Progressive Compression of Triangle Meshes

Vincent Vidal, Lucas Dubouchet, Guillaume Lavoué, Pierre Alliez

Published in Image Processing On Line on 2023–01–04.
Submitted on 2022–07–23, accepted on 2022–11–27.
ISSN 2105–1232 © 2023 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
https://doi.org/10.5201/ipol.2023.418

Progressive Compression of Triangle Meshes

Vincent Vidal[1], Lucas Dubouchet[2], Guillaume Lavoué[3], Pierre Alliez[2]

[1]Univ Lyon, UCBL, CNRS, INSA Lyon, LIRIS, UMR5205, Villeurbanne, France
(vincent.vidal@liris.cnrs.fr)
[2]Université Côte d'Azur, Inria, France
(pierre.alliez@inria.fr)
[3]Univ Lyon, Centrale Lyon, CNRS, INSA Lyon, UCBL, LIRIS, UMR5205, Ecully, France
(guillaume.lavoue@liris.cnrs.fr)

Communicated by Bertrand Kerautret *Demo edited by* Bertrand Kerautret

Abstract

This paper details the first publicly available implementation of the progressive mesh compression algorithm described in the paper entitled "Compressed Progressive Meshes" [R. Pajarola and J. Rossignac, IEEE Transactions on Visualization and Computer Graphics, 6 (2000), pp. 79-93]. Our implementation is generic, modular, and includes several improvements in the stopping criteria and final encoding. Given an input 2-manifold triangle mesh, an iterative simplification is performed, involving batches of edge collapse operations guided by an error metric. During this compression step, all the information necessary for the reconstruction (at the decompression step) is recorded and compressed using several key features: geometric quantization, prediction, and spanning tree encoding. Our implementation allowed us to carry out an experimental comparison of several settings for the key parameters of the algorithm: the local error metric, the position type of the resulting vertex (after collapse), and the geometric predictor.

Source Code

The proposed implementation is publicly available through the MEPP2 platform [20]. The algorithm can be used either as a command-line executable or integrated into the MEPP2 GUI. The source code is written in C++ and is accessible on the IPOL web page of this article[1], as well as on the GitHub page of MEPP2 (MEPP-team/MEPP2 project[2]).

Keywords: triangle mesh compression; 2-manifold mesh; progressive compression

[1]https://doi.org/10.5201/ipol.2023.418
[2]https://github.com/MEPP-team/MEPP2

1 Introduction

The development of computer graphics technologies and scanning devices has lead to a global increase in the complexity and quality of 3D models being created, manipulated, stored, and transmitted over networks. This increase in complexity and quality translates into an augmentation of the number of elements describing these models (mostly represented by surface meshes). For instance, 3D meshes generated by photogrammetric reconstruction now commonly attain several millions of vertices. With the development of Web3D technologies, an increasing number of 3D applications consider data stored on remote servers. As a concrete illustration of this fact, many companies are now proposing VR web browsers since they envision the explosion of online VR applications (video games, virtual museums, virtual courses). Similarly, dire needs of remote data apply to mixed reality (MR), for which an important intrinsically online application is telepresence. For these online applications, strong latency and frame-rate problems may be encountered, mostly due to the constraint of delivering the 3D content from the server to the end-user, and to the imperfect management of heterogeneous transmission networks and heterogeneous visualization devices.

The performance issues raised above can be resolved by the use of compressed multi-scale representations (often referred to as progressive compression techniques [13]). Indeed, progressive compression yields a high compression ratio (and thus fast transmission) and produces different Levels of Detail (LoD), enabling the complexity of the data to adapt to the remote device by stopping the transmission when a sufficient LoD is reached. These functionalities can reduce the time latency even for huge data and enable real-time visualization and interactions (i.e. high frame rate) even for low-end devices (e.g. autonomous HMD, smartphones). With these techniques, users instantly get a coarse version of the mesh which is then progressively refined as more data are decompressed until the initial model has been restored (or an intermediate model according to the device constraints).

Many progressive compression methods for static surface meshes have already been introduced for 20 years. An exhaustive literature review can be found in [13]. Most of the existing approaches only deal with manifold triangle meshes and few can compress either polygonal or non-manifold meshes. There are two main kinds of progressive approaches: *connectivity-based approaches* [10, 15, 1, 12, 3, 17] consider the mesh connectivity to guide the simplification, while *geometry-based approaches* [7, 16] operate on the geometry using a spatial tree structure (octree or kd-tree) to conduct the simplification. In this paper, we describe an implementation of "Compressed Progressive Meshes" [15] (CPM for short) with several improvements in the stopping criteria and final encoding, as proposed in recent approaches [3, 17].

Before going into further details, we outline below several benefits of the CPM method concerning its counterparts:

- *Granularity control*: the size of each simplification batch can be freely parameterized.
- *Geometric predictor control*: any geometric predictor can be integrated into the algorithm. We implemented three predictors that can be freely selected.
- *Local error control*: any error metric can be integrated into the algorithm, e.g. according to the final application (geometric or perceptual metric). We implemented three error metrics that can be freely selected.
- *Efficient encoding of vertex splits*: the encoding has been optimized; e.g. there is no need to store costly indices.

The rest of this paper is organized as follows. An overview of the implemented algorithm is given in Section 2, while Section 3 provides details of algorithms and implementation. Experimental results are detailed in Section 4 with a performance comparison of several combinations of local error metric, position type, and geometric predictor. Finally, Section 5 presents possible extensions.

2 Overview

2.1 Input Data, Data Structures, and Library

The input of the algorithm presented in this paper is a 2-manifold triangulated surface mesh. A 2-manifold surface is characterized by the fact that each point of the surface admits a neighborhood homeomorphic to a disc and each border point admits a neighborhood homeomorphic to a half-disc. A triangle mesh is a type of polygon mesh commonly used in computer graphics. It is defined as a collection of vertices, edges, and triangular faces over which incidence and adjacency relationships are defined. Manipulating a 2-manifold triangle mesh, with reasonable time complexity, requires a dedicated data structure such as *halfedge*. This is particularly true for topological modifications. Our source code is based on the MEPP2 platform[3], a C++ software development kit and GUI for processing and visualizing 3D surface meshes and point clouds. It offers an application programming interface (API) for creating new processing filters and a graphical user interface (GUI) to integrate filters as plugins. This API integrates several *halfedge*-like data structure implementations (e.g. OpenMesh [2], CGAL Surface Mesh [19], CGAL Polyhedral Surface [19], CGAL Linear Cell Complex [19]) and it offers generic-programming abstraction layers, allowing to invoke any data structure with the same source code. The core of the platform is the central layer: the FEVV template library (FEVV holds for *Face Edge Vertex Volume*). It relies on a set of concepts [9], which provide an abstraction layer over several third-party mesh data structures. Generic filters can then be created based on this template library. Here are the main concepts used in the proposed compression algorithm implementation:

- the 2-manifold mesh concept (CGAL HalfedgeGraph and FaceGraph concepts), which deals with browsing of the mesh elements (vertices, edges, and faces); note that *halfedge* is a data structure commonly used to represent 2-manifold oriented polygon meshes;
- the (FEVV) geometry concept, which deals with the Point, Vector, and Scalar operations;
- the (FEVV) property map concept, which deals with the generic management of vertex, edge, face, and mesh attributes associated with a mesh.

The proposed source code is generic and can run with the following data structures: CGAL Polyhedral Surface, CGAL Surface Mesh, CGAL Linear Cell Complex, and OpenMesh. However, some CGAL restrictions occur currently on the geometric distance (e.g. Hausdorff or RMSE) computation between two meshes as this currently requires the CGAL 3D Fast Intersection and Distance Computation (AABB Tree) package which is not yet compatible with OpenMesh.

2.2 Compression Pipeline Overview

CPM is a *progressive compression* algorithm, meaning that the compressed bitstream can be progressively decoded to allow for the progressive transmission of the encoded triangle mesh through a series of mesh refinements. The compression algorithm comprises the following main steps (see Figure 1):

- The input triangle mesh is pre-processed to ensure that it is 2-manifold, then a uniform quantization of XYZ coordinates (on *nb_q_bits* bits) and a duplicate vertex removal step are applied. The latter is performed to ensure a unique vertex spanning tree traversal.

[3] https://github.com/MEPP-team/MEPP2

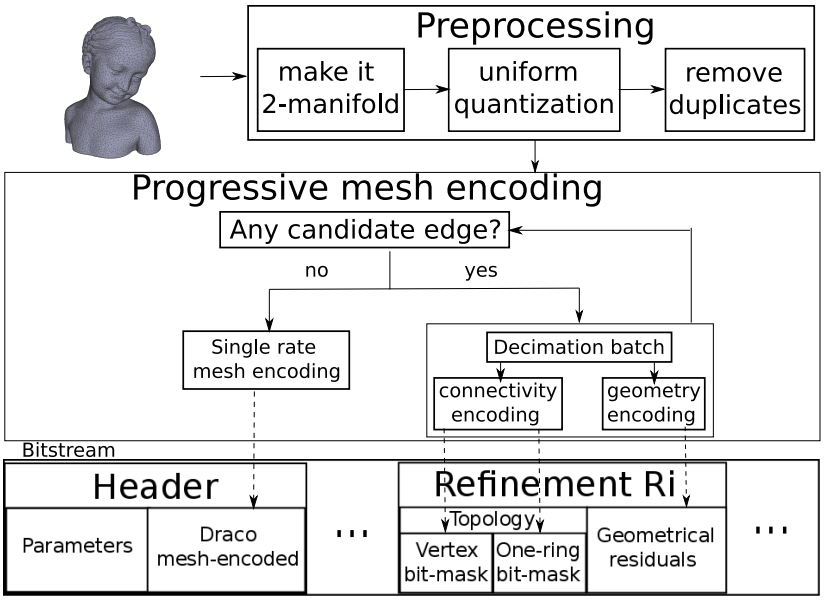


Figure 1: Overview of the progressive encoding process.

- The mesh is iteratively simplified, using batches of *edge collapse* operators (see Section 3.1.3). This simplification is guided by a *local error metric* and is repeated for a given number of batches (*nb_max_batches*) or until the mesh reaches a given minimum number of vertices (*nb_min_vertices*).

- At each simplification step, the corresponding batch of vertex-split operations (which will refine the mesh during decompression) is encoded efficiently using *geometric predictors*.

- At the end of the simplification, the base mesh (i.e. the coarsest level of detail) is encoded via a state-of-the-art single-rate compression algorithm (see Section 3.2.4).

The compressed bitstream is thus composed of (1) a few parameters with the encoded base model and (2) several chunks of compressed data, where each chunk contains the encoded refinement information (connectivity+geometry) that allows the next level of detail to be reconstructed. Each refinement step corresponds to a batch of vertex-split operations that increase the number of vertices by up to 50% while keeping the model 2-manifold and triangular. This decompression process terminates either when the last chunk has been decoded or when a sufficient LoD is reached.

2.3 Relevant Parameters

During the compression stage to generate the successive levels of detail (LoD), the CPM method selects a subset of edges to collapse, with the following rules:

- *topological constraints* decrease the set of candidates to guarantee either their decoding (edge collapse transformations are invertible – the associated vertex splits are independent) or the topology preservation. The restrictions are as follows: (1) if an edge e=(v_1,v_2) is contracted into a vertex v, then all incident edges to v cannot be contracted in the same batch (the grey edges in Figure 2); (2) only edges satisfying the link condition [4] can be collapsed; and (3) forbid edges that are incident to pivot vertices and whose opposite vertex is adjacent to v (the blue edges in Figure 2); consequently, no cut-edge is used for more than one vertex split;

- *an error metric* calculated per edge is used to select first the edges that introduce minimal error and that do not violate the aforementioned topological constraints.

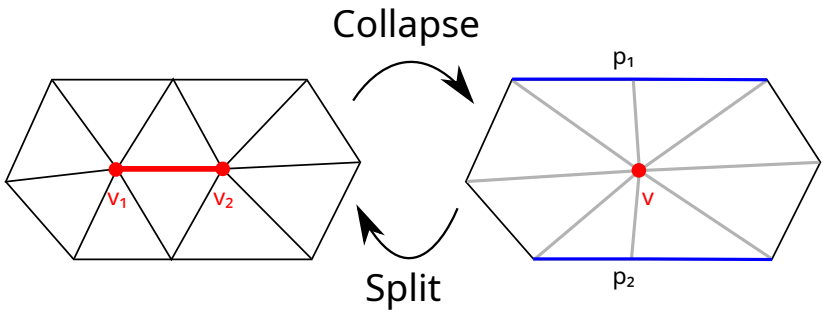


Figure 2: Edge collapse and vertex split operators: the edge (v_1,v_2) is contracted into a vertex v and two pivot vertices (p_1 and p_2) are required to retrieve the right topology during the decoding. In this example, there are 2 cut-edges (v,p_1) and (v,p_2). Gray and blue edges are forbidden from the remaining edges to collapse in the current batch.

We now list all parameters that impact the CPM results:

- the single-rate encoding method of the base mesh (the coarsest model);
- the restrictions on the edge candidates to collapse, which can also include stopping criteria such as the minimal number of vertices in the base mesh or the maximal number of different LoDs to generate;
- the selected error metric whose results depend on the position of the remaining vertex upon edge contraction;
- the connectivity coding of a batch of refinements;
- and the geometry coding of a batch of refinements whose result depends on the selected geometry predictor.

Note that geometry coding performance depends on the chosen coordinates quantization. Therefore, the number of bits for uniform quantization is also an important parameter. Moreover, the selected entropy coding method further impacts the connectivity and geometry coding performance.

3 Algorithms and Implementation Details

In this section, the progressive simplification and the associated encoding are detailed. Then, we present the main parameters of our compression framework. Afterward, the progressive refinement achieved by the decoder is summarized. Finally, we point out the correspondence between algorithms and source codes.

3.1 Progressive Simplification

Figure 1 gives an overview of the encoding process and Algorithm 1 details the progressive encoder. Let g be an input triangle mesh composed of vertices, edges, and faces (triangles). Let pm be the point map that associates a floating-point precision position to each vertex from g. Let gt be a geometry trait that enables applying operations to points, vectors, and scalars.

In what follows, we describe the preprocessing and uniform quantization step, the edge cost computation, as well as the batch decimation.

3.1.1 Preprocessing and Uniform Quantization

In the current implementation, there are 2 preprocesses. The first one ensures that the mesh g will be 2-manifold afterward: isolated vertices and edges are removed. At this stage, the mesh g cannot have other local non-manifold configurations: cut vertices, i.e. vertices with at least four incident border edges, and complex edges, i.e. edges with at least three incident triangles, have been managed during the mesh file reading by throwing an exception; dangling edges are impossible as the mesh reader cannot handle them.

Then, the vertex positions of the obtained 2-manifold triangle mesh are uniformly quantized, and the point map pm is updated consequently. The uniform quantization needs to select the number of bits to represent each coordinate of a 3D point. This number of bits is usually set to 10, 12 or 16. Alternately, the optimal number of bits can be set accordingly to the JND profile [14].

The second preprocessing is applied after uniform quantization to eliminate the initial tie-breaks (for the spanning tree construction, see Section 3.2.1). In the current implementation, all duplicate vertices are instead relocated by increasing their Z quantized coordinated (and pm is updated). Therefore, more vertex positions are modified than strictly needed.

3.1.2 Edge Costs

In Algorithm 1, at the beginning of a simplification batch, edge costs are computed to carry a greedy simplification of edges (via edge collapses). A common approach [8] is the use of a mutable priority queue (working with a heap) where the pair (edge, cost) with the minimum cost is popped at the top of the queue to achieve the next local simplification. Then, the edges whose cost has changed due to the last edge collapse have their cost updated. Note, however, that in our decimation scenario, it is not necessary to update the cost of edges and in particular of

1. incident edges to a vertex to split (which results from a terminated edge collapse),
2. and incident edges to an adjacent vertex to a vertex to split.

Indeed, these edges are forbidden at the time of the collapse (see Section 3.1.3 for more details on constraints). Moreover, only local error metrics are proposed in this work to compute the edge costs (see Section 3.3.1). Therefore, the priority queue of Algorithm 1 can be replaced by a sequence container of (edge, cost) pairs that are sorted by increasing cost. During the selection of the first pair, we only check whether the edge is forbidden and take the next pair in the positive.

3.1.3 Batch Decimation

For a given simplification batch with fixed local error metric, position type, and geometric predictor, there are two key ingredients: the simplification operator which is the edge collapse, and the predicate function which tells us whether a given edge is collapsible.

The *edge collapse* [10] is the local simplification operator of our pipeline used to remove one vertex, two or three edges, and one or two triangles (depending on the collapsed edge being a boundary or an interior edge). Conversely, its reverse operator is the *vertex split* which is a local refinement operator that permits the addition of one vertex, two or three edges and one or two triangles. Figure 2 illustrates the edge collapse and vertex split operators.

The *is_collapsible predicate function* indicates whether a given edge is collapsible. *Topology constraints* play a crucial role as they ensure that the mesh topology is preserved (link condition [4]), and they monitor edge collapse operations to guarantee the possibility to decode the associated vertex splits during the decompression, that is, if the collapse of an edge breaks the invertibility of one previous collapse, then this edge is not collapsible. *Encoding constraints* are also taken into account,

Algorithm 1: Progressive encoder

Inputs : A triangle mesh g, positions map pm, a geometry trait gt, $metric\_type$, $position\_type$, $prediction\_type$, nb_q_bits, $nb\_max\_batches$, $nb\_min\_vertices$, use_mean_th

Output: An encoded $buffer$ (that can be written into a binary file)

preprocess(g, pm, gt, nb_q_bits); `// 2-manifold,uniform quantiz.,remove vertex dupli.`
$nb\_v \leftarrow nb\_vertices(g)$;
$refinement\_list \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $nb\_max\_batches - 1$ **do**
  $list\_info \leftarrow \emptyset$; `// collapse info: psource,ptarget,pivot vertices,vkept,pkept`
  $forbidden\_edges\_set \leftarrow \emptyset$;
  `/* compute all edge costs and associated kept vertex position */`
  $edge\_pq \leftarrow \emptyset$; `// sort edges by increasing cost, here using a priority queue`
  forall *e in edges(g)* **do**
    $pkept \leftarrow get\_kept\_position(e, g, pm, gt, position\_type)$;
    $edge\_pq.insert(e, cost(e, g, pm, gt, pkept, metric\_type), pkept)$;
  end
  `/* start a new simplification batch */`
  while *not edge_pq.empty() and (nb_min_vertices* $< nb\_vertices(g)$*) and (not use_mean_th or edge_pq.get_top_cost()* $\leq$ *edge_pq.get_mean_threshold())* **do**
    $(e, pkept) \leftarrow edge\_pq.pop()$; `// get the edge with lowest cost and its position`
    if $is\_collapsible(e, g, pm, gt, forbidden\_edges\_set)$ **then**
      $info \leftarrow \{psource, ptarget, pivot\ vertices\}$;
      $forbidden\_edges\_set.insert(incident\_edges(adjacent\_vertices(source(e, g), g), g))$;
      $forbidden\_edges\_set.insert(incident\_edges(adjacent\_vertices(target(e, g), g), g))$;
      $vkept \leftarrow collapse(e, g)$; `// update pq after collapse when needed`
      put(pm, $vkept$, $pkept$); `// update vkept position`
      $info \leftarrow info \cup \{vkept, pkept\}$;
      $list\_info.push(info)$;
    end
  end
  `/* generate refinement info in the spanning tree vertex traversal order */`
  $st \leftarrow compute\_vertex\_spanning\_tree(g, pm)$;
  sort($list\_info$, st); `// sort tuples using vkept according to st traversal`
  $(vertex\_bitmask, one\_ring\_bitmask, reverse\_bitmask) \leftarrow get\_topology\_info(list\_info, st)$;
  $residuals \leftarrow get\_geometry\_info(list\_info, prediction\_type, g, pm)$;
  $refinement\_list.push(\{vertex\_bitmask, one\_ring\_bitmask, residuals, reverse\_bitmask\})$;
  $nb\_v\_current \leftarrow nb\_vertices(g)$;
  if $nb\_v = nb\_v\_current$ *or* $nb\_v\_current \leq nb\_min\_vertices$ **then**
    break; `// did not remove any vertex or the nb_min_vertices is reached`
  end
  $nb\_v \leftarrow nb\_v\_current$;
end
$buffer \leftarrow \emptyset$;
`/* encode file header */`
$encode\_param(position\_type, prediction\_type, nb\_q\_bits, AABB, buffer)$;
$encode\_base\_mesh\_draco(g, pm, gt, buffer)$;

```
   /* encode refinement information - from coarser to finer LoDs                */
38 for i ← refinement_list.size() − 1 to 0 do
39     {vertex_bitmask, one_ring_bitmask, residuals, reverse_bitmask} ← refinement_list[i];
40     encode_bits_draco(vertex_bitmask, buffer);
41     encode_bits_draco(one_ring_bitmask, buffer);
42     if position_type = target then
           /* pivot vertices based on st traversal order need to be reversed?   */
43         encode_bits_draco(reverse_bitmask, buffer);
44     encode_symbols_draco(residuals, buffer);
45 return buffer;
```

we follow the strategy which consists in prohibiting the presence of two adjacent vertices to split during the decoding of the current LoD [3]. Even if this constraint is more restrictive than the original CPM method' constraints, it enables saving about 50% of zero codes in the vertex bitmask (see Section 3.2.2). *Geometric constraints* are taken into account as well: if an edge collapse has a resulting vertex associated with at least one incident triangle normal flip, then this edge collapse is discarded. Note that geometric predictions that use several neighboring vertex positions to predict a position could add many constraints (forbid the collapse of many edges), especially when vertex positions in the k-ring neighborhood are used with $k \geq 2$. To avoid these superfluous constraints, we generate geometric predictions at the end of a batch of edge collapses during the compression and before a batch of vertex splits during the decompression.

3.2 Encoding

The encoding of the refinement information is composed of three to four bitstream chunks. There are, in the vertex spanning tree traversal order, the vertex-to-split locations (the vertex bitmask), the cut-edge locations of the vertex to split (the one-ring bitmask), the reverse bitmask only for the target position type (allows the reversal of cut-edge order), and the residuals (geometry prediction errors for the new vertices). These chunks are computed at the end of a decimation batch.

In this section, the algorithm for vertex spanning tree traversal is detailed. Then, the encoding of the connectivity bitmasks and geometry residuals are described. Finally, the encoding of the file header is presented.

3.2.1 Vertex Spanning Tree Traversal

The algorithm for the spanning tree construction is an acyclic traversal of the vertices of each connected component, which sorts all vertices and edges of the input mesh by traversal order. It assumes that no *tie-breaks* are present, which means that the spanning tree construction is deterministic. For each connected component, a pair of a root vertex r with its minimum incident edge min_{e^r} is found (see Figure 3). The root vertex is the minimum vertex. We, therefore, need a vertex comparator $<$ defined as follows: $v_1 < v_2$ if the coordinates of v_1 are strictly smaller than the coordinates of v_2, in the lexicographic order. If v_1 and v_2 have the same coordinates, then $v_1 < v_2$ if degree(v_1) $<$ degree(v_2). In the case of equality, three other conditions are successively tested: $v_1 < v_2$ if the sum (resp. min, max) of the degree of adjacent vertices to v_1 is lower than the sum (resp. min, max) of the degree of adjacent vertices to v_2. The minimum edge min_{e^r} is the edge incident to the root vertex r and the minimum adjacent vertex to r.

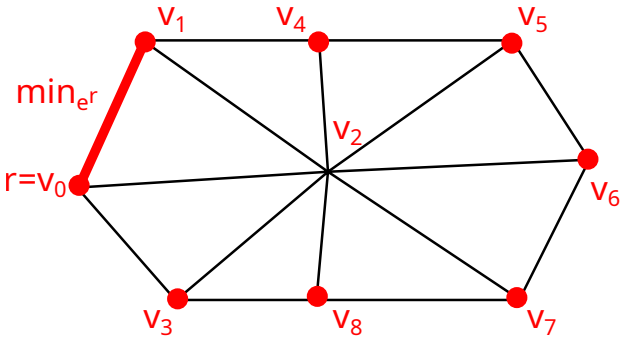


Figure 3: Vertex spanning tree traversal. The vertex indices correspond to their traversal order.

Once a root vertex and its associated minimum edge are set for a given connected component, a region growing occurs through all vertices of the connected component using the orientation of the mesh triangles (usually counter-clockwise). To order several connected components, their root vertices are sorted according to the vertex comparator $<$ defined above. Therefore, to guarantee the absence of tie-break, a sufficient condition is that all root vertices have a different position and their associated minimum incident edge is unique. No constraint applies to other vertices in a 2-manifold mesh.

3.2.2 Connectivity

The encoding of the mesh connectivity enables, during the decoding, the retrieval of the mesh connectivity of the next refined LoD. It consists of the encoding of the vertex to split locations and their associated cut-edge locations. Two to three bitmasks are encoded:

- the *vertex bitmask* identifies the vertices to split during the decoding (one means vertex split and zero doing nothing). Bits are given in the spanning tree traversal order. A direct optimization is to not encode the remaining zero bits once we have no more vertex to split. Because all adjacent vertices to the vertex to split cannot be split (a constraint that is set during the encoding), then most of the 0 codes of the vertex bitmask can be predicted and do not need to be encoded [3] (see Figure 4).

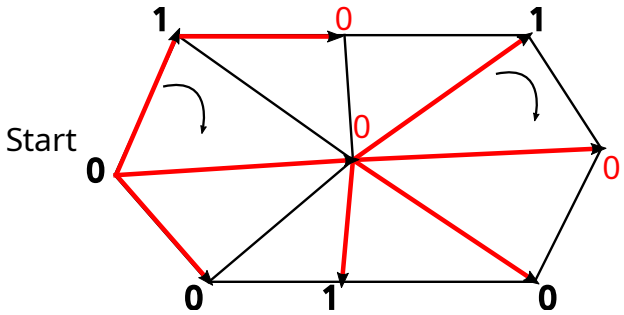


Figure 4: Encoding of vertex-to-split locations (vertex bitmask): Vertex spanning tree traversal is defined in Figure 3. The associated vertex bitmask is 0100010001. Ones (black) are generated for vertices to split. Zeros (red) can be predicted and are thus not generated.

- the *one-ring bitmask* specifies either one (border case) or two cut-edges around each vertex to split. Bits are given in the clockwise order (if the triangles are counter-clockwise ordered), starting from the min edge (given by the spanning tree). A direct optimization is to not encode

the zeros after we met two ones. Note that our coding solution is different from the CPM method [15].

- the *reverse bitmask* states whether the order of cut-edges must be reversed for a vertex to split before the geometric prediction of the new vertex position (one means reversal and zero no-reversal). This bitmask is needed only for the target position type (see Section 3.3.2).

Bitmasks are entropy coded using the Draco RAns bit encoder [5, 6].

3.2.3 Geometry

Encoding the mesh geometry enables retrieving the v_1 and v_2 vertex positions computed for each vertex v that is split during the decoding of the connectivity (see Figure 2). Residuals are the error vectors between the true positions and the predicted positions. They are entropy encoded using the Draco symbol encoder [6].

3.2.4 File Header

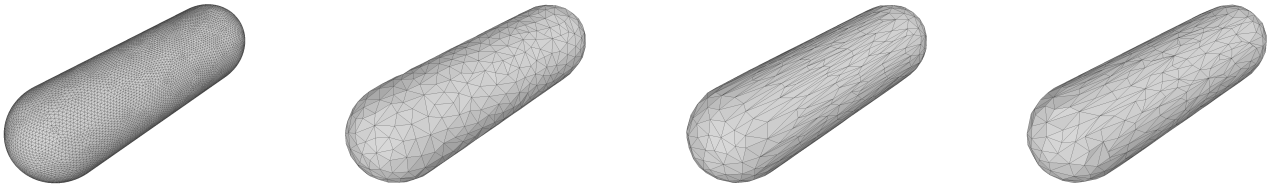
After all decimation batches have been computed, the file header is encoded. The header is composed of a few parameters (position type, geometric predictor, quantization bits, and mesh Axis-Aligned Bounding-Box) and of the base mesh. The base mesh is encoded via a state-of-the-art single-rate compression algorithm from the Draco library [6, 17] based on the Edgebreaker coding method [18] (as in the original CPM method).

3.3 Parameter Choice

In this section, we present the main parameters of our compression framework.

3.3.1 Local Error Metric

The metric type sets the priority of simplification operations (apply first the operation that introduces the minimal error according to the selected metric). Three geometric error metrics are implemented: the *edge length* (shortest edges first), the *local absolute volume error* (part of the metric used in [21]), and the memoryless variant of *QEM error* (Quadric Error Metric) [11]. The metric type impacts the LoD quality in terms of global geometric error and triangle regularity (see Figure 5). For instance, the edge length tends to produce more uniform meshes with higher geometric error, while the two other metrics better preserve the shape with less uniform mesh. Note that the original CPM method uses a variation of QEM [8] using a normalization factor for every error quadric – the number of planes.



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

Figure 5: Capsule model: from left to right (a) the original model, the simplified model respectively with (b) edge length, (c) local absolute volume error [21], and (d) QEM [11].

3.3.2 Position Type

The position type for the edge collapse sets the location of the remaining vertex after a collapse. Two position types are available in our approach: the edges target position (v_2 in Figure 2) and the midpoint position $\frac{v_1+v_2}{2}$. For experimental results, the target position is the only position considered among the source (v_1 in Figure 2) and target positions, as we expect similar performances for both. We do not consider the optimized position (available for QEM, but needs two residuals per collapse). Note that the original CPM method uses the midpoint position.

3.3.3 Geometric Predictor

The predictor type sets the geometric prediction for the two point positions v_1 and v_2 of the resulting vertex split (see Figure 6). Three predictors are available: *no-prediction* (encode the two positions v_1 and v_2), *delta* prediction (encode the vector $D = v_1 - v_2$), and *butterfly* prediction (encode one error residual $D - D'$, $D' = v'_1 - v'_2$ where v'_i is estimated using the butterfly prediction). Note that the original CPM method uses butterfly prediction.

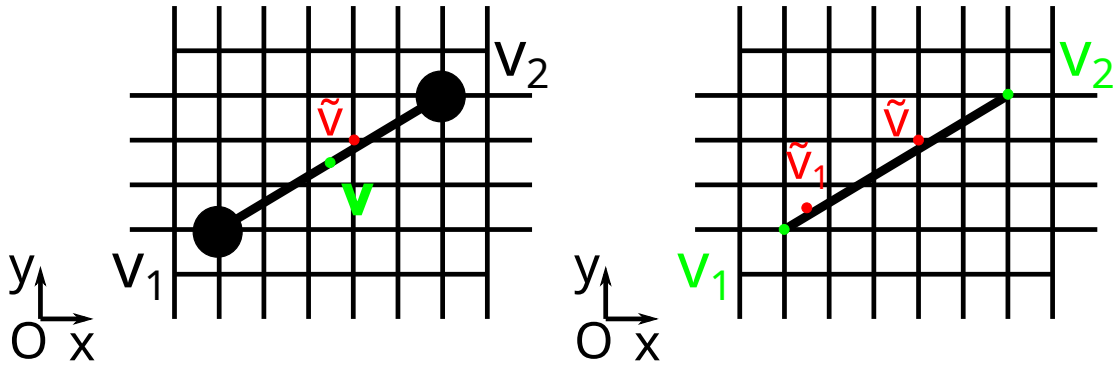


Figure 6: Geometric prediction with quantized coordinates and midpoint position type: (a) during compression the midpoint v is approximated by $\tilde{v}$ by rounding coordinates towards positive infinity and (b) during decompression for delta and butterfly prediction types (that enable retrieving the vector $D = v_1 - v_2$) the reconstructed v_1 is obtained by rounding the $\tilde{v_1}$ coordinates towards negative infinity; then v_2 is obtained using the $v_2 = v_1 - D$ equation. Green (resp. red and black) color means exact (resp. approximation and original).

3.3.4 Quantization Bits

The number of quantization bits is used to apply a uniform quantization on the input mesh during the preprocessing step.

3.3.5 Global Termination Criteria

The maximum number of LoDs/batches and the minimum number of vertices in the base mesh are two global stopping parameters. Note that the condition "minimum number of vertices" is currently tested after a batch decimation, and the final mesh has thus a number of vertices smaller or equal to the desired minimum.

3.3.6 Stopping Condition per Batch

A stopping condition parameter decides, for a decimation batch, to collapse either all candidate edges that fulfill the *is_collapsible* predicate function or only those edges up to a given threshold. This

threshold is set to the averaged metric distortion associated with edges present in the current mesh LoD (see Section 3.3.1 for available metric types).

3.4 Progressive Decoder

Algorithm 2 details the progressive decoder. It starts with the decoding of a few parameters (position type, geometric predictor, quantization bits, and mesh Axis-Aligned Bounding-Box), followed by the decoding of the base mesh. Then, batches of refinements occur until the input buffer is entirely decoded. Eventually, the decoded mesh is uniformly dequantized.

Algorithm 2: Progressive decoder

Input : A $buffer$ to decode (that can be read from a binary file)
Outputs: A 2-manifold triangle mesh g, positions map pm

```
/* decode file header                                                    */
```
$pos \leftarrow 0$; `// parsing position, updated at each decoding step`
$(position\_type, prediction\_type, nb\_q\_bits, AABB) \leftarrow decode\_param(buffer, pos)$;
$(g, pm) \leftarrow decode\_base\_mesh\_draco(buffer, pos)$;
while $buffer.size() > pos$ **do**
```
    /* decode refinement information                                     */
```
    $vertex\_bitmask \leftarrow decode\_bits\_draco(buffer, pos)$;
    $one\_ring\_bitmask \leftarrow decode\_bits\_draco(buffer, pos)$;
    if $position\_type = target$ **then**
```
        /* pivot vertices based on st traversal order need to be reversed?  */
```
        $reverse\_bitmask \leftarrow decode\_bits\_draco(buffer, pos)$;
    end
    $residuals \leftarrow decode\_symbols\_draco(buffer, pos)$;
```
    /* generate vertex split info in the spanning tree vertex traversal order
    */
```
    $st \leftarrow compute\_vertex\_spanning\_tree(g, pm)$;
    $list\_cut\_edges \leftarrow get\_topology\_info(st, vertex\_bitmask, one\_ring\_bitmask)$; `// list of cut-edges pointing towards their vertex to split (for each vertex to split: 1 edge duplicated if 1 pivot, 2 otherwise)`
    $pair\_new\_positions\_list \leftarrow$ $get\_geometry\_info(residuals, reverse\_bitmask, list\_cut\_edges, prediction\_type, g, pm)$;
```
    /* apply vertex splits of the decoded refinement batch                  */
```
    forall $(e1, e2)$ *in* $list\_cut\_edges$ **do**
        $enew \leftarrow vertex\_split(e1, e2, g)$; `// new edge whose target is the new vertex`
        $pair\_new\_positions \leftarrow pair\_new\_positions\_list.pop\_front()$;
        put(pm, $target(enew, g)$, $pair\_new\_positions.first()$);
        put(pm, $source(enew, g)$, $pair\_new\_positions.second()$);
    end
end
postprocess(g, pm, gt, AABB); `// uniform dequantiz`
return (g, pm);

3.5 Correspondence Between Algorithms and Source Codes

In this section, we point to the associated source files and functions that correspond to the two algorithms. Source files related to the progressive compression and decompression are located in the src/FEVV/Filters/CGAL/Progressive_Compression folder. These source files are processing filters that work with the CGAL data structures: CGAL Polyhedral Surface, CGAL Surface Mesh, and CGAL Linear Cell Complex.

3.5.1 Algorithm 1 (Progressive Encoder)

Algorithm 1 corresponds to the function progressive_compression_filter line 126 of file progressive_compression_filter.hpp.

- The preprocess function at line 1 (see Section 3.1.1 for description) corresponds to the preprocess_mesh function given at line 59 of the file progressive_compression_filter.hpp.
- The main iteration loop at line 4 appears at lines 215 to 236 of file progressive_compression_filter-.hpp.
- Lines 5 to 28 (inside the main iteration loop) are located in the collapse_batch function line 236 of file Compression/Batch_collapser.h. This function is called on line 217 of file progressive_compression_filter.hpp. Lines 29 to 33 (end of the main iteration loop) correspond to lines 226 to 235 of file progressive_compression_filter.hpp.
- The computation of edge costs and their sorting at lines 7 to 11 correspond to the function compute_error called on line 242 of file Compression/Batch_collapser.h. This (virtual) function is declared in the class Error_metric from Metrics/Error_metric.h file, and three implementations are proposed from the derived classes Edge_length_metric, QEM_3D, and Volume_preserving.
- The simplification batch of lines 12 to 23 is given at lines 250 to 258 in the collapse_batch function of file Compression/Batch_collapser.h.
- The generation of the refinement information at lines 24 to 28 corresponds to lines 272 to 300 in the collapse_batch function of file Compression/Batch_collapser.h.
- The encoding of the file header at lines 36 to 37 is located at lines 258 to 263 of file progressive_compression_filter.hpp.
- The encoding of refinement information at lines 38 to 44 is associated with lines 279 to 337 of file progressive_compression_filter.hpp.

3.5.2 Algorithm 2 (Progressive Decoder)

Algorithm 2 corresponds to the function progressive_decompression_filter line 67 of file progressive_decompression_filter.hpp.

- The decoding of the file header at lines 2 to 3 is located at lines 80 to 140 of file progressive_decompression_filter.hpp.
- The main iteration loop at line 4 appears at lines 162 to 166 of file progressive_decompression_filter.hpp.

- Lines 5 to 19 (inside the main iteration loop) are located in the decompress_binary_batch function line 207 of file Decompression/Batch_decompressor.h. This function is called on line 164 of file progressive_decompression_filter.hpp.
- The decoding of the refinement information at lines 5 to 10 is associated with lines 210 to 225 of file Decompression/Batch_decompressor.h.
- The generation of vertex split information at lines 11 to 12 corresponds to lines 231 to 241 in the decompress_binary_batch function of file Decompression/Batch_decompressor.h. The decoding of new vertex positions at line 13 is done at the beginning of the split_vertices function of file Decompression/Batch_decompressor.h.
- The refinement batch of lines 14 to 19 is given at lines 291 to 343 in the split_vertices function of file Decompression/Batch_decompressor.h.
- The postprocess function at line 21 corresponds to the uniform dequantization at lines 170 to 172 of file progressive_decompression_filter.hpp.

4 Experimental Results and Demonstration

In this section, we describe our data set and analyze the compression performances using several combinations of local error metric (three), position type (two), and geometric predictor (two).

4.1 Data Set

On the one hand, we may expect that the regularity of the edges and triangles may affect the performance of the geometry predictor. On the other hand, the local error metric will modify the regularity of the mesh as mentioned in Section 3.3.1. The data set is, therefore, composed of four regular mesh models and four irregular mesh models (see Figure 7). The selected meshes represent smooth surfaces and are expected to work well with the butterfly geometry prediction (especially when the mesh is sufficiently regular). A uniform quantization of XYZ coordinates is applied as a preprocess. See Table 1 for model details.

We set the minimum number of vertices (*nb_min_vertices*) to 5% of the number of vertices present in the full resolution mesh (as in CPM), and we set the maximum number of batches (*nb_max_batches*) to a value large enough to ensure that the minimum number of vertices is reached for the base mesh. We performed this experiment with 10, 12, and 16 quantization bits.

4.2 Compression Performances

4.2.1 Bit Allocation

In Table 1, we carried out an experiment with the local absolute volume error metric, the midpoint position, and the butterfly prediction. The obtained results show that the header information is about 1.23 bpv (bits per vertex) per model. The connectivity information takes about 7.26 bpv and can be reduced by about 1.5 bpv for regular meshes (less improvement is expected for irregular meshes) by following the extension proposed in the next section to optimize the connectivity bitmask. Depending on the chosen quantization, the geometry information varies between 5.91 and 23.35 bpv for regular meshes and between 6.46 and 23.86 bpv for irregular meshes. The CPM original implementation [15] yields about 3.5 bpt (bits per triangle) $\approx$ 7 bpv for the connectivity information and about 8.1 bpt $\approx$ 16.2 bpv for the geometry information quantized with 10 bits. Our CPM implementation is therefore

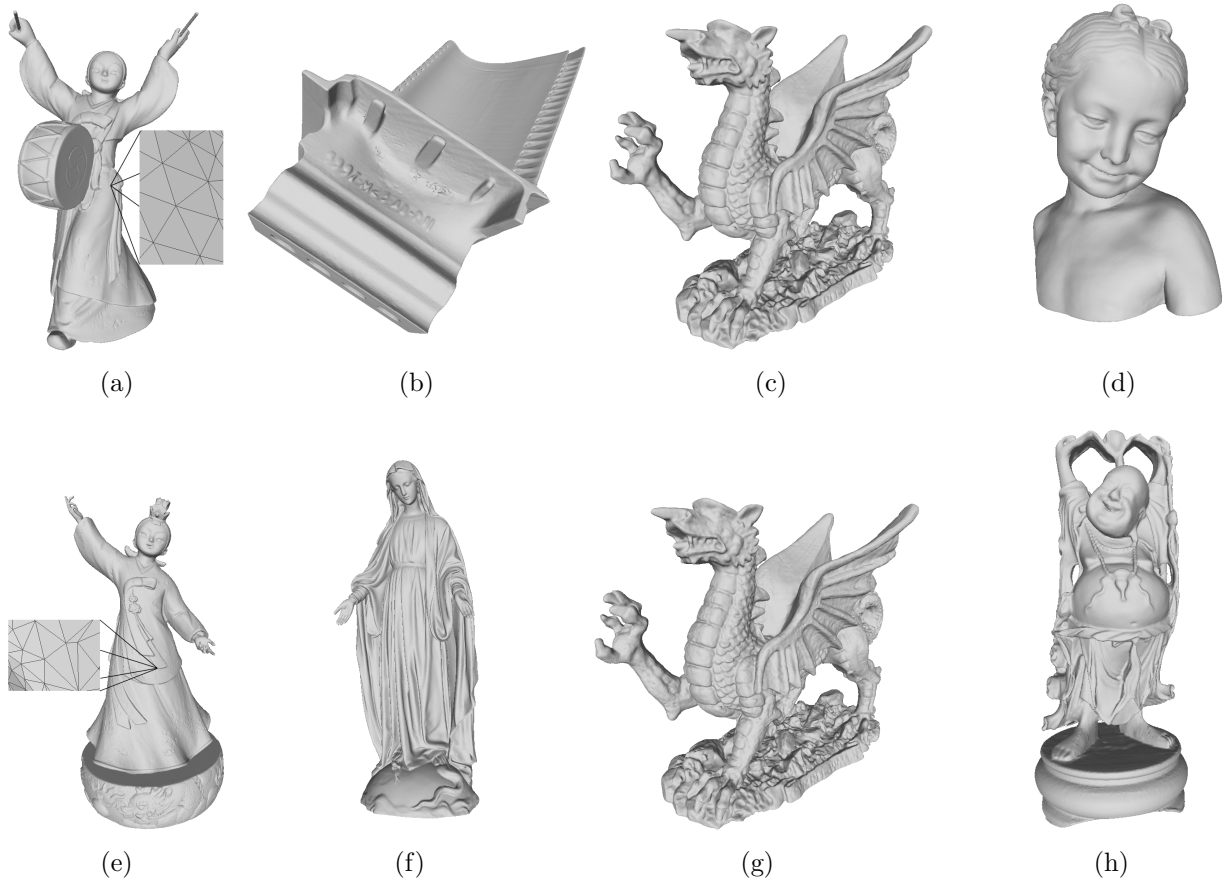


Figure 7: Data set: the first row is composed of regular mesh models while the second row is made of irregular models: (a) drum, (b) blade, (c) dragon, (d) bimba, (e) ari, (f) artec, (g) dragon, and (h) happy.

a little bit less effective for the connectivity information (but we can apply a connectivity bitmask optimization as already mentioned) and more efficient than the original CPM for the geometry information. As we noticed that the geometry information bpv of our implementation improves with the size of the input mesh (the more there are residuals, the more the entropy coders are efficient), we compare the geometry bpv for the fandisk model presented in [15] (only 6475 vertices) and our method improves about 1.67 bpv for the geometry information. Table 1 shows, for regular meshes, results between 17.74 and 19.99 bpv for 12 bits quantization which are of the same order as results obtained for state-of-the-art progressive mesh compression algorithms [13] when including the connectivity optimization of about 1.5 bpv.

4.2.2 Rate-Distortion (R-D) Curves

Figures 8 and 9 present Rate-Distortion (RD) curves. A RD curve represents the amount of time (proportional to the number of bits per vertex to be transmitted) necessary to approximately reconstruct an input signal without exceeding a specified distortion. We observe that the butterfly prediction and the midpoint position are consistently better in terms of bpv than, respectively, the delta prediction and the target position. As expected, the midpoint position introduces a little bit more distortion than the target position, because the midpoint position is not necessarily onto the original surface, while the target position is. We also notice that the edge metric usually provides the highest distortion with the smallest bpv and that the local absolute volume error is the best metric for the 12 bits quantized data set.

Model	#vertices	Header bpv	Conn. bpv	Geom bpv	Total bpv	Comp. time in s	Decomp. time in s	#LoD
drum reg. Q16		1.55	7.21	20.59	29.35	27		
Q12	218294	1.05	7.27	9.42	17.74	27	7	18
Q10		0.85	7.35	5.91	14.11	26		
blade reg. Q16		1.78	7.22	23.35	32.35		5	
Q12	169730	1.26	7.26	11.47	19.99	20	5	18
Q10		1.02	7.34	7.09	15.45		6	
dragon reg. Q16		1.63	7.20	22.69	31.52		7	
Q12	203575	1.13	7.23	10.90	19.26	24	7	18
Q10		0.90	7.29	6.51	14.70		8	
bimba reg. Q16		1.64	7.28	21.74	30.66	26		
Q12	192135	1.14	7.32	10.39	18.85	26	6	18
Q10		0.89	7.32	6.19	14.40	27		
ari irr. Q16		1.61	7.23	22.60	31.44	26		
Q12	202502	1.09	7.26	10.92	19.27	25	6	18
Q10		0.83	7.28	6.46	14.57	25		
artec irr. Q16		1.85	7.12	23.09	32.06	19	5	17
Q12	165621	1.11	7.22	11.42	19.75	18	6	18
Q10		0.90	7.32	7.21	15.43	19	5	18
dragon irr. Q16		1.66	7.20	23.86	32.72		6	
Q12	202502	1.13	7.22	11.96	20.31	25	6	18
Q10		0.87	7.25	7.11	15.23		7	
happy irr. Q16		1.62	7.22	23.07	31.91	24		
Q12	199925	1.09	7.25	11.29	19.63	26	6	18
Q10		0.85	7.27	6.64	14.76	25		

Table 1: Data set and compression results were obtained with the local absolute volume error metric, the midpoint position, and the butterfly prediction. *bpv* denotes the number of bits per vertex. Running times are given for the CGAL Surface Mesh data structure. See Figure 7 for model screenshots.

4.2.3 Timings

We ran our experiments on a laptop with an Intel Core i7 clocked at 2.8Ghz, with 32GBytes of memory. Table 1 shows that compression is more computation intensive than decompression. That is due to the computation of all edge costs for each batch but also to the *is_collapsible* predicate checks. To avoid exceeding 20 seconds during compression, meshes with less than 165k vertices are necessary.

During the decompression for the presented data set in Table 1, the first ten LoDs are usually decoded in less than 2 seconds. We focused on writing generic code. Hence our implementation is not as fast as a specific data structure implementation could be. We have not taken into account the possibilities of parallelizing certain parts of code. However, we expect that our code will enable fast prototyping of new ideas for the community.

4.3 Online Demonstration

The online demonstration illustrates the algorithm described in the paper entitled "Compressed Progressive Meshes" initially proposed in [15]. The idea of this demonstration is to show how a 2-manifold triangle mesh can be compressed and decompressed, as well as how the partial decompression feature works. First, the selected 2-manifold triangle mesh is taken as input and compressed into a file using a set of input parameters (see Section 3.3 for description). Then, the compressed

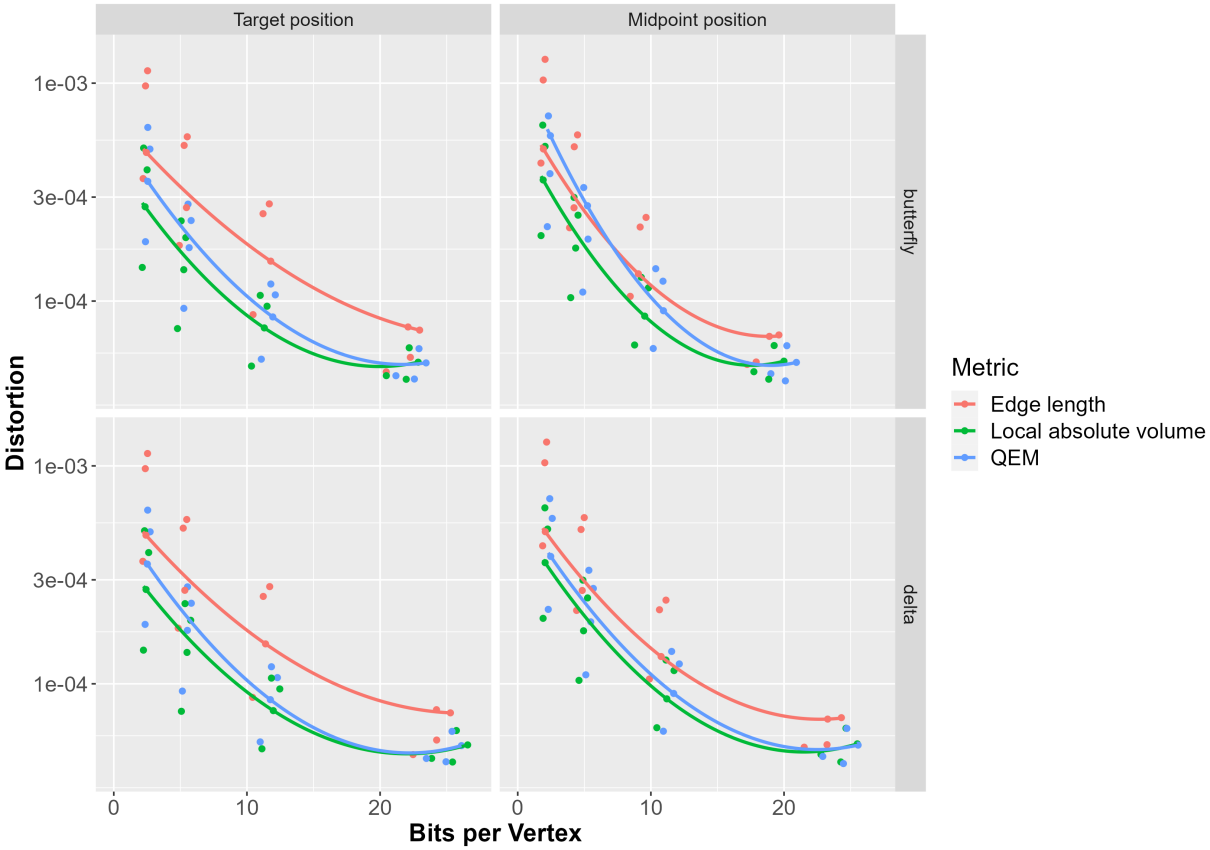


Figure 8: Median degree 2 polynomial fit of Rate-Distortion data for a set of 4 regular models. The distortion is the RMSE normalized by the diagonal length of the mesh AABB. The chosen quantization is 12 bits.

file is decompressed, allowing us to see the full decompression of the mesh. Next, the partial decompression of the mesh is demonstrated. Finally, the input mesh and the resulting full and partially decompressed meshes are displayed in the 3D viewer, allowing for comparison. This allows us to see the progressive reconstruction features of the algorithm.

In the following section, we discuss possible extensions of this work.

5 Possible Extensions

5.1 Remove Only Tie-breaks

For 2-manifold meshes, the uniqueness of the spanning tree construction is guaranteed when all root vertices have a different position and their associated minimum incident edge is unique. Therefore, the *remove duplicates* preprocessing step should be replaced by a *remove root tie-breaks* step. This limits the number of relocated vertices to only a few and restrain the visual artifacts observed when a relocated vertex creates a geometric fold, especially when the number of quantization bits is low. Also, the vertex relocation strategy needs to be improved to avoid visual artifacts.

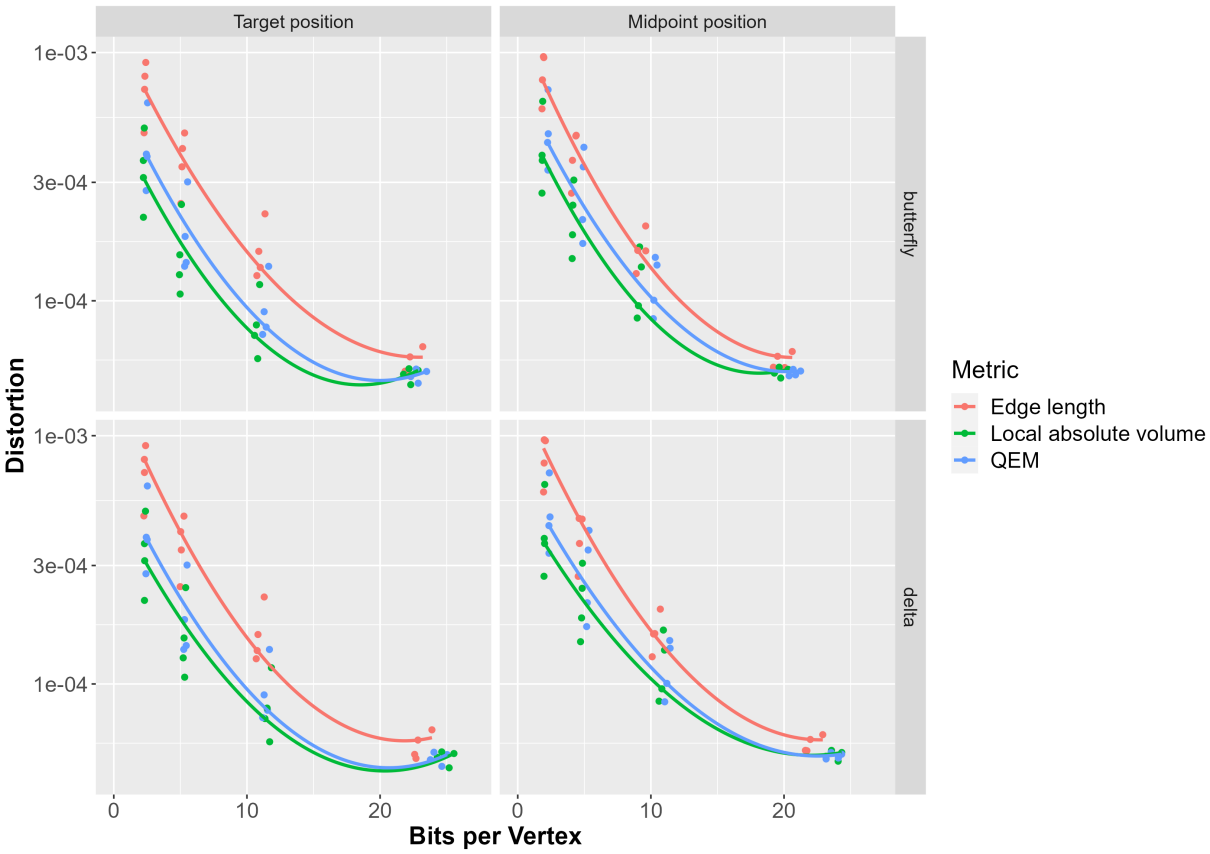


Figure 9: Median degree 2 polynomial fit of Rate-Distortion data for a set of 4 irregular models. The distortion is the RMSE normalized by the diagonal length of the mesh AABB. The chosen quantization is 12 bits.

5.2 Connectivity Bitmask Optimizations

In the current implementation, we do not apply all possible bit optimization techniques. For instance, an optimization of the one-ring bitmask required to retrieve the topology information (the cut-edges) can be performed as follows. To improve the encoding of the cut-edge symbols for each vertex to split, the idea is to sort adjacent vertices (stable sort) in descending order of probability of being a pivot vertex (a pivot vertex is incident to a cut-edge as can be seen in Figure 2). If the probability estimation is large enough, then instead of having the two activated bits at a random position, they should be rather located at the beginning of the cut-edge symbol to code, which improves in our initial experiments entropy coding performance. See [16, 17] for two illustrating examples.

There are other avenues to explore, for example, it seems that the entropy coding of the connectivity buffers can be optimized according to the batch information size. During our experiments, we obtained that, for small batches/LoDs, 8 bits packing seems to be more efficient than the Draco RAns encoder. Conversely, for large batches, the Draco symbol encoder is more efficient than the Draco RAns encoder.

5.3 Entropy Coding of Geometry Residuals

Experimentally, another way to improve the entropy coding performance, especially for geometrical residuals of finer LoDs, is to group the encoding of two or three successive LoDs, namely the same entropy coder is shared for two or three successive LoDs instead of one LoD. We can also see such an

approach as a way to compensate for a low vertex removal rate during compression due to selected edge constraints, which results in a high number of LoDs. For instance, in the CPM approach [15] we expect an average vertex removal rate for all decimation batches of about 25 percent. If our average vertex removal rate $\overline{\tau_v}$ is smaller, then we can group the encoding of $\lfloor ln(1-0,25)/ln(1-\overline{\tau_v})\rfloor$ successive LoDs.

5.4 Adaptive Quantization

Noticing that at very low mesh resolution, the geometric error introduced by quantization is insignificant compared to that caused by the reduced number of mesh elements, several authors [12, 17] propose the adaptive quantization of attributes. It improves coding performances, especially at low bit rates. Moreover, the number of bits for the initial quantization can be set sufficiently high (e.g. ≥ 16) and the adaptive quantization will adjust it.

5.5 Local Frenet Frame

Alliez et al. [1] showed the interest of projecting the residual vectors into a local Frenet frame when the displacement is rather concentrated in a direction (e.g. normal direction). However, even if there is a benefit in our case [3], the displacements are rather localized in the tangent plane and the expected gain is smaller than in [1].

5.6 Mesh Attributes

Any other vertex/edge/face attribute can be compressed the same way we encode the vertex locations, namely, we need to set an initial number of quantization bits, an error metric to monitor the degradation during the decimation, and a predictor to decrease the magnitude of the error residuals. The attribute order can be set according to the spanning tree traversal order. Then, the attribute' bitstream chunk can be added to its LoD refinement chunk.

Note that mesh texture coordinates require particular attention, as texture seams (when two or more regions meet at a vertex) must be dealt with properly during encoding, especially if region destruction is authorized during the simplification (which results in a region creation during the decoding). Eventually, note that for a textured mesh, an initial texture re-atlasing and an optimal multiplexing between mesh and textures data must be set up [3, 17].

5.7 Polygonal and Non-Manifold Meshes

Polygonal or non-manifold meshes require using additional symbols to describe the topological modifications to apply [3]. To avoid such costly symbols for non-manifold triangular meshes, non-manifold edges (complex edges) and vertices (cut vertices) can be duplicated before encoding and merged after decoding.

6 Conclusion

We presented an implementation of the Compressed Progressive Meshes (CPM) method and extensions. This progressive compression method allows for the transmission of several Levels of Detail. We show that the combination of the local absolute volume error metric, the midpoint position, and the butterfly prediction parameters yields the best results in terms of rate-distortion for our selection of several meshes, with 12 bits uniform XYZ quantization. We proposed a few avenues for improvement. This contribution can be seen as a baseline for the community.

The code for this implementation is available through the MEPP2 platform.

Image Credits

All images copyright Vincent Vidal, license CC BY-SA.

Acknowledgment

This work was supported by French National Research Agency as part of ANR-PISCo project (ANR-17-CE33-0005).

References

[1] P. Alliez and M. Desbrun, *Progressive compression for lossless transmission of triangle meshes*, in Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), 2001, pp. 195–202. https://doi.org/10.1145/383259.383281.

[2] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt, *OpenMesh - a generic and efficient polygon mesh data structure*, 2002.

[3] F. Caillaud, V. Vidal, F. Dupont, and G. Lavoué, *Progressive compression of arbitrary textured meshes*, Computer Graphics Forum, 35 (2016), pp. 475–484. https://doi.org/10.1111/cgf.13044.

[4] T.K. Dey, H. Edelsbrunner, S. Guha, and D.V. Nekhayev, *Topology preserving edge contraction*, Publications de l'Institut Mathématique (BEOGRAD) (N.S.), 66 (1999), pp. 23–45.

[5] J. Duda, K. Tahboub, N.J. Gadgil, and E.J. Delp, *The use of asymmetric numeral systems as an accurate replacement for Huffman coding*, in Picture Coding Symposium (PCS), 2015, pp. 65–69. https://doi.org/10.1109/PCS.2015.7170048.

[6] F. Galligan, M. Hemmer, O. Stava, F. Zhang, and J. Brettle, *Google/draco: a library for compressing and decompressing 3d geometric meshes and point clouds.*, 2018.

[7] P-M. Gandoin and O. Devillers, *Progressive lossless compression of arbitrary simplicial complexes*, ACM Transactions on Graphics, 21 (2002), pp. 372–379. https://doi.org/10.1145/566654.566591.

[8] M. Garland and P.S. Heckbert, *Surface simplification using quadric error metrics*, in Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), New York, NY, USA, 1997, ACM Press/Addison-Wesley Publishing Co., pp. 209–216. https://doi.org/10.1145/258734.258849.

[9] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, *Concepts: Linguistic support for generic programming in C++*, ACM SIGPLAN Notices, 41 (2006), pp. 291–310. https://doi.org/10.1145/1167515.1167499.

[10] H. Hoppe, *Progressive meshes*, in Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Association for Computing Machinery, 1996, pp. 99–108. https://dl.acm.org/doi/pdf/10.1145/237170.237216.

[11] H. Hoppe, *New quadric metric for simplifying meshes with appearance attributes*, in Conference on Visualization, 1999, pp. 59–66. https://doi.org/10.1109/VISUAL.1999.809869.

[12] H. Lee, G. Lavoué, and F. Dupont, *Adaptive coarse-to-fine quantization for optimizing rate-distortion of progressive mesh compression*, in Vision, Modeling, and Visualization Workshop, 2009.

[13] A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot, *3D mesh compression : Survey, comparisons, and emerging trends*, ACM Computing Surveys, 47 (2015). https://doi.org/10.1145/2693443.

[14] G. Nader, K. Wang, H. Franck, and F. Dupont, *Just Noticeable Distortion Profile for Flat-Shaded 3D Mesh Surfaces*, IEEE Transactions on Visualization and Computer Graphics, 22 (2016). https://doi.org/10.1109/TVCG.2015.2507578.

[15] R. Pajarola and J. Rossignac, *Compressed progressive meshes*, IEEE Transactions on Visualization and Computer Graphics, 6 (2000), pp. 79–93. https://doi.org/10.1109/2945.841122.

[16] J. Peng and C.-C.J. Kuo, *Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition*, ACM Transactions on Graphics, 24 (2005), pp. 609–616. https://doi.org/10.1145/1073204.1073237.

[17] C. Portaneri, P. Alliez, M. Hemmer, L. Birklein, and E. Schoemer, *Cost-driven framework for progressive compression of textured meshes*, in ACM Multimedia Systems Conference, New York, NY, USA, jun 2019, Association for Computing Machinery, pp. 175–188. https://doi.org/10.1145/3304109.3306225.

[18] J. Rossignac, *Edgebreaker: connectivity compression for triangle meshes*, IEEE Transactions on Visualization and Computer Graphics, 5 (1999), pp. 47–61. https://doi.org/10.1109/2945.764870.

[19] The CGAL Project, *CGAL user and reference manual*, 2022. https://doc.cgal.org/5.5/Manual/packages.html.

[20] V. Vidal, E. Lombardi, M. Tola, F. Dupont, and G. Lavoué, *MEPP2 : a generic platform for processing 3D meshes and point clouds*, in Eurographics 2020 - Short Papers, The Eurographics Association, 2020, pp. 29–32. https://doi.org/10.2312/egs.20201010.

[21] V. Vidal, C. Wolf, and F. Dupont, *Combinatorial Mesh Optimization*, The Visual Computer, 28 (2012), pp. 511–525. https://doi.org/10.1007/s00371-011-0649-9.