Edgebreaker compression for arbitrary meshes

Žiga Leskovec, Matija Marolt, Žiga Lesar

University of Ljubljana, Faculty of Computer and Information Science
E-pošta: zl5596@student.uni-lj.si, matija.marolt@fri.uni-lj.si, ziga.lesar@fri.uni-lj.si

Abstract

Edgebreaker is an algorithm for compressing vertex connectivity of triangular meshes by traversing over the triangles and employing a specialized encoding scheme to represent their connectivity. Edgebreaker notably does not support non-manifold and non-orientable meshes. This work focuses on implementing and extending Edgebreaker to support a wider range of topologies. We provide an implementation capable of losslessly compressing OBJ files that contain meshes with holes or handles, non-orientable meshes and non-manifold meshes. The implementation is capable of compressing roughly a million triangles per second on commodity hardware.

1 Introduction

To represent a triangular meshes, we typically need a list of vertices and their connectivity. Typically, we store positions as 3-tuples of floating numbers and connectivity with 3-tuples of vertex indices which represent mesh faces (triangles). While GPUs need these data in an uncompressed form, such a representation is highly redundant and meshes are usually compressed for storage and transfer to decrease network traffic, speed up data transfers, and reduce storage requirements.

In this paper, we focus on vertex connectivity compression, more specifically on the Edgebreaker algorithm described by Rossignac [6]. Edgebreaker is a well-studied algorithm with numerous extensions and improvements made over the years. Due to its simplicity, speed, and efficiency, it has been included in one of the most popular mesh compression formats, Draco.[1] However, the original formulation has a few limitations on topology, which we address with an extension presented in this paper.

2 Related work

Triangular mesh compression has been extensively studied with numerous approaches focusing on connectivity encoding and attribute quantization. The seminal Edgebreaker algorithm by Rossignac [6] introduced a highly efficient connectivity compression scheme for orientable manifold meshes, achieving near-optimal compression rates through a clever traversal strategy that encodes the mesh topology using a sequence of five operations (see Section 4). Subsequent improvements include the Wrap & Zip algorithm by Rossignac and Szymczak [7] and the Cut-Border machine by Gumhold et al. [2], which further optimized the encoding process and extended the algorithm to meshes with arbitrary genus. However, these methods are fundamentally limited to orientable manifold surfaces, which limits their applicability to practical meshes. Alternative approaches such as the valence-driven compression by Alliez and Desbrun [1] and the spectral compression scheme by Karni and Gotsman [4] have addressed different aspects of mesh compression but similarly assume manifold topology. The extension of connectivity compression to non-manifold surfaces was partially addressed by Gurung et al. [3] using the SQuad data structure. As highlighted by Maglo et al. [5] in their extensive survey on mesh compression, even state-of-the-art techniques still struggle with complex topological structures such as non-orientable and non-manifold meshes.

Our work builds upon the foundational Edgebreaker framework while addressing its topological limitations, providing a unified approach that handles both non-orientable and non-manifold triangular meshes without sacrificing compression efficiency.

3 Half-edge data structure

Edgebreaker uses a half-edge data structure for mesh traversal. A half-edge is an edge with orientation. The half-edge that is currently being processed is called the gate. Two triangles are connected when their shared edge contains two half-edges with opposing orientation. Each half-edge has the following fields (see also Figure 1):

- `other`: the opposite half-edge. When `NULL`, this half edge lies on the mesh boundary.
- `t_next`: the next half-edge in a triangle – this value can be computed on the fly.
- `t_prev`: the previous half-edge in a triangle – this value can be computed on the fly.
- `b_next`: the next half-edge along the boundary.
- `b_prev`: the previous half-edge along the boundary.
- `v`: the vertex in a triangle not contained in the half-edge, opposite to the half-edge – this value can be computed on the fly.

We construct the half-edge data structure by adding tri-

[1] `https://google.github.io/draco/`

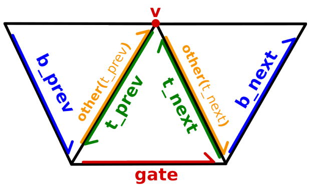

Figure 1: The half-edge data structure used in Edgebreaker.

angles one by one and keeping track of added half-edges (see Algorithm 1).

```
input  : Mesh triangles
output : Half-edges (structure of parallel arrays of
         $b\_next$, $b\_prev$ and $other$)
1  initialize set of added half edges $S_{he}$; initialize
   parallel arrays for $b\_next$, $b\_prev$ and $other$;
2  for each triangle do
3      add all edges as half-edges; connect them to
       boundary loop using $b\_next$ and $b\_prev$;
4      for each triangle half-edge $h$ do
5          if opposite half-edge $g$ exists in $S_{he}$ then
6              connect boundary loops using $b\_next$
               and $b\_prev$; link half-edges with
               $other$;
7          else
8              add $h$ to $S_{he}$;
9          end
10     end
11 end
```

Algorithm 1: Construction of half-edge data structure.

4 Manifold meshes

Edgebreaker maintains a stack of gates, which divide the mesh into distinct parts. Each part is encoded as a sequence of operations that can be used to uniquely reconstruct the mesh. There are 5 basic operations in Edgebreaker: C, L, R, S and E, illustrated in Figure 2. These operations describe the relation between a triangle and the remaining boundary of the mesh as follows:

- **C**: The triangle is not contained in the boundary (with the exception of the gate edge). Remove it and continue with the rightmost edge.
- **L**: The triangle has a boundary on its **left** side. Remove the triangle and continue traversal with the right edge.
- **R**: The triangle has a boundary on its **right** side. Remove the triangle and continue traversal with the left edge.
- **S**: Removing a triangle **splits** the mesh into two parts. Remove the triangle, add the left edge to the stack and continue traversal with the right edge.
- **E**: This is the only remaining triangle in this part of the mesh. Remove the triangle and continue with the edge on the top of the stack. The algorithm ends when the stack is empty.

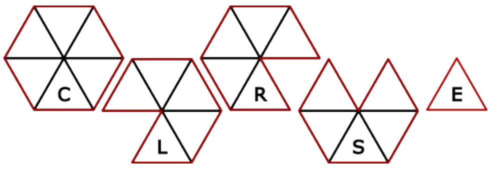

Figure 2: Basic Edgebreaker operations. Red lines represent the boundary of the remaining mesh.

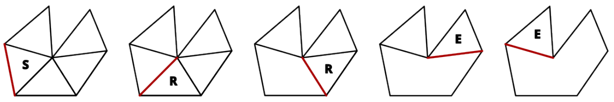

Figure 3: Compression process for a simple planar mesh. Red lines represent the edge on top of the stack.

Algorithm 2 describes the compression steps. We illustrate the compression logic for a simple planar mesh in figure 3. The output of the compression process is a sequence of operations and the permutation of vertices, which uniquely defines the vertex indices produced by the decompression process.

```
input  : Half-edges (from algorithm 1)
output : EdgeBreaker structure (history, H/M
         tables and list of duplicated vertices
1  initialize arrays of $history$, $M\_table$, $H\_table$
   and $duplicated$ vertices;
2  mark boundary edges by iterating the structure
   using $b\_next$;
3  $gate \leftarrow$ arbitrary boundary edge;
4  $push(stack, gate)$;
5  while $edge \leftarrow pop(stack)$ do
6      switch $edge$ $state$ do
7          handle cases $C$, $L$, $R$, $S$ and $E$;
8      end
9  end
```

Algorithm 2: The Edgebreaker compression algorithm.

The decompression process is more involved. It starts with a preprocessing phase in which we iterate through the operation sequence and calculate the boundary length and where the S operations split the boundary (see Figure 4). Then, we iterate through the operation sequence again and connect triangles to the boundary based on the operation semantics. Figure 4 shows the decompression process for the same planar mesh used for the compression example.

5 Meshes with holes

When the mesh contains a hole, we essentially have two external boundaries, which produces a split operation that does not actually split the mesh into two components, but rather cuts the mesh by connecting the two boundaries (see example in Figure 5). To address this situation, Rossignac proposes a new operation, which we call H.

During compression, we mark the first encountered boundary as external, and all the others as hole boundaries. When we encounter a split containing a hole edge, we

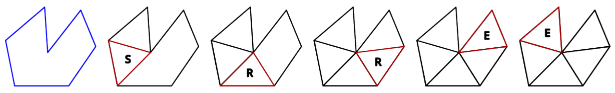

Figure 4: Decompression process for a simple planar mesh. Blue lines represent the boundary added in the preprocessing phase, red lines represent the decompressed triangles.

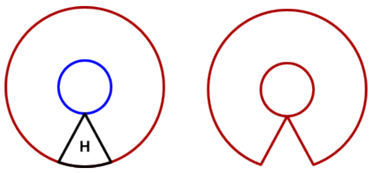

Figure 5: H operation during compression. The blue boundary is marked as a hole, while the red boundary is marked as an external edge.

mark the operation as H and calculate the length of the hole. We also mark the encountered hole as external, as shown in Figure 5. We then store the hole length in a so called H-table, which we use when decompressing to determine the initial boundary.

6 Meshes with handles

Similarly to meshes with holes, meshes with handles also break the S operation, because they produce loops in mesh topology, and cause splits to merge prematurely (see figure 6). An incomplete solution to this problem is already proposed by Rossignac [6], where for compression and preprocessing phase of decompression there exists only a rough description of how to tackle the problem. Here we provide a complete description of compression and decompression technique that was implemented to support meshes with handles.

During compression, when we encounter a split, we mark the left loop with a reference to the gate that the loop belongs to (the gate we push to stack for later evaluation). When we later encounter a vertex marked with the left loop, we mark the edges as external and merge the two loops into one and add operation M (as in **m**erge) to the operation sequence. We also compute sub-loop length, the stack position of the S operation that caused the loop, and the offset where the merge happened. We store these three values in the so called M-table.

During decompression in preprocessing phase we use the M-table's S stack position and offset values to compute correct S triangle indices. Algorithm 3 describes this process (notation taken from [6]). The code for triangle decompression in the original paper is correct and therefore omitted.

7 Non-orientable and non-manifold meshes

When constructing a half-edge data structure for a non-orientable mesh (e.g., a Möbius strip), we encounter neighbouring triangles with incompatible orientations. This causes the shared edge to be split into two half-edges with the same orientation, instead of the opposite, and prevents us from linking the aforementioned half-edges with field

```
1 case $M$ do
2     $(position, \_, length) \leftarrow next(mTable)$;
3     $e \leftarrow e - 1$;
4     $d \leftarrow d - 1$;
5     $(e', s') \leftarrow remove(stack, position)$;
6     $offsets[s'] \leftarrow -e' - length$;
7 end
```

Algorithm 3: Preprocessing of the M operation.

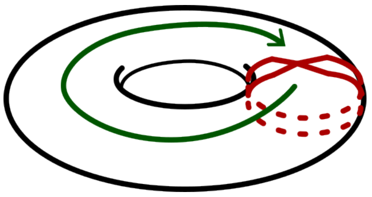

Figure 6: In meshes homeomorphic to a torus, S operation splits boundary into two loops, which are later merged, as mesh traversal takes the green path.

`other` and connecting their boundary loops with b_next and b_prev.

We handle this case by keeping `other` set to `NULL` and marking the edge as collided. Later, when we search for the initial boundary and holes and encounter a collided edge, we duplicate its vertices, effectively cutting the mesh, and continue with algorithm normally. The positions and indices of duplicated vertices are stored in a separate table, allowing us to reverse the cutting at the end of the decompression algorithm.

When dealing with non-manifold meshes, we encounter a similar problem, but with multiple collisions on the same edge. Instead of just marking the edge as collided, we count the number of collisions and duplicate vertices for every collision.

When duplicating vertices, the mesh can be split into multiple parts. To account for this in the compression algorithm, we check for any unprocessed half-edges and continue execution. When decompressing, we have to be careful in the preprocessing phase, since now we have multiple boundaries. They can be detected by matching each S with an E operation and starting a new boundary when the number of E operations exceeds the number of S operations while traversing the operation history.

8 Extended OBJ format

To store compressed meshes in a file, the OBJ file specification was extended to support 3 additional fields:

- `ebh [base64] [padding]`: the base64-encoded operation sequence.
- `ebt s/l s/p/o/l ...`: H and M tables with values `s` appended, which encodes which S operations are actually H or M operations.
- `ebd pos/idx ...`: the list of duplicated vertices.

Name	Specifics	Triangles	Size	C. time	D. time	EB	LZMA	Deflate	EB+LZMA	EB+Deflate
bunny-min.obj	contains holes	4968	167.4 kB	3 ms	< 1 ms	1.8	4.5	3.2	6.1	4.8
bunny.obj	triangle count	144046	4.5 MB	121 ms	7 ms	2.6	8.0	4.7	12.4	9.5
igea.obj	triangle count	268686	9.5 MB	223 ms	12 ms	2.3	8.5	5.0	12.6	10.0
capsule.obj	no boundaries	540	12.2 kB	< 1 ms	< 1 ms	2.2	5.3	4.0	10.7	10.9
klein-bottle.obj	non-orientable	5760	175.1 kB	3 ms	< 1 ms	2.0	4.9	3.3	6.7	5.3
suzanne.obj	multiple components	967	27.2 kB	< 1 ms	< 1 ms	1.9	5.8	4.3	9.7	9.6

Table 1: Compression (C) and decompression (D) efficiency and speed of Edgebreaker (EB) combined with LZMA and Deflate on test models.

9 Results

We implemented the described algorithm (single-threaded) in Rust. The source code has been made publicly available on Github.[2] Both compression and decompression processes are implemented with support for handles, holes, non-orientable edges and non-manifold edges.

We collected a set of test 3D models in OBJ format to be used for testing (see Table 1). The models contained topological features that the original Edgebreaker is not able to process. Some of the models contained a large amount of triangles to test how our implementation handles such cases. Figure 7 shows a subset of the test 3D models side by side, indicating no loss of data after decompression.

We measured compression and decompression times and compression ratios for each test model, using a commodity laptop with an AMD Ryzen 5 4500U CPU. The measurements are collected in Table 1. In addition to the compression ratio achieved by using only Edgebreaker and the storage format described in Section 8, we tested how stream compression algorithms compare to and combine with Edgebreaker. To do so, we used a POSIX port of 7zip[3] with LZMA (flag `-mm=LZMA`) and Deflate (flag `-mm=deflate`). Table 1 lists compression ratios achieved with five different combinations of Edgebreaker, LZMA and Deflate, showing that Edgebreaker effectively complements stream compression algorithms to achieve even better compression.

10 Conclusion

This work successfully extends the Edgebreaker algorithm to handle arbitrary triangular meshes, including those with holes, handles, non-orientable surfaces, and non-manifold structures, thereby addressing the significant topological limitations of the original formulation. Through the introduction of H and M operations, we maintain linear time complexity while achieving compression ratios between 1.8 and 2.6 across test meshes, and reaching compression ratio of 12.6 when combined with LZMA algorithm. Our implementation demonstrates practical viability with processing speeds of approximately one million triangles per second on commodity hardware, making it suitable for real-world applications. While the current implementation focuses exclusively on vertex connectivity compression and ignores other vertex attributes such as normals and texture coordinates, the foundation established here provides a robust platform for future extensions to complete mesh compression.

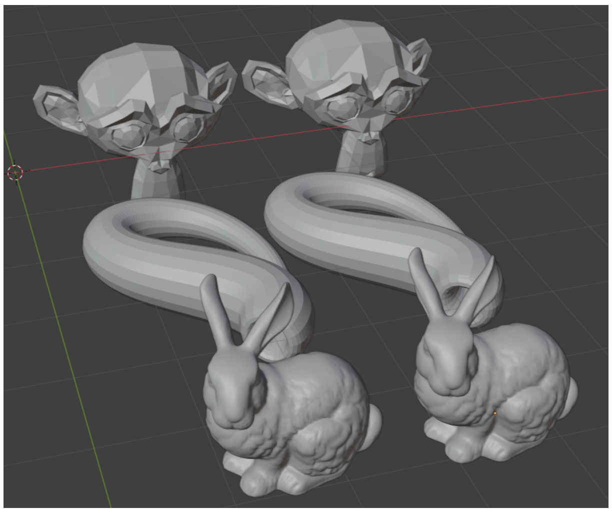
Figure 7: Screenshot of original and decompressed files loaded in Blender. From top to bottom: suzanne.obj, klein-bottle.obj, bunny.obj.

References

[1] P. Alliez and M. Desbrun. Valence-driven connectivity encoding for 3d meshes. *Computer Graphics Forum*, 20:480–489, 2001.

[2] S. Gumhold, S. Guthe, and W. Strasser. Tetrahedral mesh compression with the cut-border machine. In *Proceedings Visualization '99*, pages 51–509. IEEE, 1999.

[3] T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac. Squad: Compact representation for triangle meshes. *Computer Graphics Forum*, 30:355–364, 2011.

[4] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, pages 279–286. ACM Press, 2000.

[5] A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot. 3d mesh compression. *ACM Computing Surveys*, 47:1–41, 2015.

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

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

[2]`https://github.com/siggsy/edge-breaker`
[3]`https://p7zip.sourceforge.net/`