Compression of Non-Manifold Polygonal Meshes Revisited

Max von Buelow, Stefan Guthe and Michael Goesele

TU Darmstadt
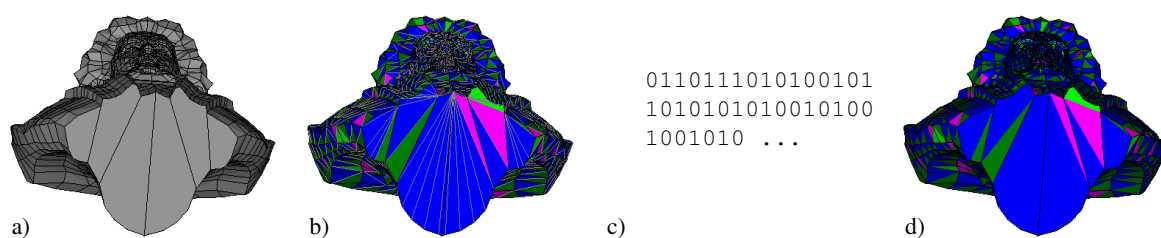
Figure 1: *A polygonal input mesh (a) is traversed and encoded as a stream of operations (marked using different colors) and attributes. During traversal, the mesh is implicitly converted into a triangle mesh (b) and stored as a bit stream (c). The final compressed representation allows us to recover the original polygons (d) rather than the intermediate triangle mesh.*

Abstract

Polygonal meshes are used in various fields ranging from CAD to gaming and web based applications. Reducing the size required for storing and transmitting these meshes by taking advantage of redundancies is an important aspect in all of these cases. In this paper, we present a connectivity based compression approach that predicts attributes and stores differences to the predictions together with minimal connectivity information. It is an extension to the Cut-Border Machine and applicable to arbitrary manifold and non-manifold polygonal meshes containing multiple attributes of different types. It compresses both the connectivity and attributes without loss outside of re-ordering vertices and polygons. In addition, an optional quantization step can be used to further reduce the data if a certain loss of accuracy is acceptable. Our method outperforms state-of-the-art compression techniques, including specialized triangle mesh compression approaches when applicable. Typical compression rates for our approach range from 2:1 to 6:1 for lossless compression and up to 25:1 when quantizing to 14 bit accuracy.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—
Graphics data structures and data types

1. Introduction

Triangle meshes and, more general, polygonal meshes are ubiquitous and have many applications. Currently used meshes typically contain millions or even billions of polygons. In order to store these efficiently, many compression approaches and, more recently, compression standards have been proposed. Initially, the meshes were stored in plain ASCII or binary format with generic compression (e.g. GZip). While such an approach already significantly reduces the storage requirements, it fails to take advantage of most of the redundancies in the underlying mesh, yielding sub-optimal compression rates.

Compression approaches can be classified as either progressive or single rate. Progressive approaches allow different levels of de-

tail and instant visualization of coarser levels while loading the remainder of the data. They tend, however, to offer lower compression rates than the single rate approaches. In terms of single rate approaches, several standards such as OpenCTM, WebGL Loader, Open3DGC and Google Draco have been proposed over time. However, all of these standards impose severe restrictions on the type of meshes (i.e. only triangle meshes) and/or the types of attributes they support.

In this paper, we propose a novel mesh compression scheme based on the Cut-Border Machine [GS98, Gum99] that allows for compression of generic, manifold and non-manifold polygonal meshes. More specifically, our contributions are:

- a general and efficient single rate connectivity compression scheme for polygon meshes of arbitrary topology,

- a compression scheme for different types and number of attributes attached to vertices, faces or corners, and
- compression rates that are on par or better than the state-of-the-art in current approaches for triangle or polygonal mesh compression even for meshes supported by other approaches.

2. Related Work

Starting with the first geometry compression approach by Deering [Dee95], the basic idea of *single rate* encoders is to start with an initial triangle and build the entire mesh by attaching triangles to the already encoded part of the mesh. Touma and Gotsman [TG98] predict the position of new vertices using a parallelogram rule rather than storing the absolute position. They also encode the free valence of new vertices instead of transmitting a special code for triangles that close a cycle. This approach has been extended to polygons by Alliez and Isenburg [IA02]. Context based predictive coding was added by Kälberer et al. [KPRW05]. Gumhold and Strasser [GS98] on the other hand explicitly store triangles in their cut-border machine and instead rely on arithmetic coding to efficiently represent triangle operations. In an extension Gumhold [Gum99] adapts the traversal order of the mesh and uses context sensitive models to further reduce storage costs. The edge-breaker algorithm [Ros99] is very similar to the cut-border machine but it instead encodes the boundary of the mesh up front. Google Draco is based on this approach. Extending Deering [Dee95], Bajaj *et al.* [BPZ99] introduce predictive coding of arbitrary attributes similar to the predictive coding of vertex positions. By restricting triangles meshes to be two-manifold [KADS02] or piecewise regular [SRK02], the coding efficiency can further be improved. Unfortunately, all algorithms to this point are capable of only encoding manifold, orientable triangle meshes. In order to encode non-manifold meshes or general polygonal meshes, the original mesh has to be split into separate triangle meshes. Guéziec *et al.* [GBTS99] compress general manifold polygonal meshes and use vertex clustering to avoid transmitting the same vertex multiple times. The TFAN algorithm [MZP09] separates any manifold or non-manifold mesh into a set of triangles fans and a set of triangle fan configurations. This approach has been implemented as part of MPEG4 and Open3DGC. Isenburg and Lindstrom [ILS05] propose a different approach to geometry compression by re-ordering of triangles and vertices. This re-oderend list is then compressed by using relative indices counting from the tail of the already streamed vertex data [IL05]. This approach is able to compress any mesh topology and can easily be extended to general polygonal meshes. The compression ratio on the other hand is not optimal as indices are still stored explicitly instead of using connectivity information. Jakob et al. [JBG17] implemented a parallel version of the cut-border machine with local predictions. However, due to the parallel implementation and the restrictions of the GPU based decompression approach, their compression approach only supports quantized attribute data and sub-optimal connectivity compression rates.

3. Preliminaries

Similar to Maglo *et al.* [MLDH15], we define a mesh as geometry, connectivity and attributes. The geometry of a mesh is the collection of polygons, i.e. which and how many vertices belong to each
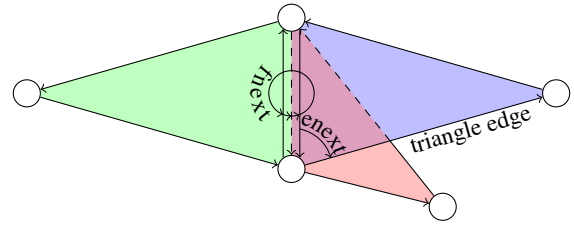

Figure 2: *Visualization of the Triangle Edge Data Structure.* fnext *points to the same edge on the next triangle (from green to red, then blue and back to green).* enext *iterates over all edges of one triangle.*

individual polygon or face. A face can also be interpreted as a list of edges. The connectivity of the mesh is defined by shared edges between polygons. The attributes are all information attached to vertices, edges and faces. According to Isenburg *et al.* [IGG01], the position of a vertex can simply be seen as one attribute.

3.1. Connectivity

Based on the topology of the input mesh, there exist certain constraints on the local connectivity. In a 2-manifold mesh, all faces attached to any given vertex form an open or closed polygon fan. In turn, if a mesh is non-manifold, there exists at least one vertex that is connected to multiple polygon fans.

Most mesh formats, e.g. PLY, OBJ and OFF, represent the connectivity by storing vertex indices for each corner of a polygon. This indirect approach of storing connectivity information requires a linear-time lookup for finding an adjacent polygon because the whole face list has to be traversed for it. Since connectivity compression algorithms make aggressive use of adjacency information, an efficient direct connectivity data structure has to be used to make the compression feasible.

In order to support non-manifold meshes, we extend the *Triangle Edge Data Structure* [Müc93]. In this data structure, the half edges form a tuple with their triangle, called *triangle edge*. Each *triangle edge* stores a reference (*fnext*) to the next and the previous triangle attached to the same edge. These references form two cycles around the half edge. There is also a pointer that references the next edge of the current triangle called *enext* (see Figure 2). It can easily be adapted to polygonal meshes by attaching each *fnext* pointer to the corresponding edges instead of the triangle/polygon. We call this modified data structure *Polygon Edge Data Structure*.

3.2. Attributes

Due to the structure of our algorithm, we make no distinction between the vertex position and further vertex attributes. An attribute can be seen as a vector of real or integral scalar numbers. The position p of a vertex, for example, can be represented as the vector (p_x, p_y, p_z). Multiple attributes that always occur in a union can be grouped together. E.g., vertex normals are always bound to the vertex positions and shared by all neighboring faces if there are no sharp edges in a mesh. Attributes with the same signature form a series, called *attribute list*. Attributes bound to faces are called
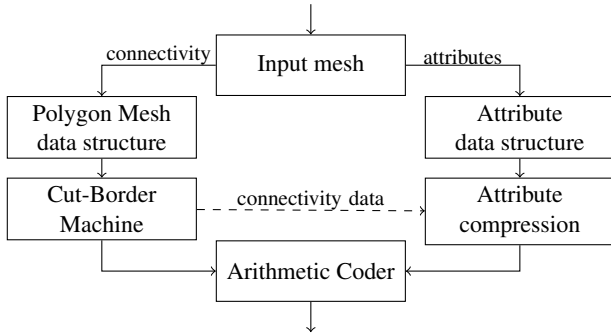
Figure 3: *Algorithm structure: Data flow of the connectivity and the attributes.*

Figure 4: *Terms of the Cut-Border Machine using the example of the new vertex operation.*

new vertex (∗) conn. forward (→) conn. backward (←)

border (_) $split_i$ (∞$_i$) $union_{p,i}$ (∪$_i^p$)

Figure 5: *Cut-border operations [Gum99]. The current cut-border is marked blue, the gate is marked as a bold blue arrow and the gate-adjacent triangle is shaded dark grey.*

face attributes and the ones bound to vertices are called *vertex attributes* respectively. Attributes that are bound to face corners are called *corner attributes*. Groups of corners around a specific vertex are called wedges if they share the same attribute data. Due to the indexed input structure, wedges can always be determined by grouping the corner attributes while walking around a vertex. Edges and half edges can also hold attributes but they will not be further discussed in this paper due to their rare use.

Usually, certain attributes are defined only for specific *regions*, e.g. texture coordinates or vertex colors are only available for part of a mesh. Regions are defined for faces and vertices separately, as a region of faces cannot be determined by analyzing the vertex regions and vice versa. Different regions can share connectivity which has to be taken into account for compression, as different regions cannot be handled as different meshes.

3.3. Compression

Our overall compression approach transforms both connectivity and attributes into a string of symbols. For maximum compression efficiency, all of these symbols are encoded using context adaptive arithmetic coding where each context consists of an adaptive model [Gum99]. Each model predicts the probability of a certain symbol based on an initial estimate and the number of symbols previously encoded through this model [MNW98].

4. Connectivity Compression

Fundamentally, our approach first splits the input mesh into connectivity and attribute data and encodes both parts separately (see Figure 3). The connectivity information is used to set up the Polygon Edge Data Structure that is used for querying adjacency information. This information is then used to encode the connectivity using our extended Cut-Border Machine. The attributes on the other hand are compressed using the connectivity data for both defining an ordering and prediction.

4.1. Cut-Border Machine

The original Cut-Border Machine of Gumhold *et al.* [GS98, Gum99] is used to compress triangle mesh connectivity. By travers-
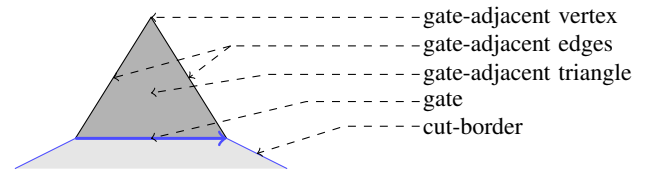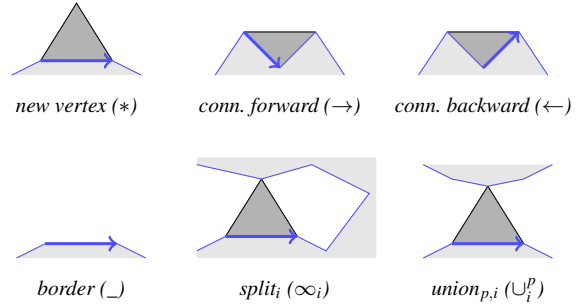
ing the mesh in a structured way and writing only information required for correct traversal during decoding, nearly optimal compression can be performed.

The main part of the Cut-Border Machine is the so called *cut-border*. The cut-border splits the mesh into an encoded and to-be-encoded part and is stored as a circular data structure containing half edges. For each iteration, the Cut-Border Machine stores a reference to the currently active cut-border element, the so called *gate*. The gate leads from the compressed to the uncompressed region and is the only part of the cut-border where new triangles are appended to the cut-border. In the following, we define multiple terms for better understanding of the algorithm (Figure 4). We call the triangle of the twin of the gate *gate-adjacent triangle* and the vertex of the gate-adjacent triangle not connected to the gate *gate-adjacent vertex*. The two edges of the gate-adjacent triangle connecting the gate with the gate-adjacent vertex are called *gate-adjacent edges*.

At the beginning, the Cut-Border Machine chooses a triangle from the set of triangles, initializes the cut-border to the three edges, and sets the gate to the first edge. Now, the Cut-Border Machine checks for the gate-adjacent vertex, which can be classified according to the following conditions (see Figure 5): If the gate-adjacent vertex is not part of the cut-border, the gate-adjacent edges will be added to the cut-border and a *new vertex* operation will be written. If the vertex is the next vertex on the cut-border after the gate, the gate will be directly connected to the edge after the next element and a *connect forward* operation will be encoded. Analogous, a *connect backward* operation will be encoded, when the gate-adjacent vertex is the previous one. If the vertex does not exist because the gate is on a mesh border a *border* operation will be written and the gate will be removed from the cut-border. If the vertex is on the current cut-border but neither the previous nor the
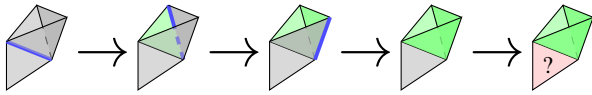
Figure 6: *Traversal of a non-manifold mesh using the Cut-Border Machine. The current gate is marked blue. The pink triangle can not be encoded with the original Cut-Border Machine.*

next, the cut-border must be split into two *parts*; the part in traversal direction from the gate to the adjacent vertex and the part in opposite direction. This will be encoded as a *split* operation, followed by the offset of the adjacent vertex on the cut-border. After splitting the cut-border a new condition can occur. When the adjacent vertex is part of another cut-border, they have to be concatenated at this vertex using the *union* operation, followed by the offset and the part index.

After checking the conditions, updating the cut-border and encoding the cut-border operations, the first new edge of the encoded triangle becomes the new gate; this corresponds to a depth-first search [Gum99]. When there is no edge left on the cut-border, the Cut-Border Machine tries to find a distinct edge connected component of the mesh and repeats the previous steps. If all edge connected components are encoded, the Cut-Border Machine stops.

An important feature the original Cut-Border Machine [GS98] lacks, is the capability to encode non-manifold edges and vertices. The improved Cut-Border Machine [Gum99] can only encode non-manifold vertices on mesh borders and non-orientable vertices which forces the approach to cut meshes into manifold edge-connected components while losing adjacency information at the cuts. In addition, neither of these are able to compress polygonal meshes. We thus propose a generalized compression scheme that works for arbitrary polygonal meshes. In Section 4.2, we will look into extending the Cut-Border Machine to encode general non-manifold triangle meshes and finally non-manifold polygonal meshes in Section 4.3.

4.2. Non-manifold meshes

When analyzing why the Cut-Border Machine cannot encode non-manifold vertices or edges, we observed that it assumes every gate-adjacent vertex to be located on the cut-border when it is flagged as previously encoded. This is exactly the condition for any *split* operation. On 2-manifold meshes, this is always the case because each edge is only handled twice during compression – once when it is added to the cut-border and again when it becomes the gate or some gate connects to that edge. During the second encounter, the edge is always removed from the cut-border. On non-manifold meshes, an edge can get connected or become the gate multiple times. Due to the removal of the edge this can no longer be encoded. However, the removal is required to ensure a correct traversal order. The problems caused by the traversal of non-manifold meshes can be seen in Figure 6 where the final triangle cannot be added as neither of the two encoded vertices can be indexed anymore.

Our approach solves this problem by handling the case where a

gate-adjacent vertex was flagged as already encoded but does not exist on the cut-border as a *new vertex* operation. During encoding the global vertex offset is transmitted instead of transmitting the whole vertex attribute as it is already present in the encoded stream. Using this technique, the mesh is split into multiple pieces implicitly. The individual parts are compressed separately but the decoder is able to concatenate the parts correctly using the global vertex offset. We call the modified *new vertex* operation *non-manifold new vertex* ($*_i$). Having to transmit a global vertex offset is acceptable, due to the few occurrences of non-manifold vertices or edges in most meshes. This solution does not need any costly preprocessing steps to recognize non-manifold vertices or edges by analyzing all triangle fans prior to encoding.

Similar to the improved Cut-Border Machine [Gum99], we further introduced three new *initial* operations to initialize a new edge-connected mesh component with the offsets of one, two or three already encoded (non-manifold) vertices (*non-manifold initial 1* $\triangle_i$, *non-manifold initial 2* $\triangle_{i,j}$ and *non-manifold initial 3* $\triangle_{i,j,k}$). In Figure 6 this is equivalent to re-initializing the gate to the already encoded edge of the pink triangle.

4.3. Polygonal Meshes

In the improved Cut-Border Machine [Gum99], Gumhold notes that the compression can be improved by traversing the mesh using a depth-first search. This keeps the start vertex of the gate fixed as long as possible and a traversal around the triangle fan of this vertex happens as long as *new vertex* operations occur. When the last triangle of the triangle fan is about to be encoded, a *connect forward* operation closes the triangle fan and finally changes the start vertex of the gate to traverse a new triangle fan. This traversal order is locally equal to the traversal orders used in current valence based systems.

Since polygons can always be split up into an open triangle fan, we can exploit the traversal order to continuously encode a triangulation of the polygon. Note that we do not actually use this triangulation outside of encoding or decoding the polygon and thus don't require special treatment of concave polygons. To recover the original polygon, we simply have to encode the number of additional vertices that belong to this polygon. This can be done by adding a symbol after the first triangle of every polygon. In pure triangle or quad meshes this will automatically be omitted as the model of the encoder only contains a single symbol.

To avoid redundant encoding of attributes, the face attribute is only attached to the first triangle of the polygon and only a single new corner attribute is attached to every following triangle, i.e. number of triangles plus two instead of three times the number of triangles.

4.4. Coding Example

When encoding the mesh seen in Figure 7, the initial triangle $f_{0,1,2}$ is encoded using an *initial* ($\triangle$) operation. Next, the half-edge $e_{2,0}$ becomes the initial gate and the algorithm traverses the triangle fan from there generating *new vertex* operations until triangle $f_{0,6,1}$ is reached. Encoding this triangle generates a *connect forward* operation which closes the triangle fan. The next two triangles ($f_{1,6,7}$
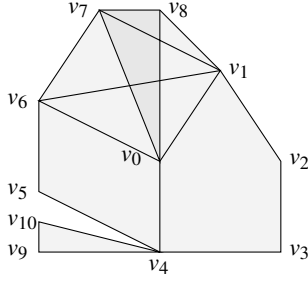
Figure 7: *Example mesh. It contains a pentagon ($f_{0,1,2,3,4}$) and a quad ($f_{0,4,5,6}$), which are split into triangles during compression and two kinds of non-manifold triangles ($f_{0,8,7}$), which overlaps the rest of the mesh and $f_{4,9,10}$, which is connected by one single vertex to the rest of the mesh).*

and $f_{1,7,8}$) cause additional *new vertex* operations. Next, the gate $e_{1,8}$ and the following cut-border elements are part of the mesh border and we thus have to store 7 *border* operations. The last triangle of this edge-connected component $f_{0,8,7}$ contains a non-manifold vertex v_0, because this vertex is already marked as encoded but not on the current cut-border. This triangle has to be encoded using a *non-manifold new vertex* operation. After that we have to store 2 more *border* operations to encode the entire boundary of the first edge connected component. The triangle $f_{4,9,10}$, that forms a new edge-connected component, contains a non-manifold vertex v_4 as well and thus causes a *non-manifold initial 1* operation. Finally, we have to encode another 3 *border* operations for the second component and then mark the stream as complete with a *end of mesh* ($\bigtriangledown$) operation. In addition to the operations, we also need to store how many triangle belong to a certain face (encoded after the operation that starts the new face) and any attributes that we would like to encode.

In summary, the bit stream will contain the following symbols for encoding the connectivity (see Figure 5 for additional symbol definition; faces are given for reference only):

$$\underbrace{\triangle 2**}_{f_{0,1,2,3,4}}\underbrace{*1*}_{f_{0,4,5,6}}\underbrace{\to 0}_{f_{0,6,1}}\underbrace{*0}_{f_{1,6,7}}\underbrace{*0}_{f_{1,7,8}}_____\underbrace{*_00}_{f_{0,8,7}}\_\_\underbrace{\triangle_40}_{f_{4,9,10}}\_\_\_\bigtriangledown$$

5. Attribute Compression

Every time we encounter a certain attribute for the first time, it needs to be encoded within the compressed stream. In order to achieve maximum efficiency, we encode all connectivity information first and use it for predicting attribute values.

5.1. Number representation

As mentioned in Section 3, an attribute is a vector that can either be stored as fixed or floating point numbers. Fixed point numbers result in a smaller entropy of delta values but also incur some loss on the original input data.

If some loss of accuracy is acceptable, floating point numbers can be transformed into fixed point numbers using quantization.

For most smaller meshes, 14 bits are often enough for vertex positions and 10 bits suffice for normals. Note, when using ASCII files as input, the attributes are usually stored as fixed point numbers which may allow for "quantization" of attributes without any loss of information.

5.2. Prediction & Delta Coding

To keep the number of bits required for encoding attributes low, we only encode differences to a predicted attribute [MLDH15]. For maximum coding efficiency, the differences should require as few bits as possible and should be similar to each other. The prediction algorithms need to be different for each attribute type and each attribute will use its own arithmetic model as the statistical distribution of possible values is unique for each attribute.

- Vertex attributes are predicted using multiple parallelogram prediction.
- Face attributes are encoded as the difference to one of their neighbors.
- Corner attributes are encoded as the difference to a former encoded attribute of that vertex.

The multiple parallelogram prediction is based on a simple parallelogram prediction [TG98, GGS99] for all neighboring triangles where all vertices have already been encoded. In our coding example, v_4 would be predicted as $v_0 + v_3 - v_2$. If there is more than one parallelogram available, e.g. v_6 in our example ($v_0 + v_5 - v_4$ and $v_0 + v_1 - v_2$), we average all predictions. For lossy compression, we use the average while for lossless compression, we pick the one closest to the average to avoid numerical instabilities. This effectively removes outliers in the prediction while still adapting to the overall average. Once all predictions have been calculated, we need to store the differences to the predictions in the bit stream.

As computing differences in floating point arithmetic suffers from loss of accuracy, a transformation to a signed integer representation that keeps the relative ordering of numbers is required. A floating point number is stored (from msb to lsb) as a sign bit, followed by the exponent and finally the mantissa. For positive floating point numbers this already creates the correct ordering. For negative floating point numbers, we simply have to invert the 31 lsbs to get the correct ordering as well. This was already described by Lindstrom and Isenburg [LI06]. Positive and negative infinity representations (maximum exponent) as well as de-norm values (minimum exponent) are also ordered correctly. Thus, this transformation keeps differences between very small positive and negative numbers close to zero as well.

Similar to Guthe and Goesele [GG16], we know the range of possible positive and negative numbers given the predicted value and can thus reorder the differences in a way that their absolute value increases (see Figure 8 for two examples).

5.3. Attribute bindings

As explained in Section 3, the mapping from faces and vertices to regions is surjective. Because the connectivity encoder decides which attribute is sent based on its traversal, redundancies can occur when one attribute is assigned multiple times. To avoid these
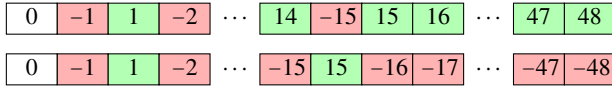
Figure 8: *Reordering of difference values for ranges* $[-15..48]$ *and* $[-48..15]$. *Possible differences are sorted by their absolute value.*

redundancies, our algorithm tracks all attribute IDs in a so called *history*. If an attribute is about to be encoded, a lookup into the history is performed. In case the attribute does not exist in the history, the raw attribute data is encoded and a new entry is added to the history. Otherwise, only the global attribute ID is encoded in the data stream. This history is only an optimization in cases where a small number of very different attributes are assigned very often and the prediction approach was not able to reduce the data sufficiently.

Because simple mesh formats such as PLY, OBJ or OFF do not support surjective indexed attributes and allow only a bijective mapping, most exporters duplicate the attribute values. This results in transmitting the same attribute multiple times because it was bound to multiple faces, vertices or corners. To restore the surjective mapping, a hash based approach is used to detect redundant values: All attributes are added to a hash map and discarded when the same value was previously inserted.

5.4. Regions

When calculating the differences to the predictions, the same attribute types are assumed for the prediction and the values to be predicted. By supporting regions, this is not always the case. As a solution to this problem, we encode the region ID prior to each attribute and omit the predictions step in the case of inconsistent regions by encoding absolute values. This is better than splitting the mesh at region borders as this would result in copying attributes at region borders instead of encoding them once. If only one region exists for faces or vertices, the encoding of the region ID will automatically be omitted by the arithmetic coder.

5.5. Header

Due to the variety of attributes supported by our format, a file header must be provided in order to decode the compressed data. The header is defined as follows:

- The number of faces N_f and vertices N_v to allow allocations of the attribute bindings.
- The number of regions N_r and their bindings to the attribute lists for faces B_f^i, vertices B_v^i and corners B_c^i (with $0 \leq i < N_r$). This needs to be defined per region as each region can have a variable number of attribute lists bound to it.
- For each attribute list the format F_i (with $0 \leq i < N_a$, N_a can be determined from the maximum of the bindings) must be transmitted (data types and quantization) and how the attributes are interpreted (positions, normals, etc.).
- In case of quantization, the format also includes the minimum and maximum values for each attribute list.
- A list of possible number of edges per face to initialize the corresponding arithmetic coder.

6. Results

To evaluate our compression algorithm, we compare it against several other approaches for both lossless and lossy compression on a variety of meshes.

6.1. Meshes

The Lucy mesh (Figure 9a) is a non-manifold triangle mesh consisting of $14,027,872$ vertices and $28,055,742$ triangles. An indexed face set would require $481.6MB$. The Bunny mesh (Figure 9b) is a manifold triangle mesh with 5 holes and consists of $35,947$ vertices and $69,451$ triangles. An indexed face set would require $1,235kB$. The Power Plant mesh (Figure 9c) is a non-manifold triangle mesh consisting of $11,070,509$ vertices with normals and $12,748,510$ triangles. An indexed face set would require $399MB$. The Beethoven mesh (Figure 9d) is a non-manifold polygon mesh consisting of $2,655$ vertices with normals and $2,814$ faces ($5,030$ triangles). An indexed face set (including a per face edge count) would require $115kB$. The Galleon mesh (Figure 9e) is a non-manifold polygon mesh consisting of $2,372$ vertices with normals and $2,385$ faces ($4,698$ triangles). An indexed face set (including a per face edge count) would require $102kB$. The Kobe mesh (Figure 9f) is a non-manifold polygon mesh consisting of $27,915$ vertices with normals and $13,066$ faces ($25,109$ triangles). An indexed face set (including a per face edge count) would require $905kB$.

6.2. Lossless Comparison

When comparing the compression ratio of our lossless compression scheme against Google Draco, OpenCTM and GZip compression of the original file (see Table 1), we can clearly see that we are always able to outperform the GZip compression of the original file. For both the Lucy and the Bunny mesh, our approach outperforms all other compression schemes while OpenCTM performs best on the very regular Power Plant model as well as on the Beethoven, Galleon and Triceratops meshes. The reason for this lies most likely in the dictionary based compression while both our approach and Google Draco use entropy coding. In terms of compression speed, we are always faster than GZip except for very small meshes and require on average about the same time as OpenCTM. In addition, our approach is the only one outside of GZip that is able to encode the polygonal meshes without converting them to triangle meshes. As a result, for the triangular meshes (except for the power plant) our approach is outperforming all other approaches in terms of compression ratio. For the polygonal meshes, our approach produces better compression rates than the only other applicable approach (GZip).

6.3. Lossy Comparison

When comparing lossy compression approaches, Google Draco always produces the smallest output with our approach in second place for the Lucy and Bunny models (see Table 2). The reason for the compression performance difference lies in the attribute prediction scheme. While our approach stores the attributes within the connectivity compression symbol stream, Google Draco stores all attributes after the connectivity has been encoded. This way, multiple existing triangles can be used to predict the position of the
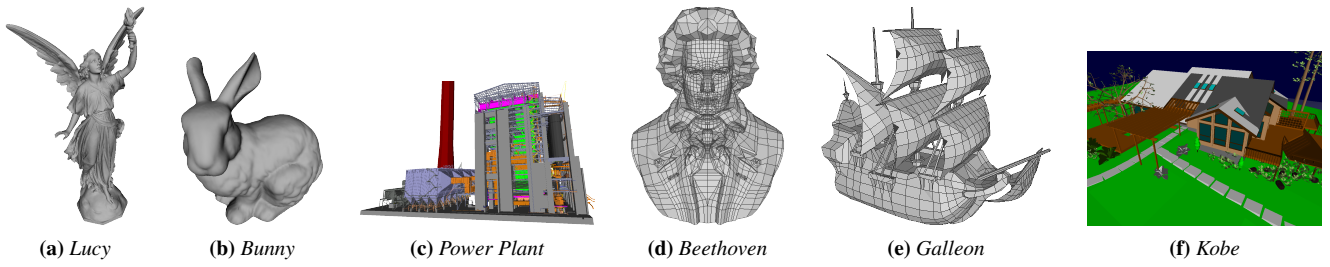
(a) *Lucy*   **(b)** *Bunny*   **(c)** *Power Plant*   **(d)** *Beethoven*   **(e)** *Galleon*   **(f)** *Kobe*

Figure 9: *Renderings of the datasets.*

Model	our		drc		ctm		gz	
	Size	Time	Size	Time	Size	Time	Size	Time
Lucy	**78.35MB**	69.18s	163.23MB	**27.96s**	123.76MB	93.12s	254.95MB	147.02s
Bunny	**231.48kB**	0.41s	413.71kB	**0.15s**	456.15kB	0.29s	809.31kB	0.52s
Power Plant	105.07MB	55.76s	224.55MB	**21.77s**	**27.07MB**	56.39s	114.37MB	97.43s
Beethoven	70.21kB	0.22s	63.17kB	**0.09s**	**47.31kB**	0.11s	81.07kB	0.16s
Galleon	64.34kB	0.25s	56.46kB	**0.09s**	**38.78kB**	0.1s	73.40kB	0.17s
Kobe	404.00kB	1.54s	697.52kB	**0.12s**	**264.73kB**	0.38s	510.94kB	0.19s

Table 1: *Lossless compression using our approach compared against Google Draco (drc), OpenCTM (ctm) and GZip (gz). Note that Google Draco and OpenCTM encode Beethoven, Galleon and Triceratops as triangle meshes and are therefore not able to recover the original polygonal mesh during decompression.*

encoded vertices leading to fewer bits to store the differences to the predicted values. In order to implement the same prediction stream, we would need to change our compression scheme to also encode all attributes after the entire connectivity has been encoded. Again, our approach is the only one that is able to encode the polygonal meshes without converting them to triangle meshes and therefore still the best choice in terms of compression ratio for these meshes.

6.4. Detailed Compression Analysis

Looking at the allocation of bits within the bit stream for both the connectivity and the attributes (Table 3), we can see that the connectivity of a triangle mesh requires 1 to 3 bit per vertex while there is an increase of about 2 bit per vertex for polygonal meshes. The attribute compression scheme achieves between 10 and 24 bit per vertex/corner during lossless compression of three 32 bit floating point values. When quantizing attributes to 14 bit (position) or 10 bit (other) per components, the compression scheme achieves between 3 and 11 bit per vertex/corner. This corresponds to a compression ratio of 3:1 to 10:1 on top of the quantization if enabled.

7. Conclusion & Future Work

In this paper, we have shown a general single rate connectivity compression scheme for polygon meshes of arbitrary topology that is capable of compressing different types and number of attributes attached to vertices, faces or corners. With and without quantization of attributes, we achieved compression rates that are on par or better than the state-of-the-art in current approaches for triangle or polygonal mesh compression. In the future, we would like to add support for edge attributes.

References

[BPZ99] BAJAJ C. L., PASCUCCI V., ZHUANG G.: Single-resolution compression of arbitrary triangular meshes with properties. In *Data Compression Conference, 1999. Proceedings. DCC'99* (1999), IEEE. 2

[Dee95] DEERING M.: Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), ACM. 2

[GBTS99] GUEZIEC A., BOSSEN F., TAUBIN G., SILVA C.: Efficient compression of non-manifold polygonal meshes. In *Visualization '99. Proceedings* (Oct 1999). 2

[GG16] GUTHE S., GOESELE M.: Gpu-based lossless volume data compression. In *3DTV-Conference: The True Vision-Capture, Transmission and Display of 3D Video (3DTV-CON), 2016* (2016). 5

[GGS99] GUMHOLD S., GUTHE S., STRASSER W.: Tetrahedral mesh compression with the cut-border machine. In *Proceedings of the conference on Visualization'99: celebrating ten years* (1999), IEEE Computer Society Press. 5

[GS98] GUMHOLD S., STRASSER W.: Real time compression of triangle mesh connectivity. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM. 1, 2, 3, 4

[Gum99] GUMHOLD S.: Improved cut-border machine for triangle mesh compression. In *Erlangen Workshop* (1999), vol. 99. 1, 2, 3, 4

[IA02] ISENBURG M., ALLIEZ P.: Compressing polygon mesh geometry

Model	our		drc		o3d		wgl	
	Size	Time	Size	Time	Size	Time	Size	Time
Lucy	17.75*MB*	59.83*s*	**17.60MB**	**34.4s**	19.83*MB*	174.54*s*	70.68*MB*	852.72*s*
Bunny	82.45*kB*	0.4*s*	**74.76kB**	**0.19s**	78.61*kB*	0.87*s*	255.94*kB*	0.75*s*
Power Plant	51.05*MB*	38.17*s*	**26.54MB**	24.5*s*	33.59*MB*	**0.12s**		
Beethoven	36.41*kB*	0.24*s*	**16.72kB**	**0.1s**	17.29*kB*	0.2*s*		
Galleon	32.59*kB*	0.24*s*	**14.36kB**	**0.09s**	15.14*kB*	0.19*s*		
Kobe	207.89*kB*	1.48*s*	**98.88kB**	**0.13s**	109.03*kB*	0.69*s*		

Table 2: *Comparison of our algorithm with Google Draco (drc), Open3DGC (o3d) and WebGL Loader (wgl) using a quantization setting of 14 bit positions and 10 bit normals (Power Plant, Beethoven, Galleon and Triceratops). WebGL Loader does not support normals. Note that Google Draco and Open3DGC encode Beethoven, Galleon and Triceratops as triangle meshes and are therefore not able to recover the original mesh during decompression.*

Model		Connect. bpv_c	Vertex attributes				Face attr. bpf_{col}	Corner attr. bpc_n
			bpv_{pos}	bpv_n	bpv_{col}	$bpv_{oth.}$		
Lucy	lossless	2.01	14.95					
	14/10bit		2.87					
Bunny	lossless	1.37	17.12			18.45		
	14/10bit		5.80			6.75		
Power Plant	lossless	2.80	16.77	6.80	2.03		0.18	
	14/10bit		6.08	3.86	2.03		0.18	
Beethoven	lossless	4.04	24.70					11.39
	14/10bit		10.85					6.18
Galleon	lossless	4.08	22.32					12.52
	14/10bit		10.11					6.42
Kobe	lossless	4.43	22.77					8.28
	14/10bit		8.89					5.39

Table 3: *Compression rates of all datasets with and without quantization.*

with parallelogram prediction. In *IEEE Visualization, 2002. VIS 2002.* (Nov 2002). 2

[IGG01] ISENBURG M., GUMHOLD S., GOTSMAN C.: Connectivity shapes. In *Proceedings of the conference on Visualization'01* (2001), IEEE Computer Society. 2

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *Visualization, 2005. VIS 05. IEEE* (2005), IEEE. 2

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Streaming compression of triangle meshes. In *ACM SIGGRAPH 2005 Sketches* (2005), ACM. 2

[JBG17] JAKOB J., BUCHENAU C., GUTHE M.: A Parallel Approach to Compression and Decompression of Triangle Meshes using the GPU. *Computer Graphics Forum* (2017). 2

[KADS02] KHODAKOVSKY A., ALLIEZ P., DESBRUN M., SCHRÖDER P.: Near-optimal connectivity encoding of 2-manifold polygon meshes. *Graphical Models 64*, 3-4 (2002). 2

[KPRW05] KÄLBERER F., POLTHIER K., REITEBUCH U., WARDET-ZKY M.: Freelence - coding with free valences. *Computer Graphics Forum 24*, 3 (2005). 2

[LI06] LINDSTROM P., ISENBURG M.: Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics 12*, 5 (2006). 5

[MLDH15] MAGLO A., LAVOUÉ G., DUPONT F., HUDELOT C.: 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys (CSUR) 47*, 3 (2015). 2, 5

[MNW98] MOFFAT A., NEAL R. M., WITTEN I. H.: Arithmetic cod-ing revisited. *ACM Transactions on Information Systems (TOIS) 16*, 3 (1998). 3

[Müc93] MÜCKE E. P.: *"Shapes and Implementations in Three-Dimensional Geometry"*. PhD thesis, "Department of Computer Science, University of Illinois at Urbana-Champaign", Ubana, Illinois, 1993. 2

[MZP09] MAMOU K., ZAHARIA T., PRÊTEUX F.: Tfan: A low complexity 3d mesh compression algorithm. *Computer Animation and Virtual Worlds 20*, 2-3 (2009). 2

[Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE transactions on visualization and computer graphics 5*, 1 (1999). 2

[SRK02] SZYMCZAK A., ROSSIGNAC J., KING D.: Piecewise regular meshes: Construction and compression. *Graphical Models 64*, 3-4 (2002). 2

[TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Proceedings of the Graphics Interface 1998 Conference, June 18-20, 1998, Vancouver, BC, Canada* (June 1998). 2, 5