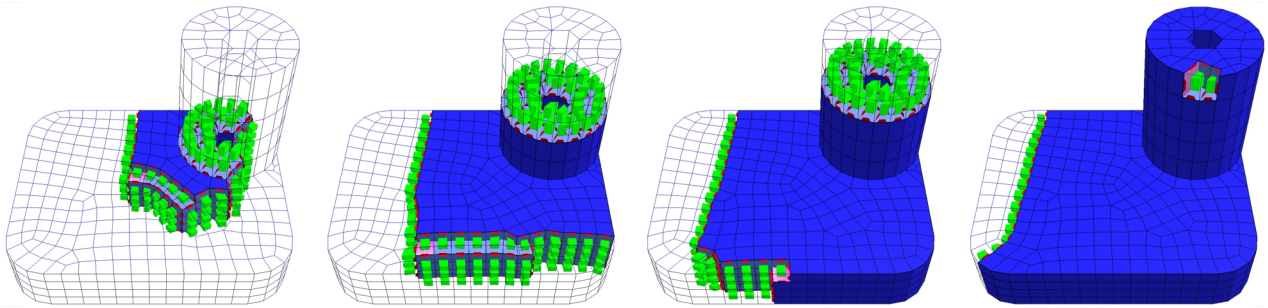
Compressing Hexahedral Volume Meshes

Martin Isenburg*
University of North Carolina
at Chapel Hill

Pierre Alliez†
INRIA
Sophia-Antipolis



Abstract

Unstructured hexahedral volume meshes are of particular interest for visualization and simulation applications. They allow regular tiling of the three-dimensional space and show good numerical behaviour in finite element computations. Beside such appealing properties, volume meshes take huge amount of space when stored in a raw format. In this paper we present a technique for encoding connectivity and geometry of unstructured hexahedral volume meshes.

For connectivity compression, we extend the idea of coding with degrees as pioneered by Touma and Gotsman [30] to volume meshes. Hexahedral connectivity is coded as a sequence of edge degrees. This naturally exploits the regularity of typical hexahedral meshes. We achieve compression rates of around 1.5 *bits per hexahedron (bph) that go down to* 0.18 *bph for regular meshes. On our test meshes the average connectivity compression ratio is* $1:162.7$.

For geometry compression, we perform simple parallelogram prediction on uniformly quantized vertices within the side of a hexahedron. Tests show an average geometry compression ratio of $1:3.7$ *at a quantization level of* 16 *bits.*

1. Introduction

Unstructured volume meshes can be found in a broad spectrum of scientific and industrial applications including fluid mechanics, thermodynamics and structural mechanics, where such volumetric data is used for both, computation and visualization. Traditionally unstructured volume meshes were composed of tetrahedral elements, but recently also other polyhedra have become popular. Especially hexahedral volume meshes are often used, because of their numerical advantages in finite element computations.

The generation of hexahedral meshes turned out to be much more complex than that of tetrahedral meshes, but the research efforts of the last years have produced several efficient techniques [29, 24, 25, 5, 21] (see [19] for a more complete list). At the same time researchers have proposed strategies for efficient visualization of unstructured volume meshes using screen-based ray-casting [8, 3, 34] or object-based sweeping [32, 6, 18] (see [7] for a survey about rendering unstructured volume grids).

The basic ingredients of unstructured hexahedral volume meshes can be classified into three things: mesh *connectivity*, that is the incidence relation among the vertices, edges, faces, and hexahedra, mesh *geometry*, that is the 3D position associated with each vertex, and application-specific mesh *properties* such as density or pressure values that are typically attached to the vertices.

The standard representation for hexahedral meshes uses three floating-point coordinates per vertex to store geometry and eight integer indices per hexahedron to store connectivity. For hexahedral meshes of v vertices and h hexahedra, this requires $96v$ bits for the geometry and $256h$ bits for the connectivity, if standard 4 byte data types are used. The mesh *c1* from our test set has 78618 vertices and 71572 hexahedra. The storage requirements for geometry and connectivity of this mesh can be estimated as 3.23 mega-bytes.

For archival and fast transmission of the data more compact representations are beneficial. In order represent mesh geometry more compactly, each coordinate can be quan-

*isenburg@cs.unc.edu
†pierre.alliez@sophia.inria.fr

tized with, for example, 16 bits. For data sets destined to be used in exact computations a loss in precision is sometimes not acceptable. However, for the purpose of volume mesh visualization this is usually not a problem as long as visual artifacts are avoided. In order represent mesh connectivity more compactly, each index can be specified with $\lceil \log_2 v \rceil$ bits by crossing the byte boundaries. For the mesh *c1* this more compact representation still requires 1.69 mega-bytes. Using the compression technique proposed here, this mesh can be represented at the same quality with less than 84 kilobytes—a compression by a factor of twenty.

Although we only focus on compression of connectivity and geometry, the same technique used to compress vertex positions can be adapted to also compress the properties. There have been several publications concerning the compression of tetrahedral volume meshes [27, 9, 22, 34], but we are not aware of a compression scheme that can handle hexahedral volume meshes.

We code the connectivity of a hexahedral mesh mainly as a sequence of its edge degrees that is subsequently compressed with an arithmetic coder [33]. Degree-based connectivity coding has already been successfully used for surface meshes. It was first proposed for purely triangular meshes [30] and was later generalized to the polygonal case [11, 15]. In this paper we extend this approach to volume meshes. We code the geometry of a hexahedral mesh as a sequence of corrective vectors that are also compressed with arithmetic coding. Whenever possible, a vertex position is predicted within the side of a hexahedron using a single parallelogram prediction [30].

2. Related Work

Compared to the number of publications on compression of polygonal surface meshes [4, 28, 30, 10, 20, 23, 2, 13, 14, 1, 11, 15, 16, 17] there are relatively few on compression of polyhedral volume meshes [27, 9, 22, 34]. Reason for this is probably the fact that volume meshes are not as widespread as surface meshes. Volumetric data sets are mostly found in scientific and industrial applications.

The immense amount of data required to represent polyhedral volume meshes makes compression even more worthwhile than in the surface case. This is especially true for the connectivity: The standard indexed representation uses $6/4$ indices per vertex for triangular/quadrilateral surface meshes, but approximately $12/8$ indices per vertex for typical tetrahedral/hexahedral volume meshes.

The challenge to compress the connectivity of tetrahedral volume meshes has first been approached by Szymczak and Rossignac [27]. Their "Grow&Fold" technique codes tetrahedral connectivity using only slightly more than 7 bits per tetrahedron for meshes with a manifold border surface. The encoding process builds a tetrahedral spanning tree that is rooted in an arbitrary border triangle. This tree is encoded with 3 bits per tetrahedron that indicate for all faces but the *entry face* whether the spanning tree will continue growing. The boundary of the tetrahedron spanning tree, a triangular surface mesh, has an associated a *folding string* that is represented with 4 bits per tetrahedron. This string describes how to "fold" and occasionally "glue" the boundary triangles of the spanning tree to reconstruct the original connectivity. The indices associated with the "glue" operations lift the bit-rate above 7 bits per tetrahedron, but their rare occurrence introduces only a small overhead.

Gumhold *et al.* have extended their connectivity coder for triangular surface meshes [10] to tetrahedral volume meshes [9]. Their algorithm performs a space growing process that maintains a *cut-border*, a (possibly non-manifold) triangle surface mesh, that separates at any time the processed tetrahedra from the unprocessed ones. Each iteration of the algorithm processes a triangle on the cut-border either by declaring it a "border" face or by including its adjacent tetrahedron into the cut-border. The latter requires to specify the fourth vertex of the tetrahedron: Either it is a "new vertex" or it is already on the cut-border, in which case a "connect" operation is needed. This operation uses a local indexing scheme to specify the fourth vertex on the cut-border. Because of the order in which the cut-border triangles are processed, the fourth vertex is often very close to the processed triangle, which results in small local indices. The average bit-rate for connectivity is about 2 bits per tetrahedron, a result that has not been challenged since.

Besides coding the mesh connectivity, the authors also describe two approaches to compress mesh geometry. Vertex coordinates are compressed when a vertex is encountered for the first time (e.g. during the "new vertex" operation). The first approach uses pre-quantized vertices, predicts their position as the center of the currently processed cut-border triangle, and codes only a corrective vector. The second approach quantizes a vertex after expressing it in a local coordinate frame whose z-axis is the normal of the currently processed cut-border triangle. In both approaches the resulting 16-bits correction vectors are split into four packages of 4 bits, which are then entropy encoded with separate arithmetic contexts. The authors report that more sophisticated prediction schemes failed, essentially because "tetrahedral meshes are too irregular to predict vertex coordinates much better than with the proximity information of the connectivity alone". At 16 bits of precision they report an average geometry compression ratio of $1 : 1.6$.

Yang *et al.* propose a compression technique for tetrahedral meshes that allows to streamline decoding and rendering of a volume mesh [34]. Their technique can significantly reduce the memory requirements of a ray-casting-based volume mesh renderer. The contribution of tetrahedra to the intersected rays is incrementally composited as they are decompressed. As soon as a decoded tetrahedron is no

longer needed it is discarded and its memory is de-allocated. This allows to render compressed tetrahedral meshes without ever having to store a completely uncompressed mesh.

First, they encode the surface formed by the border triangles using a triangle mesh compression scheme [20]. Then, they grow the border surface *inwards* by processing the adjacent tetrahedra using a breadth-first traversal. Similar to [9] a tetrahedron is encoded by specifying its fourth vertex. In case the fourth vertex was already visited they specify it using one of three different operations instead of the universal "connect" from [9]. When the fourth vertex is connected across a "face" or an "edge", they use a local index into an enumeration of adjacent faces or adjacent edges. Otherwise they use a global "index" into the list of all already visited vertices. The resulting connectivity compression rates are slightly above those of [9].

Simplification techniques for tetrahedral meshes have been proposed independently by Staadt and Gross [26] and Trotts *et al.* [31]. An iterative process collapses edge after edge, thereby removing all tetrahedra incident to them. At each stage it picks the edge whose collapse results in the minimal error according to some cost function. This simplification technique can be used to create a single mesh of a certain resolution, but it also allows to construct a progressive multi-resolution representation from which meshes at various levels of resolution can be extracted on the fly. The latter requires to store a sequence of inverse edge collapse operations, often referred to as *vertex splits*.

A compact and progressive encoding of the sequence of vertex splits was proposed by Pajarola *et al.* [22]. Instead of coding each vertex split individually, their *Implant Spray* technique codes entire batches of *independent* refinement operations at once. This reduces the average cost for identifying a split vertex from $O(log_2 v)$ to $O(1)$. Additionally the *skirt* of each split vertex has to be encoded, which specifies the set of faces that are split. The bit-rates for this progressive representation of tetrahedral mesh connectivity are reported to be less than 6 bits per tetrahedron. The authors note that the progressive nature of the connectivity encoding suggests that efficient geometry compression should be possible, but no experimental results are given.

3. Preliminaries

A *hexahedral mesh* or a *hexahedralization* is a collection of *hexahedra* that intersect only along shared faces, edges, or vertices. A hexahedron is a polyhedron that has six faces, eight vertices, and twelve edges, where each edge is adjacent to two faces, each vertex is adjacent to three faces and each face is a quadrilateral. A face is an *interior face* if it is shared by two hexahedra, otherwise it is a *border face*. Around each edge we find a cycle of faces and hexahedra. An edge is an *interior edge* if all its surrounding faces are interior faces, otherwise it is a *border edge*. A vertex is an *interior vertex* if all its incident edges are interior edges, otherwise it is a *border vertex*. In the following we denote the number of hexahedra with h, the number of faces with $f = f_i + f_b$, the number of edges with $e = e_i + e_b$, the number of vertices with $v = v_i + v_b$, where i stands for interior and b for border. A volume mesh has genus g if one can perform cuts through g closed border loops without disconnecting the underlying volume; such a volume is topologically equivalent to a sphere with g handles.

A volume mesh is *manifold* if each edge has a neighborhood that is homeomorphic to a cylinder or a half-cylinder and each vertex has a neighborhood that is homeomorphic to a sphere or a half-sphere. Edges with half-cylinder neighborhoods and vertices with half-sphere neighborhoods are on the border. The border of a manifold volume mesh is a manifold surface mesh.

The *degree* of an edge is the number of faces adjacent to the edge. For interior edges this corresponds to the number of hexahedra adjacent to the edge. For border edges this corresponds to the number of hexahedra adjacent to the edge plus the number of border openings. In the manifold case a border edge has only one border opening. The degrees of interior edges tend to have a different distribution (e.g. tend to be higher) than the degrees of border edges.

Two hexahedra are *face-adjacent* if they share a face, *edge-adjacent* if they only share an edge, and *vertex-adjacent* if they only share a vertex. A hexahedral mesh may consist of one or more connected components. A component is *face-connected* if there is a path of face-adjacent hexahedra between any two hexahedra. A component is still *edge-connected* if there is at least a path of edge-adjacent hexahedra between any two hexahedra. Otherwise the component is only *vertex-connected*.

4. Coding Connectivity with Degrees

The concept of coding connectivity with degrees was introduced by Touma and Gotsman [30] for the case of triangular surface meshes, which can be coded through a sequence of vertex degrees. The achieved bit-rates are mainly dictated by the distribution of vertex degrees. This automatically adapts to *regularity* in the mesh, which we loosely define as how regular it tiles the domain it lives in. A surface mesh consisting of only equilateral triangles constitutes a perfectly regular tiling of the 2D domain. Since the degree of all vertices of such a mesh is 6, the vertex degree distribution has an entropy of zero.

Degree coding was recently generalized to polygonal connectivity [11, 15] by using both, a sequence of vertex degrees and a sequence of face degrees. The adaptivity of the coding scheme naturally extends to the other two regular tilings of the 2D domain: using squares, all face degrees and also all vertex degrees are 4; and using regular hexagons, all face degrees are 6 and all vertex degrees are 3.

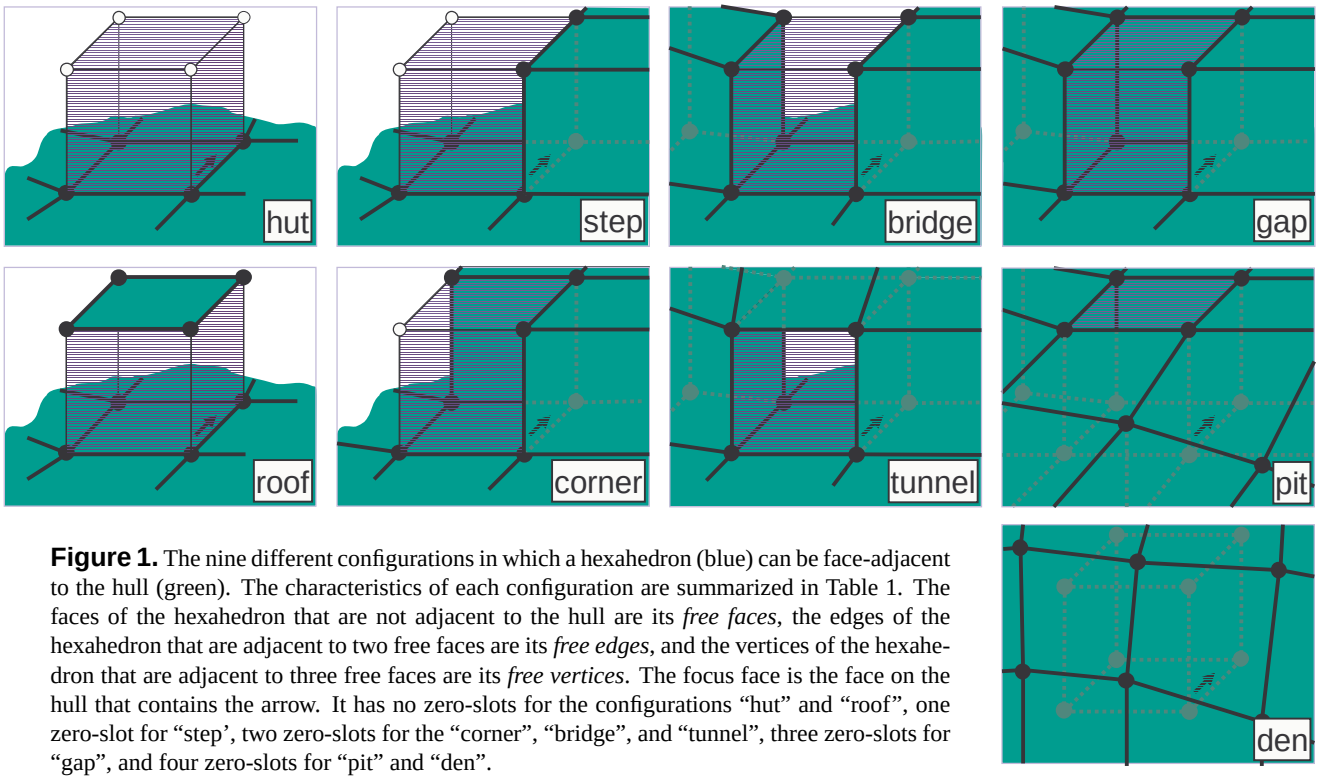


Figure 1. The nine different configurations in which a hexahedron (blue) can be face-adjacent to the hull (green). The characteristics of each configuration are summarized in Table 1. The faces of the hexahedron that are not adjacent to the hull are its *free faces*, the edges of the hexahedron that are adjacent to two free faces are its *free edges*, and the vertices of the hexahedron that are adjacent to three free faces are its *free vertices*. The focus face is the face on the hull that contains the arrow. It has no zero-slots for the configurations "hut" and "roof", one zero-slot for "step', two zero-slots for the "corner", "bridge", and "tunnel", three zero-slots for "gap", and four zero-slots for "pit" and "den".

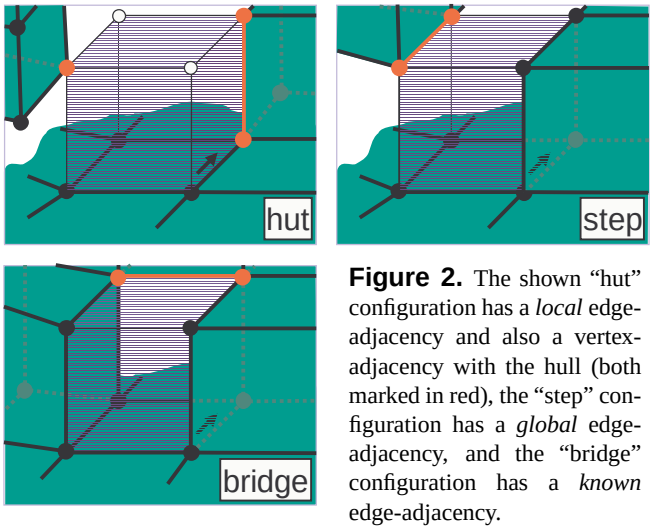


Figure 2. The shown "hut" configuration has a *local* edge-adjacency and also a vertex-adjacency with the hull (both marked in red), the "step" configuration has a *global* edge-adjacency, and the "bridge" configuration has a *known* edge-adjacency.

In the following we show that the concept of degree coding can be extended to compress the connectivity of hexahedral meshes using its *edge degrees*. Going from surface meshes to volume meshes we can think of the vertices getting stretched into edges; what was a vertex degree in the surface mesh, becomes an edge degree in the volume mesh.

Hexahedral meshes allow a regular tiling of the 3D domain. A cube is a hexahedron whose six faces are square and meet each other at right angles. It is the only of the five platonic solids that regularly tiles the 3D domain. The interior edges of a perfectly regular hexahedral mesh all have degree 4. Fortunately, many hexahedral meshes found in practice are fairly regular and exhibit a low dispersion in edge degrees. The equilateral tetrahedron, on the other hand, does not permit a tiling of 3D space. In fact, tetrahedral meshes seem irregular by nature. Although degree coding can be adapted for tetrahedral connectivity, initial measurements on the edge degree distributions of various tetrahedral meshes suggests that the achievable compression rates will be lower than those of [9, 34].

5. Compressing the Connectivity

The encoder and the decoder perform the same space growing process to compress and uncompress a connected component of a hexahedral mesh. Each iteration of the algorithm processes a hexahedron that is adjacent to one or more previously processed hexahedra. In face-connected components this hexahedron is always face-adjacent; in edge-connected or vertex-connected components this hexahedron is sometimes only edge-adjacent or vertex-adjacent. In order to simplify the description of our compression method we assume face-connected components. The two necessary extensions for dealing with components that are only edge-connected or vertex-connected are straight-forward.

Four arithmetic contexts [33] are used for compressing the symbols that encode hexahedral connectivity. One for border edge degrees, one for interior edge degrees, and two binary contexts. One of two will distinguish border elements from interior elements, and the other will mark the

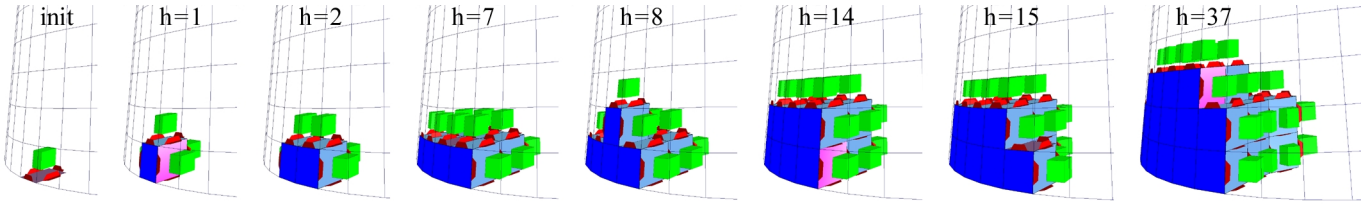


Figure 3. A close-up on the *fru* mesh at the beginning of the encoding process. Final faces are dark blue, incomplete faces are light blue, the focus face is pink, the slots are red, and all hexahedra face-adjacent to the hull are shown in green: The leftmost frame shows the initial hull. The next two frames show the hull after processing the first two tetrahedra. The rightmost frame shows the hull after 37 tetrahedra have been processed.

# of	hut	roof	step	corner	bridge	tunnel	gap	pit	den
adjacent faces	1	2	2	3	3	4	4	5	6
zero-slots	0	0	1	2	2	2	3	4	4
free faces	5	4	4	3	3	2	2	1	-
free vertices	4	-	2	1	-	-	-	-	-
free edges	8	4	5	3	2	-	1	-	-
(global)	4	-	1	-	-	-	-	-	-
(local)	4	-	4	3	-	-	-	-	-
(known)	-	4	-	-	2	-	1	-	-

Table 1. This table characterizes the nine configurations in which a hexahedron can be face-adjacent to the hull (see Figure 1). It lists the number of adjacent faces, the number of zero-slots of the focus face, and the number of free vertices, free faces, and free edges. The free edges are further classified into the number of potential candidates for global, local, or known edge-adjacency with the hull (see Figure 2).

infrequent occurances of "join" operations discussed below.

The algorithm maintains a *hull* that encloses at any time all processed hexahedra. This hull is a quadrilateral surface mesh, possibly non-manifold, whose edges and faces are called *hull edges* and *hull faces* respectively. The hull faces are classified as *final faces* and *incomplete faces*. A final face is a border face whose corresponding hexahedron has already been processed. An incomplete face is an interior face that has a processed hexahedron on one side and an unprocessed hexahedron on the other side. Each hull edge maintains a *slot-count* that specifies the remaining number of faces still to be added between its two adjacent hull faces. A hull edge is a *zero-slot* if its two hull faces are incomplete and its slot-count is zero. A hull edge is a *border-slot* if one of its hull faces is final and the other incomplete and its slot-count is one. The number of zero-slots and border-slots around an incomplete face is always between 0 and 4.

The initial hull is defined around a border face by recording the degrees of its four border edges. It has one final face, one incomplete face, and four hull edges. The slot-count of the hull edges is initialized to their degree minus one. In each iteration the algorithm selects an incomplete face as the *focus face* and processes the unprocessed hexahedron it is adjacent to (see Figure 3). Processing of a (face-adjacent) hexahedral mesh component is completed, when the hull consists only of final faces.

The currently processed hexahedron can be in one out of nine configurations face-adjacent to the hull; these are shown in Figure 1 and characterized in Table 1. Both, encoder and decoder, can determine the actual configuration based on the number of zero-slots in the vicinity of the focus face. Only when the focus face has no zero-slots, the ambiguity between the "hut" and the "roof" configuration needs to be coded explicitly. In case of the latter the encoder also needs to specify the incomplete face that the "roof" is formed with. Processing the hexahedron involves:

- recording if its free faces are border or interior faces;
- recording if its free edges are border or interior edges;
- recording the degrees of its free edges;
- predicting the positions of its free vertices;
- and updating the hull and the slot-counts appropriately.

The edge degree distribution of border edges is different from that of interior edges. While border edge degrees typically have a spread around 3, interior edge degrees average around 4 as documented in Table 2. It is therefore beneficial to compress them with different arithmetic contexts.

5.1. Propagating the Border Information

The proposed algorithm only needs to distinguish border faces from interior faces. Using this information all edges can eventually be classified as border or interior. However, in order to compress an edge degree with the appropriate arithmetic context, we need know this in the moment its degree is encoded. By using simple rules and by selecting a suitable focus face (see Subsection 5.3) we can propagate the information about the border. Most of the time encoder and decoder can deduce whether faces or edges are on the border without explicitly encoding it. The rules are:

rule $\mathbf{R}_1$ A free face is a border face if it connects to a border face across an edge with a slot-count of zero.

rule $\mathbf{R}_2$ A free face is an interior face if any adjacent edge is known to be an interior edge.

rule $\mathbf{R}_3$ A free edge is a border edge if any adjacent face is known to be a border face.

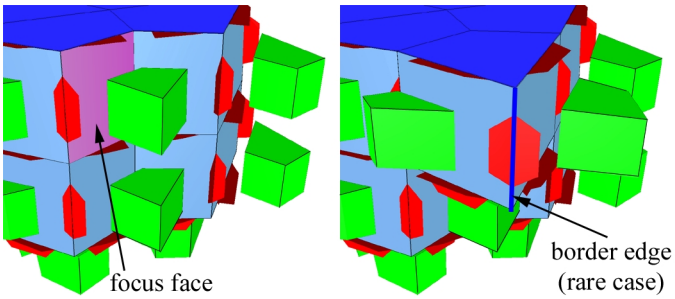


Figure 4. Propagating the border information in the "step" configuration: The free face at the top is a border face because of rule R_1. All other free faces are interior faces because of rule R_2. The two free edges at the top are border edges because of rule R_3. For the remaining three free edges this needs to be specified explicitly. Usually these would all be interior edges, but this example shows a rare scenario where one of them is a border edge.

The example in Figure 4 illustrates these rules. Whenever none of the rules applies a binary arithmetic context is used to encode explicitly if an edge or a face is on the border or not. Using these rules on our test meshes approximately 99 percent of all border elements are classified as such. The arithmetic coder is mostly used to specify that an edge or a face is interior. This requires only very few bits because the same symbol will be coded again and again.

5.2. Join Operations

For every "roof" configuration it is necessary to specify the incomplete face on the hull that forms the "roof". Furthermore, sometimes free edges are edge-adjacent or free vertices are vertex-adjacent to the hull as illustrated in Figure 2. Instead of recording the degree of such an edge or predicting the position of such a vertex, the encoder has to specify *how* they are adjacent to the hull such that the decoder can replay exactly the same updates. We use the following "join" operations for this:

Joining free vertices is done by identifying the respective vertex with an index between 0 and the current count of vertices v_{cc} minus one, which is coded with $log_2(v_{cc})$ bits.

Joining free edges is done by identifying the respective hull edge, which has at least two slots, and by specifying how the "join" divides its slot-count.

We identify the respective hull edge in three different ways, depending on the type of edge-adjacency: *known*, *local*, or *global* (see Figure 2). For the *known* type we know the two vertices in whose linked lists the respective hull edge must appear. In most cases this will leave us with a unique candidate. For the *local* type we know only one vertex in whose linked list the respective hull edge must appear. Its position in this list is addressed with an index between 0 and the current number of edges $e_{s>=2}$ of this list that have a slot-count of 2 or higher minus one, which is coded with $log_2(e_{s>=2})$ bits. For the *global* type we must furthermore explicitly address one of the vertices in whose linked list the respective hull edge appears using $log_2(v_{cc})$ bits.

Specifying how the "join" divides the s slots of the respective hull edge can be coded with $log_2(s-2)$ bits, as 2 slots are consumed during the "join".

Joining the "roof" is done by identifying one of the hull edges of the respective incomplete face. We specify this edge, which has at least one slot, by addressing the vertex in whose linked list it appears and its position in this list. Addressing the vertex is again coded with $log_2(v_{cc})$ bits. The position of the respective hull edge in this list is addressed with an index between 0 and the current number of edges $e_{s>=1}$ of this list that have a slot-count s of 1 or higher minus one, which is coded with $log_2(e_{s>=1})$ bits.

mesh	border edge degrees					interior edge degrees					
name	total	2	3	4	>4	total	2	3	4	5	>5
hanger	768	.17	.77	.06	–	149	–	.01	.98	.01	–
ra	792	.17	.79	.04	–	856	–	.03	.95	.02	–
bump2	1780	.08	.88	.03	.01	2708	–	.04	.94	.01	–
test	2928	.12	.87	.01	–	5774	–	–	1.0	–	–
mdg-1	3004	.06	.94	–	–	9676	–	.01	.98	.01	–
c2	3924	.07	.91	.02	–	10247	–	.02	.96	.02	–
fru	2872	.04	.97	–	–	11689	–	.03	.96	.02	–
shaft	8788	.08	.90	.02	.01	16392	.01	.03	.95	.02	.01
warped	4800	.05	.95	–	–	21660	–	–	1.0	–	–
hutch	2336	.03	.94	.02	–	23381	–	.01	.98	.01	–
c1	27428	.03	.97	.01	–	201190	–	.01	.98	.01	–
average		.08	.90	.02	.00		.00	.02	.97	.01	.00

Table 2. This table reports the degree distribution for border and interior edges in our data sets. Border edge degrees spread around 3; interior edge degrees spread around 4.

5.3. Minimizing the Number of Join Operations

Coding "join" operations requires local or even global indexing. This is expensive and we would like to do this as seldom as possible. If the mesh has handles then there will always be at least one "roof" configuration, one global edge-adjacency, or one vertex-adjacency per handle (see Figure 5). Unfortunately these can also happen otherwise and the frequency of their occurance is strongly dependent on the strategy used for selecting the next focus face. This problem is very similar to the occurance of "split" operations in surface mesh connectivity coding [30, 13, 10, 23].

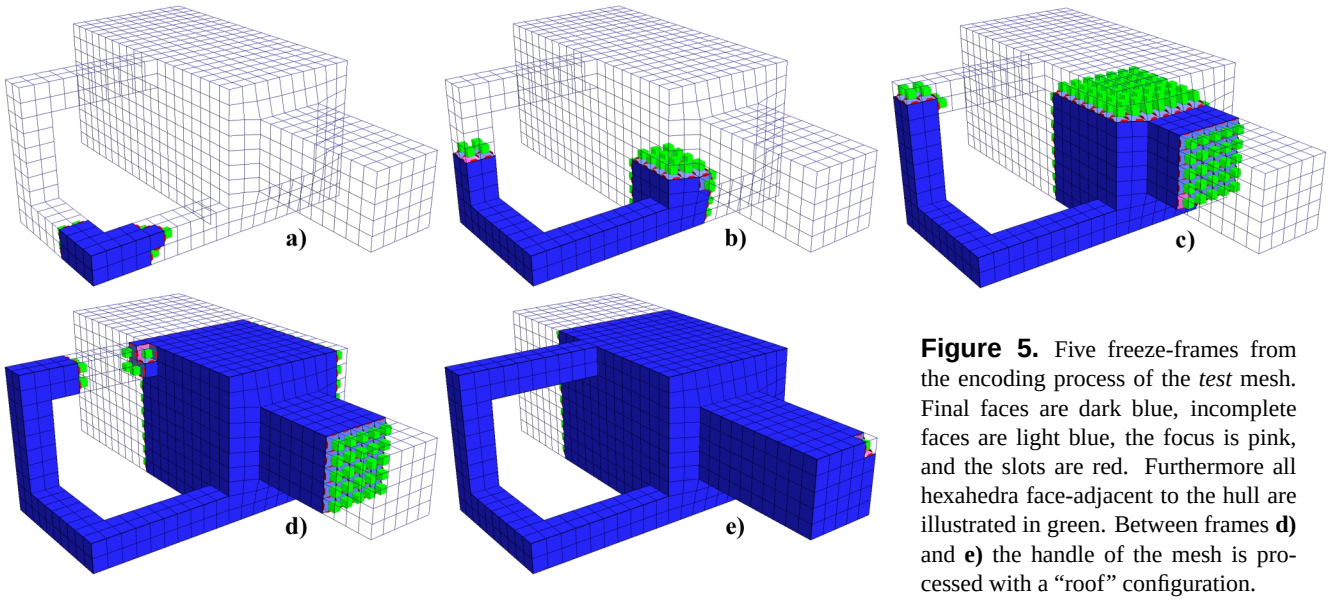


Figure 5. Five freeze-frames from the encoding process of the *test* mesh. Final faces are dark blue, incomplete faces are light blue, the focus is pink, and the slots are red. Furthermore all hexahedra face-adjacent to the hull are illustrated in green. Between frames **d)** and **e)** the handle of the mesh is processed with a "roof" configuration.

Adaptive traversal strategies have been proposed that successfully reduce the number of these operations [1, 11].

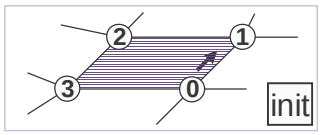
The heuristics used by adaptive traversal strategies pick a focus such that the creation of cavities during the region growing process is avoided. The focus is first moved to vertices on the boundary that are nearly completed (e.g. that have a low slot-count). We use a similar heuristic for avoiding the creation of cavities in our space growing process. The focus is moved to an incomplete face with the highest number of zero-slots. This strategy is very successful on our set of hexahedral meshes. Only for one data set, the *hutch* mesh, we need a "join" operation that is not due to a handle. This happens because during encoding the hull has temporarily the topology of a torus (see Table 4).

In case there is no face with zero-slots, a face with border-slots is selected as the focus face. This increases the success rate of the border propagation described earlier. If there is also no face with border-slots, an arbitrary incomplete face is selected in some agreed-upon way.

6. Compressing the Geometry

We use the traversal order on the vertices induced by the connectivity coder to compress their associated positions with a predictive coding scheme. In order to use such a scheme the floating-point positions are first uniformly quantized using a user-defined precision of for example 10, 12, 14, 16 or even 18 bits per coordinate. This introduces a quantization error as some of the floating-point precision is lost. In some applications it might therefore be preferable not to compress the mesh geometry. Then a prediction rule is applied that represents each quantized position as an offset vector that corrects the predicted position to the actual position. The values of these corrective vectors tend to spread around zero, which means they can be efficiently compressed with, for example, an arithmetic coder [33].

For compressing the vertex positions of triangle meshes several prediction methods have been proposed. The simplest prediction method that predicts the next position as the last position was suggested by Deering [4]. This is also known as delta coding. A more sophisticated scheme is the *spanning tree predictor* by Taubin and Rossignac [28] that uses a weighted linear combination of previously decoded vertices; the particular coefficients used can be optimized for each mesh. A similar, but much simpler scheme is the *parallelogram predictor* introduced by Touma and Gotsman [30]. This is the predictor we will use.



vertex	prediction rule
v_0	0
v_1	v_0
v_2	v_1
v_3	$v_0 - v_1 + v_2$
v_4	$2v_0 - v_8$ (*or* v_0)
v_5	$v_1 - v_0 + v_4$
v_6	$v_2 - v_1 + v_5$
v_7	$v_3 - v_2 + v_6$

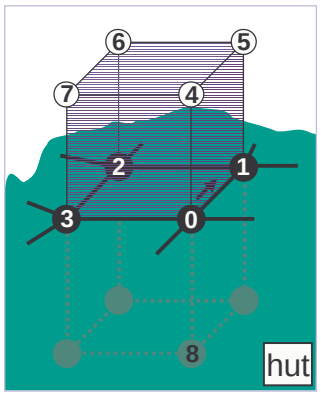


Figure 6. This figure illustrates how vertex positions are predicted: The rules for v_0 to v_3 are only used for the vertices of the initial hull. All other vertices are predicted during a "hut", a "step, or a "corner" configuration using the rules for v_4 to v_7. The first "hut" configuration needs to use a different prediction rule for v_4, since v_8 will not exist.

This scheme predicts vertex positions to complete a parallelogram spanned by the three previously processed vertices. Good predictions are those that predict a position close to its actual position. In the triangle mesh case the parallelogram predictor gives good predictions if used across two triangles that are in a fairly planar and convex position to each other. Consequently, the parallelogram predictor gives bad predictions if used across triangles that are in a highly non-planar and/or non-convex position.

When compressing polygonal meshes it is possible to improve the number of good predictions by letting the polygons dictate where to apply the parallelogram predictor [12]. Since polygons tend to be fairly planar and fairly convex, it is beneficial to make predictions *within* a polygon rather than *across* polygons. This, for example, avoids poor predictions due to a crease angle between polygons.

In similar spirit we predict most vertex positions within the side of a hexahedron in the moment they are first encountered using the rules illustrated in Figure 6. Four vertices are encountered during initialization of the hull, all others are encountered as free vertices of a "hut", "step", or "corner" configuration. The first vertex v_0 has no obvious predictor and is predicted as 0. Also the next two vertices v_1 and v_2 cannot yet use parallelogram prediction and are predicted as a previously processed position. This makes a systematic prediction error, but there will be only two such predictions per mesh component. For most following vertex positions we use the parallelogram predictor. An exception is vertex v_4 of the "hut" configuration, which is predicted by extending the ray from v_8 to v_0 (if vertex v_8 exists).

Predictive geometry compression does not scale with increasing precision. The achievable compression ratio is strongly dependent on the number of precision bits. Since this technique mainly predict away the high-order bits, the compression ratios decrease if more precision (= low bits) is added. This is clearly demonstrated by the results in Table 3, which reports the performance of our geometry compression scheme at different levels of precision.

mesh	10 bits		12 bits		14 bits		16 bits		18 bits	
name	bpv	ratio	bpv	ratio	bpv	ratio	bpv	ratio	bpv	ratio
hanger	11.2	2.7	15.4	2.3	19.6	2.1	23.2	2.1	26.5	2.0
ra	14.5	2.1	19.9	1.8	25.2	1.7	30.8	1.6	36.2	1.5
bump2	9.5	3.1	14.2	2.5	19.1	2.2	24.4	2.0	29.8	1.8
test	1.8	17.0	3.3	11.0	4.3	9.8	5.9	8.2	6.5	8.3
mdg-1	5.3	5.6	7.7	4.7	10.1	4.2	12.3	3.9	14.4	3.8
c2	5.0	6.0	7.5	4.8	10.7	3.9	14.2	3.4	17.6	3.1
fru	7.1	4.2	12.0	3.0	17.1	2.5	23.1	2.1	29.1	1.9
shaft	6.8	4.4	10.6	3.4	15.2	2.8	19.9	2.4	24.8	2.2
warped	3.4	8.8	5.1	7.1	7.9	5.3	10.5	4.6	13.2	4.1
hutch	8.1	3.7	11.6	3.1	16.1	2.6	19.9	2.4	23.9	2.3
c1	1.5	19.7	2.7	13.3	4.1	10.2	5.9	8.1	8.0	6.8
average		7.0		5.2		4.3		3.7		3.4

Table 3. This table reports bit-rates for compressed geometry in bits per vertex (bpv) at different quantization levels and gives the corresponding compression ratio compared to uncompressed geometry. The bit-rate for uncompressed geometry is simply three times the number of precision bits.

```
class SpinEdge {
  Vertex* vertex;
  SpinEdge* next;
  SpinEdge* inv;
  SpinEdge* spin;
  SpinEdge* list;
  int on_border;
  int slots;
}
```

```
class Vertex {
  int index;
  SpinEdge* edge_list;
  float p[3];
}

SpinEdge* face_list[5];
Vertex* permutation[];
```

Figure 7. The data structures used for compression: The connectivity of the hexahedral mesh is captured by the `next`, `inv`, and `spin` pointers. The geometry is attached by the `vertex` pointer. The `list` pointer is used for all linked-lists: One list per vertex, starting at the `edge_list` pointers, links all incomplete edges incident to a vertex. Furthermore five lists, starting at the `face_list` pointers, link incomplete faces that have either border-slots, or one, two, three, or four zero-slots.

7. Implementation and Results

The data structures used by encoder and decoder are shown in Figure 7. The spin-edges that store mesh connectivity are a straight-forward extension of standard twin-edges and are similar to those used in [18]. Each hexahedron uses 24 spin-edges, 4 per face, and also border faces are represented explicitly. Therefore each face has two sets of 4 spin-edges, whose `list` pointers are used to maintain two kinds of single-linked lists during encoding and decoding. One set is used to link all spin-edges a vertex has on the hull to its `edge_list` pointer. These lists are used to address an edge during a "join" operation. The other set is used to link spin-edges on the hull that either have border-slots, or one, two, three, or four zero-slots into five priority lists. These five lists are used to select the next focus face. Spin-edges are inserted into and removed from a list at most once. After leaving the hull they are not explicitly deleted, but marked invalid and removed the next time encountered. Hence, maintaining these lists has a linear time complexity.

Compression results for connectivity and geometry for a set of eleven test meshes are listed in Table 4. The bit-rates for connectivity are strongly dependent on the *compacity* of the mesh, which can be characterized by the ratio of border elements. The fraction of border vertices v_b/v and border edges e_b/e, for example, but also the number border faces per hexahedron f_b/h can be used as a measure of compacity. The less compact a mesh, the bigger the impact of the costs for encoding its border. The *hanger* mesh, for exam-

ple, is closer to a surface mesh than to a volume mesh. Although its bit-rate of 5.30 bits per hexahedron seems high, expressed as 2.65 bits per vertex it is comparable to results in surface connectivity compression.

8. Summary and Future Work

We have introduced the first scheme for compressing hexahedral volume meshes. The connectivity is coded using an edge-degree based approach that naturally adapts to the regularity typically found in hexahedral meshes. For regular meshes the bit-rates go down to 0.18 bits per hexahedron while averaging at around 1.5, which corresponds to a compression ratio of $1 : 162$. The geometry is compressed by parallelogram prediction within a hexahedron, leading to a compression ratio of $1 : 3.7$ at a quantization level of 16 bits. Furthermore, we describe a data structure well suited to efficiently implement the selection strategy for the focus face and maintain the hull during encoding and decoding.

In the future we plan to generalize the degree-based approach to unstructured volume meshes containing arbitrary polyhedra. The final goal is an universal degree-based coder for irregular surface and volume meshes that obtains bit-rates competitive to those of a specialized coder. We would also like to compute a combinatorial worst-case analysis for our hexahedral connectivity compression rates.

9. Acknowledgements

We thank several researchers for providing their data sets: Alla Sheffer for *bump2*, *fru*, *shaft*, *c1*, and *c2*, Steven Owen for *test*, *mdg-1*, and *warped*, and Scott Mitchell for *hanger*, *hutch*, and *ra*. This work was done while both authors were at INRIA, Sophia-Antipolis and was supported in part by the ARC TeleGeo grant from INRIA.

References

[1] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3D meshes. In *Eurographics'01 Conf. Proc.*, pages 480–489, 2001.

[2] C. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In *Data Compression Conference'99 Conference Proceedings*, pages 247–256, 1999.

[3] P. Bunyk, A. Kaufmann, and C. Silva. Simple, fast, and robust ray casting of irregular grids. In *Proceedings of Dagstuhl'97*, pages 30–36, 2000.

[4] M. Deering. Geometry compression. In *SIGGRAPH'95 Conference Proceedings*, pages 13–20, 1995.

[5] D. Eppstein. Linear complexity hexahedral mesh generation. *Computational Geometry Theory and Applications*, 12:3–16, 1999.

[6] R. Farias, J. Mitchell, and C. Silva. Zsweep: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of Volume Visualization Symposium'00*, pages 91–99, 2000.

[7] R. Farias and C. Silva. Out-of-core rendering of large unstructured grids. *IEEE Computer Graphics & Applications*, 21(4):42–50, 2001.

[8] M. Garrity. Raytracing irregular volume data. *Computer Graphics*, 24(5):35–40, 1990.

[9] S. Gumhold, S. Guthe, and W. Strasser. Tetrahedral mesh compression with the cut-border machine. In *Visualization'99 Conference Proceedings*, pages 51–58, 1999.

[10] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *SIGGRAPH'98 Conf. Proc.*, pages 133–140, 1998.

[11] M. Isenburg. Compressing polygon mesh connectivity with degree duality prediction. In *Graphics Interface'02 Conference Proceedings*, pages 161–170, 2002.

[12] M. Isenburg and P. Alliez. Compressing polygon mesh geometry with parallelogram prediction. In *Visualization'02 Conference Proceedings*, pages 141–146, 2002.

[13] M. Isenburg and J. Snoeyink. Face Fixer: Compressing polygon meshes with properties. In *SIGGRAPH'00 Conference Proceedings*, pages 263–270, 2000.

[14] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *SIGGRAPH'00 Conference Proceedings*, pages 279–286, 2000.

[15] A. Khodakovsky, P. Alliez, M. Desbrun, and P. Schroeder. Near-optimal connectivity encoding of 2-manifold polygon meshes. to appear in *Graphic Models (Special Issue)*, 2002.

[16] B. Kronrod and C. Gotsman. Optimized compression of triangle mesh geometry using prediction trees. In *Proceedings of 1st International Symposium on 3D Data Processing, Visualization and Transmission*, pages 602–608, 2002.

[17] H. Lee, P. Alliez, and M. Desbrun. Angle-analyzer: A triangle-quad mesh codec. In *Eurographics'02 Proc.*, pages 383–392, 2002.

[18] B. Levy, G. Caumon, S. Conreaux, and X. Cavin. Circular incident edge list: A data structure for rendering complex structured grids. In *Visualization'01 Conference Proceedings*, pages 191–198, 2001.

[19] Meshing Corner *http://www.andrew.cmu.edu/~sowen/mesh.html*

[20] T. Mitra and T. Chiueh. A breadth-first approach to efficient mesh traversal. In *Proceedings of Eurographics Workshop on Graphics Hardware*, pages 31–38, 1998.

[21] M. Mueller-Hannemann. Shelling hexahedral complexes for mesh generation. *Journal of Graph Algo. and Appl.*, 5(5):59–91, 2001.

[22] R. Pajarola, J. Rossignac, and A. Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Visualization'99 Conference Proceedings*, pages 299–306, 1999.

[23] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. on Vis. and Comp. Graph.*, 5(1):47–61, 1999.

[24] R. Schneider, R. Schindler, and F. Weiler. Octree-based generation of hexahedral element meshes. In *Proceedings of the 5th International Meshing Roundtable*, pages 205–215, 1996.

[25] A. Sheffer, M. Etzion, A. Rappoport, and M. Bercovier. Hexahedral mesh generation using the embedded voronoi graph. In *Proceedings of the 7th International Meshing Roundtable*, pages 347–364, 1998.

[26] O. Staadt and M. Gross. Progressive tetrahedralizations. In *Visualization'98 Conference Proceedings*, pages 397–402, 1998.

[27] A. Szymczak and J. Rossignac. Grow & fold: Compression of tetrahedral meshes. In *Proceedings of the 5th ACM Symposium on Solid Modeling and Applications*, pages 54–64, 1999.

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

[29] T. Tautges and S. Mitchell. Whisker weaving: A connectivity-based based method for constructing all-hexahedral finite element meshes. In *Proc. of the 4th Intern. Mesh. Roundtable*, pages 115–127, 1995.

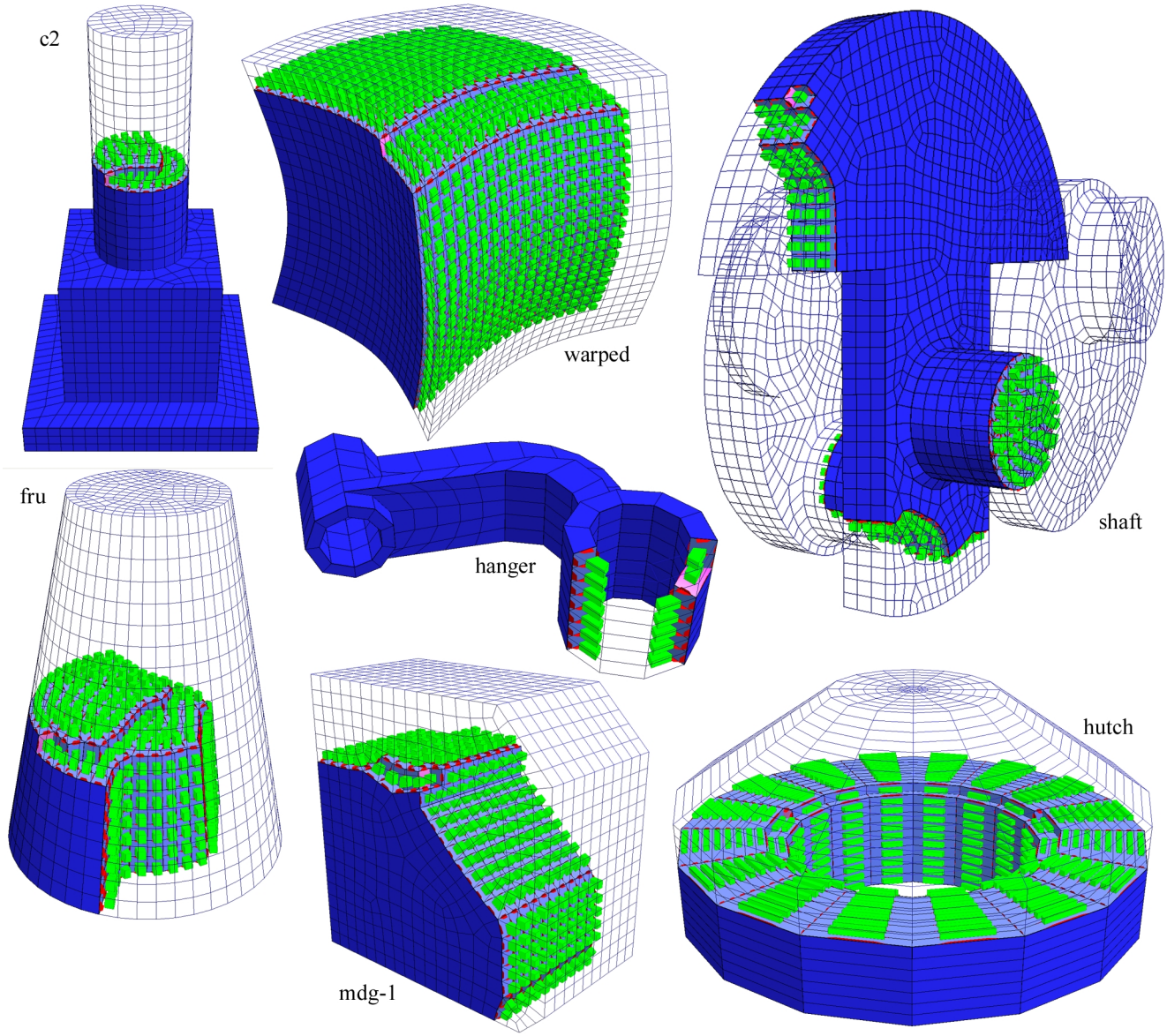
[30] C. Touma and C. Gotsman. Triangle mesh compression. In *Graphics Interface'98 Conference Proceedings*, pages 26–34, 1998.

[31] I. Trotts, B. Hamann, K. Joy, and D. Wiley. Simplification of tetrahedral meshes. In *Visualization'98 Conf. Proc.*, pages 287–295, 1998.

[32] J. Wilhelms, A. V. Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Visualization'96 Conf. Proc.*, pages 57–64, 1996.

[33] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[34] C. Yang, T. Mitra, and T. Chiueh. On-the-fly rendering of losslessly compressed irregular volume data. In *Visualization'00 Conference Proceedings*, pages 101–108, 2000.



mesh	mesh characteristics								mesh	connectivity (bph)			geometry (bpv)		
name	g	h	v	e	f_b	v_b/v	e_b/e	f_b/h	name	raw	coded	ratio	raw	coded	ratio
hanger	2	171	382	917	384	1.0	.84	2.25	hanger	72.0	5.30	13.6	48.0	23.19	2.1
ra	0	408	635	1648	396	.63	.48	0.97	ra	80.0	2.89	27.7	48.0	30.83	1.6
bump2	1	1189	1665	4480	890	.53	.40	0.75	bump2	88.0	2.10	41.9	48.0	24.41	2.0
test	1	2386	3198	8702	1464	.46	.34	0.61	test	96.0	0.87	110.3	48.0	5.85	8.2
mdg-1	0	3710	4510	12680	1502	.33	.24	0.40	mdg-1	104.0	0.77	135.1	48.0	12.30	3.9
c2	0	4046	5099	14171	1962	.39	.28	0.48	c2	104.0	1.31	79.4	48.0	14.24	3.4
fru	0	4360	5124	14561	1436	.28	.20	0.33	fru	104.0	0.98	106.1	48.0	23.12	2.1
shaft	0	6883	9218	25180	4394	.48	.35	0.64	shaft	112.0	1.70	65.9	48.0	19.93	2.4
warped	0	8000	9261	26460	2400	.26	.18	0.30	warped	112.0	0.18	622.2	48.0	10.45	4.6
hutch	0	8172	8790	25717	1168	.13	.09	0.14	hutch	112.0	0.31	361.3	48.0	19.88	2.4
c1	0	71572	78618	228618	13714	.17	.12	0.19	c1	136.0	0.60	226.7	48.0	5.91	8.1
average						.37	.27	0.48	**average**	101.8	1.55	162.7	48.0	17.28	3.7

Table 4. The table lists the genus g and number of hexahedra h, vertices v, edges e, and border faces f_b for each mesh. Furthermore the fraction of border vertices v_b/v and border edges e_b/e is given together with the number of border faces per hexahedra f_b/h. The bit-rates for uncompressed and compressed connectivity are reported in bits per hexahedron (bph). The bit rates for uncompressed and compressed geometry at 16 bits of precision are reported in bits per vertex (bpv). The corresponding compression ratios are also listed.