3D compression: from A to Zip: a first complete example

THOMAS LEWINER

Department of Mathematics — Pontifícia Universidade Católica — Rio de Janeiro — Brazil
http://www.mat.puc-rio.br/~tomlew/.

Abstract. Images invaded most of contemporary publications and communications. This expansion has accelerated with the development of efficient schemes dedicated to image compression. Nowadays, the image creation process relies on multidimensional objects generated from computer aided design, physical simulations, data representation or optimisation problem solutions. This variety of sources motivates the design of compression schemes adapted to specific class of models.

The recent launch of Google Sketch'up and its 3D models warehouse has accelerated the shift from two-dimensional images to three-dimensional ones. However, these kind of systems require fast access to eventually huge models, which is possible only through the use of efficient compression schemes.

This work is part of a tutorial given at the XX^{th} Brazilian Symposium on Computer Graphics and Image Processing (Sibgrapi 2007).

Keywords: ***Mesh Compression. Tutorial.***

Figure 1: *3D Compression: fitting 3D models into a small storage.*

1 Introduction

Images surpassed the simple function of illustrations. In particular, artificial and digital images invaded most of published works, from commercial identification to scientific explanation, together with the specific graphics industry. Technical advances created supports, formats and transmission protocols for these images, and these contributed to this expansion. Among these, high quality formats requiring low resources appeared with the development of generic, and then specific, compression schemes for images. More recently drew on the sustained trend to incorporate the third dimension into images, and this motivates orienting the developments of compression towards higher dimensional images.

There exists a wide variety of images, from photographic material to drawings and artificial pictures. Similarly, higher dimensional models are produced from many sources: The graphics industry designers draw three–dimensional objects by their contouring surface, using geometric primitives. The recent developments of radiology make intense use of three–dimensional images of the human body, and extract isosurfaces to represent organs and tissues. Geographic and geologic models of terrain and underground consist in surfaces in the multi–dimensional of physical measures. Engineering usually generate finite elements solid meshes in similar multi–dimensional spaces to support physical simulations, while reverse engineering, archæological heritage preservation and commercial marketing reconstruct real objects from points.

Compression methods for three–dimensional models appeared mainly in the mid 1990's with [16] and developed quickly since then. This evolution turned out to be a technical necessity, since the size and complexity of the typical models used in practical applications increases rapidly. The most performing practical strategies for surfaces are based on the Edgebreaker of [57] and the valence coding of [66]. These are classified as connectivity–driven mesh compression, since the proximity of triangles guides the sequence of the surface vertices to be encoded. More recently, dual approaches proposed to guide the encoding of the triangle proximity by the geometry, such as done in [20].

Actually, the diversity of images requires this multiplicity of compression programs, since specific algorithms usually perform better than generic one (such as the popular Zip method), if they are well adapted. This tutorial aims at introducing the basic concepts of compression with examples on 3D models compression for an audience without prior knowledge in modeling or compression. The examples and algorithms are chosen from works published in the Sibgrapi.

Preprint MAT. 13/07, communicated on August 25^{th}, 2007 to the Department of Mathematics, Pontifícia Universidade Católica — Rio de Janeiro, Brazil. The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.

2 Information Representation

We would like first to briefly introduce what we mean by compression, in particular the relation of the abstract tool of information theory [53, 26, 61], the asymptotic entropy of codes [61, 29, 67] and the practical performance of coding algorithms [29, 56, 39, 51]. We will then focus on the arithmetic coder [56, 51] since we will mainly use it in practise. This coder can be enhanced by taking in account deterministic or statistic information of the object to encode, which translates technically by a shift from Shannon's entropy [61] to Kolmogorov complexity [43]. Finally, we will describe how this coding takes part in a compression scheme. General references on data compression can be found in [60].

(a) Coding

Source and codes. Coding refers to a simple translation process that converts symbols from one set, called the *source* to another, this last one being called the set of *codes*. The conversion can then be applied in a reverse way, in order to recover the original sequence of symbols, called *message*. The purpose is to represent any message of the source into a more convenient way, typically a way adapted to a specific transmission channel. This coding can intend to reduce the size of the message [60], for example for compression applications, or on the contrary increase its redundancy to be able to detect transmission errors [25].

Enumeration. A simple example coder would rely on enumerating all the possible messages, indexing them from 1 to n during the enumeration. The coder would then simply assign one code for each message. In practise, the number of possibilities is huge and difficult to enumerate, and it is hard to recover the original message from its index without enumerating again all the possible messages. However, this can work for specific cases [10]. These enumerative coders give a reference for comparing performance of coders. However, in practical cases, we would like the coding to be more efficient for the most frequent messages, even if the performance is altered for less frequent ones. This reference will thus not be our main target.

Coder performance. Two different encodings of the same source will in general generate two coded messages of different sizes. If we intend to reduce the size of the message, we will prefer the coder that generated the smallest message. On a specific example, this can be directly measured. Moreover, for the enumerative coder, the performance is simply the logarithm of the number of elements, since a number n can be represented by $\log(n)$ digits. However, this performance is hard to measure it for all the possible messages of a given application. [53], [26] and [61] introduced a general tool to measure the asymptotic, theoretic performance of a code, called the *entropy*.

(b) Information Theory

Entropy. The entropy is defined in general for a random message, which entails message generators as symbol sources or encoders, or in particular to a specific message (an observation) when the probabilities of its symbols are defined. If a random message $\mathbf{m}$ of the code is composed of n symbols $\mathbf{s}_1 \dots \mathbf{s}_n$, with probability $p_1 \dots p_n$ respectively, then its entropy $\mathfrak{h}(\mathbf{m})$ is defined by $\mathfrak{h}(\mathbf{m}) = \sum_i -p_i \log(p_i)$.

As referred in [61], this definition is natural for telecommunication systems, but it is only one possible measure that respects the following criteria:

1. $\mathfrak{h}()$ should be continuous in the p_i.
2. If all p_i are equal, $p_i = \frac{1}{n}$, then $\mathfrak{h}()$ should increase with n, since there are more possible messages.
3. If the random message $\mathbf{m}$ be broken down into two successive messages $\mathbf{m}_1$ and $\mathbf{m}_2$, then $\mathfrak{h}(\mathbf{m})$ should be the weighted sum of $\mathfrak{h}(\mathbf{m}_1)$ and $\mathfrak{h}(\mathbf{m}_2)$.

Huffman coder. [29] introduced a simple and efficient coder that writes each symbol of the source with a code of variable size. For example, consider that a digital image is represented by a sequence of colours $\mathbf{s}_{black}, \mathbf{s}_{red}, \mathbf{s}_{darkblue}, \mathbf{s}_{lightblue}, \mathbf{s}_{white}$. A simple coder will assign a symbol to each colour, and encode the image as the sequence of colours. This kind of coder will be called next an *order 0 coder*.

If the image is a photo of a seascape, as the one of Figure 2,

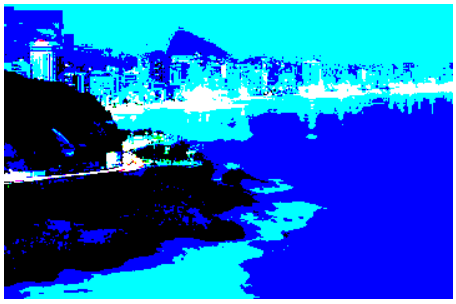


Figure 2: *Huffman coding relies on the frequency of symbols of a message, here the colours inside an image.*

the probability to have blue colours in the message will be higher than for red colours. Huffman proposed a simple way to encode with less bits the more frequent colours, here blue ones, and with more bits the less frequent symbols. Consider that each of the colour probabilities is a power of 2: $p_{black} = 2^{-3}, p_{red} = 2^{-4}, p_{darkblue} = 2^{-1}, p_{lightblue} = 2^{-2}, p_{white} = 2^{-4}$.

These probabilities can be represented by a binary tree, such as each symbol of probability 2^{-b} is a leaf of depth b in the binary tree. Then each symbol is encoded by the left (0) and right (1) choices to get from the root of the tree to that symbol. The decoding is then performed by following the left and right codes until reaching a leaf, and the symbol of that leaf is a new element of the decoded message. In that context, the probability of each left and right operation is $\frac{1}{2}$, which maximises the entropy ($\mathfrak{h}(\mathbf{m}) = 1$), i.e., the theoretical performance.

Entropy coder. The original Huffman code also worked out for general probabilities, but without maximising the entropy. It uses a greedy algorithm to choose how to round off the probabilities towards powers of 2 [29]. However, Shannon proved that it is asymptotically possible to find a coder of maximum entropy [61], and that no other coder can asymptotically work better in general. This is the main theoretical justification for the definition of $\mathfrak{h}()$. [29] introduced a simpler proof of that theorem, by grouping sequence of symbols until their probability become small enough to be well approximated by a power of 2.

(c) Levels of Information

In practise, although the entropy of a given coder can be computed, the theoretical entropy of a source is very hard to seize. The symbols of the source are generally not independent, since they represent global information. In the case of dependent symbols, the entropy would be better computed through the Kolmogorov complexity [43]. For example, by increasing the contrast of an image, as human we believe that we loose some of its details, but from the information theory point of view, we added a (mostly) random value to the colours, therefore increasing the information of the image.

An explanation for that phenomenon is that the representation of an image as a sequence of colours is not significant to us. This sequence could be shuffled in a deterministic way, it would not change the coding, but we would not recognise anymore the information of the image. In order to design and evaluate an efficient coding system, we need to represent the **exact** amount of information that is needed for our application, through an **independent** set of codes. If we achieve such a coding, then its entropy can be maximised through a universal coder, such as the Huffman coder or the *arithmetic coder*.

3 Arithmetic Coding

The *arithmetic coding* [56, 51] encodes the symbols source with code sizes very close to their probabilities. In particular, it achieves an average size of codes that can be a fraction of bits. Moreover, it can encode simultaneously different sources, and provide a flexible way of adapting probabilities of the source symbols. This adaptation can be monitored through rules depending on a *context*, or automatically looking at the previous encoded symbols by varying the *order* of the coder. This section details these points, and provides some sample code inspired from [5]. In particular, we would like to detail why the arithmetic coder is usually presented as a universal solution, and how much parameters are hidden behind this universal behaviour. This coder is widely used, and we will detail for each algorithm the hidden parameters which have a significant impact on the final compression ratio.

(a) Arithmetic Coder

Instead of assigning a code for each of the source symbol, an arithmetic coder represent the whole message by a single binary number $\mathbf{m} \in [0,1[$, with a large number of digits. Where Huffman decoder read a sequence of left/right codes to rebuild the source message, the arithmetic coder will read a sequence of interval shrinking, until finding a small enough interval containing $\mathbf{m}$. At each step, the interval is not shrunk by splitting it in two as in Huffman coding, but each possible symbol of the source is assigned a part of the interval proportional to its probability, and the interval is shrunk to the part of the source symbol. Therefore, a splitting in two corresponds in general to several shrinking, and thus a single bit can encode many symbols, as the next table details on the beginning of the example of Figure 2, using the above probabilities.

Example. We will first illustrate the arithmetic coding with our previous example of Figure 2, reduced to Figure 3: compressing an image by directly encoding its colours: $\mathbf{s}_{black}, \mathbf{s}_{red}, \mathbf{s}_{darkblue}, \mathbf{s}_{lightblue}, \mathbf{s}_{white}$. In order to simplify the writing, we will consider that the probabilities of the colours are decimals, but this does not make any difference. Each probability is assigned an distinct interval of $[0,1[$:

symbol	probability	interval
$\mathbf{s}_{black}$	0.1	[0 , 0.1 [
$\mathbf{s}_{red}$	0.1	[0.1 , 0.2 [
$\mathbf{s}_{darkblue}$	0.4	[0.2 , 0.6 [
$\mathbf{s}_{lightblue}$	0.3	[0.6 , 0.9 [
$\mathbf{s}_{white}$	0.1	[0.9 , 1 [

Then, the image to be encoded is the sequence of Figure 3 :

$\mathbf{s}_{red}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{darkblue}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{lightblue}$
$\mathbf{s}_{black}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{darkblue}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{darkblue}$
$\mathbf{s}_{darkblue}$	$\mathbf{s}_{black}$	$\mathbf{s}_{black}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{darkblue}$	$\mathbf{s}_{darkblue}$
$\mathbf{s}_{black}$	$\mathbf{s}_{black}$	$\mathbf{s}_{lightblue}$	$\mathbf{s}_{darkblue}$	$\mathbf{s}_{darkblue}$	$\mathbf{s}_{darkblue}$

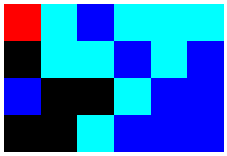


Figure 3: *Reduced image extracted of Figure 2.*

This sequence will be encoded by progressively shrinking the original interval. The final message is the lower bound of the last interval.

symbol	proba	I^j	interval $I^{j+1} = \inf I^j + I^j \cdot p$
	[, [	1	[0 ,1
$\mathbf{s}_{red}$	[0.1 ,0.2 [	0.1	[0.1 ,0.2
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	0.03	[0.16 ,0.19
$\mathbf{s}_{darkblue}$	[0.2 ,0.6 [	0.012	[0.166 ,0.178
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	0.0036	[0.1732 ,0.1768
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	0.00108	[0.17536 ,0.17644
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	0.000324	[0.176008 ,0.176332
$\mathbf{s}_{black}$	[0 ,0.1 [	$3.24\ 10^{-05}$	[0.176008 ,0.1760404
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	$9.72\ 10^{-06}$	[0.17602744 ,0.17603716
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	$2.92\ 10^{-06}$	[0.176033272 ,0.176036188
$\mathbf{s}_{darkblue}$	[0.2 ,0.6 [	$1.17\ 10^{-06}$	[0.1760338552 ,0.1760350216
$\mathbf{s}_{lightblue}$	[0.6 ,0.9 [	$3.50\ 10^{-07}$	[0.17603455504 ,0.17603490496
$\mathbf{s}_{darkblue}$	[0.2 ,0.6 [	$1.40\ 10^{-07}$	[0.176034625024 ,0.176034764992

(b) Algorithms

Decoding algorithm. The decoding procedure is easy to understand once the message $\mathbf{m} \in [0,1[$ has been completely read. In practise, it is progressively decoded since it is too long to be conveniently represented by a single number in memory, which introduces some extra work referred to as renormalisation. First, the probability p_i^j of the source symbols must be known at each step j of the decoding procedure. The initial interval is set to $I^0 = [0,1[$. Then, at each step j, the interval I^{j-1} is subdivided into parts proportional to the symbol probabilities p_i^j into subintervals sI_i^j as follows :

$$\begin{array}{rcrcl} sI_1^j & = & [& 0 , p_1^j & [\\ sI_2^j & = & [& p_1^j , p_1^j + p_2^j & [\\ sI_3^j & = & [& p_1^j + p_2^j , p_1^j + p_2^j + p_3^j & [\\ \cdots & & & & \\ sI_{n-1}^j & = & [& 1 - p_n^j - p_{n-1}^j , 1 - p_n^j & [\\ sI_n^j & = & [& 1 - p_n^j , 1 & [\end{array}$$

Then, the message $\mathbf{m}$ belongs to one of the subintervals sI_i^j, corresponding to symbol $\mathbf{s}_i$. This symbol $\mathbf{s}_i$ is added to the decoded message, and the next interval is set to sI_i^j : $I^j = sI_i^j$.

The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.

Algorithm 1 aritdecode(in,out) : decodes stream in to out

$I \leftarrow [0,1[$ // *initial interval*
$in \xrightarrow{+32} \mathbf{m}$ // *reads the first bits of the input*
repeat
  $(p_i)_{i\in[\![1,n]\!]} \leftarrow$ get_model () // *retrieves the probabilities*
  $count \leftarrow p_1$ // *upper bound of* sI_i^j
  for $i \in [\![2,n]\!]$ **do** // *look the interval containing* $\mathbf{m}$
    if $\mathbf{m} < count$ **then** // *found the subinterval*
      break // *exits the* ***for*** *loop*
    $count \leftarrow count + p_i$ // *next* i
  $\mathbf{s} \leftarrow \mathbf{s}_i$; $out \xleftarrow{+\mathbf{s}} \mathbf{s}_i$ // *decoded symbol* $\mathbf{s}_i$
  $I \leftarrow [count - p_{i-1}, count[$ // *updates the current interval*
  while $\frac{1}{2} \notin I$ **or** $|I| < \frac{1}{2}$ **do** // *renormalisation*
    if $I \subset [\frac{1}{2}, 1[$ **then** // *higher half*
      $I \leftarrow I - \frac{1}{2}$; $\mathbf{m} \leftarrow \mathbf{m} - \frac{1}{2}$ // *shifts higher half to lower half*
    else if $I \subset [\frac{1}{4}, \frac{3}{4}[$ **then** // *central half*
      $I \leftarrow I - \frac{1}{4}$; $\mathbf{m} \leftarrow \mathbf{m} - \frac{1}{4}$ // *shifts central half to lower half*
    $I \leftarrow 2 \cdot I$ // *lower half is directly renormalised*
    $\mathbf{m} \leftarrow 2 \cdot \mathbf{m}$; $in \xrightarrow{+1} \mathbf{m}$ // *message is shifted by reading* in
until $\mathbf{s} = stop$ // *read a stop symbol*

Algorithm 2 aritencode(in,out) : encodes stream in to out

$I \leftarrow [0,1[$ // *initial interval*
repeat
  $(p_i)_{i\in[\![1,n]\!]} \leftarrow$ get_model () // *retrieves the probabilities*
  $in \xrightarrow{-\mathbf{s}} \mathbf{s}_i$ // *retrieves symbol to encode*
  $I \leftarrow \left[\sum_{k\in[\![1,i-1]\!]} p_k, \sum_{k\in[\![1,i]\!]} p_k\right[$ // *deduce the current interval*
  while $\frac{1}{2} \notin I$ **or** $|I| < \frac{1}{2}$ **do** // *renormalisation*
    if $I \subset [0, \frac{1}{2}[$ **then** // *lower half*
      $out \xleftarrow{+1} 0$ // *appends a* 0 *to the coded message*
    else if $I \subset [\frac{1}{2}, 1[$ **then** // *higher half*
      $out \xleftarrow{+1} 1$ // *appends a* 1 *to the coded message*
      $I \leftarrow I - \frac{1}{2}$ // *shifts higher half to lower half*
    else if $I \subset [\frac{1}{4}, \frac{3}{4}[$ **then** // *central half*
      out.repeat_next_bit // *set out to repeat the next bit to be output*
      $I \leftarrow I - \frac{1}{4}$ // *shifts central half to lower half*
    $I \leftarrow 2 \cdot I$ // *scaling*
until $\mathbf{s} = stop$ // *read a stop symbol*

Renormalisation. Observe that unless the decoder does not stop on its own, as for Huffman coding. The source must have a stop symbol or the decompression must know how to stop the decoder. For large messages, the intervals I^j require more memory to be represented than the usual 2×64 bits offered by computers. Therefore, when an interval I^j is contained in $[0, \frac{1}{2}[$ or $[\frac{1}{2}, 1[$, one bit of the message is transmitted, the interval is shifted (scaled by two), and the algorithm goes on. Moreover, the intervals I^j can get arbitrarily small around $\frac{1}{2}$. In order to prevent this, when $I^j \subset [\frac{1}{4}, \frac{3}{4}[$, two bits of the message are transmitted, the interval is shifted twice (scaled by four). These processes are called *renormalisation*. In parallel, the message does not need to be read entirely, since the only precision needed to decode one symbol is given by the intervals I^j. At each renormalisation, a complement of the message is read to ensure that the precision of the interval matches the precision of the message. This whole process is implemented by Algorithm 1: aritdecode.

Encoding algorithm. The encoding procedure is very similar to the decoding one, as details on Algorithm 2: aritencode. Note that in both cases, the finite precision of computer representations forces to use only half of the available bits to represent the probabilities and the intervals, since both have to be multiplied with exact precision. Also, the open intervals are actually represented by closed ones: $[n_i, n_s[= [n_i, n_s - 1[$.

(c) Statistical Modelling

This arithmetic coder provides a powerful engine to encode a universal source. However, it is very sensible to the tuning, which is a very hard task. First, the probability model is very important. Consider a zero–entropy message, i.e. a message with a constant symbol $\mathbf{s}_0$. If the probability model states that $\mathbf{s}_0$ has probability $p_0 \ll 1$, then the encoded stream will have a huge length. Therefore, the arithmetic coder is not close to an entropy coder, unless very well tuned. We will see some generic techniques to improve these aspects.

Adaptive models. A simple solution to the adaptability of the probabilities consists in updating the probability model along the encoding, in a deterministic way. For example, the probability of a symbol can increase each time it is encoded, or the probability of a $stop$ symbol can increase at each new symbol encoded, as on the table below. This can be easily implemented through the function get_model() of Algorithm 1: aritdecode and Algorithm 2: aritencode. For the case of the zero–entropy stream, there would be a reasonable amount of encoded stream where p_0 goes closer to 1, and then the stream will be encoded at a better rate. Observe that p_0 cannot reach one, since the probability of each symbol must not vanish, prohibiting an asymptotic zero–length, but respecting the second item of the requirements for the entropy of section 2(b) *Information Theory*.

symbol	updated probabilites						$p_{\mathbf{s}}$	I^j
	$\frac{0}{10}$	$\frac{1}{10}$	$\frac{2}{10}$	$\frac{6}{10}$	$\frac{9}{10}$	$\frac{10}{10}$	$[\ ,\ [$	1
$\mathbf{s}_{red}$	$\frac{0}{11}$	$\frac{1}{11}$	$\frac{3}{11}$	$\frac{7}{11}$	$\frac{10}{11}$	$\frac{11}{11}$	$[\frac{1}{10},\frac{2}{10}[$	0.1
$\mathbf{s}_{lightblue}$	$\frac{0}{12}$	$\frac{1}{12}$	$\frac{3}{12}$	$\frac{7}{12}$	$\frac{11}{12}$	$\frac{12}{12}$	$[\frac{7}{11},\frac{10}{11}[$	0.027272
$\mathbf{s}_{darkblue}$	$\frac{0}{13}$	$\frac{1}{13}$	$\frac{3}{13}$	$\frac{8}{13}$	$\frac{12}{13}$	$\frac{13}{13}$	$[\frac{3}{12},\frac{7}{12}[$	0.009090
$\mathbf{s}_{lightblue}$	$\frac{0}{14}$	$\frac{1}{14}$	$\frac{3}{14}$	$\frac{8}{14}$	$\frac{13}{14}$	$\frac{14}{14}$	$[\frac{8}{13},\frac{12}{13}[$	0.002797
$\mathbf{s}_{lightblue}$	$\frac{0}{15}$	$\frac{1}{15}$	$\frac{3}{15}$	$\frac{8}{15}$	$\frac{14}{15}$	$\frac{15}{15}$	$[\frac{8}{14},\frac{13}{14}[$	0.000999
$\mathbf{s}_{lightblue}$	$\frac{0}{16}$	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{8}{16}$	$\frac{15}{16}$	$\frac{16}{16}$	$[\frac{8}{15},\frac{14}{15}[$	0.000400
$\mathbf{s}_{black}$	$\frac{0}{17}$	$\frac{2}{17}$	$\frac{4}{17}$	$\frac{9}{17}$	$\frac{16}{17}$	$\frac{17}{17}$	$[\frac{0}{16},\frac{1}{16}[$	$2.49\ 10^{-5}$
$\mathbf{s}_{lightblue}$	$\frac{0}{18}$	$\frac{2}{18}$	$\frac{4}{18}$	$\frac{9}{18}$	$\frac{17}{18}$	$\frac{18}{18}$	$[\frac{9}{17},\frac{16}{17}[$	$1.02\ 10^{-5}$
$\mathbf{s}_{lightblue}$	$\frac{0}{19}$	$\frac{2}{19}$	$\frac{4}{19}$	$\frac{9}{19}$	$\frac{18}{19}$	$\frac{19}{19}$	$[\frac{9}{18},\frac{17}{18}[$	$4.57\ 10^{-6}$
$\mathbf{s}_{darkblue}$	$\frac{0}{20}$	$\frac{2}{20}$	$\frac{4}{20}$	$\frac{10}{20}$	$\frac{19}{20}$	$\frac{20}{20}$	$[\frac{4}{19},\frac{9}{19}[$	$1.20\ 10^{-6}$
$\mathbf{s}_{lightblue}$	$\frac{0}{21}$	$\frac{2}{21}$	$\frac{4}{21}$	$\frac{10}{21}$	$\frac{20}{21}$	$\frac{21}{21}$	$[\frac{10}{20},\frac{19}{20}[$	$5.41\ 10^{-7}$
$\mathbf{s}_{darkblue}$	$\frac{0}{22}$	$\frac{2}{22}$	$\frac{4}{22}$	$\frac{11}{22}$	$\frac{21}{22}$	$\frac{22}{22}$	$[\frac{4}{21},\frac{10}{21}[$	$1.54\ 10^{-7}$
$\mathbf{s}_{darkblue}$	$\frac{0}{23}$	$\frac{2}{23}$	$\frac{4}{23}$	$\frac{12}{23}$	$\frac{22}{23}$	$\frac{23}{23}$	$[\frac{4}{22},\frac{11}{22}[$	$4.92\ 10^{-8}$
$\mathbf{s}_{black}$	$\frac{0}{24}$	$\frac{3}{24}$	$\frac{5}{24}$	$\frac{13}{24}$	$\frac{23}{24}$	$\frac{24}{24}$	$[\frac{0}{23},\frac{2}{23}[$	$4.27\ 10^{-9}$
$\mathbf{s}_{black}$	$\frac{0}{25}$	$\frac{4}{25}$	$\frac{6}{25}$	$\frac{14}{25}$	$\frac{24}{25}$	$\frac{25}{25}$	$[\frac{0}{24},\frac{3}{24}[$	$0.53\ 10^{-9}$

Order. This probability model can be enhanced by considering groups of symbols instead of only one. The number of symbols considered jointly is called the *order* of the coder. This is particularly useful for text, where syllables play an important role. An order 0 coder means that the probability model is updated continuously, whereas an order k model will use a different probability model for each combination of the k symbols preceding the encoded one.

Contexts. With this point of view, the arithmetic coder begins with one probability model, and updates it continuously along the encoding process. However, we can actually consider various probability models simultaneously, depending on the *context* of the symbol to encode. For example when coding a text, it is more probable to find a vowel after a consonant. Therefore, the probability of a vowel after another vowel could be reduced to improve the probability model.

Limits. Context modelling and order–based coding allows reducing the interdependence of the symbols (putting the entropy closer to the Kolmogorov complexity [43]). This process is the main part of describing the object to encode, but since it is a difficult one, these features can lead to significant improvements of the results. However, the number of contexts and the order must be limited, since for each context the coder builds a probability model through the regular updates, and an exponential number for each order added. This probability needs a reasonable amount of encoded stream to get closer to the real probability model. The encoded stream must be longer than this amount of time for each context.

Prediction. Another way to reduce the covariance relies on *prediction mechanisms*, i.e. deductions that can be equally obtained from the encoder and the decoder. Since we encode the lower part of the interval containing the message, a message ended by a sequence of 0 is cheaper to encode than a message ended with a 1, as on the example of section 3(a) *Arithmetic Coder*. Therefore, if the prediction always asserts the results, the message will be a sequence of 0s, with some isolated 1s. This is actually encoded by an arithmetic coder as a run–length encoded stream, since the $stop$ characters induce a very tiny last interval. If the stop can be predicted too, then the arithmetic coding spares the last sequence of 0s. In this rough point of view, the better case of arithmetic coding is, for a generic problem, the logarithm of the number of symbols.

4 Compression

Coding is only a part of a *compression scheme*. Actually, a compression scheme is composed of various steps of conversions, from the original data to a symbolic representation, from this representation to specifications of sources, from these sources to the encoded message, from this encoded message to a transmission protocol, which entails a re–coding for error detection, and the symmetric parts from the receiver.

This whole process can be designed part by part, or all together. For example, some nice compression scheme already contains error detection using the redundancy of the original data that is left after the encoding. Some lossy or progressive compression schemes perform the encoding directly from the representation and incorporate the specification of sources.

These features optimise compression for specific applications. However, a generic application usually requires a separate design of the parts of a compression scheme. In this context, arithmetic coding turns out to be a very flexible tool to work on the final *compression ratio*, i.e. the ratio of the final size and the original size of the data. Depending on the application, this compression ratio must be reduced to optimize specific characteristics of these applications, leading to different trade–offs. We will now detail three such generic trade–offs.

(a) Compaction

Compaction refers to compact data structures, also called succinct. These structures aim at reducing the size of the memory used during the execution of an application, while maintaining a small execution overhead. This trade–off between memory used and execution time must also allow a random access to the compact data structure. For example for mesh data structures, this trade–off can be simply a elegant data representation with no specific encoding such as [59, 36, 37]. It can also involve simple encoding scheme that are fast to interpret as [28], or involve a precise mixture of very efficient encoding with a higher–level data structure, such as [11, 12].

(b) Direct Compression

The most used meaning of compression refers to file compression or to compression of exchanged information. Most of the common generic algorithms are based on the

LZH (ZIP) algorithm of [39], aside from specific image and video compression algorithms such as JPEG [70, 13] and MPEG [19, 54]. In this case, the goal is to optimise the trade–off between compression rate and compression time: the enumeration method is usually too slow, while a simple coding of the data representation can be in general improved with a minor time overhead. The trade–off can also take into account the amount of memory necessary to compress or decompress the stream. In that case, the process is usually performed *out–of–core*, such as [32].

(c) Progressive Compression

The compression can also lead to a loss of information when decompressing. This can be useful either when the lost part is not significant, or when it can be recovered by a further step of the compression. In that second sense, lossy compression will generate objects at various levels of detail, i.e. in *multiresolution*. Each resolution can be compressed separately by the difference from the previous one. A variant of that scheme does not distinguish between levels of details, sending a *coarse level* by direct compression, and refining it by a sequence of local changes. In these contexts, the goal is to optimise the trade–off between compression ratio and the *distortion* of the decompressed model. For the geometrical models, the distortion is usually measured by the geometric distance between the decoded model and the original one.

5 Meshes and Geometry

Geometrical objects are usually represented through meshes. Especially for surfaces in the space, triangulations had the advantage for rendering of representing with a single element (a triangle) many pixels on screen, which reduced the number of elements to store. Although the increasing size of usual meshes reduced this advantage, graphic hardware and algorithms are optimised for these representations and meshes are still predominant over point sets models. Moreover, several parts of the alternative to meshes require local mesh generation, which becomes very costly in higher dimensions. Finally, meshes describe in a unique and explicit manner the support of the geometrical object, either by piecewise interpolation or by local parameterisation such as splines or NURBS.

To a real object correspond several meshes. These meshes represent the same geometry and topology, and thus differ by their *connectivity*. The way these objects are discretised usually depends on the application, varying from visualisation to animation and finite element methods. These variations make it harder to define the geometric quality of a mesh independently of the application, even with a common definition for the connectivity.

There is no need for a specific data structure for meshes. We will consider only the operations described in this section as the basic elements of a generic data structure. For further readings, the classical data structures for surfaces are the winged–edge [8], the split–edge [17], the quad–edge [22], the half–edge [49] and the corner–table [59, 37]. For non–manifold 2–meshes, we would mention the radial–edge [71] and [18]. Further references on the following definitions can be found in [52, 9, 27].

(a) Simplicial Complexes and Polytopes

There are various kind of meshes used in Computer Graphics, Scientific Visualisation, Geometric Modelling and Geometry Processing. However, the graphic hardware is optimised for processing triangles, line segments and points, which are all special cases of *simplices*. We will therefore focus mainly on meshes made of simplices, called *simplicial complex*, and one of its extensions to meshes made of convex elements, which we will refer as *polytopes*. This notion can be further extended to *cell complexes* [27], but these are only used for high–level modelling and we will not use them in this tutorial.

Simplicial Complexes

Simplex. A simplex is an n–dimensional analogue of a triangle. More precisely, a simplex σ^n of dimension n, or n–simplex for short, is the open convex hull of $n+1$ points $\{v_0, \ldots, v_n\}$ in general position in some Euclidean space $\mathbb{R}^d$ of dimension n or higher, i.e., such that no m–plane contains more than $(m+1)$ points. The closed simplex $\bar{\sigma^n}$ is the closed convex hull of $\{v_0, \ldots, v_n\}$. The points v_i are called the *vertices* of σ^n. For example, a 0–simplex is a point, a

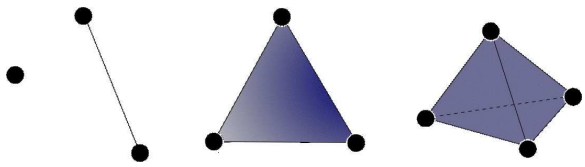
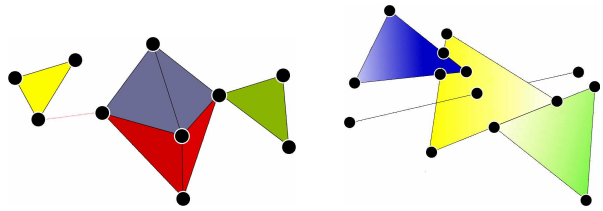


Figure 4: *Simplices from dimension* 0 *to* 3.

1–simplex is a line segment, a 2–simplex is a *triangle*, a 3–simplex is a *tetrahedron*, and a 4–simplex is a *pentachoron*, as shown on Figure 4.

Incidence. The open convex hull of any $m < n$ vertices of σ^n is also a simplex τ^m, called an *m–face* of σ^n. We will say that σ^n is *incident* to τ^m, and denote $\sigma^n \succ \tau^m$. The 0—faces are called the *vertices*, and the 1–faces are called the *edges*. The *frontier* of a simplex σ, denoted by $\partial\sigma$, is the collection of all of its faces.

Complex. A *simplicial complex* K of $\mathbb{R}^d$ is a coherent collection of simplices of $\mathbb{R}^d$, where coherent means that K contains all the faces of each simplex ($\forall \sigma \in K, \partial\sigma \subset K$), and contains also the geometrical intersection of the closure of any two simplices ($\forall\, (\sigma_1, \sigma_2) \in K^2, \bar{\sigma_1} \cap \bar{\sigma_2} \subset K$), as illustrated on Figure 5. Two simplices incident to a common simplex are said to be *adjacent*. The *geometry* of a complex usually refers to the coordinates of its vertices, while its *connectivity* refers to the incidence of higher–dimensional simplices on these vertices.



(a) Simplicial complex. (b) Non–complex.

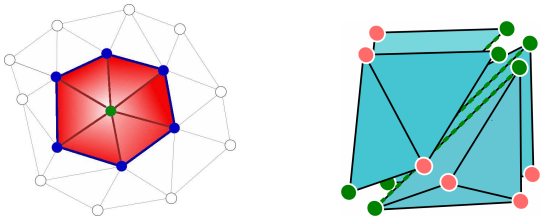
Figure 5: *Simplicial complex and a set of simplices not being a complex.*

Skeleton. If a collection K' of simplices of K is a simplicial complex, then it is called a *subcomplex* of K. The subcomplex $K_{(m)}$ of all the p–simplices, $p \leqslant m$, is called the m–*skeleton* of K.

Connected components. A complex is *connected* if it cannot be represented as a union of two non–empty disjoint subcomplexes. A *component* of a complex K is a maximal connected subcomplex of K.

Local Structure

Consider a simplex σ of a complex K. The local neighbourhood of σ is described by its *star* [1].



(a) The star of a vertex is the union of the open star (in red) with the ling (in blue).

(b) The star of an edge.

Figure 6: *Vertex star in a* 2*–complex and edge star in a* 3*–complex.*

Link. The *join* $\sigma \star \tau$ of two disjoint simplices σ and τ is the simplex that is the open convex hull of $\sigma \cup \tau$. The *link* of a simplex $\sigma \in K$ is the set of simplices whose join with σ belongs to K: $\mathrm{lk}(\sigma) = \{\tau \in K : \bar{\sigma} \cap \bar{\tau} = \varnothing, \sigma \star \tau \in K\}$.

Star. The *open star* of σ is then the join of σ with its link: $\dot{\mathrm{st}}(\sigma) = \{\sigma \star \tau, \tau \in \mathrm{lk}(\sigma)\}$. Finally, the *star* of σ is the union of all the simplices of the open star together with all their faces: $\mathrm{st}(\sigma) = \dot{\mathrm{st}}(\sigma) \cup \bigcup_{\rho \in \dot{\mathrm{st}}(\sigma)} \partial\rho$. The *valence* of a vertex is the number of maximal faces in its star.

Pure Simplicial Complexes

Dimension. The *dimension* n of a simplicial complex K is the maximal dimension of its simplices, and we will say that K is an n–complex. A *maximal face* of a simplicial complex of dimension n is an n–simplex of K.

Euler–Poincar characteristic. Denoting $\#_m(K)$ the number of m–simplices in K, the *Euler–Poincar characteristic* $\chi(K^n)$ of an n–complex K^n is a topological invariant [27] defined by $\chi(K^n) = \sum_{m \in \mathbb{N}} (-1)^m \#_m(K^n)$.

Pure complexes. Roughly speaking, a complex is pure if all the visible simplices have the same dimension. More precisely, a simplicial complex K^n of dimension n is *pure* when each p–simplex of K, $p < n$, is face of another simplex of K.

Boundary. The *boundary* ∂K of a pure simplicial complex K^n is the closure of the set, eventually empty, of its $(n-1)$–simplices that are face of only one n–simplex: $\partial K^n = \left\{\sigma^{n-1} : \#\,\mathrm{lk}\left(\sigma^{n-1}\right) = 1\right\}$. The simplices of the boundary of K and their faces are called *boundary* simplices, and the other simplices are called *interior* simplices.

Simplicial Manifolds

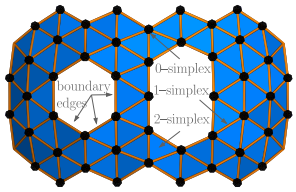


Figure 7: *A surface with two bounding curves*

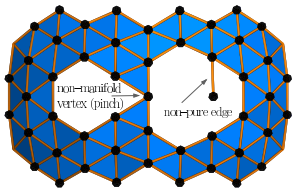


Figure 8: *A non–pure* 2*–complex with a non–manifold vertex.*

Manifolds. A simplicial n–manifold $\mathcal{M}^n$ is a pure simplicial complex of dimension n where the open star of each interior vertex is homeomorphic to an open n–ball $\mathbb{B}^n$ and the open star of each bounding vertex is homeomorphic to the intersection of $\mathbb{B}^n$ with an closed half–space. This implies that each $(n-1)$–simplex of $\mathcal{M}$ is the face of either one or two simplices. In particular, the boundary of an n–manifold is a $(n-1)$–manifold with an empty boundary.

Orientability. An orientation on a simplex is an ordering $(v_0, \ldots, v_n)$ on its vertices. Two orientations are equivalent if they differ by an even permutation. There are therefore two opposite orientations on a simplex. A simplicial manifold $\mathcal{M}^n$ is *orientable* when it is possible to choose a coherent orientation on all its simplices. More precisely, if $\sigma^{n-1} = (v_1, \ldots, v_n)$ is an oriented interior $(n-1)$–simplex of $\mathcal{M}^n$, face of $\rho = \sigma^{n-1} \star v$ and $\rho' = \sigma^{n-1} \star v'$, then the orientation of ρ and ρ' is coherent the orientation of ρ is equivalent to $(v, v_1, \ldots, v_n)$ and the orientation of ρ' is opposed to $(v', v_1, \ldots, v_n)$. This orientation thus defines the notion of *next* and *previous* vertex inside a triangle of a simplex.

Surfaces. For example, a 2–manifold is a surface, i.e. a simplicial complex made of only vertices, edges and triangles where each edge is in the frontier of either one or two triangles and where the boundary does not pinch. For example,

Figure 7 shows an example of 2–manifold and Figure 8 illustrates a 2–complex that is neither pure nor a manifold. The topology of surfaces can be easily defined from its orientability and its Euler–Poincar characteristic, using the Surface classification theorem [6]: Any oriented connected surface $\mathcal{S}$ is homeomorphic to either the sphere $\mathbb{S}^2$ ($\mathfrak{g}(\mathcal{S}) = 0$) or a connected sum of $\mathfrak{g}(\mathcal{S}) > 0$ tori, in both cases with some finite number $\mathfrak{b}(\mathcal{S})$ of open disks removed. The number $\mathfrak{g}(\mathcal{S})$ is called the *genus* of $\mathcal{S}$, and $\mathfrak{b}(\mathcal{S})$ its *number of boundaries*. The Euler–Poincar characteristic $\chi(\mathcal{S})$ of $\mathcal{S}$ is equal to $\chi(\mathcal{S}) = \#_2(\mathcal{S}) - \#_1(\mathcal{S}) + \#_0(\mathcal{S}) = 2 - 2 \cdot \mathfrak{g}(\mathcal{S}) - \mathfrak{b}(\mathcal{S})$.

Polytopes

Surfaces in finite element methods are usually represented by a mixture of triangular and quadrangular elements. Although this do not directly fits to the simplicial complexes we just introduced, this structure can be easily extended to that case. For example, one could divide each quadrangle into two coplanar triangles and get a simplicial complex. We define a *polytope* in a similar way.

Along this tutorial, a polytope in $\mathbb{R}^d$ will be a coherent collection of convex open set of $\mathbb{R}^d$, called *cells*, where coherent means again that the collection contains the frontiers and the intersections of its cells. Observe that this implies that each cell is made up with piecewise linear elements, from edges to its maximal faces.

The definition and properties described above are still valid, in particular the notion of boundary, manifold, orientability, and the classification for 2–dimensional manifold polytopes. Moreover, polytopes are useful to define the *dual* of a manifold: The dual of an n–manifold $\mathcal{M}^d$ is the manifold polytope obtained by reversing the incidence relations of its cells, i.e. creating a vertex for each n–cell of $\mathcal{M}^d$, and an m–cell for each $(n-m)$–cell of $\mathcal{M}^d$, spanning the vertices created for each n–cell of its star in $\mathcal{M}^d$.

6 Combinatorial Operators

The encoding of meshes describes, in a compact way, how to build the encoded mesh. The decoding operation thus performs a sequence of combinatorial operations on an initial empty or canonically defined mesh together with a reconstruction of its geometry. These combinatorial operations are of two kinds: purely constructive ones and their inverse, namely the Euler and Handle operators, which only increase the number of simplices (or cells) without modifying existing ones; and subdivision ones with their inverse, and Stellar operators is a complete set for this category. The Handle operators are sometimes considered as a special case of Euler operators, since these operations are similar but they alter the topology of the mesh.

We will now describe each of these operators, first because it is a complete set of operators for mesh, and second because the Edgebreaker algorithm [57] of section 7 *Connectivity–Driven Compression* uses the specificities of Handle operators together with Euler operators.

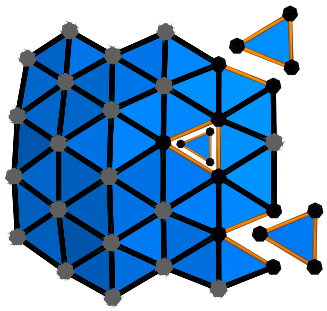


Figure 9: *3 ways of attaching a simplex to a triangulation. Centre: MSG, Down: MTE, Up: MEV + MTE.*

(a) Euler Operators

Generic Euler operators for surfaces. Euler operators were originally defined as operators on surfaces that do not change its manifoldness nor its Euler characteristic [49]. In this restricted definition, there was 5 creation and 5 destruction operators, some illustrated on Figure 9, namely:

- M/K EV adds/removes an edge and a vertex
- M/K TE adds/removes a triangle and an edge
- M/K STV adds/removes an initial triangle, without edge
- M/K SG closes/opens a bounding curve
- M/K EKL adds/removes an edge, joining two bounding curves

We will now extend these definitions, in order to first distinguish between topology–preserving operators (M/K EV and M/K TE) and Handle operators that will be introduced next, and second to extend these definitions to any dimension. All the definitions of this section also apply to polytopes.

Low–level Euler operators. With this distinction, basic Euler operators are reduced to *simplicial collapse* and *simplicial expansion* [27]. Given a simplicial complex K and σ^m a simplex of K face of only one $(m+1)$–simplex ρ^{m+1}, we say that K collapses to $K' = K \setminus \{\sigma^m, \rho^{m+1}\}$, and that K' expands to K. Observe that since we add or remove pairs of simplices of consecutive dimensions, we do not change the Euler characteristic of K. For $m = 0$, the expansion corresponds to MEV, and the collapse to KEV. For $m = 1$, the expansion corresponds to MTE, and the collapse to KTE.

Euler attachment. The simplicial expansion operator works at low–level, and in particular for pure complexes or manifolds, these operators must be used in group to preserve the purity of the mesh as on the top of Figure 9. The *simplex attachment* operations compose the minimal group of Euler operators that add a unique maximal face, preserving the purity. For 1–complexes, a simplex attachment of order 1 is a MEV operation. For 2–complexes, a simplex attachment of order 2 is a MTE operation, eventually preceded by a simplex attachment of order 1. In general, a simplex attachment for n–complexes can be described as a simplicial expansion with $m = n-1$, eventually preceded of at most $n-1$ simplex attachments of order $n-1$.

Manifold Euler operators. The simplicial expansion preserves the topology of the mesh (actually this define its simple homotopy type), and combined into simplex attachment operations, it preserves the purity of the mesh. In order to preserve the manifoldness of the mesh, we need to restrict the simplex attachment operations. From the property of manifolds, that each $(n–1)$–simplex is the face of either one or two simplices, we need to restrict these simplex attachment to the boundary of the manifold. Since Euler attachments preserve the topology of the mesh, this is actually sufficient: the manifold Euler operators are attachments involving only bounding simplices.

(b) Handle Operators

Generic attachment. We distinguished inside the generic Euler operators for surfaces between those who preserve and those who change the topology of the simplicial complex. The Euler attachment resulted in adding a maximal face using only low–level Euler operators, and thus preserving the topology. The generalization of this notion, the *generic attachment*, also adds a simplex and its faces, with the only restriction of preserving the properties of a simplicial complex. More precisely, an m–simplex σ^m can be attached to an n–complex K^n by identifying some of the faces of σ^m with some of the simplices of K. In order to preserve the simplicial complex property, we impose that if a face τ of σ^m is identified with a simplex τ' of K^n, then the faces of τ are identified with the faces of τ' in a one–to–one correspondence.
Such operation can alter the topology of the complex, and its manifoldness. In particular, observe that the first step of a mesh construction, i.e. creating the first simplex, is a generic attachment onto an empty complex.

Manifold Handle operators. In order to preserve the manifoldness, we must first restrict the attachment to a maximal face, and identify part of its frontier with the boundary of the manifold, as we did previously. However, this is not enough, since a generic attachment can create a pinch on the manifold. A sequence of Euler attachments can then fatten this pinch in order and thus recover the manifoldness. Therefore, we will define a Handle operator on $\mathcal{M}^n$ as a generic attachment involving only the boundary of $\mathcal{M}^n$ of an n–simplex followed by at most $n-1$ Euler attachments of n–simplices. This operation is also referred as *Handle attachment* [45, 46].

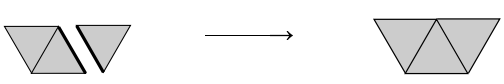


Figure 10: $\chi + 0$*: Euler attachment.*

Handle operators for surfaces. These manifold Handle operators can be exhaustively described, and especially for curves and orientable manifold, they characterise exactly the topological change they induce. For 1–complexes, there are four constructive operators: creating a first vertex (generic attachment), adding an edge and a vertex (MEV), adding an edge between two vertices of a boundary (Handle operator) and adding an edge between an interior vertex and another vertex (Non–manifold generic attachment). For surfaces, there are seven surface constructive operations, with their inverse, destructive operators [45, 46]:

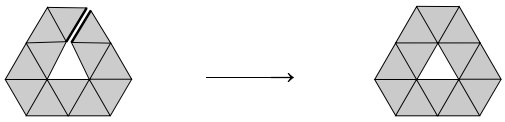


Figure 11: $\mathfrak{b}+1$, $\chi-1$*: lower* 1*–handle operator.*

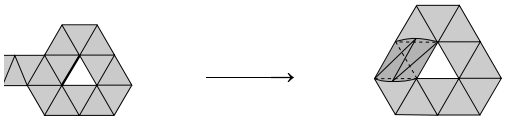


Figure 12: $\mathfrak{g}+1, \mathfrak{b}-1$, $\chi-1$*: upper* 1*–handle operator.*

$\chi + 1$ creates a new connected component with initial triangle, with its three edges and vertices (Handle operator: 0–handle)

$\chi + 0$ completes two consecutive bounding edges with a triangle (MTE, Figure 9)

$\chi + 0$ glues a triangle on a bounding edge (Euler attachment, Figure 10)

$\chi - 1$ glues two bounding edges of distinct connected components (Handle operator: 1–handle)

$\mathfrak{b} + 1$ glues two non–consecutive edges of the same bounding curve with two triangles, splitting this curve into two bounding curves (Handle operator: lower 1–handle, Figure 11)

$\mathfrak{g} + 1, \mathfrak{b} - 1$ glues two edges of different bounding curves with two triangles, creating a genus (Handle operator: upper 1–handle, Figure 12)

$\mathfrak{b} - 1$ closes a three–edges bounding curves with a triangle (Handle operator: 2–handle, Figure 13)

7 Connectivity–Driven Compression

Since there is still no strong relation between the geometry and the connectivity of these meshes for the usual objects considered by graphics applications, dedicated compression schemes consider either that the common information can be deduced from either the connectivity or the geometry. The first option assumes that the star of a simplex has a simple geometry, which can be well approximated by simple methods such as linear interpolation. Then, the geometry can be efficiently encoded by a connectivity traversal of the mesh, leading to *connectivity–driven* compression schemes. The second option predicts the connectivity from the geometry, and will be referred as *geometry–driven* compression schemes. In that case, the connectivity is usually better compressed, but it needs efficient geometry coding.

The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.

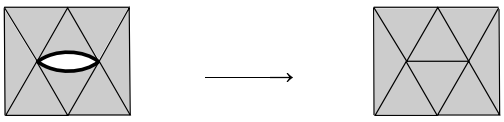


Figure 13: $\mathfrak{b} - 1$, $\chi + 1$*: 2–handle operator.*

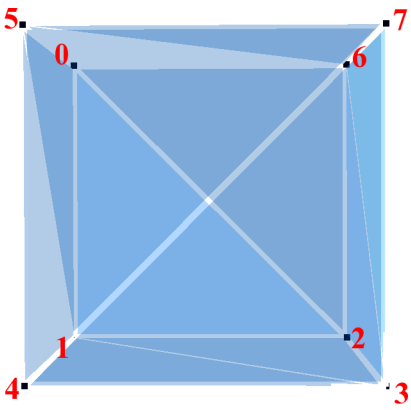
In this section, we will focus on the connectivity part of the compression. These connectivity–driven methods improved so much in the last decade that the compression ratio for usual surface connectivity turns around 2/3 bits per vertex. We will give a general framework for handling the critical elements of the connectivity: the topological singularities. These singularities are well understood for surfaces through the Handle operators of section 6(b) *Handle Operators*. We will then focus on the Edgebreaker scheme, and introduce two new improvements: the handling of boundary, as a consequence of this framework for singularities, and a small improvement of the decompression algorithm. We will conclude this section with compression ratios of the Edgebreaker on usual models with recent improvements, and we will detail the specificities of connectivity–driven compression scheme. The goal of this section is to state what connectivity–driven compression means, with the detailed example of the Edgebreaker (Figure 14 and Figure 15), and to show where these algorithms are well suited.

(a) Principles

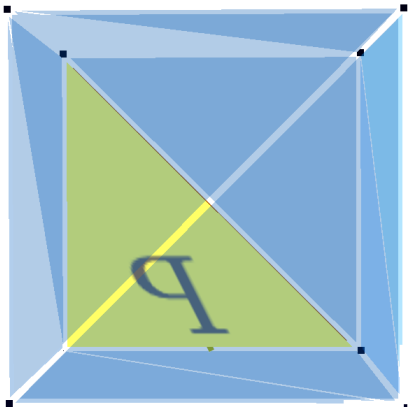
Connectivity–driven compression schemes rely on a traversal of the mesh in order to visit each vertex, and to identify it on further visits. This way, the geometry of the vertex needs to be transmitted only once, and the traversal encodes the connectivity of the mesh. This general framework suits particularly well for manifold polytopes. Most of the existing compression techniques are dedicated to surfaces, and we will focus on these algorithms. Further extensions to non–manifold cases are described in [21], while simple extensions of the most common schemes exist for solid models in [63, 31].

Connectivity–driven compression begun with cache problems in graphic cards: the rough way of transmitting triangle meshes from the main memory to the graphic card is (still) to send the three vertices of the triangle, represented by their three floating–point coordinates. Each triangle is then encoded with 96 bits! [16] proposed to represent these triangle meshes by generalised strips in order to share one or two vertices with the last triangle transmitted, reducing by at least a half the memory required previously. This mechanism uses also a small prediction scheme to optimise caching.

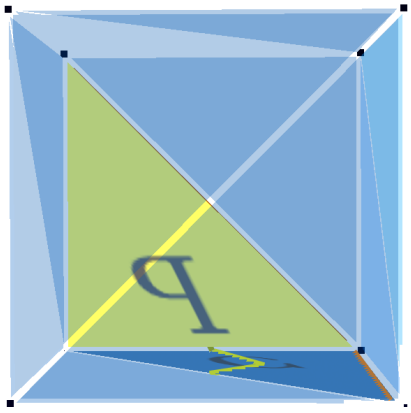
Then, these strips were generalised by a topological surgery approach in [64, 65]. These works introduced the most general framework for connectivity–driven compression, and has been efficiently derived into the Edgebreaker [57], and with a more flexible way into the valence coding of [66, 2]. The Edgebreaker has been ex-



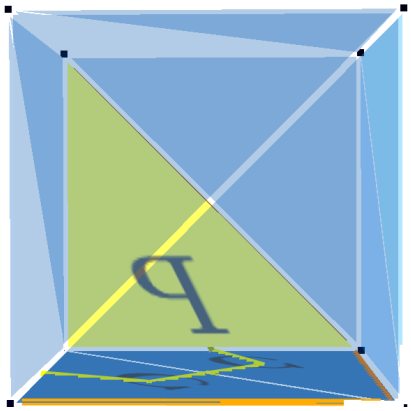
(a) Vertex labels used in the next sequence.



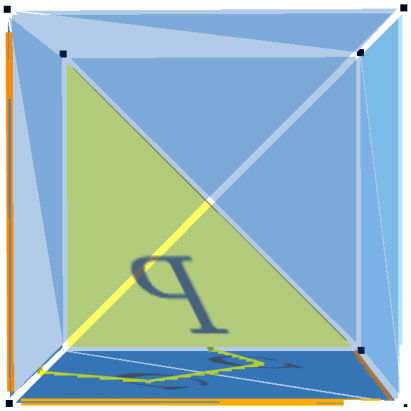
(b) First triangle not encoded: **P**, vertices 0, 1, 2 are marked. It will be the root of the dual tree. The traversal starts from edge 12.



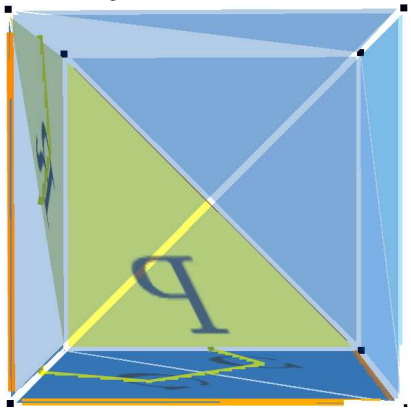
(c) Since vertex 3 is unmarked, 132 is created and 3 is marked: **C**. This extends the dual tree and the primal remainder. The traversal continues on the right.



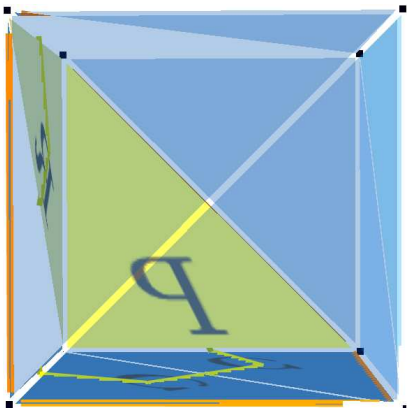
(d) Similarly, since vertex 4 is unmarked, 143 is created and 4 is marked: **C**.



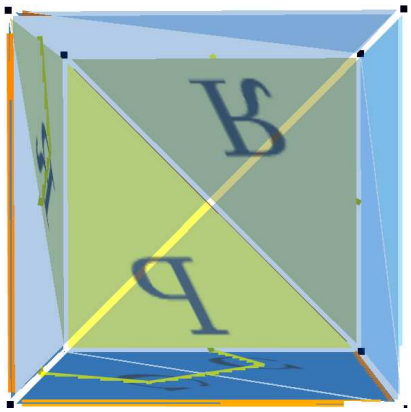
(e) Again, vertex 5 marked: **C**



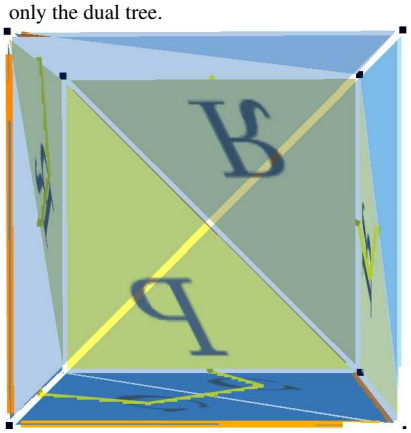
(f) Since vertex 0 is marked and the right triangle is marked already, 105 is attached and the traversal continues on the left: **R**. This extends only the dual tree.



(g) **C** again: vertex 6 is marked.



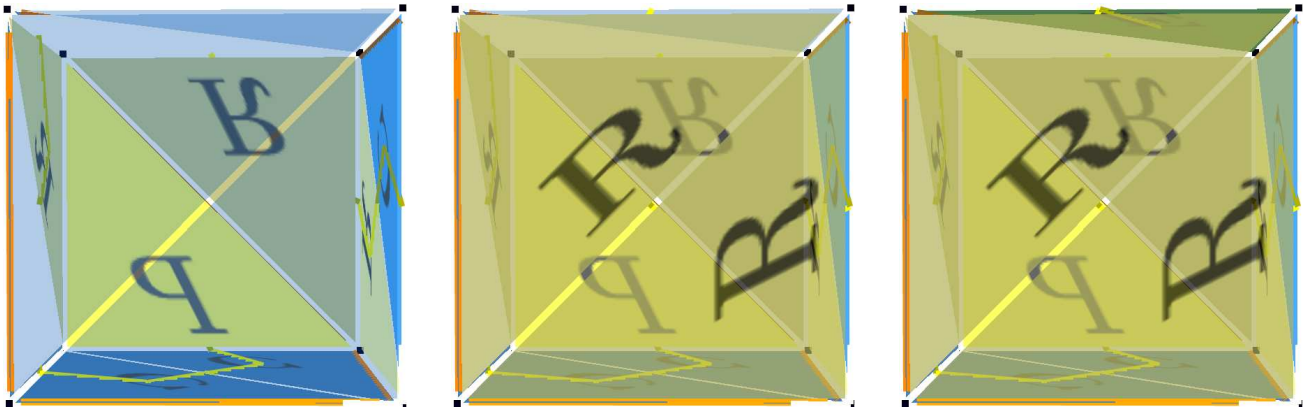
(h) Again, vertex 2 is already marked and the right triangle also: **R**.



(i) Again: **R**.

Figure 14: *Edgebreaker compression of a triangulated cube.*

The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.

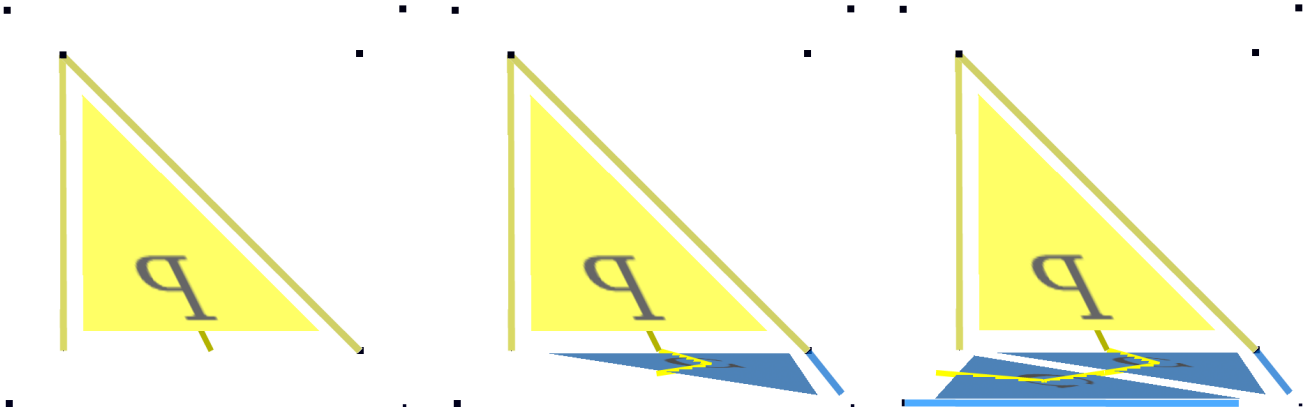


(j) **C** again: vertex 7 is marked.

(k) Again, vertices 4 and then 5 are already marked, with their right triangles also: **RR**.

(l) Since vertex 6 is marked, and both the right and left triangles are marked, attach 567: **E**. This extends the dual tree only.

Figure 14: *Edgebreaker compression of a triangulated cube (continued).*

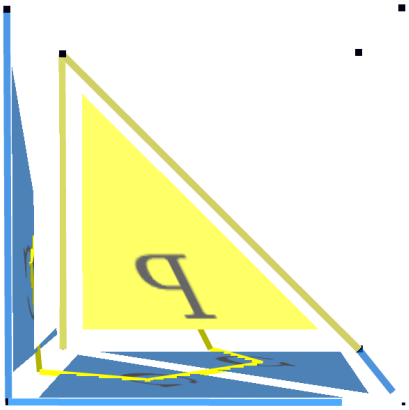


(a) Decode **P**: create the first vertex. This is a Handle operator of type 0: $\chi = 0 \rightarrow \chi = 1$

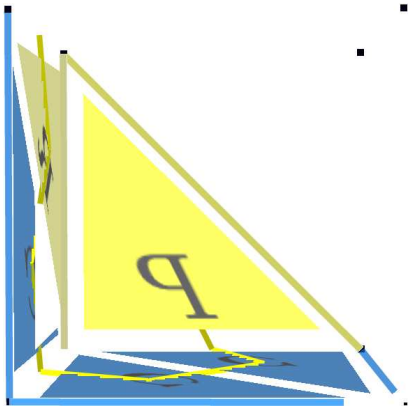
(b) Decode **C**: create a new triangle. This is an Euler attachment: $\mathsf{MEV}_3 + \mathsf{MTE}$.

(c) Decode **C**: create a new triangle: $\mathsf{MEV}_4 + \mathsf{MTE}$.

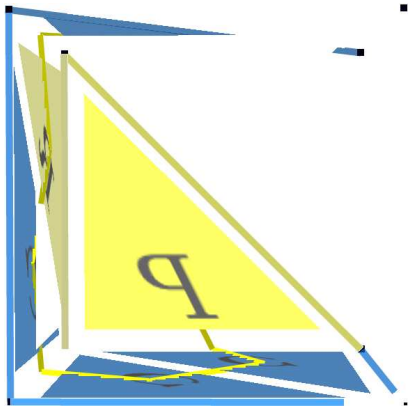
Figure 15: *Wrap&Zip decompression of a triangulated cube.*



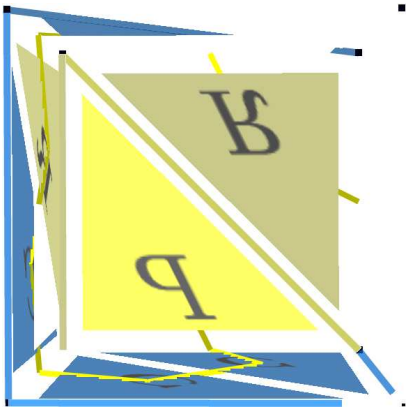
(d) Decode $\mathbf{C}$: create a new triangle: MEV$_5$ + MTE.



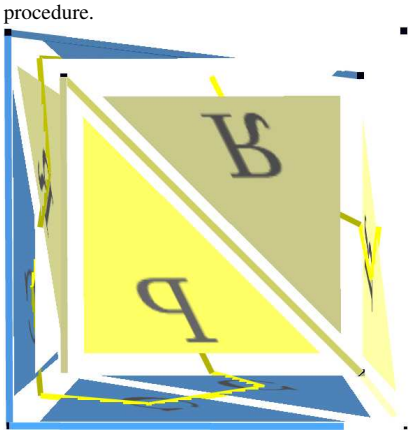
(e) Decode $\mathbf{R}$: attach one triangle: MTE. The new edge will be identified later by the Zip procedure.



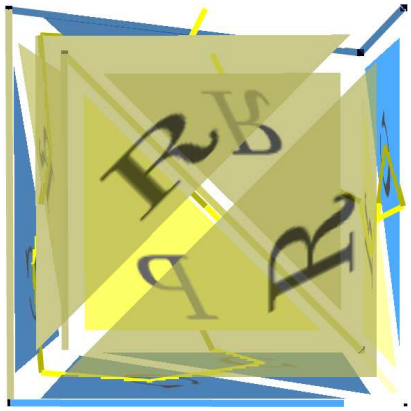
(f) Decode $\mathbf{C}$: create a new triangle: MEV$_6$ + MTE.



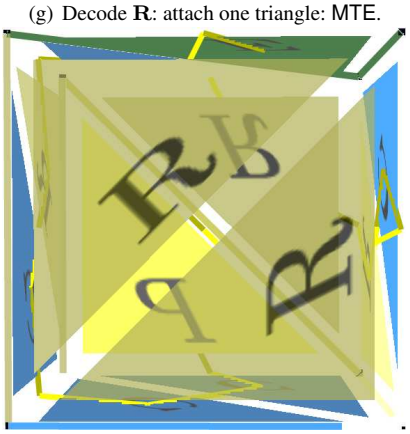
(g) Decode $\mathbf{R}$: attach one triangle: MTE.



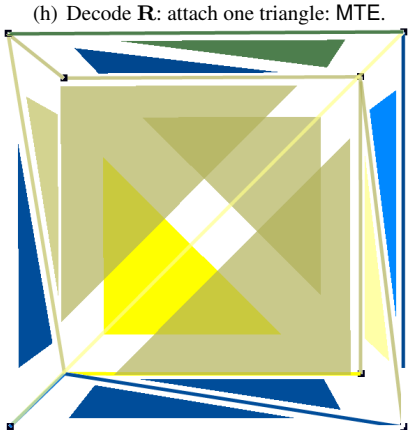
(h) Decode $\mathbf{R}$: attach one triangle: MTE.



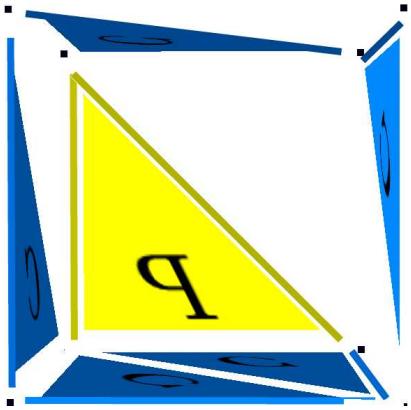
(i) Decode $\mathbf{CRR}$ as above.



(j) Decode $\mathbf{E}$: close one triangle. This is a Handle operator of type 2: $\mathfrak{b} = 1 \rightarrow \mathfrak{b} = 0$. The two new edges will be identified by the Zip procedure.



(k) The above Wrap procedure already decoded the adjacencies of the traversal: this is the dual tree.



(l) The Zip procedure will then identify the edges of the primal remainder, matching edges created by a $\mathbf{C}$ with the others.

Figure 15: *Wrap&Zip decompression of a triangulated cube (continued).*

tended to handle larger categories of surfaces in [47, 40], while valence coding has been tuned using the geometry in [2], discrete geometry [33]. In addition, the generated traversal of valence coding can be cleaned using [10].

With these improvements, the connectivity of usual models can be compressed with less than 3 bits per vertex. Geometry became the most expensive part, which can be reduced using prediction [66, 14] and high–quality quantisation [62, 24]. However, we will not focus here on the compression of the geometry.

(b) Primal or dual remainders

Primal and dual graphs. The main advance of topological surgery [64] was to substitute mesh connectivity compression by graph encoding. A graph can be considered as a simplicial complex of dimension 1. Therefore, the 1–skeleton $K_{(1)}$ of any simplicial complex is a graph, called the *primal graph* of the manifold. Moreover for manifolds, we defined the dual manifold in section 5.a(v) *Polytopes*, and the 1–skeleton of this dual manifold is also a graph, called the *dual graph* of the manifold. For example, Figure 16 represents the primal and the dual graph of a triangulated sphere.

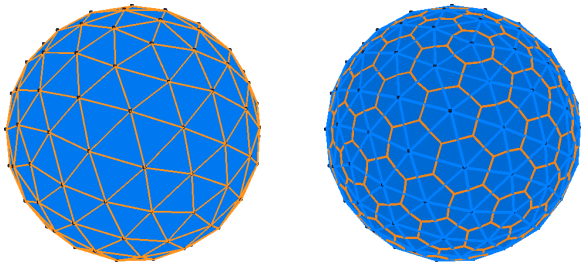


Figure 16: *(left): the primal graph and (right): the dual graph of a triangulated sphere.*

Tree encoding. For simplicial surfaces, the dual graph has a very nice property: each node of the graph has three incident links. Encoding the connectivity thus resumes to encoding this dual graph. In order to encode the geometry, this graph must be encoded by traversal, i.e. a spanning forest. Since each connected component can be encoded separately, we will consider only connected orientable surfaces, and the spanning forest is, in that case, a tree. This tree can be encoded from its root by enumerating for each node how many sons he has. This is the principle of both the valence coding and the Edgebreaker algorithms. The first one encodes the mesh by enumerating the valence of each node of a spanning tree in the primal graph, while the second one encodes a little more than the valence of each node of a spanning tree in the dual graph. In this last case, the valence is either 1, 2 or 3 since the nodes of the dual graph have a constant valence, which simplifies the coding.

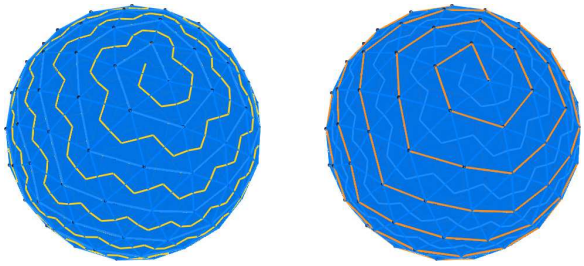


Figure 17: *(left): a dual spanning tree $\mathcal{S}^{21}$ extracted from the dual graph of Figure 16(right). (right): the primal remainder $\mathcal{S}^{01}$ of $\mathcal{S}^{21}$, which is a subgraph of the primal graph of Figure 16(left).*

Remainders. For clarity of the presentation, we will focus on spanning tree of the dual graph and the primal remainder, which is the focus of the Edgebreaker. What follows can be read identically by considering spanning tree of the primal graph and the dual remainder, which is the point of view of the valence coding. Consider a surface $\mathcal{S}$, with a spanning tree $\mathcal{S}^{21}$ of its dual graph. Observe that the links of $\mathcal{S}^{21}$ correspond to edges of $\mathcal{S}$. Then, consider the primal graph $\mathcal{S}^1$ (1–skeleton) of $\mathcal{S}$. Its links also correspond to edges of $\mathcal{S}$. The graph $\mathcal{S}^{01}$ having the same nodes as $\mathcal{S}^1$ and the links of $\mathcal{S}^1$ not represented in the dual spanning tree $\mathcal{S}^{21}$ is called the *primal remainder* of $\mathcal{S}^{21}$. This remainder is what is left to encode after the traversal of the dual mesh, i.e. $\mathcal{S}^{21}$, has been encoded. For example, the Edgebreaker encodes this primal remainder by specific symbols for the valence 1 and 2 of the dual tree. Moreover, this primal remainder contains all the vertices of the mesh, and will therefore be used to drive the encoding of the geometry.

(c) Topological Singularities

Topology of the remainders. If the remainder is a tree, then it can be easily encoded. The original Edgebreaker works directly in that case. However, this is not always the case, and the topology of the primal remainder actually characterises the topology of the (orientable) surface. For the dual remainder used by the valence coding, there is a detail to assert when the surface has a non–empty boundary. This process relies on a very simple calculus of the Euler characteristic of the remainder. According to section 5.a(iii) *Pure Simplicial Complexes*, the Euler characteristic of a surface is given by $\chi(\mathcal{S}) = \#_2 - \#_1 + \#_0$, and according to the surface classification theorem introduces in section 5.a(iv) *Simplicial Manifolds*, $\chi(\mathcal{S}) = 2 - 2 \cdot \mathfrak{g}(\mathcal{S}) - \mathfrak{b}(\mathcal{S})$. Since $\mathcal{S}^{21}$ is a tree with exactly one node for each of the $\#_2$ faces, it has $\#_2 - 1$ links. Therefore, the Euler characteristic of the primal remainder $\mathcal{S}^{01}$ is $\chi(\mathcal{S}^{01}) = \chi(\mathcal{S}) - \chi(\mathcal{S}^{21}) = 1 - 2 \cdot \mathfrak{g}(\mathcal{S}) - \mathfrak{b}(\mathcal{S})$. We get the same result for the case of a dual remainder.

Remainder of topological spheres. If the surface $\mathcal{S}$ is a topological sphere, then $\mathfrak{g}(\mathcal{S}) = \mathfrak{b}(\mathcal{S}) = 0$, and the remainders have Euler characteristic 1. From the Jordan curve theorem [6], the remainders are connected, since they cannot be disconnected by the corresponding spanning tree, which has

no closed curve. Then, the remainder is a connected graph with Euler characteristic 1: it is a tree. This primal remainder will be easy to encode, relating topological simplicity to easy compression with connectivity–driven schemes.

Morse edges. For a generic remainder, its Euler characteristic is $1 - 2 \cdot \mathfrak{g}(\mathcal{S}) - \mathfrak{b}(\mathcal{S})$. In the case of a dual spanning tree, the primal remainder is always connected. However, for primal spanning trees on surfaces with a non–empty boundary, the dual remainder can be disconnected. This can be avoided if the primal spanning tree contains all the bounding edges of the surface, except one per boundary components to keep it as a tree. With this restriction, the remainder is a connected graph with exactly $2 \cdot \mathfrak{g}(\mathcal{S}) + \mathfrak{b}(\mathcal{S})$ independent cycles, where a cycle is a sequence of distinct adjacent links whose last one is adjacent to the first one, and where independent means that removing one link of a cycle does not break any other. For each cycle, one edge that would break it will be called a *Morse edge*, since it induces a change in the topology of the surface, and corresponds to a Handle operator introduced in [46] and section 6(b) *Handle Operators*. Any connectivity–driven compression scheme designed for topological spheres can be extended to any orientable surface by encoding separately these Morse edges. For example, in the

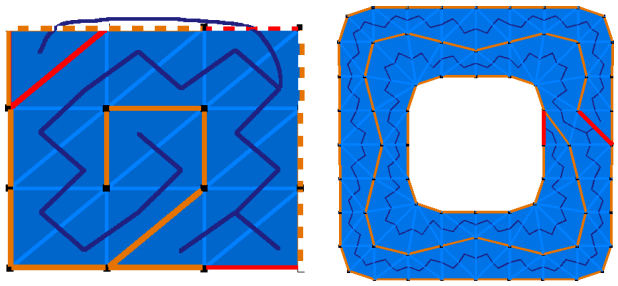


Figure 18: *(left): a primal remainder on a torus (genus 1): the topmost and bottommost horizontal edges are identified, and so do the leftmost and rightmost ones. (right) a primal remainder on an annulus (two bounding curves).*

case of a sphere, the primal remainder is a tree, as shown on Figure 17. For a mesh with genus one or with two boundary curves, the primal remainder is a graph with two cycles, as shown on Figure 18.

8 The Edgebreaker example

The Edgebreaker scheme has been enhanced and adapted from the Topological Surgery [64] to yield an efficient but initially restricted algorithm [57], which encodes the connectivity of any simplicial surface homeomorphic to a sphere with a guaranteed worst case code of 1.83 bits per triangle [34]. The Wrap&Zip algorithm introduced in [58] enhanced the original Edgebreaker decompression worst–case complexity from $O(n^2)$ to $O(n)$, where n is the number of triangles of the mesh. It decompresses the mesh in two passes, a direct and a recursive one. It is possible to decompress it in only one pass using the Spirale Reversi algorithm of [30], but it requires to read the encoded backwards, which is not appropriate for the Huffman encoding of [34] or the arithmetic encoding. But the true value of Edgebreaker lies in the efficiency and in the simplicity of its implementations [59], which is very concise. This simple algorithm has been extended to deal with non–simplicial surfaces [35] and the compression of simplicial surfaces with handles has been enhanced in [48] using *handle data*. Because of its simplicity, Edgebreaker is viewed as the emerging standard for 3D compression [60] and may provide an alternative for the current MPEG–4 standard, which is based on the Topological Surgery approach [64].

In this section, we will enhance the Edgebreaker compression for surfaces with a non–empty boundary. [34] encoded these surfaces by closing each bounding curve with a dummy vertex. This is a very simple but expensive solution: first, it requires encoding each bounding edge with a useless triangle; second, it requires extra code to localise the dummy vertex; and third, it gives bad geometrical predictors on the boundary. The original solution of [57] however encodes bounding curves a special symbol containing their length, which solves the first item but does not describe explicitly the topology of the surface, and gave bad prediction on the boundary. As we introduced in [40], we use directly the handle data to encode the boundaries, which solves the above mentioned problems and enhances the compression ratio. We will also introduce a small acceleration to the Wrap&Zip procedure in order to avoid the recursion, accelerate the decompression and reduce the amount of memory used.

(a) CLERS encoding

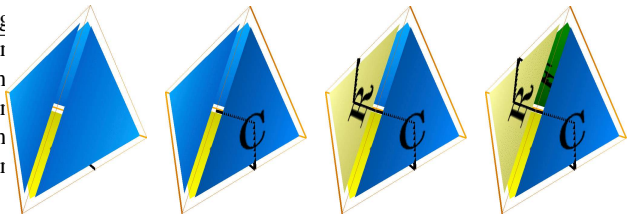
Gate based compression. Edgebreaker encodes the connectivity of the mesh by producing the stream of symbols taken from the set **C**,**L**,**E**,**R**,**S**, called the *clers stream*. It traverses spirally the dual graph of a surface in order to generate a spanning tree. At each step, a decision is made to move from one triangle t to an adjacent triangle t' through an edge e' called the *gate*. The vertex v of t not contained in the previous gate e is called the *apex* of the gate. This decision depends on the previously visited triangles, which are marked together with their incident vertices.

Right–first traversal. The spiral traversal means that the next triangle is chosen to be the one on the right if not marked, where the right triangle means that the link of the new gate e' contains the vertex next to the apex v of the previous gate e (see section 5.a(iv) *Simplicial Manifolds* for the definition of next). This gives a direct construction of the dual spanning tree and an order on it.

CLERS codes. The traversal is then encoded by the valences (1, 2 or 3) of the nodes of the dual spanning tree $\mathcal{S}^{21}$, and for the valence 2 case, by the current position ($\varnothing$, left or right) of the primal remainder $\mathcal{S}^{01}$ with respect to the new triangle. The corresponding symbols are stated on Table 1.

	operator	$\mathcal{S}^{21}$ val.	$\mathcal{S}^{21}$ pos.	apex	left tri.	rig
C	attach	2	$\varnothing$	unmarked	unmarked	unı
R	MTE	2	left	marked	unmarked	m
L	MTE	2	right	marked	marked	unı
E	2–handle	1		marked	marked	m
S	1–handle	3		marked	unmarked	unı

Table 1: *The CLERS codes.*



(a) **P** (b) **C** (c) **R** (d) **E**

Figure 20: *Coding of a tetrahedron:* **PCRE**.

The valence of the nodes of $\mathcal{S}^{21}$ can be easily detected during the traversal, using the rules of Table 1 [57].

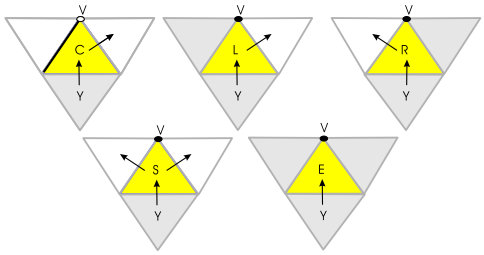


Figure 19: *The* Edgebreaker *encoding. A* **C** *corresponds to a vertex* **C***reation. With the outward orientation, an* **L** *means that the* **L***eft triangle has been visited, whereas an* **R** *means that the* **R***ight triangle has been visited.* **S** *stands for* **S***plit, and* **E** *for* **E***nd.*

Original compression. We will now describe directly the above formal presentation of the Edgebreaker. The algorithm starts by encoding the geometry of a first triangle, that will be the root of $\mathcal{S}^{21}$. In the text, we will call it a **P** triangle. It corresponds to a 0–handle Handle operator. The traversal begins right after with the rules of Table 1: if the apex is not marked, a **C** is encoded with the geometry of the apex, and the traversal continues on the right triangle. Otherwise, if the left triangle is marked, an **R** symbol is encoded and the traversal continues on the right triangle. Similarly, if the right triangle is marked, an **L** symbol is encoded and the traversal continues on the left triangle. If none of the triangles are marked (but the apex is), an **S** symbol is encoded. The traversal splits since the spanning tree has a branching here. The first traversed branch begins with the right triangle, and continues on the left one when the first branch ends. Finally, if both adjacent triangles are marked, the branch ends with an **E** symbol. This branching mechanism can be simply implemented with an **S** stack that stores the left triangle of each **S** triangle.

(b) Fast decompression

Wrap&Zip decompression. The original Wrap&Zip procedure of [58] decodes the clers stream in two passes. The Wrap simply decodes the dual spanning tree, with the geometry of each vertex at each **C** symbol. It decodes the **S**/**E** branchings and positions correctly the adjacent triangles using the branching order and the distinction between the **C** or **L** symbols and the **R** symbols. Then, the Zip procedure completes this spanning tree to obtain the dual graph. If the surface has the topology of a sphere, then there is enough information to recover the entire dual graph, as we will see next. The procedure is very similar to the enumeration of [55]: it looks for the star of each vertex v, and if its star is not closed, and if the two bounding edges of its star are associated to a **C** on one side, and on another symbol on the other side, then these two edges are identified. A recursive implementation of this procedure is necessary to achieve a linear complexity, using the fact that the closure of a star usually allows closing adjacent stars, except when reaching an **L** or **E** symbol.

Fast Zip. Actually, the Zip procedure is a recursive traversal of the dual spanning tree, and it closes the stars from the leaves to the root. Actually, since the algorithm just built the spanning tree, there is no need to traverse it all to find the leaves. It is sufficient to use a **C** stack during the Wrap that stores each **C** triangle. Popping the **C** stack reads it in the reverse way, and the algorithm closes one star at each **C** symbol, and three for each **P** symbol, instead of trying all triangles. This spares half of the tests. Moreover, stars can be closed at some **R** and **E** symbols during the Wrap. This can be used to keep the size of the **C** stack small, and allows a better usage of the multiway geometry prediction of [14].

(c) Topology encoding

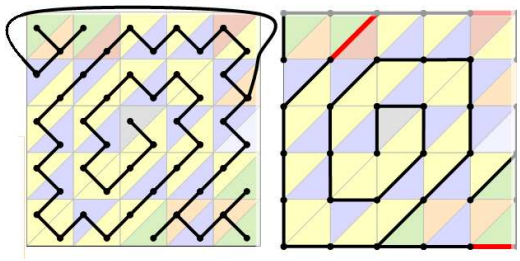
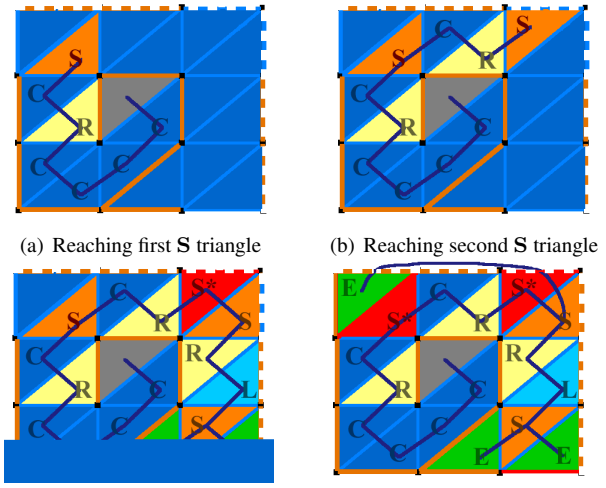


Figure 21: *Dual tree generated by the* Edgebrealer *traversal and the primal remainder, with the two Morse edges in red.*

Handle* $\mathbf{S}^h$ *symbols. As we said earlier, if the surface $\mathcal{S}$ has genus $\mathfrak{g}(\mathcal{S}) > 0$, the primal remainder $\mathcal{S}^{01}$ is not a tree anymore, as illustrated on Figure 21. For a surface with an empty boundary, $\mathcal{S}^{01}$ has $2\cdot\mathfrak{g}(\mathcal{S})$ cycles. These cycles can be simply detected during the traversal and efficiently encoded using [48], while preserving the original Edgebreaker compression scheme. These cycles correspond to a branching,

and thus to an **S** symbol. However, the two branchings induced by each genus of the surface loops back, and the left edge of the **S** triangle is visited before its right branch ends. During the execution, this is easily detected when popping the **S** stack containing the triangles left to **S** symbols: if the top of the **S** stack is not marked, the algorithm continues as normally. If the left triangle was marked, the **S** symbol actually corresponds to a handle, and will be marked as a *handle* $\mathbf{S}^h$ symbol. This symbol is encoded as a normal **S**, and special information identifying this $\mathbf{S}^h$ symbol is encoded in the *handle data*. In order to decompress handles directly, the position of the left triangle in the clers stream can be encoded, for example by the number of **S** symbol that preceded the $\mathbf{S}^h$ symbol and by the number of **R**, **L** and **E** symbols that preceded the left triangle, since *handle* $\mathbf{S}^h$ triangles are obviously closed by only these kind of triangles. These numbers can be encoded by differences to spare even more space.



(a) Reaching first **S** triangle

(b) Reaching second **S** triangle

(c) The lower–right **E** triangle closes the handle.

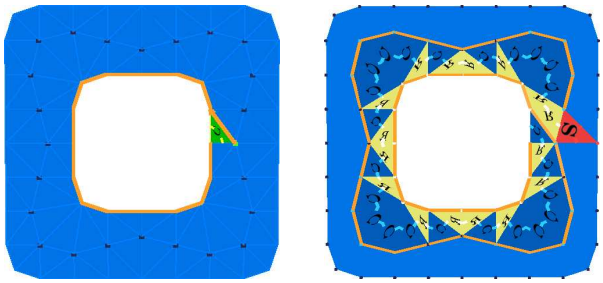
(d) The upper–left **E** triangle closes the handle.

Figure 22: *Coding of a torus: the creation of two* handle **S** *triangles: the first and the second* **S** *symbols.*

Example. To illustrate the algorithm, consider the triangulated torus of Figure 22, where the edges on the opposite sides of the rectangle are identified. This simplicial complex can be embedded in $\mathbb{R}^3$. The Edgebreaker compression algorithm encodes the connectivity of the mesh though the following clers stream: **CCCCRCSCRSSRLSEEE**, completed with the following handle data: $0\text{—}4^-, 0\text{—}3^+$. There are four triangles labelled with an **S** symbol. The left triangles of the two last ones are visited when popping the **S** stack. On the contrary, the two first ones are visited before the being popped out of the **S** stack. These two triangles are detected as handle $\mathbf{S}^h$ symbols. This is encoded in the handle data as follows: the first handle $\mathbf{S}^h$ symbol is also the first **S** symbol, and the first number encoded is therefore 0. There are four possible matches (**R**, **L** and **E** symbols) for its left triangle before the good one, which is encoded by the 4. Since it is an **E** triangle, it can be glued on both sides, and the left side is indicated by the $\epsilon = ^-$. The encoding is done the same way for the second handle $\mathbf{S}^h$ symbol.

First bounding curve. This scheme can be extended to boundary compression, since they correspond to the same Handle operators. Using the handle data to encode boundaries is then more coherent, gives a direct reading of the surface topology through this handle data even before decoding the mesh, and allows a specific prediction scheme for boundaries. Consider first a connected surface $\mathcal{S}$ with one bounding curve. Suppose that we close it by adding a face incident to each bounding edge of $\mathcal{S}$, called the *infinite face*. The resulted surface $\mathcal{S}^+$ has no boundary, and can almost be encoded by the previous algorithm. However, the infinite face is not a triangle. In the same way that the **P** triangles are not explicitly encoded, we will not encode this infinite face, and start the compression directly one of its adjacent triangle. As in the original Edgebreaker algorithm, we encode and mark first all its vertices, e.g., all the vertices belonging to the boundary of $\mathcal{S}$. Then, for the first boundary, we only need to know if the surface component has a boundary or not.

Boundary $\mathbf{S}^b$ ***symbols.*** Now, consider a connected surface has more than one bounding curve. Then, we distinguish arbitrarily one of them as the first boundary and the encoding uses the technique of the last paragraph. During the traversal, we label each triangle touching a new bounding curve as a boundary $\mathbf{S}^b$ triangle. As for handles, we encode it as a normal **S** symbol in the clers handle, and specify that it is a boundary $\mathbf{S}^b$ symbol in the handle data. To distinguish with handle $\mathbf{S}^h$ symbols, their first number is negative. Also, due to the orientation of the bounding curve, the left triangle is always glued on its left side, and we do not need to specify the last $\epsilon = ^+$ or $\epsilon = ^-$, and we can avoid counting the **L** symbols to localise it. From the Euler characteristic, we know that there is exactly one boundary $\mathbf{S}^b$ symbol per bounding curve. On Figure 23, the only *handle* **S** triangle is



(a) The first triangle is chosen adjacent to a boundary. The vertices of the central infinite face are encoded.

(b) An unmarked boundary is reached: the corresponding **S** triangle is a *boundary* **S** triangle.

Figure 23: *Coding of an annulus: initialisation and creation of* boundary **S** *triangles.*

the first triangle with a vertex on the internal boundary that we encounter during the traversal. As said before, there are

$2 \cdot \mathfrak{g}(\mathcal{S}) + \mathfrak{b}(\mathcal{S}) - 1$ such *handle* $\mathbf{S}$ triangles for each surface component with genus $\mathfrak{g}(\mathcal{S})$ and $\mathfrak{b}(\mathcal{S})$ bounding curves.

Multiple components. The compression processes successively each surface component. When the component has no boundary, the compression encodes explicitly the vertices of the first triangle (uncoded $\mathbf{P}$ symbol). Otherwise, it encodes the vertices of the first bounding curve. In practise, we only need to transmit the number of components with boundary of $\mathcal{S}$. Then we transmit first all the components with a non–empty boundary, and then the other ones.

(d) Compression algorithms

The compression scheme then decomposes in handling the multiple components and their first boundaries (Algorithm 4: compress), compress each component by the dual spanning tree traversal (Algorithm 3: traverse). The handles are tested along the traversal with Algorithm 5: check handle. The whole process is linear and performed in one pass only.

(e) Decompression algorithms

The decompression is performed in three passes, controlled by Algorithm 7: decompress. The first pass decodes the dual spanning tree (Algorithm 8: wrap), which is further zipped using the backward sequence of $\mathbf{C}$ symbols (Algorithm 9: fast zip). The compression described here encodes boundary curves, which improves prediction for the interior. However this means that the size of the boundary is not known to the decoder at the first pass, and the geometry must be decoded in a posterior step (Algorithm 10: read geometry). This pass could be done at the wrap stage if we encode the boundaries when they are closed, or if we encode the geometry of the bounding curves in a separate stream.

9 Performances

We presented in this section the fundamental concepts of connectivity–driven compression. In particular, we focused on an extension of the Edgebreaker algorithm, which handles manifold surfaces of arbitrary topology. The complexity of the compression and the decompression are both linear in execution time and memory footprint, independently of the maximal number of the active elements during the execution. However, the decompression still requires two passes, which makes it harder to stream.

There are various ways of representing a geometrical object, even for simplicial surfaces. For specific type of meshes, some algorithms show better performances than other ones. This distinction is one of the main shifts from the MPEG compression [19] to the MPEG–4 one [54], which for example encodes differently human faces than landscapes. Although it is difficult to distinguish with precision classes of meshes and to predict exactly the behaviour of compression algorithms on these, we will try to get an intuition of which characteristics of a mesh are well suited for

Algorithm 3 traverse(t): encode one component starting from triangle t

```
1:  stack Sstack ← ∅        // stack of the triangles left to S symbols
2:  repeat
3:    t.mark ← true          // mark current triangle
4:    v ← t.apex             // orient the triangle from its apex
5:    if v.mark = false then                          // C triangle
6:      write vertex (v)     // encode the geometry of v
7:      v.mark ← true        // mark the vertex
8:      write symbol (C)     // encode the clers code: C
9:      t ← t.right          // spiral traversal to the right
10:   else if is boundary(t.right) or t.right.mark then  // right triangle visited
11:     if is boundary(t.left) or t.left.mark then      // E triangle
12:       write symbol (E)   // encode the clers code: E
13:       check handle (t)   // check if it is the left triangle of a S^h triangle
14:       repeat
15:         if Sstack = ∅ then   // end of compression
16:           return             // exit the external repeat loop
17:         t ← Sstack.pop       // pop the S stack
18:       until not t.mark       // skip left of a handle S^h triangle
19:     else                                            // R triangle
20:       write symbol (R)   // encode the clers code: R
21:       check handle (t)   // check if it is the left triangle of a S^h triangle
22:       t ← t.left         // break in spiral traversal: to the left
23:   else if is boundary(t.left) or t.left.mark then   // L triangle
24:     write symbol (L)     // encode the clers code: L
25:     check handle (t)     // check if it is the left triangle of a S^h triangle
26:     t ← t.right          // spiral traversal to the right
27:   else                                              // S triangle
28:     write symbol (S)     // encode the clers code: S
29:     if is boundary(v) then   // boundary S^b triangle
30:       write boundary (t)     // encode boundary
31:       t.mark ← −#S           // mark for the handle data
32:     else                     // normal S or handle S^h triangle
33:       t.mark ← #S            // mark for the handle data
34:     Sstack.push (t.left)     // push the left triangle on the S stack
35:     t ← t.right              // spiral traversal to the right
36: until true                                          // infinite loop
```

Algorithm 4 compress($\mathcal{S}$): compress separately each component of $\mathcal{S}$

```
1: 𝔟* ← 0 // counts number of components with boundary
2: for all vertices v ∈ S do // reset marks
3:   v.mark ← is boundary (v) // mark boundary vertices
4: for all triangles t ∈ S do // compress components with boundary first
5:   if not t.mark and is boundary(t) then // not boundary or already encoded
6:     write boundary (t) // encode boundary
7:     traverse(t) // component compression
8:     𝔟* ← 𝔟* + 1 // one more component with boundary
9: for all triangles t ∈ S do // compress the other components
10:  if not t.mark then // not already encoded
11:    t.mark ← true // mark P triangle
12:    for all vertices v ∈ ∂t do // encode the 3 vertices of the P triangle
13:      write vertex (v) // encode the geometry of v
14:      v.mark ← true // mark the vertex
15:    traverse(t.right) // component compression
16: write(handle, 𝔟*) // write the number of components with boundary
```

Algorithm 5 check handle(t): check if triangle t is left to a $\mathbf{S}^h$ triangle

```
1: if not is boundary(t.right) and t.right.mark ∉ {true, false} then // handle S^h triangle to the right
2:   write(handle, t.right.mark − #^+_RE) // write the handle data
3: if not is boundary(t.left) and t.left.mark ∉ {true, false} then // handle S^h triangle to the left
4:   write(handle, t.left.mark − #^−_LE) // write the handle data
```

Algorithm 10 read geometry($Cstack'$): decompress the geometry

```
1: while Cstack' ≠ ∅ do // traverse the stack
2:   t ← Cstack'.pop () // pop the next element of the C stack
3:   if t ⩾ 0 then // not a boundary triangle
4:     read vertex(t) // read a new vertex
5:   else // boundary triangle
6:     read boundary(t) // read a new bounding curve
```

Algorithm 7 decompress(streams): decompress separately each component

```
1: repeat
2:   s − t^ε ← read(handle) // read handle data
3:   if s > 0 then // handle S^h symbol
4:     glue(s, t, ε) // glue the handle on side ε before the decompression
5:   else // boundary S^b symbol
6:     glue(−s, t, ⁻) // close the bounding curve before the decompression
7: until end of file(handle) // passed the last couple of data
8: 𝔟* ← s // last handle data counts number of components with boundary
9: stack Cstack ← ∅ // stack of the C and boundary S^b triangles
10: wrap(𝔟*, Cstack) // wrap using the clers stream
11: fast zip(Cstack) // closes the stars of the primal remainder
12: read geometry(Cstack) // reads the geometry of the surface
```

Algorithm 9 fast zip($Cstack$): decompress one primal remainder

```
1: stack Cstack' ← ∅ // reverse copy of the C stack for the geometry
2: while Cstack ≠ ∅ do // traverse the stack
3:   t ← Cstack.pop () // pop the next element of the C stack
4:   Cstack'.push (t) // copy the C stack
5:   if t ⩾ 0 then // not a boundary triangle
6:     close star(t) // close the star of the next vertex
7:   Cstack ← Cstack' // returns the copy of the C stack
```

Model	$\lvert\#_0\rvert$	$\lvert\#_2\rvert$	[58, 34]	[57]	[40]	[57]/[40]	[58, 34]/[40]
sphere	1 848	926	3.39	3.39	3.45	0.98	0.98
violin	1 508	1 498	3.16	2.21	2.25	0.98	1.41
pig	3 560	1 843	3.26	3.24	3.13	1.03	1.04
rose	3 576	2 346	3.37	2.95	2.64	1.12	1.28
cathedral	1 434	2 868	2.25	1.00	0.19	5.27	11.86
blech	7 938	4 100	3.25	3.18	2.40	1.33	1.35
mask	8 288	4 291	3.19	3.12	1.93	1.62	1.65
skull	22 104	10 952	3.51	3.51	3.30	1.06	1.06
bunny	29 783	15 000	3.36	3.34	1.27	2.62	2.64
terrain	32 768	16 641	3.03	3.00	0.40	7.43	7.51
david	47 753	24 085	3.45	3.85	3.07	1.25	1.12
gargoyle	59 940	30 059	3.28	3.27	2.11	1.55	1.55

Table 2: *Comparative results on different models drawn on Figure 26. 'Dum' stands for the dummy vertex method to encode meshes with boundaries [58, 34], and 'Ori' stands for the original* Edgebreaker *[57], and [40] for the algorithm introduced here, with the simple arithmetic coder of [50]. The size of the compressed symbols is expressed in bit per vertex.*

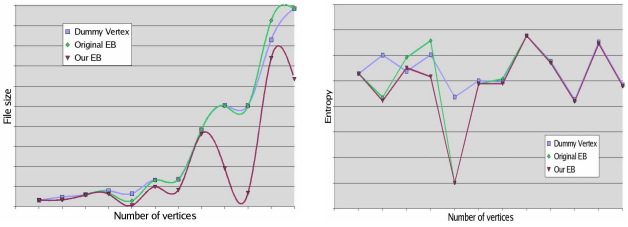
Algorithm 8 wrap($\mathfrak{b}^*, Cstack$): decompress the dual trees

```
1:  #2 ← 0                                              // initialisation
2:  stack Sstack ← ∅        // stack of the triangles left to S symbols
3:  repeat                                              // components loop
4:    if b* > 0 then                          // component with boundary
5:      b* ← b* − 1; t ← ∅                              // first boundary
6:      Cstack.push (−#2)          // push the boundary triangle for the geometry
7:    else                          // component with an empty boundary
8:      t ← #2                                          // P triangle
9:      Cstack.push (t)    // push the first triangle for the zip
10:     for all vertices v ∈ ∂t do // decode the 3 vertices of the P triangle
11:       read vertex (v)          // decode the geometry of v
12:     t ← t.right                // spiral traversal to the right
13:     #2 ← #2 + 1                                     // initialisation
14:   repeat                                  // decompress one component
15:     glue(t, #2)   // glue the next triangle eventually to the boundary
16:     s ← read symbol (clers) // reads the next symbol
17:     if s = C then                                   // C triangle
18:       Cstack.push (#2)      // push the C triangle for the zip
19:       t ← t.right // orient the new triangle to the right
20:     else if s = R then                              // R triangle
21:       t ← t.left     // orient the new triangle to the left
22:       tryclose star (t.apex)        // eventually zip the right edge
23:     else if s = L then                              // L triangle
24:       t ← t.right // orient the new triangle to the right
25:     else if s = S then                              // S triangle
26:       if not t.right.mark then          // not a handle or boundary S symbol
27:         Sstack.push (#2.left)  // push the S triangle for the next E
28:       else if is boundary(#2) then          // boundary triangle
29:         Cstack.push (−#2)      // push the boundary triangle for the geometry
30:       t ← t.right // orient the new triangle to the right
31:     else if s = E then                              // E triangle
32:       tryclose star (t.apex)        // eventually zip the right and left edges
33:       if Sstack = ∅ then      // end of the component
34:         break                  // exits the component loop
35:       t ← Sstack.pop ()      // pop the next element of the S stack
36:     #2 ← #2 + 1                                     // next triangle
37:   until true                                        // infinite loop
38: until end of file(clers)                // end of the clers stream
```

connectivity–driven compression schemes, and in particular for the Edgebreaker.

(a) Compression Rates

Experimental results for the Edgebreaker are recorded on Table 2 and Figure 24. We compared with the original Edgebreaker implementation with the Huffman encoding of [34] and the border handling of [58], and the encoding of [40] with the simple arithmetic coder of [50]. The entropy of the codes of [40] is always better than the other implementations of Edgebreaker, as shown on Figure 24(b). A compression ratio of a few bits per vertex, or even less, is a general order for efficient connectivity–driven compression schemes.



(a) Size of the compressed file vs complexity of the model.

(b) Entropy vs complexity of the model.

Figure 24: *Comparison of the final size and entropy: for the range encoder, those parameters depends more on the regularity than on the size of the model.*

(b) Good and bad cases

	Edgebreaker	Valence coding
Topology–dependent	+ + + [40]	− − −
Regular valence	− − −	+ + + [66]
Lossy connec.	+ [7]	+ + + [2]
Self–similar connec.	+ + + [57]	+ [33]
Irregular connec.	+ [34]	+ + + [10]
Geometric connec.	+ + + [15]	+ [38]
Geometry prediction	+ [42]	+ + + [14]
Low resource use	+ + + [59]	− − −

Table 3: *Good and bad cases for the two main connectivity–driven compression schemes.*

Topology–dependent applications. For the extended Edgebreaker of [40], the separate handle data informs directly the application of the topology of the mesh. Many simple parameterisations, texturing or remeshing applications work only for closed surfaces without handle. The handle data can be used to call a preprocessing step for simplifying the topology before using these kind algorithms. For the Edgebreaker, this handle data is not an overhead, since encoding the handle and boundary **S** symbols as a true/false code on the clers string is in the best case logarithmic as we saw in section 3(c) *Statistical Modelling*, which is equivalent to the handle data.

Regular connectivity. The valence coding of [66, 33] encodes particularly well meshes where the vertices have a uniform valence. This can be obtained by subdivision [44, 68] or remeshing [3, 4]. Remeshing can be done also to improve the Edgebreaker compression using the Swingwrapper of [7]. Without these regularisations, valence coding based algorithms have better performance when the connectivity is locally regular, whereas the Edgebreaker performs better on irregular meshes or meshes with a global regularity, such as those obtained by subdivision algorithms or with some self–similar connectivity. Meshes with a very irregular connectivity would be better encoded by enumeration methods of [55, 10].

Regular geometry. The geometry of the mesh is not directly considered in connectivity–driven compression, and therefore geometry–based compression will outperform these schemes for the connectivity compression of meshes with a regular geometry. However, the geometry can be used to predict the connectivity, which works specifically when the geometry is regular. This has been done for the valence coding in [2, 38] and in [15] for the Edgebreaker.

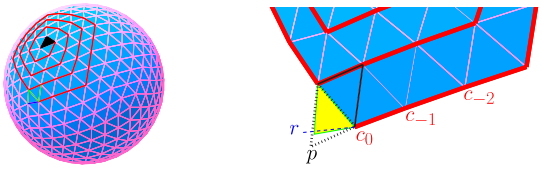


Figure 25: *The Edgebreaker cuts the compressed surface along a curve in the space. An extrapolation of this curve is used to enhance the parallelogram predictor. The predictor uses the parallelogram predictor to guess the distance from the last vertex of the curve, and rotates this estimation according to the approximating curve.*

Geometry prediction. Geometry prediction uses already decoded vertices to estimate the next vertex to be decoded, asserting that the geometry is locally regular. For connectivity–driven schemes are usually based on the parallelogram predictor of [66]. It can be enhanced by using more than one parallelogram to estimate the new position, as described in [14]. This is particularly well adapted to the valence coding since the traversal can be adapted to the prediction. For the Edgebreaker, the parallelogram can be distorted to adapt to local mean curvature of the surface, as in [38], or to torsion and curvature of the primal remainder, as described in [42] and on Figure 25.

Low resource applications. The Edgebreaker uses a deterministic traversal, independent of geometry considerations. Although this is less flexible for geometry prediction enhancements, it gives a very simple algorithm. Moreover, compared to the valence coding schemes that needs to maintain sorted active boundaries along compression and decompression, the Edgebreaker just needs a stack of past **S** symbols. The Edgebreaker thus requires much less memory for the execution, and spares a constant sort, which can become expensive. More generally, connectivity–driven compression schemes are easy to implement and quick to execute.

The above results are roughly summarised on table 3, as a general appreciation from the author.

10 Next steps

The diversity of images requires a multiplicity of compression programs, since specific algorithms usually perform better than generic one (such as the popular Zip method), if they are well adapted. In particular, the simple example of Edgebreaker can be extended in many ways, to address specific issues of particular applications. Even with the few notions of this tutorial, it is feasible to improve the state-of-the art in 3D compression. In particular, the following directions may be promising:

Non–simplicial meshes. Connectivity–driven compression schemes are easier on simplicial meshes, since the dual graph has a constant valence. Most of the mesh compression algorithms for polytope surfaces can be interpreted as a simplicial encoding preceded by a triangulation of each face. This triangulation is done in a canonical way from the traversal, and the decoder just need to know the degree of the triangulated faces. For example, the valence coding can be extended by encoding simultaneously the vertex valences and the face degrees, as in [2], and the Edgebreaker codes can be combined in a predictable way using the codes of [35].

Non–manifold meshes. Extending these methods to non–manifold meshes directly is a hard task. The usual method consists in cutting the non–manifold surface into manifold pieces, using the techniques of [21], encoding the manifold parts as separate components, and then encoding the cut operations that were performed. The encoding of cut operations can be done directly as in the handle data, or more carefully by propagating the curves formed by the non–manifold edges.

Higher dimensions. For solid meshes, the Edgebreaker compression has been directly extended to tetrahedral meshes in [23, 63], and the valence coding has been extended in [31]. The principles are the same, but the encoding needs some extra information to complete the intermediate dimension between the spanning tree and the remainders.

This extra information has necessarily some expensive parts to encode, similar to the handle S symbols that are necessary to glue distant parts of the traversal. Minimising this extra information is an NP–hard problem, as proved in [41]. For higher dimensions, the combinatory of mesh connectivity makes it difficult to find a concise set of symbols for coding, or a good statistical model for them as was done for surfaces in [34]. However, for high codimensions, the connectivity remains simple while the geometry can be efficiently predicted. Seen from the other side, this means that for low codimension, geometry–based coding can be very efficient, which is where isosurface compression outperforms any connectivity–based compression.

Robustness. The Edgebreaker is robust in the sense that it handles general manifold surfaces. However, it is not particularly robust with a noisy transmission, where the clers codes can be altered. In that case, the grammar inherent to these codes can be used to detect transmission errors, but not directly to correct them.

Deformable meshes. For animation purposes, the Edgebreaker can be used directly to compute the deformed mesh when its connectivity is constant, and using for example [62] to interpolate the geometry. Local changes in the connectivity can be further encoded using the explicit identification of vertices and triangles provided by the Edgebreaker, similarly to the description of [69].

Acknowledgments

The authors would like to thank the organization of the Sibgrapi for the opportunity of presenting this wok, and CNPq for financial support during the preparation of this paper, through project MCT/CNPq 02/2006.

References

[1] J. W. Alexander. The combinatorial theory of complexes. *Annals of Mathematics*, 31:219–320, 1930.

[2] P. Alliez and M. Desbrun. Valence–driven connectivity encoding of 3D meshes. In *Eurographics*, pages 480–489. Blackwell, 2001.

[3] P. Alliez, D. Cohen–Steiner, O. Devillers, B. Levy and M. Desbrun. Anisotropic polygonal remeshing. In *Siggraph*. ACM, 2003.

[4] P. Alliez, É. Colin de Verdière, O. Devillers and M. Isenburg. Isotropic surface remeshing. In *Shape Modeling International*. IEEE, 2003.

[5] F. W. Wheeler. Adaptive arithmetic coding source code.

[6] M. A. Armstrong. *Basic topology*. McGraw–Hill, London, 1979.

[7] M. Attene, B. Falcidieno, M. Spagnuolo and J. Rossignac. SwingWrapper: retiling triangle meshes for better Edgebreaker compression. *Transactions on Graphics*, 22(4):982–996, 2003.

[8] B. G. Baumgart. Winged edge polyhedron representation. Technical Report AIM-179 (CS-TR-74-320), Stanford University, 1972.

[9] J.-D. Boissonnat and M. Yvinec. *Algorithmic geometry*. Cambridge University Press, 1998.

[10] L. Castelli Aleardi and O. Devillers. Canonical triangulation of a graph, with a coding application. *INRIA*, 2004.

[11] L. Castelli Aleardi, O. Devillers and G. Schaeffer. Succinct representation of triangulations with a boundary. In *Workshop on Algorithms and Data Structures*, volume 5608, pages 134–145, 2005.

[12] L. Castelli Aleardi, O. Devillers and G. Schaeffer. Dynamic updates of succinct triangulations. In *Canadian Conference on Computational Geometry*, pages 135–138, 2005.

[13] C. Christopoulos, A. Skodras and T. Ebrahimi. The JPEG2000 still image coding system: An overview. *Transactions on Consumer Electronics*, 46(4):1103–1127, 2000.

[14] D. Cohen–Or, R. Cohen and T. Ironi. Multi–way geometry encoding. Technical report, Tel Aviv University, 2001.

[15] V. Coors and J. Rossignac. Delphi: geometry-based connectivity prediction in triangle mesh compression. *The Visual Computer*, 20(8–9):507–520, 2004.

[16] M. F. Deering. Geometry compression. In *Siggraph*, pages 13–20. ACM, 1995.

[17] C. M. Eastman. Introduction to computer aided design, 1982. Course Notes. Carnegie-Mellon University.

[18] L. de Floriani and A. Hui. A scalable data structure for three–dimensional non–manifold objects. In *Symposium on Geometry Processing*, pages 72–82. ACM/Eurographics, 2003.

[19] D. le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, 1991.

[20] P.-M. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. In *Siggraph*, volume 21, pages 372–379. ACM, 2002. Siggraph.

[21] A. Guéziec, G. Taubin, F. Lazarus and W. P. Horn. Converting sets of polygons to manifold surfaces by cutting and stitching. In D. Ebert, H. Hagen and H. Rushmeier, editors, *Visualization*. IEEE, 1998.

[22] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *Transactions on Graphics*, 4:74–123, 1985.

[23] S. Gumhold, S. Guthe and W. Straßer. Tetrahedral mesh compression with the Cut–Border machine. In *Visualization*, pages 51–58. IEEE, 1999.

[24] N. A. Gumerov, R. Duraiswami and E. A. Boroviko. Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in d dimensions. Technical report, University of Maryland, 2003.

[25] R. W. Hamming. Error-detecting and error-correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[26] R. V. L. Hartley. Transmission of information. *Bell System Technical Journal*, 7:535, 1928.

[27] A. Hatcher. *Algebraic topology*. Cambridge University Press, 2002.

[28] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson and K. Museth. Gigantic deformable surfaces. In *Siggraph Sketches*. ACM, 2005.

[29] D. A. Huffman. A method for the construction of minimum redundancy codes. In *I.R.E*, pages 1098–1102, 1952.

[30] M. Isenburg and J. Snoeyink. Spirale reversi: reverse decoding of the Edgebreaker encoding. In *Canadian Conference on Computational Geometry*, pages 247–256, 2000.

[31] M. Isenburg and P. Alliez. Compressing hexahedral volume meshes. In *Pacific Graphics*, pages 284–293. IEEE, 2002.

[32] M. Isenburg and S. Gumhold. Out–of–core compression for gigantic polygon meshes. *Transactions on Graphics*, 22(3):935–942, 2003.

[33] F. Kälberer, K. Polthier, U. Reitebuch and M. Wardetzky. Freelence — coding with free valences. In *Eurographics*, volume 24, pages 469–478. Blackwell, 2005.

[34] D. King and J. Rossignac. Guaranteed 3.67v bit encoding of planar triangle graphs. In *Canadian Conference on Computational Geometry*, pages 146–149, 1999.

[35] B. Kronrod and C. Gotsman. Efficient coding of nontriangular mesh connectivity. *Graphical Models*, 63:263–275, 2001.

[36] M. Lage, T. Lewiner, H. Lopes and L. Velho. CHE: a scalable topological data structure for triangular meshes. Technical report, Department of Mathematics, PUC–Rio, 2005.

[37] M. Lage, T. Lewiner, H. Lopes and L. Velho. CHF: a scalable topological data structure for tetrahedral meshes. In *Sibgrapi*, pages 349–356, Natal, Oct. 2005. IEEE.

[38] H. Lee, P. Alliez and M. Desbrun. Angle-analyzer: A triangle-quad mesh codec. In *Eurographics*, volume 21. Blackwell, 2002.

[39] A. Lempel and J. Ziv. A universal algorithm for sequential data compression. *Transactions on Information Theory*, 23(3):337–343, 1977.

[40] T. Lewiner, H. Lopes, J. Rossignac and A. W. Vieira. Efficient Edgebreaker for surfaces of arbitrary topology. In *Sibgrapi*, pages 218–225, Curitiba, Oct. 2004. IEEE.

[41] T. Lewiner, H. Lopes and G. Tavares. Applications of Forman's discrete Morse theory to topology visualization and mesh compression. *Transactions on Visualization and Computer Graphics*, 10(5):499–508, 2004.

[42] T. Lewiner, J. Gomes Jr., H. Lopes and M. Craizer. Curvature and torsion estimators based on parametric curve fitting. *Computers & Graphics*, 2005.

[43] M. Li and P. M. B. Vitanyi. *An introduction to Kolmogorov complexity and its applications*. Springer, 1997.

[44] C. T. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, *University of Utah*, 1987.

[45] H. Lopes. Algorithms to build and unbuild 2 and 3 dimensional manifolds. PhD thesis, *Department of Mathematics, PUC–Rio*, 1996. Advised by Geovan Tavares.

[46] H. Lopes and G. Tavares. Structure operators for modeling 3 dimensional manifolds. In C. Hoffman and W. Bronsvort, editors, *Solid Modeling and Applications*, pages 10–18. ACM, 1997.

[47] H. Lopes, J. Rossignac, A. Safonova, A. Szymczak and G. Tavares. Edgebreaker: a simple compression for surfaces with handles. In C. Hoffman and W. Bronsvort, editors, *Solid Modeling and Applications*, pages 289–296, Saarbrücken, 2002. ACM.

[48] H. Lopes, S. Pesco, G. Tavares, M. G. M. Maia and Á. Xavier. Handlebody representation for surfaces and its applications to terrain modeling. In *Shape Modeling International*, volume 9. IEEE, 2003.

[49] M. Mäntylä. *An introduction to solid modeling*. CS Press, Rockville, 1988.

[50] G. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video & Data Recoding*, 1979.

[51] A. Moffat, R. Neal and I. H. Witten. Arithmetic coding revisited. In *Data Compression*, pages 202–211, 1995.

[52] J. R. Munkres. *Elements of algebraic topology*. Addison-Wesley, Menlo Park, 1984.

[53] H. Nyquist. Certain topics in telegraph transmission theory. *Transactions of the American Institute of Electrical Engineers*, 47:617–644, 1928.

[54] F. Pereira and T. Ebrahimi. *The MPEG–4 Book*. Prentice Hall, Upper Saddle River, 2002.

[55] D. Poulalhon and G. Schaeffer. Optimal coding and sampling of triangulations. In *ICALP*, pages 1080–1094, 2003.

[56] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, 1976.

[57] J. Rossignac. Edgebreaker: connectivity compression for triangle meshes. *Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[58] J. Rossignac and A. Szymczak. Wrap&zip decompression of the connectivity of triangle meshes compressed with edgebreaker. *Computational Geometry*, 14(1-3):119–135, 1999.

[59] J. Rossignac, A. Safonova and A. Szymczak. 3D compression made simple: Edgebreaker on a corner–table. In *Solid Modeling International*, pages 278–283. IEEE, 2001.

[60] D. Salomon. *Data compression: the complete reference*. Springer, Berlin, 2000.

[61] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.

[62] O. Sorkine, D. Cohen–Or and S. Toledo. High-pass quantization for mesh encoding. In *Symposium on Geometry Processing*, pages 42–51. ACM/Eurographics, 2003.

[63] A. Szymczak and J. Rossignac. Grow & Fold: compressing the connectivity of tetrahedral meshes. *Computer Aided Design*, 32(8/9):527–538, 2000.

[64] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *Transactions on Graphics*, 17(2):84–115, 1998.

[65] G. Taubin, W. P. Horn, F. Lazarus and J. Rossignac. Geometry coding and VRML. *Proceedings of the IEEE*, 86(6):1228–1243, 1998.

[66] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface*, pages 26–34, 1998.

[67] W. T. Tutte. *Graph theory as I have known it*. Oxford University Press, New York, 1998.

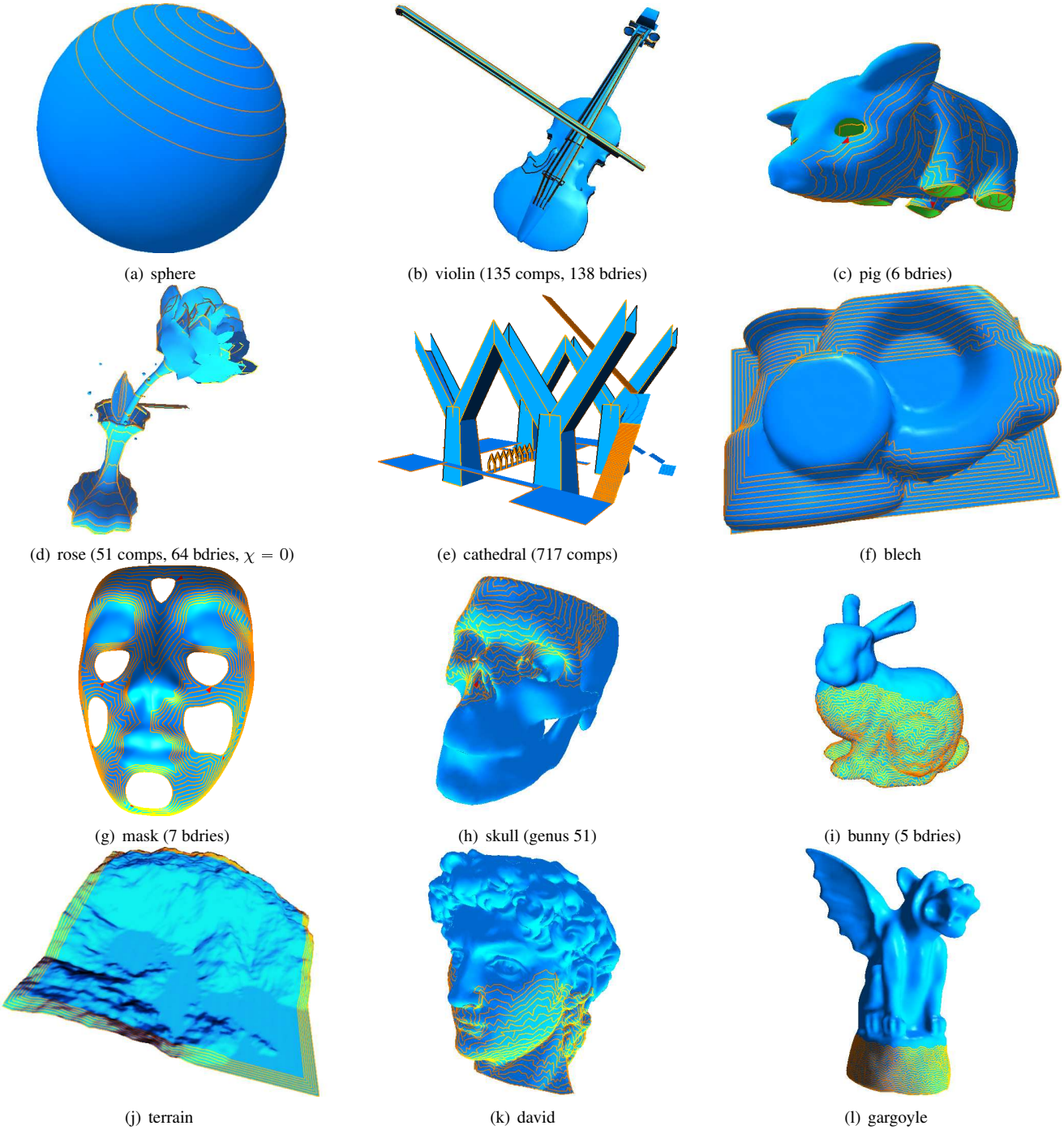
[68] L. Velho and D. Zorin. 4–8 subdivision. *Computer Aided Geometric Design*, 18(5):397–427, 2001. Special issue on Subdivision Techniques.

The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.

[69] A. W. Vieira, T. Lewiner, L. Velho, H. Lopes and G. Tavares. Stellar mesh simplification using probabilistic optimization. *Computer Graphics Forum*, 23(4):825–838, 2004.

[70] M. W. Walker, L. Shao and R. A. Volz. Estimating 3-D location parameters using dual number quaternions. *Computer Vision and Image Understanding*, 54(3):358–367, 1991.

[71] K. J. Weiler. Edge–based data structures for solid modeling in curved–surface environments. *Computer Graphics and Applications*, 5(1):21–40, 1985.



(a) sphere (b) violin (135 comps, 138 bdries) (c) pig (6 bdries)

(d) rose (51 comps, 64 bdries, $\chi = 0$) (e) cathedral (717 comps) (f) blech

(g) mask (7 bdries) (h) skull (genus 51) (i) bunny (5 bdries)

(j) terrain (k) david (l) gargoyle

Figure 26: *Some of the models used for the experiments, with the beginning of* Edgebreaker*'s dual spanning tree: The 'violin' has 135 components and 138 boundaries. The 'rose' has 51 components, genus 1 and 64 boundaries. The cathedral has 717 components with boundary. The 'mask' has 7 boundaries. The 'skull' has genus 51. The bunny has 5 boundaries.*

The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.

Index

Summary of notations

$\mathbb{R}$	set of the real number
$\mathbb{R}^n = \mathbb{R} \times \mathbb{R} \times \ldots \times \mathbb{R}$	Euclidean space of dimension n
$\mathbb{B}^p$	unit ball in $\mathbb{R}^p$
$\mathbb{S}^{p-1}$	unit sphere in $\mathbb{R}^p$
$\mathbb{N}$	sets of the natural and relative integers
$[\![m,n]\!] = \{m, m+1, \ldots, n\}$	integer interval
ρ, σ, τ	simplices
ρ^n, σ^n, τ^n	simplices of dimension n
v	vertices : simplex of dimension 0
e	edge : simplex of dimension 1
t	triangle : simplex of dimension 2
$\sigma^n \succ \tau^m$	σ^n is incident to τ^m
$\partial\sigma = \{\tau, \sigma \succ \tau\}$	frontier of a simplex σ
$K = \{\sigma\}$	simplicial complex
$K^n = \{\sigma^p, p \leqslant n\}$	simplicial complex of dimension n
$\#_m(K) = \#\{\sigma^m \in K\}$	number of m–simplices of K
$\chi(K^n) = \sum (-1)^m \#_{n-m}(K^n)$	Euler–Poincar characteristic of K
$K_{(m)} = \{\sigma^p \in K, p \leqslant m\}$	m–skeleton of K
$\sigma \star \tau = \text{hull}\, \sigma \cup \tau$	join of σ and τ
$\text{lk}(\sigma) = \{\tau \in K : \sigma \star \tau \in K\}$	link of σ
$\dot{\text{st}}(\sigma) = \{\sigma \star \tau, \tau \in \text{lk}(\sigma)\}$	open star of σ
$\text{st}(\sigma) = \dot{\text{st}}(\sigma) \cup \bigcup_{\rho \in \dot{\text{st}}(\sigma)} \partial\rho$	star of σ
$\partial K^n = \left\{\sigma^{n-1} : \#\,\text{lk}\left(\sigma^{n-1}\right) = 1\right\}$	boundary of a pure simplicial complex
$\mathcal{M}^d$	simplicial n–manifold
$\mathcal{S} = \mathcal{M}^2$	triangulated surface
$\chi(\mathcal{S}) = 2 - 2 \cdot \mathfrak{g}(\mathcal{S}) - \mathfrak{b}(\mathcal{S})$	genus and number of boundaries of $\mathcal{S}$

The corresponding work was published in Revista de Informática Teórica e Aplicada, special edition for Sibgrapi tutorials.