to [Isenburg and Alliez 2002a]) making it uncompetitive compared to a dedicated hexahedral mesh compressor.

## 4.2 Streaming Compressor

All the algorithms presented in the previous section are only able to compress relatively small datasets. Indeed, they all begin by loading the whole mesh in memory, which limits the size of the meshes that can be compressed. In order to overcome this problem, we use the *streaming paradigm* presented in Section 3.2. Our compressor requires streaming input: an interleaved sequence of vertices, hexahedra, and finalization tags. Mesh generators can easily be modified to produce meshes in a streaming format (we give such an example in Section 4.4). Existing meshes in non-streaming formats need to be converted. For coherent meshes this conversion can easily be done with *vertex-compaction* [Isenburg and Lindstrom 2005].

Most tetrahedral meshing techniques intrinsically produce poor streaming meshes. For example, the widely used Delaunay refinement operation starts from a base tetrahedrization of a reduced point set and randomly adds points, splitting cells until some property is verified [Dey 2009]. This produces meshes with very poor coherence. On the other hand, hexahedral meshes tend to have very coherent layouts, because they are mainly generated using advancing-front techniques [Blacker 1996; Muller-Hannemann 2001; Staten et al. 2005]. Figure 4.2 compares the layout coherence of typical tetrahedral and hexahedral meshes. After vertex-compaction the width (i.e. the maximum number of simultaneously active vertices) of our hexahedral models is around $2-5\%$ of their size. In comparison, tet meshes generally have a width of $80-100\%$ of their size [Isenburg et al. 2006a]. This makes streaming ever more attractive for hexahedral meshes than for tetrahedral meshes.

Our compressor starts encoding the mesh as soon as the first hexahedron and its eight vertices are received. It always encodes if and how the current hexahedron is adjacent to previously encoded hexahedra, compresses all new vertices that are referenced for the first time, and then deallocates the data structure associated with all vertices that are referenced for the last time (i.e. that are *finalized*). As only the active vertices have to be stored in memory, the width of the streaming mesh determines the maximum memory consumption. In the following sections, we detail the compression of the three components of a mesh: *connectivity*, *geometry* (and vertex attributes), and *cell attributes*.

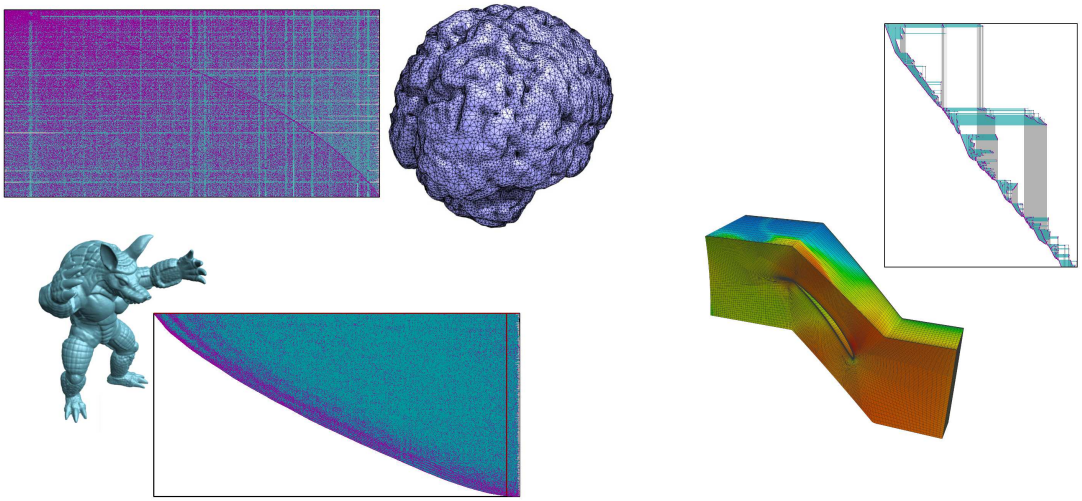

Figure 4.2: Comparison of layout quality for typical tetrahedral (*brain*, top left), triangular (*armadillo*, bottom left) and hexahedral (*blade*, right) models.

### 4.2.1 Compressing Connectivity

Like Isenburg *et al.* [Isenburg et al. 2006a], we maintain an *active surface*, a half-edge structure composed of active vertices and quadrilateral faces (see Figure 4.3). A vertex of the current hexahedron is added to the active surface when it is referenced for the first time and removed when it is finalized (see Figure 4.4). A face of the current hexahedron is added to the active surface when it was previously *not* part of it and removed otherwise. Faces are also removed after all their vertices are finalized (i.e. boundary faces).

The ten ways in which a hexahedron can be face-adjacent to the active surface are illustrated in Figure 4.5. The vertices shown in red for the *START*, *HUT*, *STEP* and *CORNER* configuration are usually new and will be compressed (see Section 4.2.2). Occasionally, however, these vertices are already part of the active surface. Such *joined* vertices are specified using dynamic indexing [Isenburg et al. 2005b] with $log_2$(number of active faces) bits.

For coding efficiency we rotate the current hexahedron into a *canonical* configuration. For the *BRIDGE* configuration, for example, the active faces will always be $f_0$, $f_1$ and $f_3$ after rotating (see Figure 4.6). Then we only need to code the configuration type and the following information:

- START: We code for all 8 vertices if they are new or joined. For coherent meshes there is usually only one *START* per component.
- HUT: We specify the face $f_0$ on the active surface the hexahedron is adjacent to and we code whether the 4 vertices are new or joined.
- ROOF: We specify $f_0$ on the active surface, code $v_4$ with dynamic indexing, and code which of $v_4$'s adjacent faces is $f_5$.
- STEP: After we specify $f_0$ we can find face $f_1$ incident to the shared edge (which is known due to the canonical order). There is often only one candidate so that specifying face $f_1$ is in most cases free. For 2 vertices we code if they are new or joined.
- BRIDGE: We proceed as for *STEP* to specify $f_1$ and $f_3$.
- CORNER: We code a *STEP* and let the decoder deduce the following: if there exists an active face which contains vertices $v_1$,$v_0$,$v_4$ then the operation is a *CORNER* and this face is $f_2$. This works in

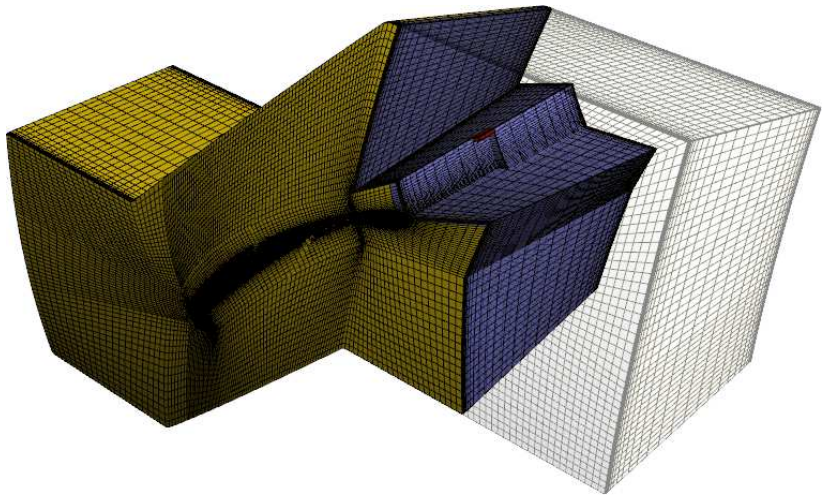

Figure 4.3: A snapshot of the streaming compression process for the *Blade* model. The *active surface* is blue, and resides in memory within a halfedge structure. The yellow cells have been released since they are *finalized*. The red cell is being compressed.

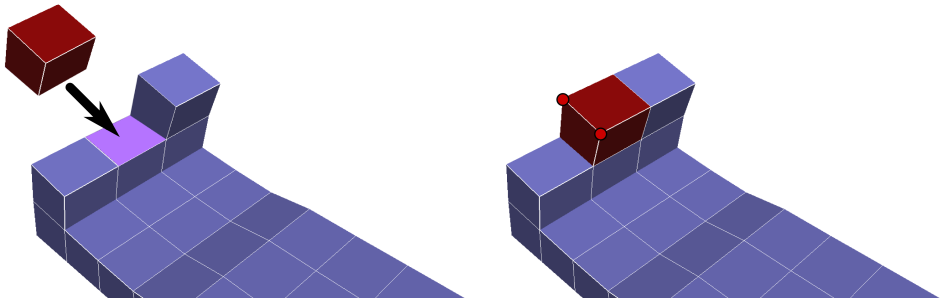

Figure 4.4: Adding a hexahedron to the active hull: The two vertices and the four faces in red are new and will be added to the active surface. At the same time, two interior faces will be removed from it.

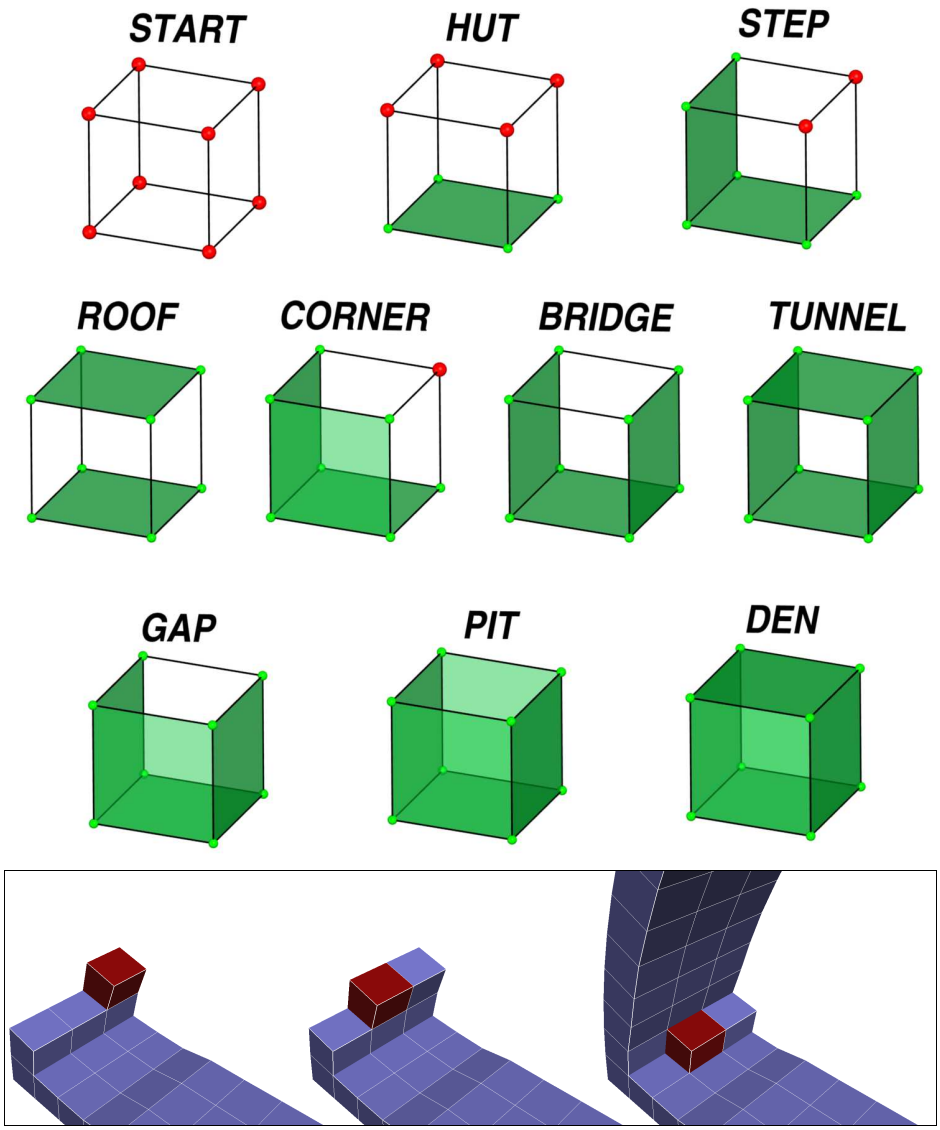


Figure 4.5: The ten possible canonical hexahedron configurations: green faces and vertices are active, and red vertices are either new or joined. At the bottom, the three most frequent operations *HUT*, *STEP* and *CORNER* in context. Together with the *START* operation, they are sufficient to encode a regular grid in scanline order.

all but one very special case illustrated in Figure 4.8. Here the face exists but is not $f_2$. To assure the decoder makes the correct decision, we output a confirmation symbol. This special case is rare and happens only once in all our test models. Thus, the confirmation symbol is nearly always the same and adds almost nothing to the bit budget. We also code if the last vertex is new or joined.

- TUNNEL: We code a *BRIDGE* and let the decoder deduce that this is in fact a *TUNNEL* operation whenever there exists an active face with vertices $v_4$,$v_5$,$v_6$,$v_7$. There is no ambiguity in this case.
- GAP: We code a *STEP* and first let the decoder deduce that it is a *CORNER* and then—with the same reasoning—that it is a *GAP*.
- PIT: We code a *STEP* and let the decoder deduce a *GAP* and then—again with the same reasoning—that it is a *PIT*.
- DEN: We take the reasoning of *PIT* one step further.

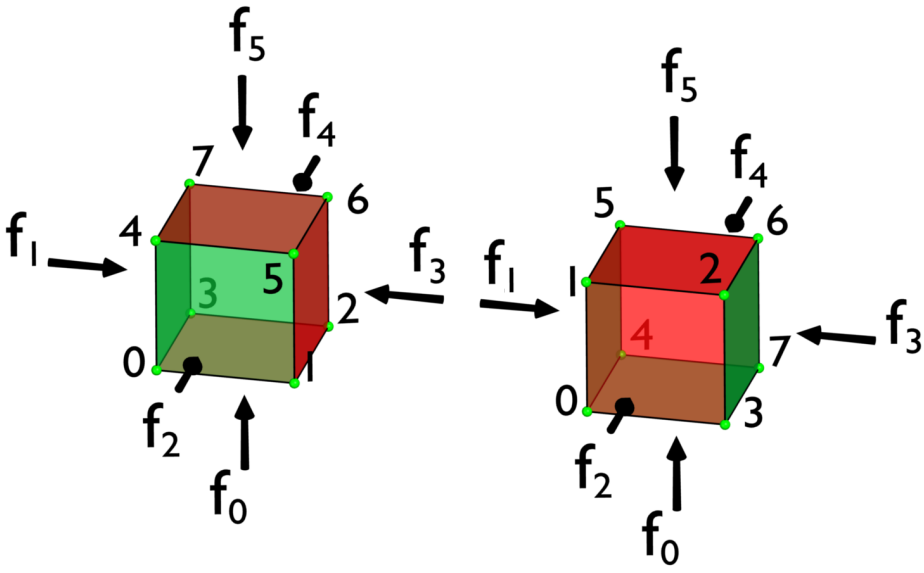


Figure 4.6: Going from a randomly oriented *BRIDGE* configuration (left) to a canonical *BRIDGE* configuration (right). On the right, the vertices have been rotated such that the active faces are $f_0$, $f_1$ and $f_3$.

To summarize: the coder will only distinguish between *START*, *HUT*, *ROOF*, *STEP* or *BRIDGE*. Everything else is deduced by the decoder. Each operation except *START* has to specify the face $f_0$ on the active surface. Doing this each time with dynamic indexing [Isenburg et al. 2005b] would be very costly.

When consecutive hexahedra share a face we can specify face $f_0$ very efficiently by caching the 6 faces of the previous hexahedron. If we find face $f_0$ in the cache we can code it with $log_2(6)$ instead of $log_2$(number of active faces) bits (see Figure 4.7). Using context-based entropy coding we can further decrease these costs as different configurations share faces with similar regularity. When face $f_0$ is not in the face cache we use the same idea with an edge cache and finally a vertex cache. We only resort to dynamic indexing [Isenburg et al. 2005b] when this all fails.

In Table 4.2 we list the compression rates of our scheme on different models and compare them to those of [Isenburg and Alliez 2002a], [Krivograd et al. 2008] and [Lindstrom and Isenburg 2008]. As expected, our rates are worse than those of degree-based methods since we compress the hexahedra in their original order. Unsurprisingly, our method outperforms the lossless coder that does not reorder vertices as we do and also preserves the original orientation of hexahedra. The penalty for streaming

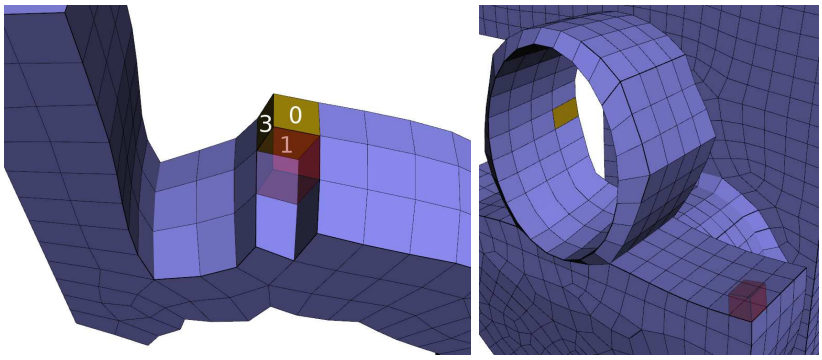


Figure 4.7: Reducing the cost of specifying $f_0$ within the active surface: Successive hexahedra will usually be adjacent. By caching the faces of the last hexahedron (yellow), we can specify $f_0$ (red) at reduced cost (left). When $f_0$ is not in the cache, we resort to dynamic indexing (right).

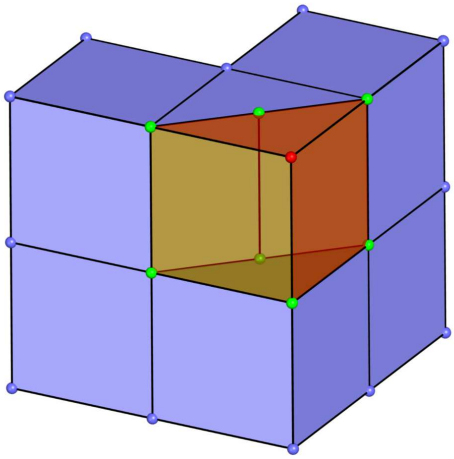

Figure 4.8: Special *CORNER* case: the bottom face is active and shares three vertices with the current hexahedron. The confirmation bit is set to 1, and the operation is a *CORNER*. However, the top face should no be shared. Thus, the second confirmation bit set to 0, meaning that the operation is not a *GAP*.

is highest for meshes with global regularity (e.g. "block" or "cylinder") that our compressor cannot exploit. Overall however, after also compressing geometry and attributes, the connectivity accounts for a comparatively small amount of the total bit budget.

#### 4.2.1.1 Local Reordering

Locally, hexahedral meshes tend to have the connectivity of a grid whose cells are often specified in scanline order (see Figure 4.9). There will be a cache miss (i.e. two non-adjacent hexahedra) for each scanline as soon as the scanlines are longer than two hexahedra. We can avoid this cache miss if we locally reorder the hexahedra.

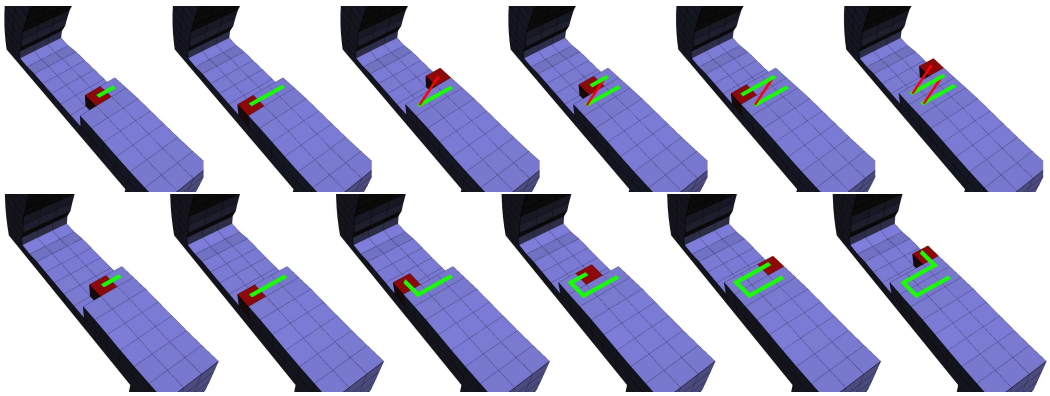

Figure 4.9: A closeup on the *shaft* model, original order (top) and reordered using a buffer of three hexahedra (bottom). The original order causes a cache miss every three hexahedra, the new order has no cache misses.

Also, on more irregular connectivities, we get fewer cache misses and better compression rates when we buffer a number of hexahedra from which we then greedily pick "the best" hexahedron and feed it to the compressor. Using a fixed-size *delay buffer* we tried a number of strategies with an emphasis on speed and simplicity. We describe here the simple *spiraling reorderer* that performed well in our experiments and that we use in the results that we report.

We label the active faces of the hexahedron in cache with *front, left, right, top,* and *bottom*. When we pick a new hexahedron among those waiting in the buffer, that has one of its faces in the cache, there are three possibilities for the label $L$ of this face:

- $L$ is *left* or *right*: We set $C(horizontal) = L$ and $D_{last} = horizontal$.
- $L$ is *top* or *bottom*: We set $C(vertical) = L$ and $D_{last} = vertical$.
- $L$ is *front*: Nothing changes.

To pick the next hexahedron we consider in decreasing priority among the hexahedra in the buffer:

- the hexahedron that shares the face $C(D_{last})$ with the currently cached hexahedron,
- the hexahedron that shares the face $C(\bar{D_{last}})$, where $\bar{D_{last}}$ is *vertical* if $D_{last}$ is *horizontal*, and *horizontal* else,
- the hexahedron that shares the face *front*,
- the hexahedron that shares one of the the other faces,
- the oldest hexahedron in the buffer.

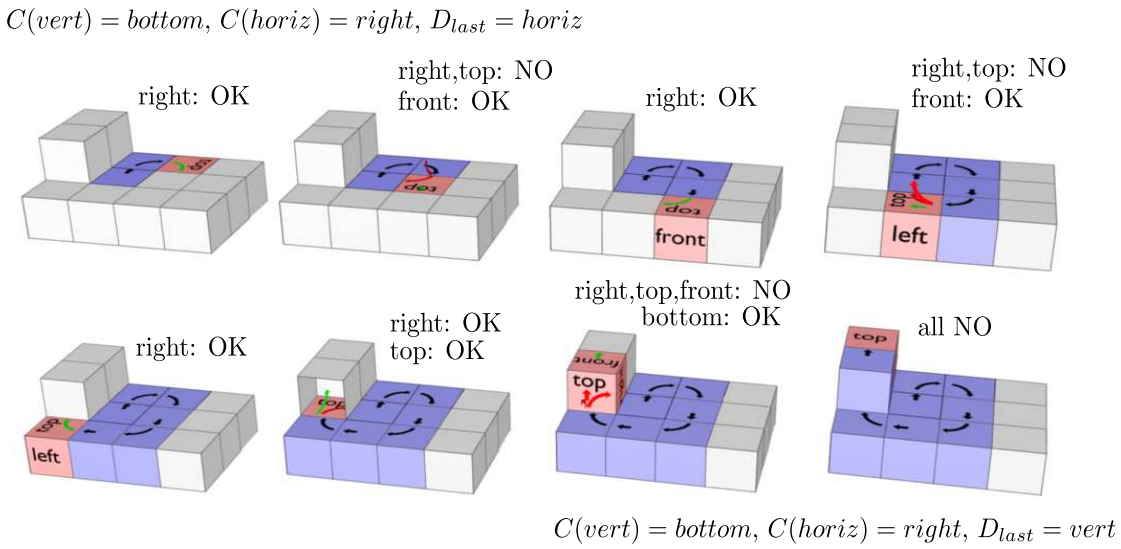


Figure 4.10: The *spiraling* reorderer running on an example mesh. The blue, red and white hexahedra are respectively visited, in cache, and waiting in the buffer. A green arrow denotes the first possible choice in order of priority, red arrows choices that would have had a higher priority but for which there was no corresponding hexahedron in the buffer. We suppose that the starting configuration is $D_{last}$ = *horizontal*, $C(horizontal)$ = *right*, and $C(vertical)$ = *bottom*.

$C(horizontal)$, $C(vertical)$ and $D_{last}$ are then updated according to this choice. The Figure 4.10 illustrates this process. This method has the advantage that the added hexahedra closely "stick" to the active surface. For example, in the case of a regular grid originally given in scanline order, the spiraling reorderer will result in a zigzagging pattern similar to Figure 4.9. There will not be cache misses within a slice as long as the buffer size is larger than the scanline size.

The curves shown in Figure 4.11 plot the compression rate versus the delay buffer size for several models. Increasing the delay to more than a couple hundred hexahedra usually does not improve the compression rate any further, because the greedy strategy then shows its limits. A global strategy could potentially overcome this drawback, but it would also greatly decrease the speed of the compressor—eventually making it equivalent to a non-streaming compressor.

### 4.2.2 Compressing Vertex Geometry and Properties

Each time a new vertex is added (this can happen only during *START*, *HUT*, *STEP* and *CORNER* operations), we predict its position and code the difference between the predicted and actual value.

We use a spectral/Taylor predictor (see Chapter 2) – these approaches are equivalent in the hex case. We list the prediction rules we use in Table 4.1. They work extremely well on hexahedral meshes that tend to be geometrically smooth.

Our implementation has two modes of operation: it can either uniformly quantize the vertices prior to compression, perform all predictions in integer arithmetic, and entropy code the resulting integer residuals or it can avoid quantization, perform all predictions in floating-point arithmetic, and compress the residuals with the method of Isenburg and Lindstrom [Lindstrom and Isenburg 2006a]. In Table 4.2 we list our geometry compression rates across our set of test meshes side by side with those of the other methods [Isenburg and Alliez 2002a; Lindstrom and Isenburg 2008; Krivograd et al.

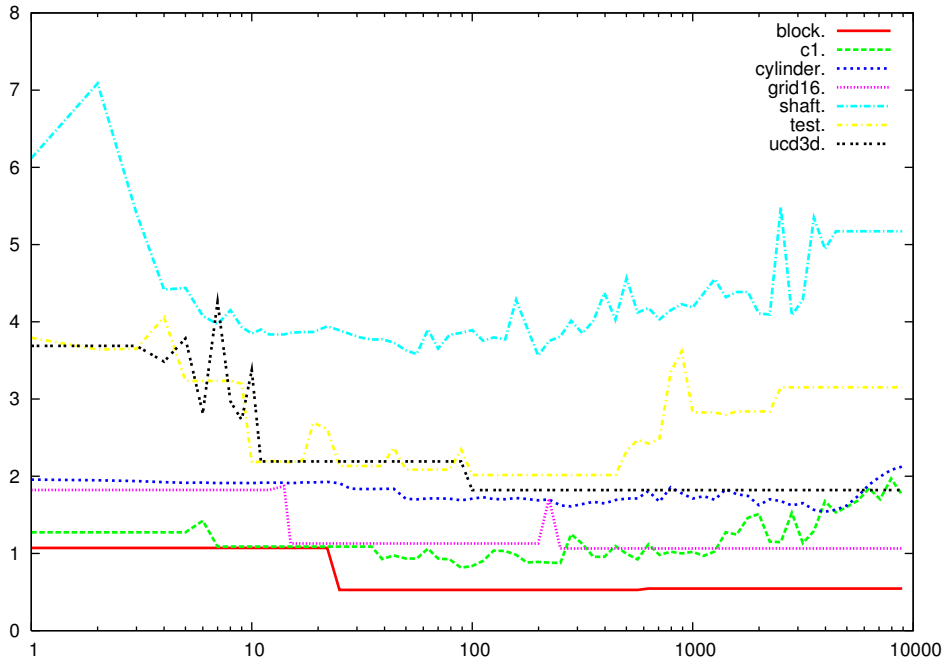


Figure 4.11: Connectivity compression (bph) versus delay buffer size for several models. Local reordering is usually very efficient, but further increasing the delay shows the limits of greedy strategies.

| **Vertex** | **Prediction** |
|---|---|
| $v_0$ | last vertex |
| $v_1$ | $v_0$ |
| $v_2$ | $v_1$ |
| $v_3$ | $v_0 + v_2 - v_3$ |
| $v_4$ | $v_0$ |
| $v_7$ | $v_3 + v_4 - v_0$ |
| $v_5$ | $v_1 + v_4 - v_0$ |
| $v_6$ | $v_0 - v_1 + v_2 - v_3 - v_4 + v_5 + v_7$ |

Table 4.1: Spectral/Taylor prediction rules. The vertices appear in the order they are predicted, i.e. for any given vertex on line $i$ of the table, the vertices of all previous lines are known.

2008]. We use an updated version of [Isenburg and Alliez 2002a] that also uses spectral prediction. The implementation of [Lindstrom and Isenburg 2008] natively uses spectral prediction.

The degree coder [Isenburg and Alliez 2002a] uses more information to predict $v_4$ in *HUT* configurations by mirroring the adjacent hexahedron. This gives slightly better compression ratios. However, most predictions (typically around 90-95%) are made using the efficient Lorenzo predictor (see Table 4.1, last row), so this will always remain a very small improvement.

| | **vertices** | **hexas** | **Connectivity (bph)** | | | | | **Geometry (bpv)** | | | | **Total (bph)** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | A | B | HZ | Ours | | A (Q) | HZ (L) | Ours (Q) | Ours (L) | A (Q) | HZ (L) | Ours (Q) | Ours (L) |
| block | 101,401 | 93,750 | 0.07 | * | 0.07 | 0.55 | (1.00) | 0.05 | 0.2 | 0.05 | 2.8 | 0.1 | 0.3 | 0.6 | 3.6 |
| c1 | 78,618 | 71,572 | 0.59 | 0.56 | 1.50 | 0.94 | (1.24) | 3.4 | 14.8 | 3.2 | 9.3 | 4.3 | 17.8 | 4.5 | 10.2 |
| cylinder | 500,055 | 482,900 | 0.22 | 0.30 | 3.01 | 1.66 | (1.91) | 0.3 | 1.8 | 0.3 | 1.9 | 0.5 | 4.9 | 2.0 | 3.6 |
| fru | 5,124 | 4,360 | 0.97 | 0.98 | 3.06 | 2.20 | (2.42) | 16.5 | 55.7 | 17.4 | 35.8 | 20.3 | 68.5 | 22.6 | 44.3 |
| grid | 4,096 | 3,375 | 0.29 | 0.4 | 0.21 | 1.35 | (1.93) | 0.4 | 0.3 | 0.4 | 19.5 | 0.8 | 0.6 | 1.8 | 25.0 |
| hutch | 8,790 | 8,172 | 0.30 | 0.48 | 8.68 | 2.56 | (2.91) | 8.5 | 26.8 | 7.8 | 24.9 | 9.4 | 37.5 | 10.9 | 29.3 |
| mdg-1b | 4,510 | 3,710 | 0.77 | 0.93 | 7.81 | 2.98 | (3.35) | 2.8 | 9.3 | 2.8 | 28.7 | 4.2 | 19.1 | 6.4 | 37.9 |
| shaft | 9,218 | 6,883 | 1.70 | * | 18.55 | 4.04 | (5.63) | 16.5 | 42.3 | 17.4 | 30.0 | 23.8 | 75.2 | 27.3 | 44.2 |
| steven | 96,030 | 81,832 | 0.05 | * | 1.96 | 1.04 | (1.00) | 3.3 | 14.7 | 3.7 | 9.7 | 3.9 | 19.2 | 5.4 | 12.4 |
| test | 3,198 | 2,386 | 0.87 | 1.09 | 20.40 | 2.53 | (4.02) | 4.5 | 10.8 | 5.7 | 34.0 | 6.9 | 34.9 | 10.2 | 48.1 |
| ucd3d | 2,646 | 2,000 | 0.47 | * | 0.30 | 2.52 | (4.50) | 1.9 | 4.67 | 2.0 | 29.9 | 3.0 | 6.4 | 5.2 | 42.1 |

Table 4.2: Compression rates of existing methods for various models: A, B, HZ respectively denote the methods of [Isenburg and Alliez 2002a], [Krivograd et al. 2008], and Hexzip. For our algorithm, we give the compression rates without reordering (between parentheses) and with a reordering buffer of 50 hexahedra, which is an effective buffer size on average. For comparison, note that [Lindstrom and Isenburg 2008] preserves the order of both vertices and cells, our method preserves cell order (up to local reordering) but reorders vertices within a cell, and [Isenburg and Alliez 2002a; Krivograd et al. 2008] do not preserve order at all. For geometry compression, we give rates using 16-bits quantization (Q) and lossless (32-bit floating-point) compression (L). The method of [Krivograd et al. 2008] does not support geometry compression. We denote with * the models for which the software provided by Krivograd *et al.* did not work.

| | **Connectivity (bph)** | | | | | | | **Geometry Prediction** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | config. | cache | dyn. | cand. | final | join | rest | last | edge | para. | lorenzo |
| block | 0.024 | 0.456 | 0.017 | 0.003 | 0.033 | 0.019 | 0.001 | 1 | 224 | 8,050 | 93,126 |
| c1 | 0.085 | 0.504 | 0.093 | 0.005 | 0.054 | 0.198 | 0.002 | 4 | 521 | 14,413 | 63,680 |
| cylinder | 0.091 | 0.796 | 0.510 | 0.068 | 0.014 | 0.183 | 0.000 | 32 | 2,604 | 54,189 | 443,230 |
| fru | 0.273 | 0.941 | 0.070 | 0.020 | 0.237 | 0.644 | 0.024 | 1 | 81 | 1,390 | 3,652 |
| grid | 0.147 | 0.834 | 0.045 | 0.014 | 0.247 | 0.033 | 0.031 | 1 | 45 | 675 | 3,375 |
| hutch | 0.348 | 1.791 | 0.113 | 0.027 | 0.141 | 0.126 | 0.014 | 3 | 196 | 2,154 | 6,437 |
| mdg-1b | 0.356 | 1.943 | 0.224 | 0.050 | 0.300 | 0.080 | 0.028 | 1 | 122 | 1,417 | 2,970 |
| shaft | 0.769 | 1.897 | 0.195 | 0.050 | 0.497 | 0.618 | 0.017 | 18 | 608 | 3,404 | 5,188 |
| steven | 0.114 | 0.606 | 0.202 | 0.015 | 0.077 | 0.033 | 0.001 | 6 | 905 | 15,664 | 79,455 |
| test | 0.453 | 1.375 | 0.077 | 0.023 | 0.503 | 0.057 | 0.044 | 4 | 114 | 849 | 2,231 |
| ucd3d | 0.272 | 1.652 | 0.084 | 0.020 | 0.392 | 0.052 | 0.052 | 1 | 46 | 619 | 1980 |

Table 4.3: Compression details for the models of Table 4.2, with the same delay buffer of 50 hexahedra. The bit budget is broken down into its *configuration*, *cache*, *dynamic indices*, *candidates*, vertex *finalization*, and the *rest*, which includes header information and confirmation bits. For geometry prediction, *last* refers to the coding for $v_0$, *edge* for $v_1, v_2, v_4$, *parallelogram* for $v_3, v_5, v_7$, and *lorenzo* for $v_6$ (table 4.1).

### 4.2.3 Compressing Cell Properties

Ideally we would like to compress cell data with the same strategy that has already proven successful for vertex data. Hence, the prediction would be made using the already processed cells of the neighbourhood with a spectral predictor on the dual of the mesh. However, two problems arise:

1. The dual of a hexahedral mesh is generally not a hexahedral mesh itself (except in the case of a grid). Thus, the configuration of the neighbourhood can vary a lot between hexahedra as shown in Figure 4.12. It would be expensive, if not impossible to store the prediction weights for every possible configuration of the neighbourhood. Alternatively the compressor and decompressor could determine the configuration of the neighbourhood and compute the weights with respect to this configuration on-the-fly.

2. However, using the complete hexahedron neighbourhood requires additional storage: We would have to store the cell data for all processed hexahedra that still have one or more unfinalized vertices (shown as grey and blue cells in Figure 4.12). This goes against our objective to keep the memory footprint of our algorithm as small as possible.

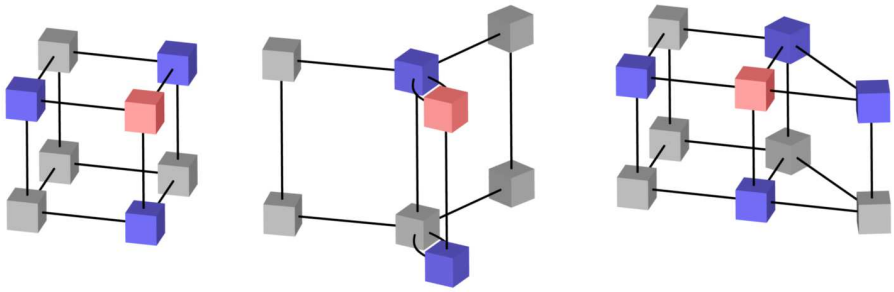

Figure 4.12: Examples of possible neighbourhoods for prediction, for a *CORNER* on a grid (left), the *CORNER* configuration of Figure 4.8 (center), and a *GAP* configuration with irregular neighbourhood (right). For readability, we show the dual of the mesh. Hexahedra are represented as colored cubes (red: new hexahedron, blue: active hexahedra, gray: hexahedra that are in the neighbourhood but may no longer be active).

For these reasons we use a simpler prediction scheme that is faster and more memory efficient. We predict cell data using only the hexahedra that are face-adjacent to the current hexahedron (illustrated by blue cells in Figure 4.12). These hexahedra have at least the corresponding active face (but usually a few more) on the active surface. We simply store copies of the cell data of a hexahedron with each of its active face on the active surface. When these faces become inactive and are removed from the active surface the cell data is released along with the face.

Another benefit of this scheme is that there are only 10 possible neighbourhood configurations, which correspond to the 10 connectivity operations. This enables us to precompute the weights and store them in a lookup table, shown in Figure 4.13. For the *HUT*, *ROOF*, *STEP*, *CORNER*, *TUNNEL* and *DEN*, the weights can be determined by symmetry (they are the same and add up to 1). We tried three different approaches to determine the weights for the non-symmetric *BRIDGE*, *GAP* and *PIT*. All these methods give the same result.

- We used the spectral method of [Ibarria et al. 2007] with a $3 \times 3 \times 3$ grid. The running time for this method is on the order of hours (under Mathematica).

- We computed the Taylor weights using a regular neighbourhood. We give here a brief derivation of the weights for the *GAP* case, but the two other cases are similar. Because of symmetry, the *GAP* case only has two weights. Let $\alpha$ be the weight of left and right cells of the canonical configuration of Figure 4.5, and $\beta$ that of the front and bottom cells. These cells have respectively

parameter coordinates $(u, v, w) = \{(-1, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1)\}$. Then the zero and first order conditions give:

$$\begin{cases} 2\alpha + 2\beta &= 1 \\ 0 &= 0 \quad (\frac{\partial f}{\partial u}) \\ \beta &= 0 \quad (\frac{\partial f}{\partial v}) \\ \beta &= 0 \quad (\frac{\partial f}{\partial w}) \end{cases} \tag{4.1}$$

The symbolic computation of the weights (using Mathematica) takes a few seconds.

- We also computed the least squares weights on the *Blade* model. The results were the same to a precision of $10^{-3}$.

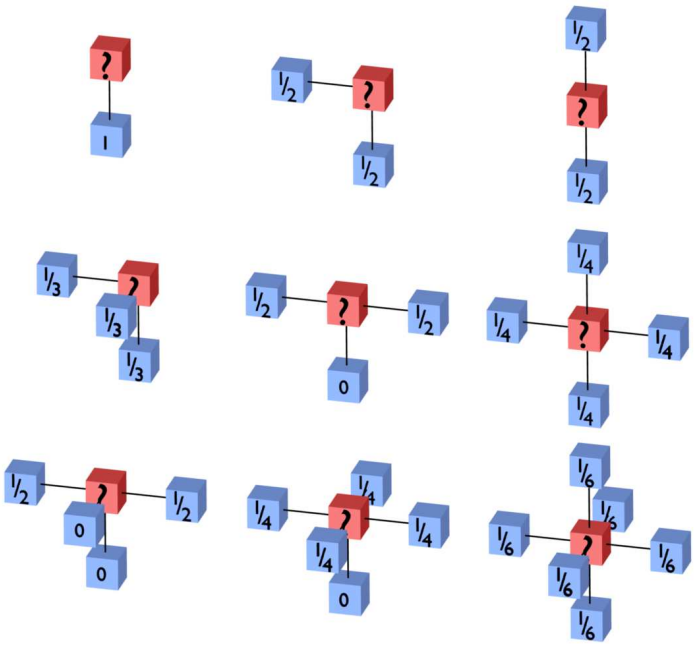


Figure 4.13: Cell data prediction weights for the different configurations. The red hexahedron is the new cell whose data is to be predicted. Blue hexahedra are the active ones that are used for prediction.

It may be noted that all the weights are positive. That means that the result of the prediction will always lie within the convex hull of the values of all the cells used for prediction. That usually results in a systematic prediction error, because the predictor is unable to accurately extrapolate, for example in the typical case of a locally monotonic scalar field. Because of this, we use a second-order predictor.

Let $c_1, \ldots, c_k$ be the data at known cells of the neighbourhood, and $w_1, \ldots, w_k$ the weights used to predict the unknown value $c_u$. The first-order prediction of $c_u$ and the prediction residual are:

$$\begin{cases} \bar{c_u} = \sum_{i=1}^{k} w_k.c_k \\ r_u = c_u - \bar{c_u} \end{cases} \tag{4.2}$$

| Quantity | 12 bits | 16 bits | Lossless |
|---|---|---|---|
| $K$ | 1.83 | 4.44 | 18.1 |
| $L$ | 1.99 | 5.17 | 11.6 |
| $P$ | 2.39 | 5.35 | 10.8 |
| $T$ | 2.32 | 5.20 | 16.0 |
| $V_x$ | 2.74 | 6.05 | 23.3 |
| $V_y$ | 2.68 | 5.94 | 23.2 |
| $V_z$ | 1.97 | 4.66 | 16.4 |
| $Y_{CH_4}$ | 1.39 | 3.47 | 14.4 |
| $Y_{CO_2}$ | 2.28 | 5.17 | 15.1 |
| $Y_{H_2O}$ | 2.28 | 5.18 | 14.9 |
| $Y_{N_2}$ | 1.48 | 3.67 | 12.2 |
| $Y_{O_2}$ | 1.94 | 4.19 | 14.7 |

Table 4.4: Cell data compression rates, in bits per hexahedron, for the twelve attributes of the "cedre" model.

We simply use the same prediction rule to predict the residual from the residuals of neighbour cells $r_1, \ldots, r_k$, and code the difference $r_u^2$ between the real and expected residuals:

$$\begin{cases} \bar{r_u} = \sum_{i=1}^{k} w_k . r_k \\ r_u^2 = r_u - \bar{r_u} \end{cases} \tag{4.3}$$

The decoder can then retrieve the original value as $\bar{c_u} + \bar{r_u} + r_u^2$. Coding the second-order residual $r_u^2$ instead of the first order residual $r_u$ drastically reduces the entropy. For our test models, the second order scheme halved the entropy of residuals on average for 12 bits quantization.

Table 4.4 gives the bitrates achieved using our algorithm on the "cedre" dataset. This model comes from a computational fluid dynamics combustion simulation made at *ONERA*[2] with twelve attributes attached to the cells (see Figure 4.15, right).

## 4.3 Streaming HexZip

The non-streaming Hexzip coder of Lindstrom and Isenburg [Lindstrom and Isenburg 2008] has a structure such that it can be easily modified to enable streaming *compression*. The original implementation reads the whole vertex geometry and cell indices tables, and then traverses the cells predicting and coding indices. This enables very fast access to vertices. We modified the original implementation by exchanging the dense table of vertex geometry with a hash table of vertices from which the vertices are removed as soon as they are finalized. This trades speed for memory efficiency: On one hand, the compressor uses less memory since only active vertices have to be stored. On the other hand, the compressor is slower, because the access to an element of a hash table is far slower than a simple indirection (see Table 4.5).

This simple change does only that. It does not change compression rates for either geometry or connectivity, since the cells and vertices are still processed in the same order.

### 4.3.1 Providing streaming decompression

One may note that only streaming *compression* is provided. To enable streaming *decompression*, one has to transmit finalization tags, and there is currently no available way of transmitting finalization information. However, this compressor specifically addresses hex and quad meshes. It would be possible to take advantage of this fact to add another stream of byte-aligned *finalization tags* along

[2]The French Aerospace Lab, http://www.onera.fr

| Model | compression time (ms) | | memory footprint (MB) | |
|---|---|---|---|---|
| | non-streaming | streaming | non-streaming | streaming |
| Block | 17 | 140 | 15.6 | 3.9 |
| c1 | 24 | 120 | 12.4 | 4.0 |
| cylinder | 67 | 650 | 97.2 | 14.9 |

Table 4.5: Speed/Memory trade-off for streaming HexZip. Note that implementations for both methods are totally different, so these numbers should only be taken as a trend and not accurately compared. Also note that reading timings are included for the streaming version, since reading and compression are interleaved.

connectivity and geometry. Indeed, writing a bit for each vertex of a cell, set to 1 if the vertex must be finalized, and to 0 if it is to be kept, natively takes 1 byte per cell for a hexahedral mesh, and 1 byte per two cells for quad meshes. In addition to that, if the mesh layout is regular enough, there will be an important redundancy in the byte stream. This would enable efficient transmission of finalization information, while remaining within the original frame of efficient byte-aligned coding. Yet, we do not currently have experiments that prove the efficiency of this idea.

## 4.4 Generating Already Compressed Meshes

Traditional methods first generate a mesh that is kept in a temporary storage, either in memory or on disk. In a second stage, the mesh is read and compressed. In contrast, the streaming paradigm enables pipelining of several processing tasks. In particular, this can be used to directly generate compressed streaming meshes, by plugging the output of the mesher to the compressor. Compression will begin as soon as meshing begin, and the mesh will never be stored on the disk in an uncompressed form.

The following section gives an example of such a compressed mesh generator. We re-implemented the *transfinite* meshing approach of the open-source hex mesher GMSH[3] as an input module to Isenburg's streaming API[4]. The modules reads a GMSH geometry file and outputs a stream of interleaved vertices/cells and finalization tags.

### 4.4.1 GMSH's transfinite mesher

GMSH uses several different techniques for hexahedral mesh generation. The simplest among them is the *transfinite approach*. The input mesh is specified as a coarse *non piecewise-linear* hexahedral mesh (NPL mesh). The elements of this mesh are cells that must have 8 vertices, 12 edges, and 6 faces. However they are not hexahedra in the sense that the edges are not line segments. Instead, the edges can be any 3D curve. GMSH comes with a user interface (Figure 4.14, left) that helps specify these curves as a combination among a choice of primitives: line segments, circular/elliptic arcs, splines, ...

To build a hexahedral mesh from these control curves, three steps are applied:

1. The control curves (edges of the NPL mesh) are discretized. The sampling can be either uniform or come from a metric. The sampling is also constrained such that two opposite edges of any face of the NPL mesh have the same number of samples.

[3] http://geuz.org/gmsh
[4] http://www.cs.unc.edu/~isenburg

2. The faces of the NPL mesh are discretized. As the sampling is similar for each pair of opposite NPL edges, the connectivity can be that of a grid. The geometry of a vertex at discrete parameter coordinates $(u, v)$ in the $N \times M$ surface is built from that of the four projections on the surface boundary:

$$p(u,v) = \frac{1}{2}\left[\frac{M-v}{M}.p(u,0) + \frac{v}{M}.p(u,N-1) + \frac{N-u}{N}.p(0,v) + \frac{u}{N}.p(0,M-1)\right] \tag{4.4}$$

3. The cells of the NPL mesh are then discretized using the same process, but using this time a 3D grid and using 6 surface points for geometry.

An example is given in Figure 4.14.

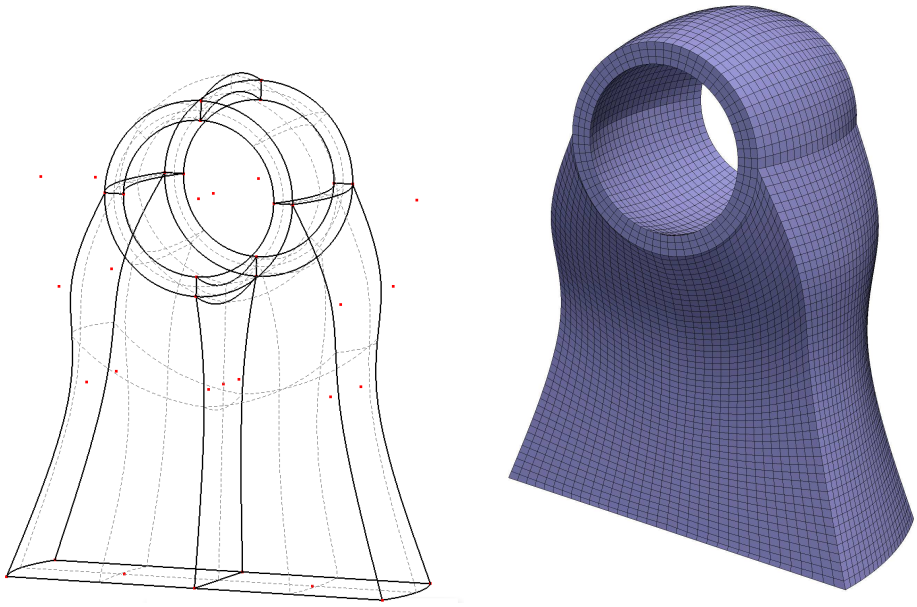

Figure 4.14: An example of *transfinite* hexahedral mesh. The control curves are on the left, and consist in line segments, circles and b-splines. The control points of these primitives are in red, and the dashed lines indicate transfinite surfaces. On the right, the output of the algorithm.

The transfinite algorithm bears some resemblance to a subdivision scheme: most vertices are regular, and interior vertices are interpolated from control curves. However, it is not a subdivision algorithm, because it can have an arbitrary (and different) number of samples in each direction. Meshes generated with this algorithm are intrinsically very redundant in both connectivity (they are piecewise grids) and geometry (the interpolation scheme is very smooth). Therefore, they compress very well.

### 4.4.2 Streaming transfinite mesh generation

The implementation of a streaming generator for the meshing scheme described above is nearly trivial. Interior vertices are independent from one block to another. Therefore, the streaming mesher considers each cell of the NPL mesh in turn, and outputs the vertices using a front advancing parallel

to one of its faces. To compute finalization information, each active vertex maintains a count of its available *cell slots*. Initially, a vertex of a control curve has as many slots as neighbour cells in the NPL mesh. Interior vertices of a boundary face of the NPL have a slot count of 4, and all other vertices have a slot count of 8. Each time a cell is output, the slot counts of all the vertices it references are decreased by 1. Then, these 8 vertices are tested for a 0 slot count, finalization tags are output according to the result of the test, and the vertices with a slot count of 0 are destroyed.

## 4.5 Compressing large models

The big advantage of our streaming method compared to other non streaming methods is for the compression of large meshes. As we release the mesh structures as they become finalized, the memory footprint of our compressor stays very low. Table 4.6 compares the memory consumption of the two streaming approaches that we introduced with that of the degree coder of Isenburg and Alliez [Isenburg and Alliez 2002a]. For fair comparison, we have obtained from the authors of [Isenburg and Alliez 2002a] an implementation that has been optimized to run faster while reducing memory consumption. The tests were run on an Intel core 2 duo running at 2.66GHz (our implementation uses only one core). We do not compare with [Krivograd et al. 2008], because their software is optimized neither for speed nor for memory efficiency.

Compared to our approach, the streaming Hexzip scheme is simpler, and thus the modified coder is faster than ours, even if it is slower than the original Hexzip. It also has a smaller memory footprint, because its structure for representing the mesh is simpler (approximately 4 times more compact). However, the compression rate is a bit lower, and, more importantly, it cannot compress cell data.

Our streaming compressor greatly outperforms the degree coder with a memory footprint orders of magnitude smaller. Using the non-streaming coder, most machines with 32-bits operating systems would not even be able to compress the *crank* dataset without swapping, because they would not support enough RAM to accommodate the memory needed by the compressor.

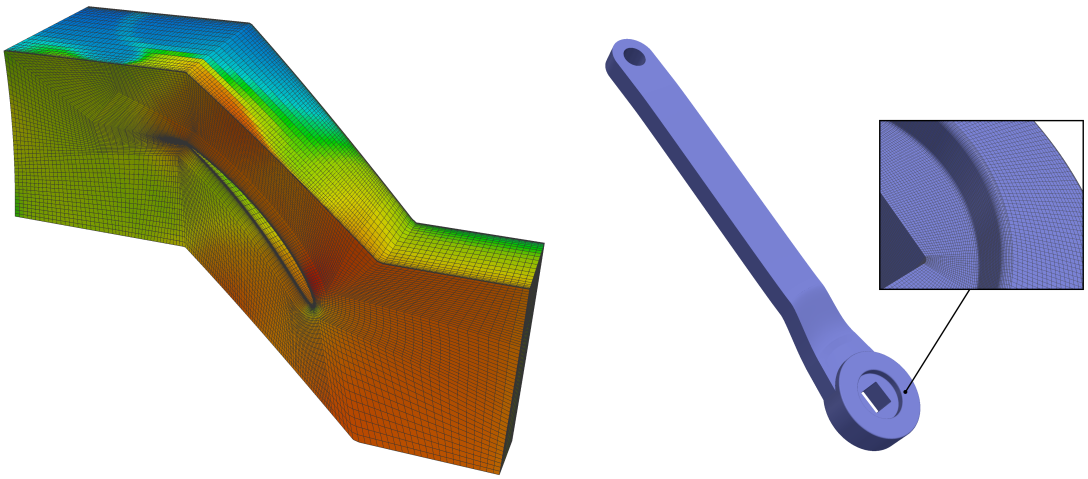

Figure 4.15: The *blade* and *crank* models used in table 4.6.

| | Vertices | Hexas | Width | Connectivity (bph) | | | Geometry (bpv) | | | Memory (MB) | | | Time (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | (%) | CHVM | SHZ | Ours | CHVM | SHZ | Ours | CHVM | SHZ | Ours | CHVM | SHZ | Ours |
| blade | 479k | 456k | 1.17 | 0.02 | 0.85 | 0.78 (0.45) | 16.8 | 20.8 | 16.7 (16.8) | 273.8 | 1.8 | 4.2 | 2.0 | 0.6 | 0.7 (1.0) |
| crank | 2M | 2M | 0.90 | | 0.98 | 0.82 (0.46) | | 12.3 | 9.1 (9.1) | | 3.2 | 12.4 | | 3.0 | 3.4 (5.3) |
| crank | 49M | 48M | 0.35 | | 0.36 | 0.48 (0.28) | | 10.7 | 6.3 (6.3) | | 16.5 | 67.5 | | 72 | 95 (142) |

Table 4.6: Comparison of the state-of-the-art non-streaming degree coder [Isenburg and Alliez 2002a] (*CHVM*) and our modification of Hexzip (*SHZ*) with our streaming approach on large models. The given memory footprint is the peak heap usage as reported by GNU `memusage`. The time for [Isenburg and Alliez 2002a] does not include reading. The time for the streaming methods include reading, since reading and compression are interleaved. The geometry rates are given for lossless compression. For our algorithm, we give the figures without delay buffer, and with a delay of 50 (between parentheses). The rates for the modified version of Hexzip do not include finalization information.

## 4.6 Conclusion

In this Chapter, we have presented several contributions to the streaming compression of hexahedral meshes. We have shown how to extend the triangle and tetrahedral (i.e. simplicial complex) streaming compressors to hexahedral meshes – but the ideas presented here can also be applied to quad meshes as well. We have also shown how to generate compressed transfinite streaming hexahedral meshes directly without ever writing the mesh in uncompressed form. Because hexahedral meshes – and especially large ones – are very coherent, the price to pay for *streaming* compression is not very high, compared to the triangle/tetrahedral case. All our large meshes were compressed to less than one bit per hexahedron for the connectivity. Therefore, we think that the streaming paradigm is specially well adapted to hexahedral mesh processing.

The main drawback of our streaming compressor is that it will fail if the mesh is not made of *only* hexahedra. However, some hexahedral meshes sometimes include a very small number of other elements (wedges, tetrahedra) to "glue" different parts together. In that case, the streaming version of *Hexzip* [Lindstrom and Isenburg 2008] that we have presented in Section 4.3 will be a better choice, since it can handle these degenerate elements. In addition, it will usually be faster that the first compressor.

While the streaming compression methods presented here enable the compression/decompression of very large hexahedral meshes, we are still limited by the problems that we have detailed in Section 3.3: The models that we compress with our algorithm can only be processed using streaming algorithms. In particular, they cannot be visualized interactively.

# Chapter 5

# Hierarchical Random Accessible Compression

In the previous section, we have presented a *symmetric* streaming method suitable for the compression of very large meshes. However, the streaming paradigm is useful for algorithms that require a *global* access to the mesh during decompression. In the context of interactive visualization of large meshes, there is rarely the need to decompress the whole mesh, for several reasons:

1. The limited computing resources make it impossible to render the whole model *on time*. For interactive visualization of the model, users will generally not tolerate a frame rate that is less than approximately 1 frame per second. For a visualization terminal with a limited graphics rendering performance, this imposes an upper bound on the complexity (number on polygons) of the model to be rendered. The streaming approach is not sufficient here, since even if it enables a rendering of the model, it will not be done in a *timely* manner.

2. Even if the computational power is such that the model can indeed be rendered in time, there is still a *display resolution* problem. Let us consider a triangle mesh with $N_f$ triangles. If the display has a resolution of $w \times h$ pixels, and the model is visualized globally, then the size of each primitive will be on the order of $\frac{w \times h}{N_f}$ pixels. For a typical desktop display of $1600 \times 1200$ pixels, this means that a model with more than 2 million faces will be rendered with sub-pixel primitives. In order to be able to see fine local behaviour, the user has to zoom in several times, which means that only a small part of the model will be visible. Everything that is compressed but falls outside the viewport can be considered as unnecessary overhead.

3. In addition to these two technical reasons, scientific data exploration naturally leads users to consider specific *regions of interest* – e.g. zones where interesting phenomena appear in the result of a simulation.

To tackle this problem, recent approaches have proposed *random-accessible* compression, as already mentioned in Chapter 3. This technique enables to select the interesting parts of the mesh without decompressing too much of the rest of the mesh (see Section 3.3).

In this chapter, we present a new contribution in the field of random-accessible compression. Classical methods provide random access by simply splitting the mesh in several pieces that are compressed independently using a single-rate or progressive coder (see Chapter 3). While this approach has proved quite efficient, it still has drawbacks. On a practical point of view, using single-rate algorithms to compress the charts means that in order for the algorithm to be efficient, each chart must be sufficiently large so that the compressor has some regularity to work with. This limits the granularity

of the random access to a few thousands of vertices. Also, the compressed random-accessible file stores a header with some information on how to *stitch* the charts together (this information corresponds to the elements shared by several charts). This means that the overhead is proportional to the number of charts. As the size of the models increases, the user must choose between keeping the number of charts constant and loosing granularity, or keeping the same granularity, but increasing the number of charts, negatively impacting the compression rate (a more quantitative study of these phenomena is given in Section 5.3). In practice, for reasonable compression rates, the existing algorithms have to keep the number of charts small.

We depart from these approaches by using a *hierarchical chartification* approach. Instead of having the user specify the number of charts to be used, we recursively split the mesh in two balanced charts, which are progressively subdivided until individual faces are reached. The information to stitch the charts together is stored in a novel succinct way, which enables the approach to work *without relying on any singe-rate compression algorithm*. This new approach is totally different from usual single-rate compression methods, but still results in reasonable compression rates. Emphasis is put on the *decompression efficiency*: The compression/decompression process is *asymmetric*. In addition, having an intrinsically hierarchical representation of the mesh enables us to embed a Bounding Volume Hierarchy (BVH) within the compressed mesh with minimal overhead. Where a previous approach required 50 to 100 bits per vertex to store the BVH, we are able to do it with only 2 bits per vertex.

Surprisingly, the divide-and-conquer approach has found very few applications in mesh compression. Only two such approaches have been proposed. The first, due to Ivrissimtzis *et al.* [Ivrissimtzis et al. 2002], finds triangle strips that cut the mesh into smaller meshes. However, this approach does not depart from the traditional approaches. It is actually a label-based method similar to Edge-Breaker [Rossignac 1999], but that traverses the mesh using the order in which the triangle strips are generated. The second one is very different. Starting from a triangle mesh, Aleardi *et al.* use a three-level chartification to succinctly represent a mesh [Aleardi et al. 2005]. They compress the lowest level by enumerating all possible triangulations with the same number of vertices (which is small), and code the actual mesh as a dictionary entry in this indexed set. Their compression method is therefore not based on the hierarchical subdivision itself, but resembles that of Choe *et al.* [Choe et al. 2009], although their goal is to provide fast neighbourhood queries rather than geometric random accessibility. Also, they do not target large meshes.

In this chapter, we present a new mesh compression method that has the following properties:

- **Hierarchical:** Our method is built on a recursive split of the mesh into two balanced independent parts.
- **Random Accessible:** The recursive partitioning allows the reconstruction of any requested part of the mesh without decoding other, less interesting parts.
- **Polygonal:** We are able to compress meshes with arbitrary polygons instead of only triangles.
- **Simplicity:** Our scheme is very simple to implement.

This chapter is organized as follows. The first part presents the hierarchical chartification process, connectivity and geometry coding, as well as how random accessibility is provided. This provides the main frame for our compression algorithm. In a second part, we show how this compression algorithm can be used for interactive visualization of very large models, by embedding a bounding volume hierarchy that only marginally increases the file size. Finally, we compare our approach with the other random accessible methods.

## 5.1 The algorithm

### 5.1.1 Hierarchical Chartification

The way we represent a mesh bears some resemblance with the approach of Choe *et al.* [Choe et al. 2009], in the sense that we also use the concept of *wire* to describe the mesh. However, in contrast to their approach, our charts are defined recursively such that each level $n$ of subdivision has $2^n$ charts with size proportional to $\frac{N_v}{2^n}$ (where $N_v$ is the number of vertices in the mesh), that can be decoded independently. Therefore, where their approach has a fixed granularity that depends on the number $K$ of charts they use, the granularity of our method can be decided by the decoder. We begin by giving a formal definition of what we call a *wire* before proceeding to the description of the mesh representation in itself.

#### 5.1.1.1 Wires

Let $G$ be a 2-manifold mesh of genus 0. We define a *wire* as a sequence of connected vertices (i.e. vertices that are joined by an edge) of $G$ such that any vertex appears at most once (Figure 5.1, $(a)$). A *closed wire* is a wire whose first and last vertices are connected. A wire that is not closed is an *open wire*. If the mesh has a boundary, a *boundary wire* (or simply *boundary*) is a closed wire containing all the vertices on the boundary (and only them) (Figure 5.1, $(b)$). A wire $C$ is a *cut wire* (Figure 5.1, $(c)$) if it joins two vertices $v_A$ and $v_B$ in the same boundary wire $W$ such that either:

- $v_A$ and $v_B$ are not adjacent within $W$
- $v_A$ and $v_B$ are adjacent and $C$ has at least one vertex in addition to $v_A$ and $v_B$ (in the case illustrated in Figure 5.1, $(d)$).

In addition, if two wires $W_A$ and $W_B$ share one endpoint but have no other vertex in common, we define the wire $W_A + W_B$ as the wire which contains all the vertices of $W_A$ and $W_B$ (and shares one endpoint with $W_A$ in the case there are several candidates, i.e. for a closed wire).

A wire is very easy to compress because it has implicit connectivity. Thus, the connectivity can be coded using only its number of vertices. Because adjacent vertices in the wire are connected in the mesh, it also exhibits good geometric correlation. Our algorithm uses the concept of wire as a basis for representing the mesh.

#### 5.1.1.2 Representing the mesh as a tree of charts

To take advantage of good compression ratios brought by wires, we store the mesh $G$ as a tree of wires (or equivalently, as a tree of charts). We begin by extracting a boundary wire $B_G$ of $G$ (if the mesh has no boundary, we remove a random face of the mesh, and use the resulting boundary). Then $G$ is split into two independent meshes $G_L$ and $G_R$ (see Figure 5.2). It is easy to see that the vertices that belong to both $G_L$ and $G_R$ form a cut wire. Let $C$ be this cut wire, and $v_A$ and $v_B$ the end vertices of the cut wire. Note that $v_A$ and $v_B$ belong to $B_G$. Therefore we can partition $B_G$ in two wires $W_A$ and $W_B$ such that $B_G = W_A + W_B$, $B_{G_L} = W_A + C$ and $B_{G_R} = W_B + C$.

$C$ as well as the indices $i_A$ and $i_B$ of $v_A$ and $v_B$ in $B_G$ are stored in the root node. Then the above process is applied recursively to $G_L$ and $G_R$ until only a face is left. The output of this process is a tree where each node stores a cut wire and two indices representing where this cut wire attaches in the parent boundary. The leaves of the tree are individual polygons. This defines a tree of charts as illustrated in Figure 5.3.

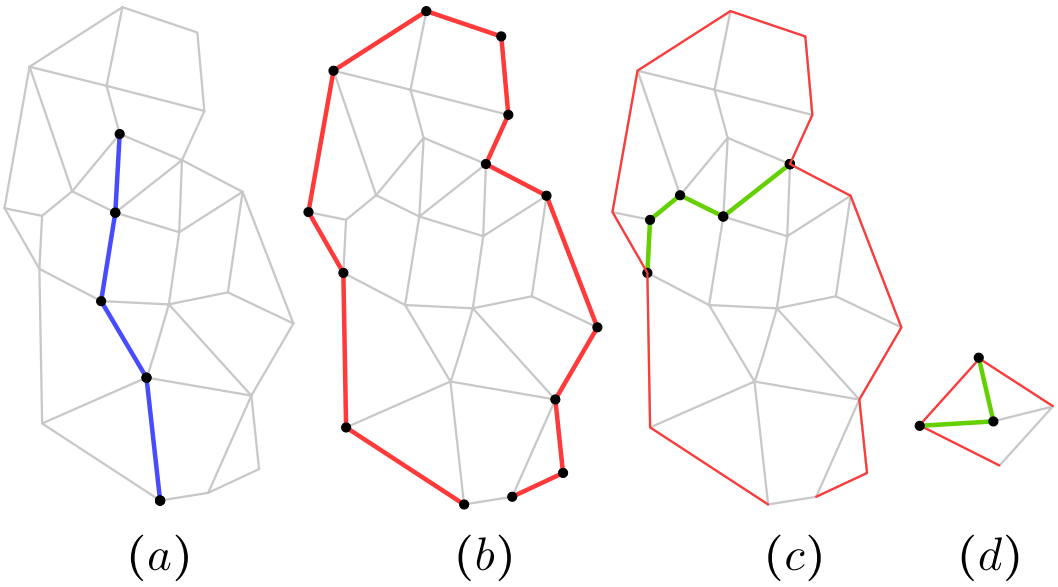


Figure 5.1: An open wire ((*a*), blue), a boundary wire ((*b*), red) and a cut wire ((*c*), (*d*), green).

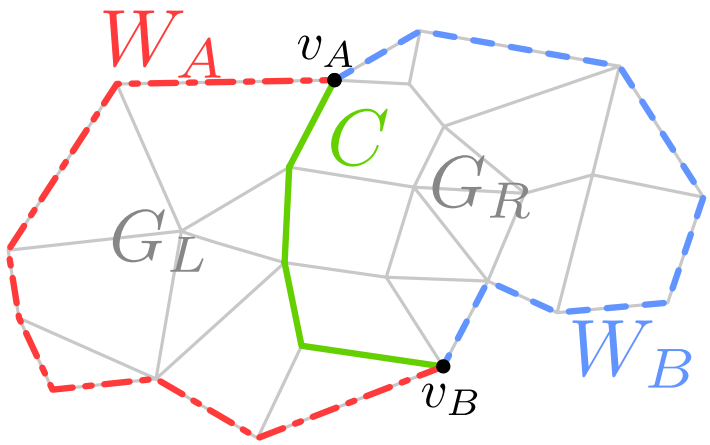


Figure 5.2: One step of the algorithm: splitting the mesh $G$ and its boundary. $C$ (green) is the cut wire, $G_L$ and $G_R$ are the two submeshes, and the original boundary $B_G$ is split into $W_A$ (red, dashed and dotted) and $W_B$ (blue, dashed).

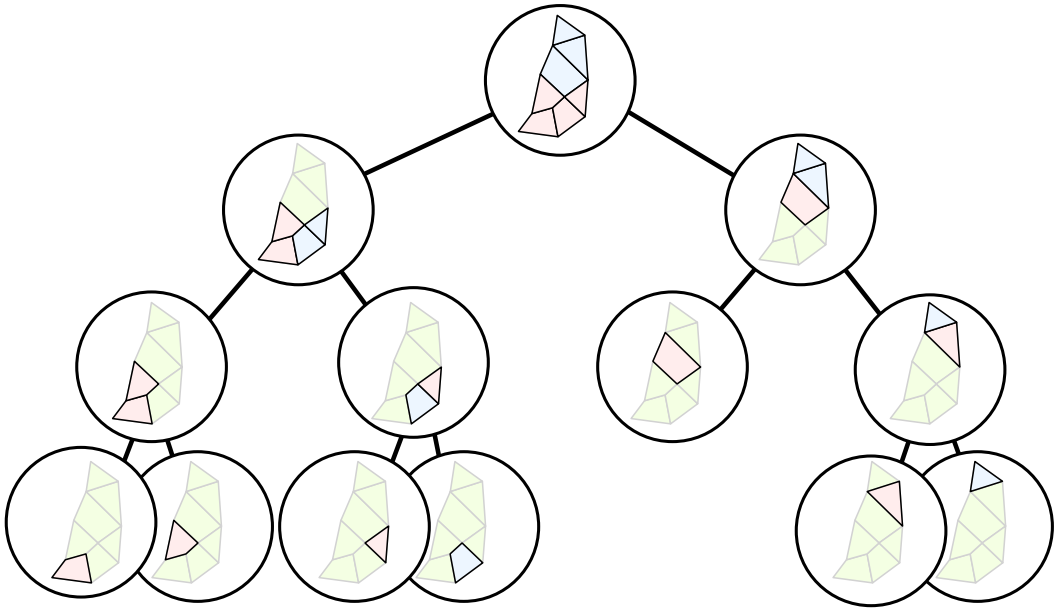

Figure 5.3: The tree of charts resulting from the recursive chartification process.

#### 5.1.1.3 Decoding an element of the mesh

Let us suppose that $B_G$ is known. The process begins at the root of the tree: $i_A$, $i_B$ and $C$ define two regions bounded by $W_A + C$ and $W_B + C$. We choose the region corresponding to the element to be decoded (★), and proceed recursively (Figure 5.4). Determining whether (★) is in the left or right region of the graph depends on the intended application. One example will be discussed in Section 5.2.

### 5.1.2 Coding Connectivity and Geometry

The above representation requires storing for each cut wire:

- The length $S_C$ of the cut wire (number of vertices),
- The indices $i_A$ and $i_B$ of the vertices $v_A$ and $v_B$,
- The position of each of the vertices of the cut (except $v_A$ and $v_B$).

Coding the geometry of the cut wire is straightforward: we apply a simple linear predictor on the sequence of cut vertices, followed by an entropy coding of the residuals. Using only the two last vertices for prediction is sufficient. The case of connectivity is more interesting: As there are approximately as many cut wires as faces in the mesh, this means that a naive storage of the cut wires will use 96 bits per face (if the length and indices are stored using 32 bit integers) for connectivity. This is equivalent to 192 bpv for triangle meshes and 96 bpv for quad meshes. However, this figure can be drastically reduced by using several techniques that we detail in the following paragraphs.

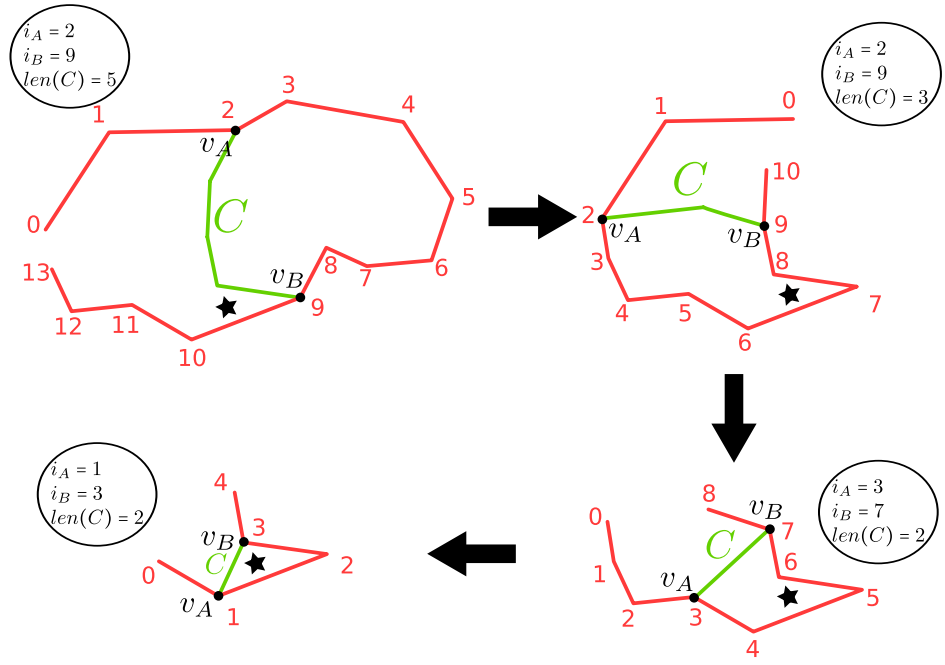


Figure 5.4: Decoding the element ★: We begin by numbering the root boundary, which has 14 vertices, from 0 to 13. We retrieve $v_A$ and $v_B$ from $i_A = 2$ and $i_B = 9$. We can then rebuild $C$ (here adding 3 vertices). We choose the left region (which contains the ★), and define the new boundary as $W_{[0,2]} + C + W_{[9,13]}$. We proceed recursively, until we hit an unsplittable polygon.

#### 5.1.2.1 Compressing cut wire lengths

The distribution of $S_C$ is very biased towards zero. Most of the final cuts will have a length of zero (not including the two end vertices). More generally, for a mesh of $N_v$ vertices, we can expect the size of a cut to be approximately $\sqrt{N_v}$. If the tree of charts is balanced, there will be $2^n$ charts with a size of around $\frac{N_v}{2^n}$ vertices, and thus approximately $2^n$ cut wires with length $\sqrt{N_v}\left(\frac{1}{\sqrt{2}}\right)^n$. Hence, the distribution of $S_C$ will be roughly geometric, and therefore $S_C$ is suitable for entropy coding. Figure 5.5 shows the typical distribution of the cut wire lengths, that has an entropy of around 1.6 bits per cut wire.

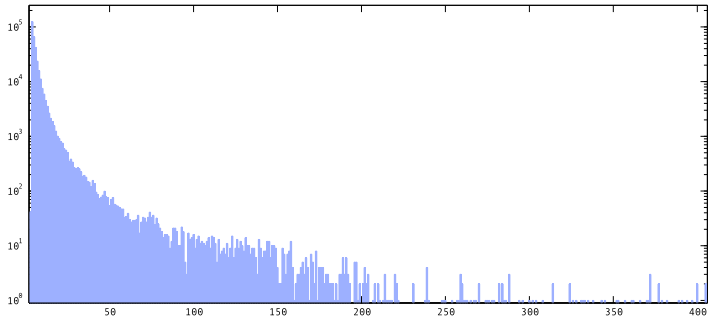


Figure 5.5: Distribution of the lengths of the cut wires for the *armadillo* model (log scale). Entropy is 1.6 bits per cut.

#### 5.1.2.2 Compressing indices

Similarly to the length of the cut wire, the size of the boundary decreases according to a geometric distribution. For each submesh with $n$ vertices, the indices $i_A$ and $i_B$ are uniformly distributed in $[0, \sqrt{n}]$. Therefore, they can also be compressed with entropy coding. The resulting bit rates are on the order of 2.5 bpv for each index, which is not very satisfying (see Figure 5.6).

However, it is possible to greatly improve the compression of the indices by introducing two notions that we call *opposite vertex* and *context-dependent numbering*.

We remark that the cuts that are well suited to building a balanced tree, i.e. cuts that split the mesh into submeshes with similar number of vertices, also tend to separate the boundary in a balanced manner. We take advantage of this property. Instead of coding $i_A$ and $i_B$ separately, we code $i_A$ and $\delta_B$. $\delta_B$ is the difference in the position of $v_B$ in the boundary between the actual cut and the balanced cut (that is, the cut that would result in $W_A$ and $W_B$ having the same number of vertices). Given $v_A$, we call the vertex $v_B$ that would cut the boundary in a balanced manner the *opposite vertex of* $v_A$, noted $o(v_A)$. By favouring cuts that tend to link a vertex to its opposite, we are able to obtain entropies of around 0.5 bits for $\delta_B$ (see Figure 5.7), versus 2.5 bits for $i_B$.

As we know how to succinctly code $i_B$, most of the bit budget is now allocated to coding $i_A$. The only available context data known at the time of decoding is the boundary information. Therefore, a scheme to code $i_A$ should only rely on this information. The numbering of the boundary vertices comes from the context of the parent subdivision in the subdivision tree, and is therefore arbitrary in the current context. Thus, all the values for $i_A$ are equiprobable, preventing us from decreasing the entropy below the geometric distribution of Figure 5.6. Our scheme uses the geometric information of the boundary to renumber the vertices of the boundary (the Section 5.1.2.3 details how this is done), and uses the cut with smallest index $i_A$. Hence, the probability density of $i_A$ is not constant

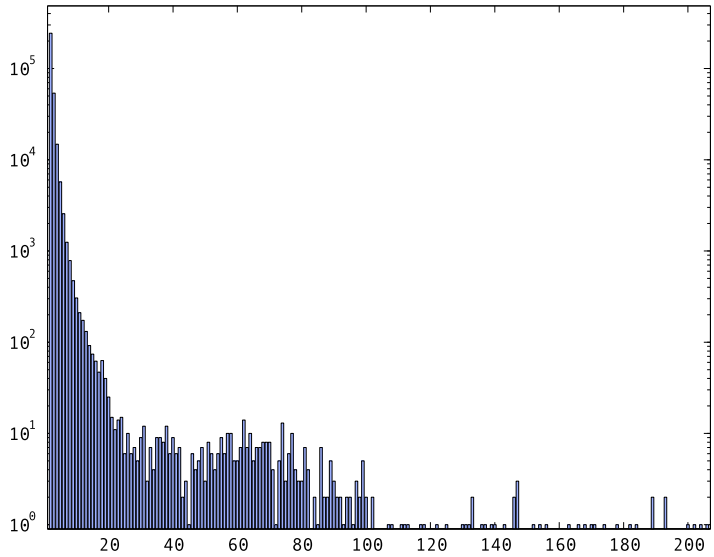


Figure 5.6: Distribution of the indices $i_A$ and $i_B$ for the *armadillo* model (log scale). Entropy is 2.5 bits per index.

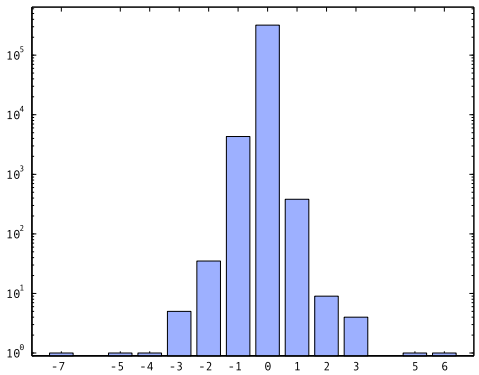


Figure 5.7: Distribution of $\delta_B$ for the *armadillo* model (log scale). Entropy is 0.5 bits per index.

anymore, but has a small variance around 0. This way, the entropy of $i_A$ drops from 2.5 to 0.5 bits, with the distribution shown in Figure 5.8.

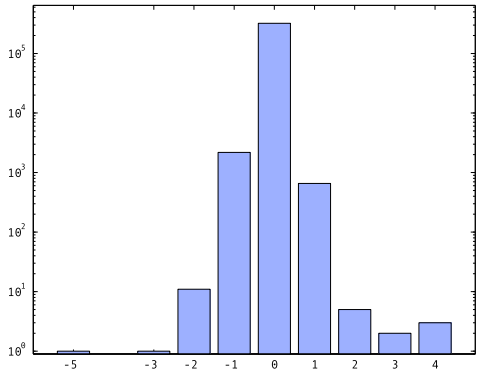


Figure 5.8: Distribution of $i_A$ after renumbering for the *armadillo* model (log scale). Entropy is 0.5 bits per index.

#### 5.1.2.3 Cut Selection

The compression ratio for a given cut depends on the length of the wire, the smoothness of its geometry, the closeness of $v_B$ to $o(v_A)$, and a good renumbering of the boundary vertices for compressing $i_A$. Thus, finding the best cut wire with respect to compression ratio is too slow for practical use. We must find heuristics to provide an efficient cut. A good cut wire should have the following properties:

1. Be short, so that the number of vertices in $C$ is small (to enable better entropy coding).
2. Cut the boundary in two parts as equal as possible, so that $\delta_B$ is biased towards zero (to enable better entropy coding).
3. Be smooth, to provide good correlation between the geometry of adjacent vertices.

In addition, the heuristic should not be too complex to compute, to reduce the time needed for compression.

We use the following heuristic to determine the best cut: For each index $i$ in the boundary, we note $v^i$ the $i$-th vertex, and $o(i)$ the index of $o(v^i)$. For all $i$, we compute the Euclidean distance between $v^i$ and $o(v^i)$. Let $i_0$ be the index such that this distance is smallest (Figure 5.9). In case of a tie, we use the first smallest distance pair. We then renumber the vertices on the boundary away from $i_0$ in each direction. We denote the original numbering by a superscript and the renumbering by a subscript. Thus $v^{i_0}$ becomes $v_0$. This new order is known to both coder and decoder.

If there exists a cut wire from $v_0$ to $v_{o(0)}$, then $\delta_B = 0$ and $i_A = 0$. Else[1], we check if there is a cut wire $(v_0 \rightarrow v_{o(0)-1})$; if there exists one, $\delta_B = -1$ and $i_A = 0$, else we proceed with $(v_k \rightarrow v_{o(0)-l})$ with increasing $k + l$ until a cut wire is found (see Figure 5.10). The actual information that we store is not $i_A$ and $\delta_B$, but the rank $R$ of the first suitable cut wire found, which enables to code both indices with one single index.

To find the actual vertices of the cut wire, we *grow* $G_R$ and $G_L$ from the newly determined $W_A$ and $W_B$ until all the vertices in $G$ are visited (We call *growing* the process of augmenting a submesh $G_{sub}$ with the unvisited vertices of $G$ that have neighbours in $G_{sub}$). The resulting cut wire is the

[1] See for example the case of Figure 5.1, (*d*).

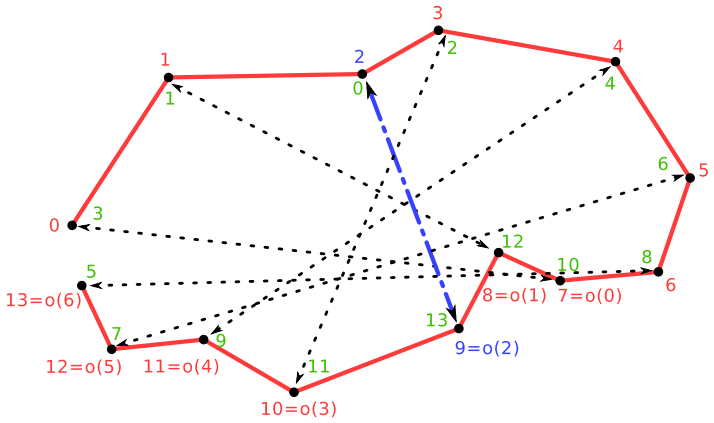


Figure 5.9: Finding the basis vertices for renumbering: The opposite vertices with shortest euclidean distance are picked (here 2-9). The resulting renumbering is in green.

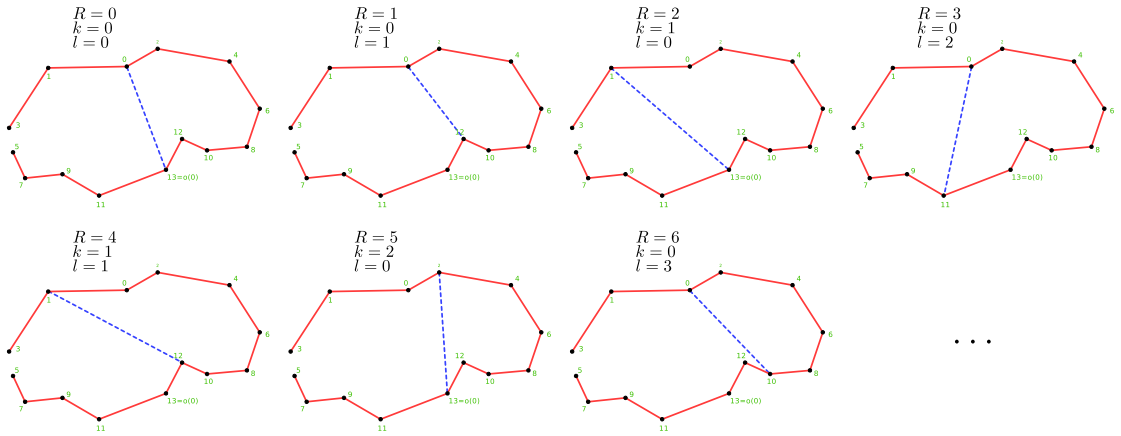


Figure 5.10: The cut wire selection process. We try to find cut wires from $v_k$ to $v_{13-l}$, with increasing $k + l$. $R$ denotes the rank of the trial. A suitable cut wire is usually found for $R = 0$.

shortest path through the vertices that have neighbours in both $G_L$ and $G_R$. The Figure 5.11 shows a simple example of the growing process.

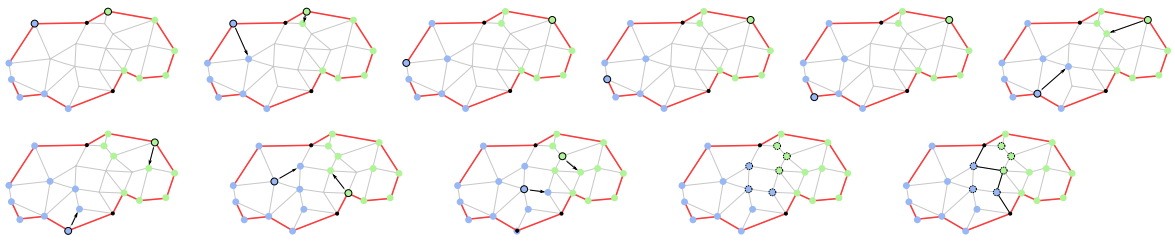

Figure 5.11: The growing process.

As we visit vertices further from the original guess (which is balanced) in increasing order, property (2) is verified. We use the smallest geometric distance, therefore the resulting cut is generally short, and thus property (1) is verified. The smoothness of the geometry of the cut wire depends on the regularity (in terms of vertex degree) of the mesh because of the use of the growing algorithm. Because we select the shortest path within the vertices that have neighbours in both $G_L$ and $G_R$, the cut wire will usually avoid zigzagging, since this would result in a longer path; therefore property (3) is generally verified (Figure 5.12). The heuristic is in $O(N^{\frac{3}{2}})$ in the worst case, but often becomes $O(N)$ practically as a path usually exists for the original guess.

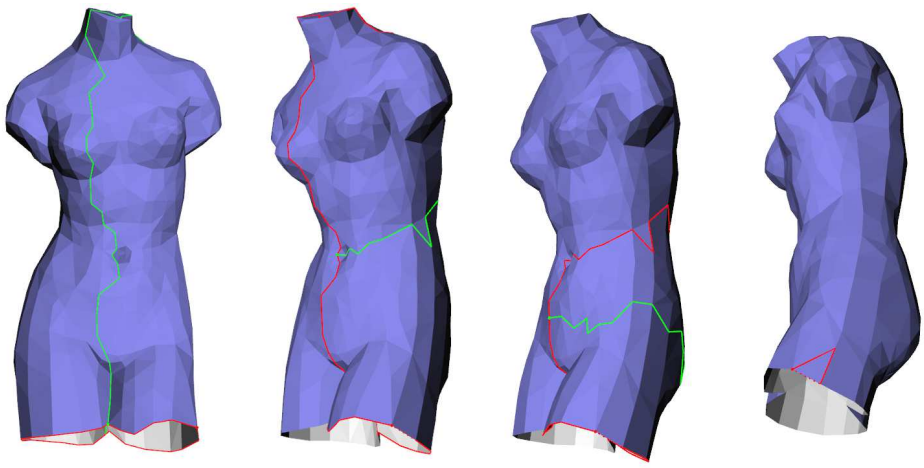

Figure 5.12: Some of the cuts determined by the proposed heuristic on the *Venus* model. The cut wires are in green, the boundary wires in red. Red points indicate the starting vertex of the boundary wire, and dashed edges close the boundaries. Note that the subdivision is balanced, and the cuts are short and reasonably smooth.

One may argue that using the *shortest path* between $v_A$ and $v_B$ instead of the growing algorithm would result in a cut that would better address properties (1) and (3). We experimented with this approach, however it has two major drawbacks. First, this approach is slower that growing. Second, and more importantly, it results in very unbalanced chartification. It is very easy to be convinced of this by looking at Figure 5.13.

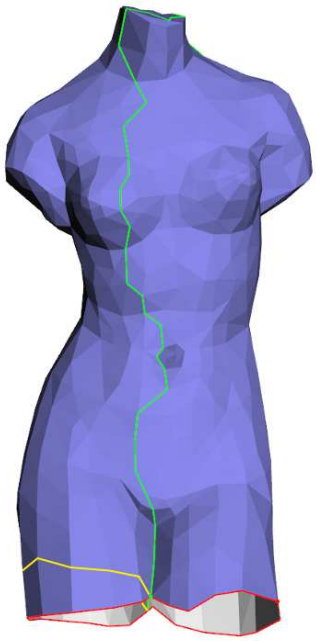

Figure 5.13: The shortest path approach to chart subdivision usually results in very badly balanced chartifications. The *Venus* model has been cut using the growing (green cut) and shortest path (yellow cut) algorithms. The latter results in an unbalanced cut. In addition, recursively subdividing the right part will once again yield a bad cut, resulting in a *comb* tree.

#### 5.1.2.4 Initial boundary selection

We use as initial boundary wire the largest available boundary. If the mesh has no boundary, we remove a random face of the mesh, and use the resulting boundary. The heuristic described above then results in a balanced cut, as shown in Figure 5.14.

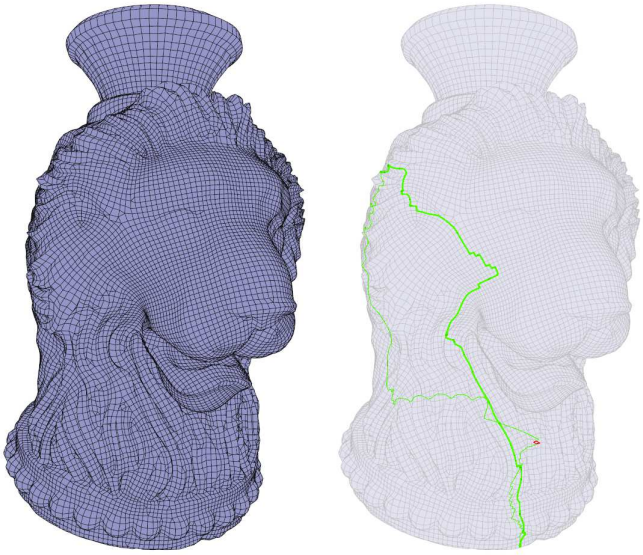

Figure 5.14: Handling meshes without boundaries: A random face is removed and its vertex loop is used as initial boundary. The proposed heuristic then splits the mesh in a balanced manner.

### 5.1.3 Providing Random Accessibility

Previous random-accessible methods used an indexing structure in the file header to enable individual access to the charts. Departing from this approach, we embed the indexing structure inside the tree of subdivisions. This means that the random accessibility information itself is random accessible. Where Choe *et al.* [Choe et al. 2009] have to first reconstruct the wire-net mesh of the whole mesh, our method enables to only build the wire-net mesh of the part that is requested. This way, we are able to determine the path to follow to decode only the required part of the mesh without decoding other parts of the tree, thus enabling random access.

If the goal of the algorithm was to achieve the best possible compression, the best way would be to compress the bit stream resulting from the above compression process using an entropy coding method (e.g. arithmetic coding). However, such a method is unable to provide random accessibility in the bit stream. This comes from the fact that an arithmetic encoder *does not use an integer number of bits* to store each symbol. This property enables better compression rates compared to Huffman coding, but in the context of random access, this is actually a drawback. Providing random accessibility means that the compression algorithm must be able to *skip* a certain quantity of information and resume decoding at a specific point in the file. Using arithmetic coding to encode the bit stream would mean that the coder has to make a jump in memory of a rational number of bits, which does not make sense.

Our random access scheme is based on the *tree of charts* representation of the mesh built earlier. The tree is stored in a depth-first manner, which uses 2 bits per node in general (see e.g. [Jacobson 1989]). The data at the nodes is entropy coded using a scheme which enables individual symbol

decoding (we used Huffman coding). For more efficiency at lower levels, the tree is coded in an autumnal fashion [Fabbrini and Montani 1986].

While storing the tree uses 2 bits per node if a uniform distribution of trees is considered, we were able to reduce this value by taking advantage of the properties of the special structure of the tree in our case. Because the tree is well balanced, approximately half the nodes have two children that are leaves (we note this situation $C_{\square,\square}$), and one quarter have two children that are internal nodes ($C_{\wedge,\wedge}$). Another quarter have either a leaf on the left and a internal node on the right ($C_{\square,\wedge}$), or an internal node on the left and a leaf of the right ($C_{\wedge,\square}$). Figure 5.15 illustrates these situations. This enables to use the prefix-free codes 1, 01, 000 and 001 to code each of these cases (Figure 5.15, bottom left), with an average code length of $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{3}{8} = 1.75$ instead of 2. If the mesh has holes, we must add the cases where there is a *hole* (or *non-face*) leaf on the left, and an internal node on the right ($C_{\boxtimes,\wedge}$), and its symmetric case ($C_{\wedge,\boxtimes}$). On manifold meshes, the case with two non-face leaves does not happen. There are usually a small number of holes in the mesh, so the probabilities for these symbols are negligible. However, they still modify the previous prefix-free codes by lengthening one of the most current codes (1, 01, 0001 and 001, Figure 5.15, bottom right), leading to an average code length of $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{8} + \alpha\epsilon = 1.875 + \alpha\epsilon$.

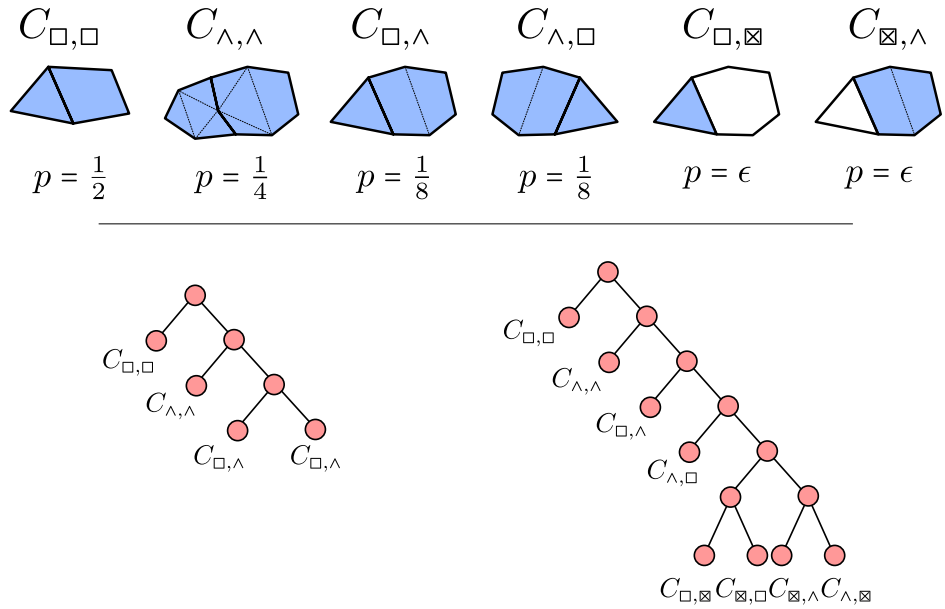


Figure 5.15: The 8 different types of nodes in a tree of charts, with associated probabilities (top). Blue denotes faces, white signals holes in the mesh. For the last two types, we did not represent the symmetric configurations. A mesh without holes can be represented using only the 4 first configurations. For this kind of meshes, we can use the Huffman tree on the bottom left, with codes 1, 01, 001 and 000. For meshes with holes, we need the tree on the bottom right.

To provide random accessibility, in addition to the cut wire data, we store in each node of the tree the information needed to reach the left and right children. This enables jumping to the right (resp. left) submesh without having to decode the left (resp. right) one. In a typical application without size constraints, this is typically done using two pointers, one to the left subtree, and another to the right subtree. As we have a continuous memory layout of the tree, we can use *relative* indexing and only give memory offsets for the children. Furthermore, because the tree is laid out in memory in depth-first order, the offset for the left subtree is 0, and that of the right child corresponds to the size of the left subtree, so only one of the offsets needs to be stored.

A naive implementation would therefore use 32 bits per node (one integer offset pointer). However, this information can be stored in a more efficient way (see Figure 5.16). Consider a node $G$ in the tree, let $D_G$ be the size (in bits) of the tree of root $G$ and $D_C$ the size of the cut wire data stored at node $G$. We know that the left subtree has a size $D_{G_L}$ which is smaller than $D_G - D_C$, so only $log_2(D_G)$ bits are enough to code $D_{G_L}$. We can also retrieve the size of the right subtree $D_{G_R}$ as $D_G - D_C - D_{G_L} - log_2(D_G)$. This way, at each node, the left and right children nodes can be decoded independently, by offsetting the memory pointer of 0 bits (left child) or $D_{G_L}$ bits (right child). A leaf is simply a node of size zero. This storage method enables random access with around 40% overhead on our test meshes. If one considers that the storage of the structure of the tree counts as connectivity, and that only the random access pointers are overhead, then the overhead is around 20% on average.

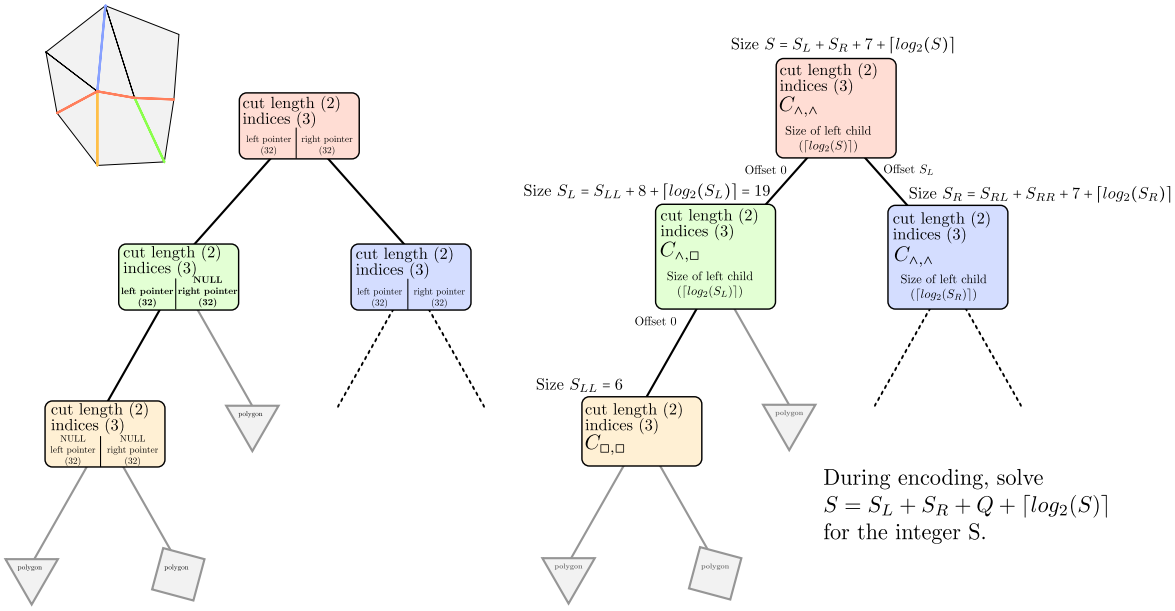


Figure 5.16: Comparison of a naive implementation (left) of a random accessible chartification and our proposed approach (right), for the mesh in the top left corner. Node colours refer to coloured cuts in the mesh. For simplicity, we suppose that there are no holes, so we are able to use the encoding of Figure 5.15, bottom left. Also, we assume that storing the indices uses a constant number of 3 bits, and storing the cut wire length uses 2 bits.

### 5.1.4 Compression rates

Table 5.1 details the compression rates for our scheme. Compression rates are better for quad than for triangle meshes, typically 19 versus 26 bits per vertex. This is because there are roughly as many cut wires as faces, but there are twice as many polygons in a triangle mesh than in a quad mesh with the same number of vertices. Our coder uses about 3 bits per polygon for connectivity (without taking into account the encoding of the structure of the tree, which counts for both connectivity, geometry and random access, and that we consider as overhead). Hence, the ratios for connectivity are roughly 6 bits per vertex for triangle meshes and 3 bpv for quad meshes. Also, as we use Huffman coding in our memory layout to enable random access, the average code length cannot drop below 1 bit per symbol. We include corresponding entropies for reference.

| Model | #V | cut wire length | indices | geom. (bpv) | overhead | total |
|---|---|---|---|---|---|---|
| Igea (T) | 130k | 1.44 (1.19) | 1.16 (0.64) | 13.88 (13.82) | 42% | 27.2 |
| Ramses (Q) | 160k | 1.83 (1.76) | 1.51 (1.35) | 10.18 (10.08) | 31% | 17.7 |
| Armadillo (T) | 170k | 1.44 (1.21) | 1.11 (0.47) | 11.48 (11.45) | 49% | 24.7 |
| Buste (T/Q) | 255k | 1.76 (1.67) | 1.75 (1.60) | 10.73 (10.51) | 36% | 19.4 |
| Eros (T) | 476k | 1.43 (1.16) | 1.32 (1.01) | 12.23 (12.20) | 45% | 25.8 |
| Neptune (Q) | 3.7M | 1.83 (1.73) | 1.54 (1.39) | 17.31 (17.23) | 21% | 24.9 |

Table 5.1: Compression results for various models (T denotes models having mostly triangles, Q models having mostly quads). Note that cut wire length and indices are given in bits per cut wire. The total compression ratio is given in bits per vertex and includes the overhead induced by the random accessible memory layout. Numbers between parentheses are entropies. Geometry is quantized to 12 bits. The *Neptune* and *Eros* models are quantized to 16 and 14 bits respectively since they have finer resolution.

## 5.2 Interactive Visualization

To illustrate the significance of our approach, we implemented as an example a view-dependent rendering framework. This approach is useful when the model is so large that rendering the whole mesh is too time consuming, or even impossible because the model does not fit into main memory. Using the random accessibility provided by our compression method, we can render only the portion of the model which is of interest to the user without decompressing the whole model. Thus, we decrease the time between request and actual display.

View frustum culling [Clark 1976] is often used to enable view-dependent rendering. The method consists of building a hierarchy of bounding volumes. If a bounding volume does not intersect the view frustum, then all its children will not lie inside the frustum. Else, the hierarchy is searched one level deeper (Figure 5.18). We can take advantage of the hierarchical representation provided by our method, by slightly modifying the compression process. For each submesh in our hierarchical representation, a bounding sphere is computed. Its center is the center of mass of the boundary, because this is the only information available to the decoder (see Figure 5.17). At the time of decoding, only a quick frustum/sphere intersection test needs to be carried out to decide whether refinement is necessary or if the whole submesh can be discarded.

The radius of the bounding sphere is stored along with the cut wire, and can be aggressively quantized since precision is not very important. The radius gets smaller in lower and more populated levels, therefore its entropy is very low. As we want to guarantee that a face that falls inside the viewport is always rendered, the radius is quantized by excess. Therefore, there is a trade off between more aggressive quantization, that enables better compression but reduces the granularity of random access, and finer quantization, that increases the size of the file but enables more precise intersection tests. Figure 5.20 illustrates this trade-off. We found that using a number of quantization bits larger than 8 usually does not provide better granularity. Also, because we are using Huffman coding, the average code length for the radius cannot drop below 1 bit. Therefore, quantization levels below 6 bits do not decrease file size, even though they decrease entropy of the quantized radius. In our experiments, we used the conservative value of 8 bits. The radius code lengths and the associated file

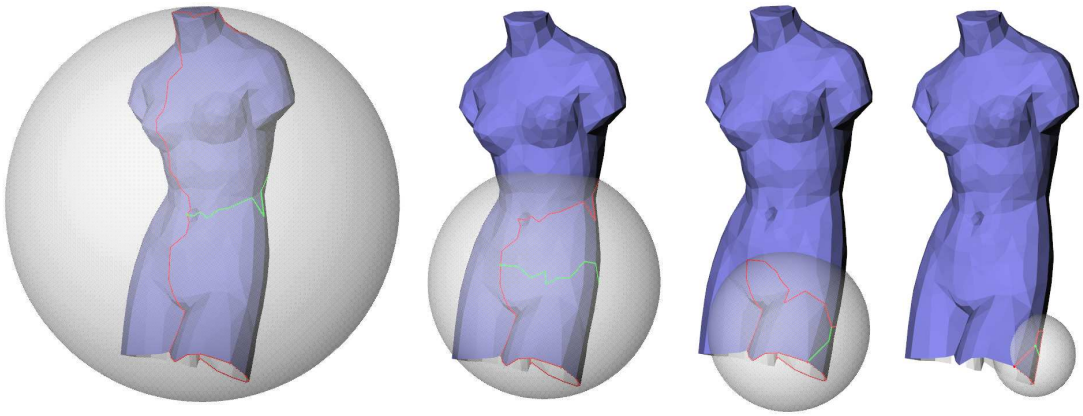

Figure 5.17: Some bounding spheres of the hierarchical chartification of the *Venus* model.

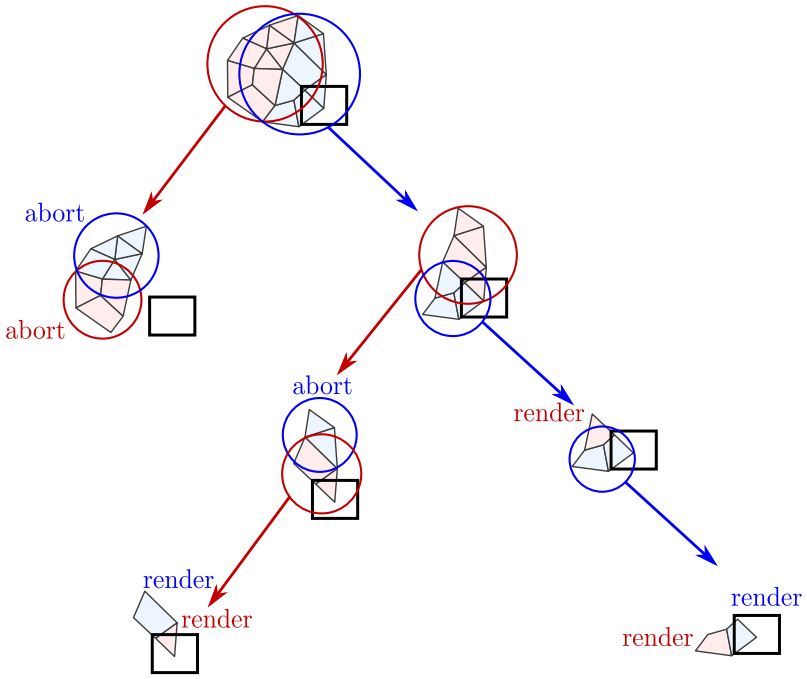


Figure 5.18: Hierarchical frustum culling in 2D: the children of a node are searched only if their bounding circle intersects the frustum.

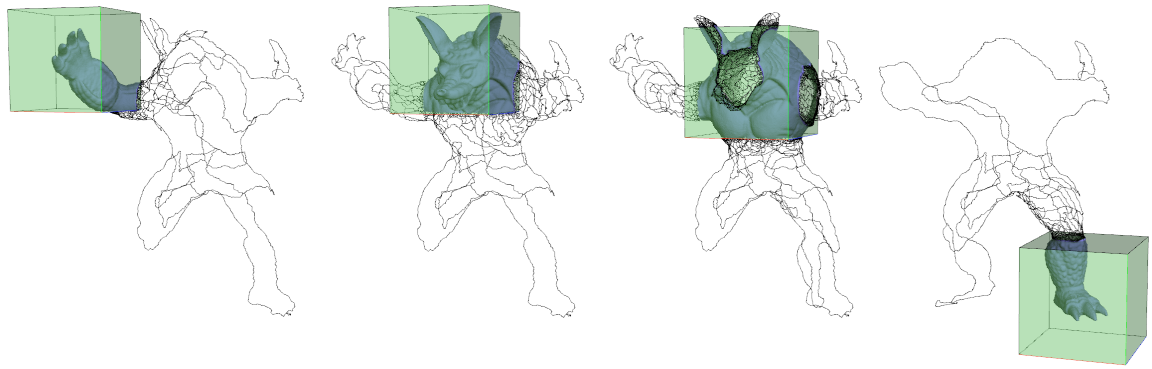

Figure 5.19: View-dependent rendering using a cube-shaped frustum. The green box represents what the user sees. The black wires are the overhead wires that need to be decoded.

sizes are shown in table 5.3.

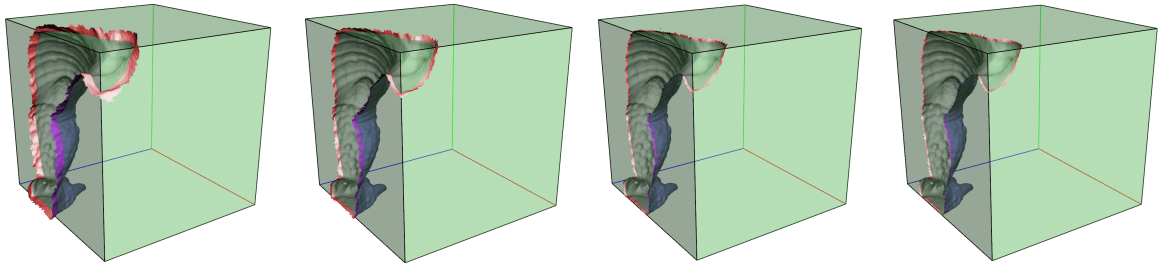

Figure 5.20: Effect of the radius quantization depth on the granularity of random accessibility. The query domain is the green box. The faces that are classified as *inside* the query domain by the sphere intersection test, but that are actually outside are shown in red. From left to right, the radius is respectively quantized to 6, 7, 8 and 10 bits. The corresponding average Huffman code lengths are 1.02, 1.06, 1.52 and 1.80.

The random access capability of our method enables interactive visualization of large models. As only the portion of the model that falls into the viewport is fully decompressed, the parts of interest can be decompressed and displayed very quickly, resulting in interactive frame rates as long as the part of interest is not too big. Table 5.2 gives the timings and memory footprints for rendering typical images like those on figure 5.19. Note that to demonstrate the decompression efficiency, our approach does not cache anything from one frame to the other. At each frame, the tree is traversed from the root and the search stops as soon as there are no nodes with bounding spheres intersecting the frustum. When a face is decoded, it is drawn but not stored. Each time we walk up from a node to its parent, all the associated structures are released. This way, memory consumption is kept to a minimum, since the part of the mesh that is displayed is never loaded into the memory as a whole.

## 5.3 Comparison with previous approaches

In [Choe et al. 2004; 2009], Choe *et al.* use their algorithm for view-frustum culling. To enable this, they must entirely decode the wire-net mesh to test whether each face intersects the viewport. Then, they decode the visible charts. As they use the Angle Analyser algorithm [Lee et al. 2002] to encode

| Model | # faces visible | # polygons per second |
|---|---|---|
| Igea | 1300 (0.5%) | 640k (max fps) |
| | 24k (9%) | 1100k (max fps) |
| | 238k (89%) | 1200k (5 fps) |
| Ramses | 5600 (3%) | 700k (max fps) |
| | 60k (37%) | 900k (14 fps) |
| | 163k (100%) | 900k (5 fps) |
| Armadillo | 1700 (0.5%) | 590k (max fps) |
| | 42k (12%) | 1100k (25 fps) |
| | 345k (100%) | 1100k (3 fps) |
| Buste | 5200 (2%) | 660k (max fps) |
| | 51k (19%) | 900k (17 fps) |
| | 269k (100%) | 900k (3 fps) |
| Eros | 9200 (1%) | 1000k (max fps) |
| | 146k (15%) | 1200k (8 fps) |
| | 953k (100%) | 1200k (1 fps) |
| Neptune | 31k (1%) | 520k (17 fps) |
| | 510k (14%) | 900k (2 fps) |
| | 3.7k (100%) | 900k (0.23 fps) |

Table 5.2: Performance results for view-dependent rendering of various models. For each model, a part of the mesh was rendered, as in figure 5.19. The table summarizes the random-access decompression and rendering times for parts of various size. The performance is computed by dividing the total time needed to decompress and draw the visible faces (including the overhead of decoding the wires that lie outside the viewport, but were needed for decompression) by the number of *displayed faces*. *max fps* means that the frame rate is limited by the display capabilities (60 fps in our case), and not by the decoding speed.

| Model | #V | radius | total (bpv) |
|---|---|---|---|
| Igea (T) | 130k | 1.51 (1.26) | 30.4 |
| Ramses (Q) | 160k | 1.23 (0.83) | 19.0 |
| Armadillo (T) | 170k | 1.28 (0.96) | 27.4 |
| Buste (T/Q) | 255k | 1.38 (1.11) | 21.0 |
| Eros (T) | 476k | 1.01 (0.09) | 28.0 |
| Neptune (Q) | 3.7M | 1.02 (0.12) | 25.9 |

Table 5.3: Effects of embedding the quantized radius on the bit rate. The center column gives the average Huffman code length and the entropy of the quantized radius (on 8 bits). The rightmost columns gives the total size of the interactive-visualization-enabled mesh, in bpv, including all overheads.

the charts, each chart must be either fully decoded or not decoded at all. Therefore, there is a large decoding overhead when only a small part of a chart is visible. On the other hand, as the charts are encoded independently, they can decode a chart, draw it, then release the memory used for decoding, and move to the other chart. However, the wire-net mesh must be maintained in memory at all times. In contrast, our method decodes wires when it needs them and releases them as soon as they are not needed anymore. The difference in the pattern of the overhead is shown in Figure 5.21.

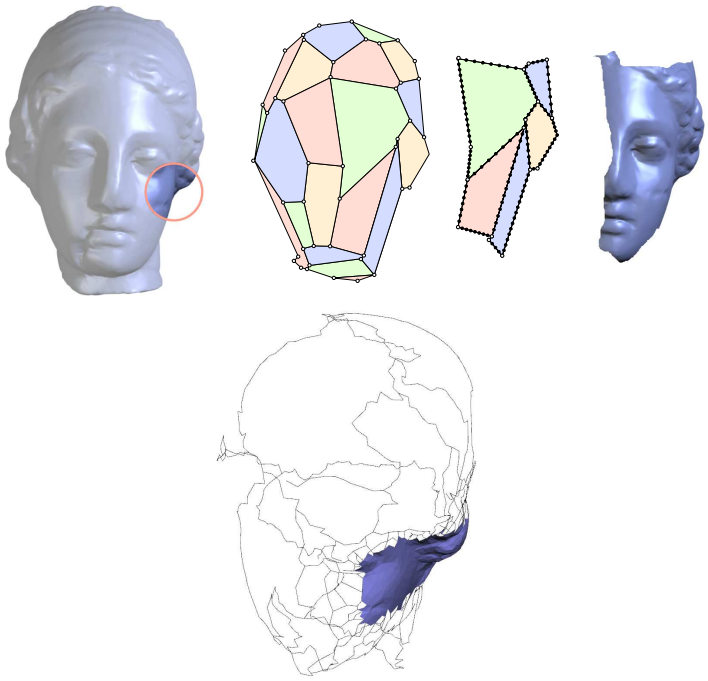

Figure 5.21: Difference in overhead patterns for the method of Choe *et al* (top), and our method (bottom). For our method, the overhead is the wires shown in black. For the approach of Choe *et al.*, the overhead consists in the wire-net mesh, the associated wires, and the parts of the charts that are decoded but not visible.

In the following, we evaluate the complexity of the two approaches, in a memory and complexity point of view. The memory footprint needed for the view-dependent rendering method of Choe *et al.* is

$$m_c(n, K) = O(K + T) = O(K + \frac{n}{K}) \tag{5.1}$$

where $n = K \times T$ is the number of vertices in the global mesh, $K$ the number of charts and $T$ the number of vertices per chart. The associated best case time complexity is

$$t_c(n, K) = O(K + \lceil \frac{\alpha n}{T} \rceil T) = O(K + \frac{\lceil \alpha K \rceil n}{K}) \tag{5.2}$$

where $\alpha$ is the proportion of vertices that fall into the viewport.

On the other hand, in the same case, we use

$$m_o(n) = O(\sum_{i=0}^{log_2(n)} \sqrt{\frac{n}{2^i}}) = O(\sqrt{n}) \tag{5.3}$$

because we offer random access with polygon granularity. The corresponding time complexity is

$$t_o(n) = O(\sum_{i=0}^{log_2(n)-log_2(\alpha n)} \sqrt{\frac{n}{2^i}} + \alpha n) = O(\sqrt{n} \times (1 - \sqrt{\alpha}) + \alpha n) \quad (5.4)$$

When main memory availability becomes a problem, the equation 5.1 suggests that the best choice for $K$ is $\sqrt{n}$, in which case the memory complexity of both algorithms are roughly equivalent. In that case, our algorithm has a small time complexity advantage. The results given in [Choe et al. 2004; 2009] favor high compression ratios and speed of decompression over memory usage by using small values for $K$ ($100 \ll \sqrt{n}$). Therefore, we cannot fairly compare compression ratios with their scheme because the compression ratios for their method increase with $K$. However, by extrapolating on their results, we can suppose that their scheme is better for triangle meshes.

As far as compression ratios are concerned, our method seems better than that of Choe *et al.* on quad or higher degree polygonal meshes. As they use the single-rate compression scheme of [Lee et al. 2002] to compress the charts [Choe et al. 2004], we can expect their rates to be roughly the same for higher degree polygons as for triangles. However, our scheme is more efficient on higher degree polygons, because the vertices/faces ratio is lower. This is confirmed experimentally as quad meshes are compressed to 20 bpv instead of 27 for triangle meshes.

Once again, it is difficult to compare our view-dependent rendering times with the ones in [Choe et al. 2009] fairly, as the overhead for their method depends on $K$. Choe *et al.* render a vertex in 1.35 microseconds, not including the overhead of decoding the vertices that lie outside the viewport but inside the partially visible charts. For reference, our algorithm renders approximately 1 million faces per second, **including overhead**, i.e. a vertex every 2 microseconds for triangle meshes, or every 1 microsecond for quad meshes.

Another concern, as evoked before, is that the algorithm of Choe *et al.* sometimes results in incorrect answers to geometric queries, because the geometric random access criterion is based on the intersection of the wire-net mesh faces with the viewport. As the charts are not perfectly planar, some queries can return no answers whereas there were actually polygons inside the viewport (see Figure 3.9). On the other hand, our algorithm *guarantees* no false negatives, because an empty intersection between the viewport and a bounding sphere means that all the child charts are outside the viewport.

We do not compare our results with those of Yoon and Lindstrom [Yoon and Lindstrom 2007], because their method is very different since it does not provide the same type of random accessibility – they address indexed and adjacent RA.

Compared to the only other method that explicitly codes a bounding volume hierarchy [Kim et al. 2010] to enable random access, our method is able to embed the BVH *within* the compressed mesh. Therefore, there is a very small overhead for the BVH: Where the approach of Kim *et al.* uses approximately 100 bpv and triples the bit rate for mesh representation alone, the overhead of embedding the BVH in our method is usually less than 2 bpv. This demonstrates the advantage of using an intrinsically hierarchical representation of the mesh for geometric random access.

Note that the kd-tree based method of Du *et al.* provides a kind of BVH in the sense that it is possible to use the kd-tree as an acceleration structure for geometric queries [Du et al. 2009]. However, as they only provide random accessibility with a block granularity, the hierarchy stops as soon as the $L$-th level is reached. Also, the decompression timings are worse than our method: They decompress around $150k$ vertices per second – *not including overhead* – to visualize $300k$ vertices of a mesh that has $3.4M$ vertices, which is roughly 3 times less efficient than our method (we decompress and render around $900k$ faces, or $450k$ vertices, per second). The decompression speed of CHuMI viewer [Jamin et al. 2009] is roughly equivalent to [Du et al. 2009]: The authors report a decompression performance

of up to $300k$ triangles (i.e. $150k$ vertices) per second. Note that the hardware used for benchmarks in both articles is better than ours, so the difference should be even more pronounced.

An interesting feature of the algorithms of Choe *et al.*, Du *et al.* and CHuMI viewer is that they provide an actual *coarse polygon mesh* that can be used as a guide for the view-dependent exploration of the mesh. In contrast, our method only has a *wireframe mesh* that is less comfortable to work with. It would be possible to fill this wireframe mesh with as-flat-as-possible polygons, but this would not be efficient in terms of rendering speed. As a future work, we would like to investigate a complementary scheme that would replace these high-degree polygons by good low-degree approximations.

## 5.4 Discussion

There is a limitation to the categories of meshes on which our algorithm applies. When chartifying the mesh, we suppose that each cut splits the mesh in two parts. It is easy to see that this will not be the case for meshes with handles (see Figure 5.22 for an example). In these cases, there will be some cuts that only remove a handle, but the associated node in the tree of charts will only have one child. We propose to deal with this problem by introducing a ninth tree structure code for the removal of a handle ($C_{\circlearrowleft}$). The probability of this symbol will depend of the genus $g$ of the surface. For usual meshes, $g$ is small, thus the addition of this symbol does not change the codes for the 4 more frequent symbols, thus not impacting compression rate. We did not yet implement this method in our prototype, so we cannot give experimental results for this approach.

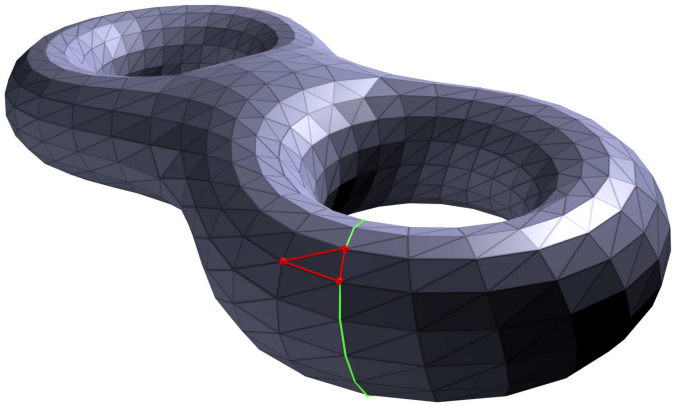

Figure 5.22: A cut through a mesh with genus 2 does not split the mesh in two, but only removes a handle.

Another limitation of this approach is that it cannot be extended to 3D or higher-dimensional meshes. The efficiency of our method for surface meshes comes from the fact that we found a very succinct way of coding the cut wires, given that they have essentially a *free* connectivity. In the $d$-D case, the cuts would be $(d-1)$-D meshes with arbitrary connectivity. In addition to the fact that coding the connectivity of meshes would require additional space, random accessibility would also be impaired. Instead of having a number of vertices on the order of $\sqrt{n}$, the cuts would contain $n^{(1-\frac{1}{d})}$ vertices, leading to a very large decompression overhead. Note that this problem is not specific to our algorithm, since nearly all random-accessible compression algorithms (with the exception of [Kim et al. 2006]) have to deal with boundaries.

Despite these limitations, our mesh compression method has some advantages apart from random accessibility. Because it uses an approach that totally departs from traditional methods, its efficiency

is not linked to the mesh regularity. However, for the same reason, it it very hard to find theoretical results concerning its bit rate.

There are still a lot of aspects that can be improved. For example, in our current implementation, a new wire is allocated each time we go down a node in the tree, and released when we walk up from the node. Decompression could be made drastically more efficient by *reusing* these structures (similarly to the streaming approaches) instead of constantly allocating and freeing them. An other necessary improvement would be to introduce an out-of-core compression scheme. Currently, only the *decompression* algorithm if efficient, but compression of huge meshes remains problematic. Also, it should not be too difficult to make the compression process run in parallel, since random accessibility inherently makes the compression of charts independent from each other. In addition, compression rates could be improved by using *local quantization* for wire geometry instead of first uniformly quantizing to model.

# General conclusion

**Large meshes:** In this dissertation, we have presented two approaches specifically targeted towards the compression of large meshes. On the one hand, we added a tool to the streaming arsenal. This tool compresses hexahedral meshes, where the existing tools could only deal with simplicial complexes (i.e. triangle and tetrahedral meshes). As with other streaming tools, this enables fast compression of huge meshes using commodity hardware. On the other hand, we proposed a random-accessible algorithm for cases where the stream processing is not adapted (e.g. visualization). The latter algorithm essentially emancipates the decoder from following the strict ordering imposed by the compressor. In contrast, in the streaming paradigm, both compressor and decompressor blindly follow the ordering of the mesh generation process.

**Geometry compression:** In addition, we presented a completely generic way of deriving efficient prediction weights for linear prediction, with tokens of optimality given some smoothness assumptions. These weights perform well in practice, and the formalism retroactively supports some previous *experimental* predictors. For example, we were able to theoretically back up the weights used by the *Freelence* approach [Kälberer et al. 2005]: We have shown that these weights were actually optimal if the input meshes were 2-smooth. In addition, we have derived new weights that can be used e.g. for progressive compression, that improve geometry compression ratios of about 9%

As evoked in Chapter 2, Taylor prediction is limited by the Runge phenomenon. Therefore, it is doubtable that making the assumption of higher order smoothness will bring further improvement of prediction weights. In practice, our experiments show that making the assumption of 2-smoothness is generally sufficient to determine efficient weights. Therefore we recommend to either use small stencils – to limit the number of interpolation weights – or limit the interpolation order to 2 and using another method such as minimizing the norm of the weights to further constrain the weights.

**Publications:** The three main contributions presented in this dissertation have been published – or will shortly appear – in international conferences and journals [Courbet and Hudelot 2009; Courbet and Isenburg 2010; Courbet and Hudelot].

**Implementation:** During the three years of my PhD, I spent a non-negligible time implementing the various ideas I had to confront them with experiments. For the evaluation of algorithms, I will now take into account another criterion: easiness of implementation. Designing generic compression algorithms can be hindered by the complexity of the implementation. For example, the source for streaming compression of hexahedral meshes has several thousands of lines (of C++ code) just for the hex compression part (i.e. without all the streaming mesh code). Dealing with mixed tet/hex meshes was not implemented – not because of theoretical problems, nor efficiency, but only because the implementation would have grown drastically in complexity. This would have been the case, too, for handling general polygon meshes. Therefore, I will now take into account this aspect when

rating an algorithm. In that respect, the random accessible algorithm presented in Chapter 5 handles gracefully meshes with mixed elements.

## 6.5 Future work

**Short term:**

**Out-of-core compression:** The major drawback of my current implementation of the hierarchical random access compressor is the fact that it is not out-of-core. As of now, the growing algorithm requires that the whole mesh be loaded in-core. However, the growing algorithm is essentially a region-growing algorithm where all processing occurs along two vertex fronts advancing towards the cut wire. Therefore, it is possible to use the out-of-core technique presented in [Isenburg and Gumhold 2003] to implement an out-of-core compressor, without having to modify the core of the method.

**Progressive random access:** Adrien Maglo of Ecole Centrale Paris is currently investigating a random-access compression scheme based on a chartification approach similar to that of Choe *et al.* [Choe et al. 2009], but compressing the charts using progressive compression [Lee et al. 2010a]. In order to address the compression of general polygon meshes, we are investigating a progressive polygon compression scheme. Preliminary results are encouraging (see Figure 6.23).

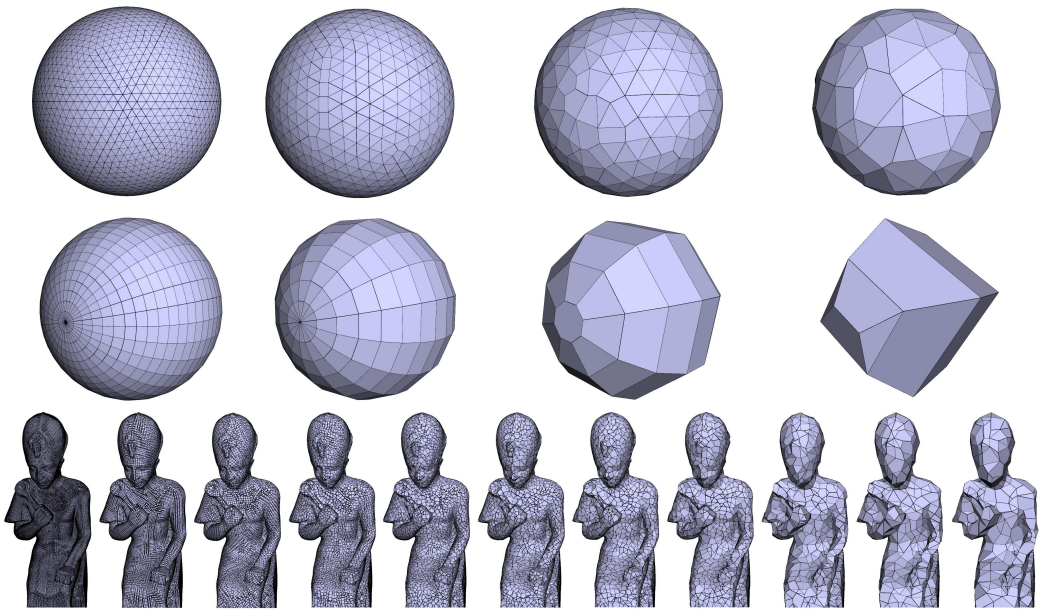

Figure 6.23: Preliminary results for progressive polygon compression.

**Longer term:**

**Beyond blockwise access:** Although quite efficient at decompressing arbitrary parts of a mesh, random-accessible approaches still have a fairly large overhead in terms of how much more of

the mesh has to be decompressed. Another (but somehow related) concern I have is the inelegance of random-accessible approaches. All methods use more or less a split-and-independently-compress-blocks paradigm, with explicit pointers stored in a header. This limits the number of blocks used for compression lest the header becomes too large, and therefore granularity. In the hierarchical approach presented in Chapter 5, I tried to depart as much as possible from the blockwise approach. I achieved a much finer granularity, and succeeded in interleaving random-accessible information and mesh data with limited overhead. However, for efficient coding the algorithm still relies on encoding a non-negligible amount of data in batches (here provided by wires). I wonder if it be possible to develop methods that are not based on a blockwise compression. In particular, there are no stochastic approaches to random accessible compression. In all random-accessible approaches, the decoder first figures out where to jump in memory from explicit memory pointers – either stored in the header in previous approaches, or interleaved with compressed data in our algorithm. One of the things I would like to investigate, that is yet a very vague idea, would be based on a probabilistic auto-correcting traversal of the compressed mesh representation, that does not need explicit pointers. The mesh would be stored in a spatially sorted way such that the compressor can know when it decodes a specific element whether the required element is placed before or after the current position in memory. Judging from the difference in position, the decompressor would make a new memory position estimate and make a new try. The main problem of this approach is ensuring that an element can be decoded unambiguously given any of the possible ways of reaching it via a sequence of guessed memory positions. The latter problem is similar in spirit to that found in [Kim et al. 2006] for geometry coding.

**Compression via learning:** In Chapter 3, we have briefly described a random-accessible method based on neural network modelling of the input mesh [Piperakis and Kumazawa 2001]. This approach is very interesting because it enables incredibly efficient geometric random access. Although the idea of representing a surface as a compressed implicit surface is very interesting, we think that the choice of classical neural networks is not very well adapted to mesh compression. Piperakis and Kumasawa use the sigmoid $f(x) = 1/[1 + exp(-\sigma x)]$ activation function, which is the most commonly used activation function for neuron networks. However, this function is not well localized in space, which is obviously not well adapted to the case of real-world objects, which have limited extents. Therefore, more compact functions as Radial Basis Function [Buhmann 2003] may be more adapted, either using neural networks or Support Vector Machines. We have begun conducting some experiments in this respect, and preliminary results indicate that these functions are well adapted to representing curves in 2D. We still have to extend these results in 3D and compare them with the approach of Piperakis and Kumasawa. However, even if this approach is efficient in terms of compression rates and random access, it has the drawback that the surface is not given in the usual mesh representation that most computer graphics algorithms natively use. Converting from implicit surface to mesh representation is too expensive to be a credible solution. Therefore, if this approach is to be used, algorithms to efficiently display the resulting compressed implicit surface remain to be developped. These algorithms would ideally run on GPU hardware for fast rendering.

**Volume meshes:** As noted before, none of the existing approaches – including ours – can be generalized to volume meshes. Indeed, as boundaries are always shared between blocks, the data must be either coded independently at a higher level – as chosen by most algorithms – or simply duplicated – this is the solution of CHuMI viewer [Jamin et al. 2009]. Doing things this way is not possible in dimension $d$ higher than 2, since the boundaries have size $n^{(1-\frac{1}{d})}$. Therefore, handling boundaries this way would result in a very high overhead in decompression time and bit rate. Random access for volume meshes remains an open problem.

**Evaluation:** Another important issue raised in Chapters 3 and 5 is the evaluation of random-accessible approaches. There is currently no *objective* criterion to compare methods. To evaluate our approach, we have computed the *complexity* of decoding a part of the mesh, and the associated *overhead* in terms of how much more of the mesh is decoded when requesting a given part. Another important – and somehow related – criterion is random access *granularity*. Algorithms that typically decode parts that are small compared to the size of the mesh need higher granularity if overhead is not to become the largest computational load. All compression algorithms trade granularity for bit rate efficiency, so the latter criterion is of lesser importance when evaluating random-accessible methods than when comparing traditional compression algorithms. Because different applications have very different memory access patterns, it is doubtable whether it will be possible to come up with an objective evaluation scheme fitting a large panel of applications. Even in the limited scope of visualization applications, it is hard to evaluate the overhead of a method. When visualization targets a part of a mesh, decoding other parts of the mesh may be considered overhead by some applications; however in the case of progressive approaches, this overhead includes the coarser approximation that helps navigating within the model. In this respect, part of the overhead is actually useful and must not be regarded as being irrelevant.

# Bibliography


Aleardi, Devillers, and Schaeffer. Succinct representation of triangulations with a boundary. In *Workshop on Algorithms and Data Structures*, volume 3608, pages 134 – 145, 2005. 124

Alliez and Desbrun. Progressive encoding for lossless transmission of triangle meshes. In *SIGGRAPH*, volume 28, pages 198–205, 2001a. 38, 56, 57

Alliez and Desbrun. Valence-driven connectivity encoding of 3d meshes. *Computer Graphics Forum*, 20:480–489, 2001b. 36, 56

Alliez and Gotsman. *Recent Advances in Compression of 3D Meshes*, pages 3–26. Springer-Verlag, 2005. 38

Alliez, Meyer, and Desbrun. Interactive geometry remeshing. *ACM Transaction on Graphics*, 21(3): 347–354, 2002. 50

Alliez, Colin de Verdière, Devillers, and Isenburg. Centroidal voronoi diagrams for isotropic surface remeshing. *Graphical Models*, 67:204–231, 2005. 50

Alliez, Attene, Gotsman, and Ucelli. *Recent Advances in Remeshing of Surfaces*. Springer, 2007. 47

Attene, Falcidieno, Spagnuolo, and Rossignac. Swingwrapper: Retiling triangle meshes for better edgebreaker compression. *ACM Transaction on Graphics*, 22(4):982–996, 2003. 50, 51, 52

Bajaj, Pascucci, and Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In *Data Compression Conference*, 1999a. 32

Bajaj, Pascucci, and Zhuang. Progressive compression and transmission of arbitrary triangular meshes. In *IEEE Visualization*, pages 307–316, 1999b. 38

Ben-Chen and Gotsman. On the optimality of spectral compression of mesh data. *ACM Transaction on Graphics*, 24(1):60–80, 2005. 40, 70

Benzley, Perry, Merkley, Clark, and Sjaardema. A comparison of all-hexahedral and all-tetrahedral finite element meshes for elastic and elasto-plastic analysis. In *International Meshing Roundtable*, 1995. 103

Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2:1–13, 1997. 93

Bernardini, Mittleman, Rushmeier, Silva, and Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5:349–359, 1999. 90

Julien Berta. Integrating vr and cad. *IEEE Computer Graphics and Applications*, 19:14–19, 1999. 17

Blacker. The cooper tool. In *International Meshing Roundtable*, pages 13–29, 1996. 103, 105

Buhmann. *Radial Basis Functions: Theory and Implementations*. Cambridge University Press, 2003. 149

Campagna, Kobbelt, and Seidel. Directed edges - a scalable representation for triangle meshes. *ACM Journal of Graphics Tools*, 3:21–40, 1998. 85, 93

Chaine, Gandoin, and Roudet. Mesh connectivity compression using convection reconstruction. In *Symposium on Solid and Physical Modeling*, pages 41–49, 2007. 41, 44

Chen, Chiang, Memon, and Wu. Optimized prediction for geometry compression of triangle meshes. In *Data Compression Conference*, 2005. 56

Choe, Kim, Lee, Lee, and Seidel. Mesh compression with random accessibility. In *Israel-Korea Binational Conference on Geometric Modeling and Computer Graphics*, 2004. 92, 94, 96, 97, 100, 140, 143

Choe, Kim, Lee, and Seidel. Random accessible mesh compression using mesh chartification. In *IEEE Transactions on Visualization and Computer Graphics*, 2009. 85, 94, 95, 96, 100, 101, 124, 125, 135, 140, 143, 148

Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19:547 – 554, 1976. 138

Cohen-Or, Levin, and Remez. Progressive compression of arbitrary triangular meshes. In *IEEE Visualization*, pages 67–72, 1999. 38

Cohen-or, Cohen, and Irony. Multi-way geometry encoding. Technical report, 2002. 56, 57, 62, 73

Courbet and Hudelot. Taylor prediction for mesh geometry compression. To appear in Computer Graphics Forum. 147

Courbet and Hudelot. Random accessible hierarchical mesh compression for interactive visualization. In *Eurographics Symposium on Geometry Processing*, pages 1311–1318, 2009. 147

Courbet and Isenburg. Streaming compression of hexahedral meshes. *The Visual Computer (proceedings of Computer Graphics International)*, 26:1113–1122, 2010. 147

Deering. Geometry compression. In *SIGGRAPH*, pages 13–20, 1995. 32

Dey. *Delaunay mesh generation of three dimensional domains*. Springer-Verlag, 2009. 105

Du, Jaromersky, Chiang, and Memon. Out-of-core progressive lossless compression and selective decompression of large triangle meshes. In *Data Compression Conference*, pages 420–429, 2009. 85, 98, 99, 100, 101, 143

Dyn, Levin, and Gregory. A four-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design*, 4:257 – 268, 1987. 78, 81

Dyn, Levin, and Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transaction on graphics*, 9:160–169, 1990. 50, 59, 79

Fabbrini and Montani. Autumnal quadtrees. *Computer Journal*, 29:472–474, 1986. 136

Friedel, Schröder, and Khodakovsky. Variational normal meshes. *ACM Transaction on Graphics*, 23 (4):1061–1073, 2004. 50

Gandoin and Devillers. Progressive lossless compression of arbitrary simplicial complexes. *ACM Transaction on graphics (proceedings of SIGGRAPH)*, 21:372–379, 2002. 41, 42, 43, 45, 55, 98, 100

Garcia-Palacios, Hoffman, Carlin, Furness, and Botella. Virtual reality in the treatment of spider phobia: a controlled study. *Behaviour Research and Therapy*, 40:983–993, 2002. 17

Gotsman. On the optimality of valence-based connectivity coding. *Computer Graphics Forum*, 22: 99–102, 2003. 36

Gottschalk, Lin, and Manocha. Obb-tree: A hierarchical structure for rapid interference detection. In *SIGGGRAPH*, pages 171–180, 1996. 93

Gu, Gortler, and Hoppe. Geometry images. In *SIGGRAPH*, pages 355–361, 2002. 47, 51

Gueziec, Taubin, Lazarus, and Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In *IEEE Visualization*, 1998. 41

Gueziec, Bossen, Taubin, and Silva. Efficient compression of nonmanifold polygonal meshes. In *IEEE Visualization*, 1999. 41

Guibas and Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4:74–123, 1985. 93

Gumhold and Amjoun. Higher order prediction for geometry compression. *Shape Modeling International*, 2003. 55

Gumhold and Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH*, pages 133–140, 1998. 34, 35

Gumhold, Karni, Isenburg, and Seidel. Predictive point-cloud compression. In *SIGGRAPH*, 2005. 41, 52

Guskov, Vidimče, Sweldens, and Schröder. Normal meshes. In *SIGGRAPH*, pages 95–102, 2000. 50, 51

Ho, Lee, and Kriegman. Compressing large polygonal models. In *IEEE Visualization*, 2001. 85, 86, 88, 94

Hopf and Ertl. Hierarchical splatting of scattered data. In *IEEE Visualization*, 2003. 41

Hoppe. Progressive meshes. In *SIGGRAPH*, pages 99–108, 1996. 38

Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH*, 1997. 98

Huang, Peng, Kuo, and Gopi. Octree-based progressive geometry coding of point clouds. In *Point-Based Graphics*, 2006. 41, 43

Huang, Peng, Kuo, and Gopi. A generic scheme for progressive point cloud coding. *IEEE Transactions on Visualization and Computer Graphics*, 14:440–453, 2008. 41, 43

Hubbard. Interactive collision detection. In *Symposium on Research Frontiers in Virtual Reality*, pages 24–31, 1993. 93

Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*, pages 1098–1102, 1952. 31

Ibarria, Lindstrom, and Rossignac. Spectral interpolation on 3x3 stencils for prediction and compression. *Journal of Computers*, 2007. 37, 55, 59, 60, 61, 77, 78, 83, 115

Ihm and Park. Wavelet-based 3d compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18:3–15, 1999. 93

Isenburg. Compressing polygon mesh connectivity with degree duality prediction. In *Graphics Interface*, 2002. 37, 88

Isenburg. Compression and streaming of polygon meshes, 2005. Ph.D. Thesis, Chapell Hill. 52

Isenburg and Alliez. Compressing hexahedral volume meshes. In *Pacific Graphics*, pages 284–293, 2002a. 37, 104, 105, 109, 112, 114, 120, 121

Isenburg and Alliez. Compressing polygon mesh geometry with parallelogram prediction. In *IEEE Visualization*, 2002b. 57

Isenburg and Gumhold. Out-of-core compression for gigantic polygon meshes. In *SIGGRAPH*, 2003. 41, 85, 88, 89, 148

Isenburg and Lindstrom. Streaming meshes. In *IEEE Visualization*, 2005. 24, 85, 88, 90, 91, 105

Isenburg and Snoeyink. Face fixer: Compressing polygon meshes with properties. In *SIGGRAPH*, pages 263–270, 2000. 34

Isenburg and Snoeyink. Graph coding and connectivity compression, 2004. draft: http://www.cs.unc.edu/ isenburg/research/papers/is-gccc-04.pdf. 33

Isenburg, Ivrissimtzis, Gumhold, and Seidel. Geometry prediction for high degree polygons. In *Spring Conference on Computer Graphics*, 2005a. 57, 58, 59, 60, 68, 76, 77, 78, 83, 94

Isenburg, Lindstrom, and Snoeyink. Streaming compression of tetrahedral volume meshes. In *Eurographics Symposium on Geometry Processing*, 2005b. 85, 88, 91, 92, 95, 96, 106, 109

Isenburg, Lindstrom, Gumhold, and Shewchuk. Streaming compression of tetrahedral volume meshes. In *Graphics Interface*, 2006a. 85, 92, 105, 106

Isenburg, Liu, Shewchuk, and Snoeyink. Streaming computation of delaunay triangulations. In *SIGGRAPH*, pages 1049–1056, 2006b. 20, 90

Ivrissimtzis, Rössl, and Seidel. A divide and conquer algorithm for triangle mesh connectivity encoding. In *Pacific Graphics*, page 294, 2002. 34, 124

Jacobson. Succinct static data structures, 1989. PhD Thesis, Carnegie-Mellon. 135

Jamin, Gandoin, and Akkouche. Chumi viewer: Compressive huge mesh interactive viewer. *Computers & Graphics*, 33(4):542–553, 2009. 85, 98, 100, 101, 143, 149

Ju, Losasso, Schaefer, and Warren. Dual contouring of hermite data. In *SIGGRAPH*, pages 339–346, 2002. 90

Kälberer, Polthier, Reitebuch, and Wardetzky. Freelence: Coding with free valences. In *Eurographics*, 2005. 34, 56, 57, 73, 75, 95, 147

Karni and Gotsman. Spectral compression of mesh geometry. In *SIGGRAPH*, pages 279–286, 2000. 39, 40, 55

Karni and Gotsman. 3d mesh compression using fixed spectral bases. In *Graphics interface*, pages 1–8, 2001. 40

Karypis and Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998. 86

Keeler and Westbrook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 14:239–252, 1995. 34

Kettner. Using generic programming for designing a data structure for polyhedral surfaces. In *Symposium on Computational Geometry*, pages 3–36, 1998. 93

Khodakovsky and Guskov. *Compression of Normal Meshes*. Springer-Verlag, 2003. 50, 52, 60

Khodakovsky, Schröder, and Sweldens. Progressive geometry compression. In *SIGGRAPH*, 2000. 50, 51, 52, 55

Khodakovsky, Alliez, Desbrun, and Schröder. Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models*, 64:147–168, 2002. 37, 94

Kim and Lee. Truly selective refinement of progressive meshes. In *Graphics interface*, 2001. 97, 98

Kim, Choe, and Lee. Multiresolution random accessible mesh compression. In *EUROGRAPHICS*, 2006. 85, 97, 100, 101, 144, 149

Kim, Moon, Kim, and Yoon. RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):273–286, 2010. 85, 97, 101, 143

King and Rossignac. Guaranteed 3.67v bit encoding of planar triangle graphs. In *Canadian Conference on Computational*, 1999a. 34

King and Rossignac. Connectivity compression for irregular quadrilateral meshes. Technical Report GIT-GVU-99-36, GVU, 1999b. 34

Klosowski, Held, Mitchell, Sowizral, and Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *Transactions on Visualization and Computer Graphics*, 4:21–36, 1998. 93

Kobbelt. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. *Computer Graphics Forum*, pages 409–420, 1996. 59, 78, 81

Kobbelt. $\sqrt{3}$ subdivision. In *SIGGRAPH*, 2000. 59, 81

Krivograd, Trlep, and Zalik. A hexahedral mesh connectivity compression with vertex degrees. *Computer-Aided Design*, 2008. 37, 104, 109, 112, 114, 120

Kronrod and Gotsman. Optimized compression of triangle mesh geometry using prediction trees. In *Symposium on 3D Data Processing, Visualization and Transmission*, 2002. 56

Labsik and Greiner. Interpolatory $\sqrt{3}$ subdivision. In *Eurographics*, 2000. 50, 59, 78, 79, 82

Lee. Thesis proposal on 3d mesh single-rate compression, 2002. http://www-scf.usc.edu/leeh/qual/qual.pdf. 55

Lee, Alliez, and Desbrun. Angle-analyzer: A triangle-quad mesh codec. In *Eurographics*, pages 383–392, 2002. 56, 95, 140, 143

Lee, Lavoue, and Dupont. New methods for progressive compression of colored 3d mesh. In *WSCG*, 2010a. URL http://liris.cnrs.fr/publis/?id=4548. 38, 56, 148

Lee, Lavoue, and Dupont. Adaptive coarse-to-fine quantization for optimizing rate-distortion of progressive mesh compression. In *Vision, Modeling, and Visualization Workshop*, 2010b. 38

Lefebvre and Hoppe. Compressed random-access trees for spatially coherent data. In *Eurographics Symposium on Rendering*. Eurographics, 2007. URL http://www-sop.inria.fr/reves/Basilic/2007/LH07. 93

Lewiner, Craizer, Lopes, Pesco, Velho, and Medeiros. Gencode: Geometry-driven compression in arbitrary dimension and co-dimension. In *SIBGRAPI*, page 249, 2005. 41, 43, 44

Li and Kuo. A dual graph approach to 3d triangular mesh compression. In *ICIP*, pages 891–894, 1998a. 34, 35

Li and Kuo. Progressive coding of 3-d graphic models. In *IEEE Visualization*, volume 86, pages 1052–1063, 1998b. 38

Lin and Manocha. *Collision and proximity queries.* 2003. 93

Lindstrom and Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 2006a. 112

Lindstrom and Isenburg. Lossless compression of predicted floating-point data. *lookup the journal on my webpage*, 2006b. 39

Lindstrom and Isenburg. Lossless compression of hexahedral meshes. In *IEEE Data Compression Conference*, 2008. 31, 32, 33, 37, 92, 104, 105, 109, 112, 114, 117, 121

Loop, 1987. Smooth Subdivision Surfaces Based on Triangles, Master's thesis, University of Utah, Department of Mathematics. 50, 59, 78, 81

Lopes, Rossignac, Safonova, Szymczak, and Tavares. Edgebreaker: A simple compression algorithm for surfaces with handles. *Computers & Graphics*, 27:553–567, 2003. 34

Lorensen and Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH*, pages 163–169, 1987. 90

Losasso, Hoppe, Schaefer, and Warren. Smooth geometry images. In *Symposium on Geometry Processing*, pages 138–145, 2003. 49

Mamou, Zaharia, and Preteux. A triangle-fan-based approach for low complexity 3d mesh compression. In *ICIP09*, pages 3513–3516, 2009. 33, 37

Marais and Gain. Distance ranked connectivity compression of triangle meshes. *Computer Graphics Forum*, 26:813 – 823, 2007. 41, 44, 55

Mascarenhas, Isenburg, Pascucci, and Snoeyink. Encoding volumetric grids for streaming isosurface extraction. In *3DPVT*, pages 665–672, 2004. 90

Moffat and Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45:1200–1027, 1997. 31

Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd, 1966. 25

Muller-Hannemann. Shelling hexahedral complexes for mesh generation. *Journal of Graph Algorithms and Applications*, 2001. 103, 105

Pajarola and Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualisation and Computer Graphics*, 6(1):79–93, 2000a. 38, 57, 62, 75, 76

Pajarola and Rossignac. Squeeze: fast and progressive decompression of triangle meshes. In *Computer Graphics International*, pages 173–182, 2000b. 38

Payan and Antonini. An efficient bit allocation for compressing normal meshes with an error-driven quantization. *Computer Aided Geometric Design*, 22(5):466–486, 2005. 50

Peng and Kuo. Geometry-guided progressive lossless 3d mesh coding with octree (ot) decomposition. *ACM Transaction on graphics*, 2005. 41, 45, 46

Peng, Kim, and Kuo. Technologies for 3d mesh compression : A survey. *Journal of Visual Communication and Image Representation*, 16:688–733, 2005. 38

Piperakis and Kumazawa. 3d polygon mesh compression with multi layer feed forward neural networks. *Systemics, Cybernetics and Informatics*, 1(3), 2001. 94, 101, 149

Popovic and Hoppe. Progressive simplicial complexes. In *SIGGRAPH*, pages 217–224, 1997. 38

Poulalhon and Schaeffer. Optimal coding and sampling of triangulations. In *International colloquium on automata, languages and programming*, 2003. 32

Prat, Gioia, Bertrand, and Meneveaux. Connectivity compression in an arbitrary dimension. *The Visual Computer*, 2005. 105

Praun and Hoppe. Spherical parametrization and remeshing. *ACM Transactions on Graphics*, 22(3): 340–349, 2003. 47

Reif. A unified approach to subdivision algorithms near extraordinary vertices. *Computer Aided Geometric Design*, 12:153–174, 1995. 60, 81

Rissanen and Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162, 1979. 31

Rodler. *Compression with Fast Random Access*. PhD thesis, 2001. 93

Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5:47–61, 1999. 34, 35, 124

Rossignac and Szymczak. Wrap & zip decompression of the connectivity of triangle meshes compressed with edgebreaker. *Journal of Computational Geometry, Theory and Applications*, 14:119–135, 1999. 34

Rubin and Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14:110–116, 1980. 93

Runge. Über empirische funktionen und die interpolation zwischen öquidistanten ordinaten. *Zeitschrift für Mathematik und Physik*, 46:224–243, 1901. 83

Rusinkiewicz and Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH*, 2000. 41

Sander, Gortler, Snyder, and Hoppe. Signal-specialized parametrization. In *Eurographics workshop on Rendering*, pages 87–98, 2002. 47

Sander, Wood, Gortler, Snyder, and Hoppe. Multi-chart geometry images. In *Symposium on Geometry Processing*, pages 146–155, 2003. 47

Schnabel and Klein. Octree-based point-cloud compression. In *Point-Based Graphics*, 2006. 41, 43

Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948. 30

Silva and Mitchell. Greedy cuts: An advancing front terrain triangulation algorithm. In *Symposium on Geographic Information Systems*, pages 137–144, 1998. 90

Sim, Kim, and Lee. An efficient 3d mesh compression technique based on triangle fan structure. *Signal processing Image communication*, 18:17–32, 2003. 56, 57, 62, 73

Sorkine, Cohen-Or, and Toldeo. High-pass quantization for mesh encoding. In *Symposium on Geometry Processing*, 2003. 39

Staten, Owen, and Blacker. Unconstrained paving and plastering: A new idea for all hexahedral mesh generation. In *International Meshing Roundtable*, 2005. 103, 105

Strang and Nguyen. *Wavelets and filter banks*. Wellesley – Cambridge Press, 1996. 49

Surazhsky and Gotsman. Explicit surface remeshing. In *Symposium on Geometry processing*, pages 20–30, 2003. 50

Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM Journal on Mathematical Analysis*, 29(2):511–546, 1998. 49

Szymczak and Rossignac. Grow & fold: Compression of tetrahedral meshes. In *ACM Symposium on Solid Modeling*, 1999. 34, 35

Szymczak, Rossignac, and King. Piecewise regular meshes: construction and compression. *Graphical Models*, 64(3-4):183–198, 2002. 50

Taubin and Rossignac. Geometric compression through topological surgery. *ACM transactions on Graphics*, 17:84–115, 1998. 32

Taubin, Gueziec, Horn, and Lazarus. Progressive forest split compression. In *SIGGRAPH*, pages 123–132, 1998. 38

Teschner, Kimmerle, Heidelberger, Zachmann, Raghupathi, Fuhrmann, Cani, Faure, Magnenat-Thalmann, Strasser, and Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 19:61–81, 2005. 93

Touma and Gotsman. Triangle mesh compression. In *Graphics Interface*, 1998. 35, 36, 56, 57, 86, 88

Turan. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984. 32, 33

Tutte. A census of planar triangulations. *Canadian Journal of Mathematics*, 14:21–38, 1962. 32

Tutte. A census of planar maps. *Canadian Journal of Mathematics*, 15:249–271, 1963. 32

Valette and Prost. Wavelet-based multiresolution analysis of irregular surface meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2004. 38, 46

Valette, Chaine, and Prost. Progressive lossless mesh compression via incremental parametric refinement. In *Symposium on Geometry Processing*, pages 1301–1310, 2009. 45, 46

Warren. Subdivision methos for geometric design, 1995. Unpublished manuscript. 59

Weiler. Edge-based data structures for solid modeling in a curved surface environment. *IEEE Computer Graphics and Applications*, pages 21–40, 1985. 93

Weiler. The radial-edge structure: A topological representation for non-manifold geometric boundary representations. In *Geometric Modelling for CAD Applications*, pages 3–36, 1988. 93

Weinberger, Seroussi, and Sapiro. The loco-i lossless image compression algorithm: Principles and standardization into jpeg-ls. *IEEE Transactions on Image Processing*, 9:1309–1324, 2000. 59

Witten, Neal, and Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30:520–540, 1987. 31

Xia and Varshney. Dynamic viewdependent simplification for polygonal models. In *IEEE Visualization*, pages 327–334, 1996. 98

Yao and Lee. Adaptive geometry image. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):948–960, 2008. 47

Yoon and Lindstrom. Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2007. 12, 85, 93, 95, 96, 97, 143

Zorin, Schröder, and Sweldens. Interpolating subdivision for meshes with arbitrary topology. In *SIGGRAPH*, 1996a. 49, 50, 59, 78, 79, 81, 82

Zorin, Schröder, and Sweldens. Interpolating subdivision for meshes with arbitrary topology. Technical Report CS-TR-96-06, GVU, 1996b. 59, 60

# Appendix

## 7.6 Analysis of the small support $\sqrt{3}$ interpolating subdivision scheme.

In the following, we prove that the $\sqrt{3}$ interpolatory subdivision scheme that we proposed in Section 2.3.5.2 has $C^1$ limit surfaces. Around extraordinary points, we use the same weights as Labsik and Greiner, therefore the scheme is also $C^1$ around extraordinary points. Therefore, we only need to consider ordinary points.

Using the vertex numbering of Figure 7.24, the subdivision matrix is:

$$S = \frac{1}{9}\begin{bmatrix} 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 4 & -1 & 4 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 4 & 0 & -1 & 4 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 4 & 0 & 0 & -1 & 4 & 4 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 4 & -1 & 0 & 0 & -1 & 4 & 4 & 0 & 0 & 0 & 0 & -1 & 0 \\ 4 & 4 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.5}$$

The neighbourhood is 2-step invariant, meaning that the set of vertices in *blue* on Figure 7.24 is mapped to the set of vertices in *red* in two subdivision steps, with a rotation of $\frac{\pi}{6}$, by the matrix $\tilde{S} = RS^2$, where:

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{7.6}$$

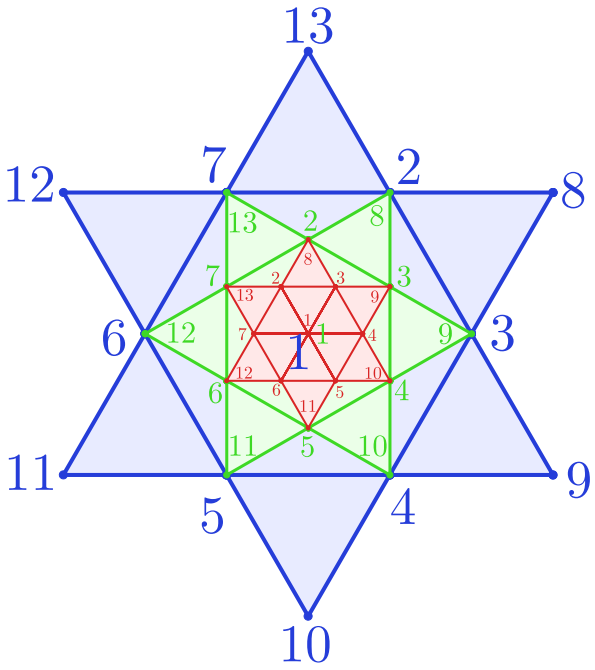


Figure 7.24: 2-step invariant neighbourhood of a regular vertex for the reduced-support interpolating $\sqrt{3}$ subdivision scheme.

The eigenvalues of $\tilde{S}$ are :

$$\left[1, \frac{1}{3}, \frac{1}{3}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, -\frac{1}{9}, -\frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{27}, \frac{1}{27}\right] \tag{7.7}$$

They verify the sufficient condition for $C^1$ smoothness ($\lambda_1 = 1 > |\lambda_2| > |\lambda_4|$, $\lambda_2 = \lambda_3$), and therefore the limit surface is $C^1$ around ordinary points.