Lossless Triangle Mesh Compression

by

Junjie Zhu

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

July 2013

Copyright © Junjie Zhu, 2013

Abstract

Triangle meshes have been widely used in many fields such as digital archives, computer-assisted design and in the game industry. The topic we are particularly interested is triangle mesh compression for fossil models. This thesis explores the research area of triangle mesh compression and implements a good compressor for our data set. The compressor is mainly based on a previously proposed algorithm, Edgebreaker, with modifications and improvements. The details of the implementation and our improvements are described in detail in this thesis.

Acknowledgments

I would like to take this opportunity to thank my supervisor, David Rappaport, for his help over the course of this thesis and his understanding about my work-life balance.

I would also like to thank our research sponsor, FedDev, and our research partner, Research Casting International. It is a wonderful experiment to combine my thesis with real-world challenging problems.

I would also thank my friends, Hessian Hasic and Jia Xin Qiu, who spent their time on proofreading my draft document and giving me useful editing advices.

Table of Contents

Abstract **i**

Acknowledgments **ii**

Table of Contents **iii**

List of Tables **v**

List of Figures **vi**

Chapter 1:
Introduction . . . 1
1.1 Motivation . . . 1
1.2 Problem . . . 2
1.3 Contributions . . . 3
1.4 Organization of Thesis . . . 4

Chapter 2:
Background . . . 5
2.1 Lossless Connectivity Compression . . . 5
2.2 Lossless Geometry Compression . . . 8
2.3 Out-of-Core Solution . . . 9

Chapter 3:
Implementation and Results . . . 11
3.1 Choices of Edgebreaker algorithm . . . 12
3.2 Mesh Input Format . . . 13
3.3 Half-edge Data Structure . . . 15
3.4 Basic Edgebreaker for simple meshes . . . 17
3.5 Supporting Holes . . . 27
3.6 Supporting Handles . . . 29

3.7 Supporting non-manifold vertices 37
3.8 Supporting multiple surfaces and isolated vertices 40
3.9 Encoding Schema 43
3.10 Correctness Measurement and Testing 47
3.11 Experimental Results 51

Chapter 4:
Summary and Conclusions 57
4.1 Summary 57
4.2 Future Work 57
4.3 Conclusion 58

Bibliography 60

List of Tables

3.1 Five Basic Edgebreaker operators 18
3.2 Decompression for the five basic Edgebreaker operators 22
3.3 left:right ratio of the split(S) operations 33
3.4 ratio of three-edge right loop 33
3.5 non-manifold vertices conversion 39
3.6 Original fixed schema 46
3.7 New fixed schema 47
3.8 vertex and triangle sequence comparison 48
3.9 vertex and triangle sequence comparison (sorted) 50
3.10 Test Dataset's Property Matrix 51
3.11 Uncolored Mesh's Compression Ratio 52
3.12 RGB-colored Mesh's Compression Ratio 55
3.13 RGBA-colored Mesh's Compression Ratio 55

List of Figures

3.1 Left: Mesh native representation and PLY format 14
3.2 Half-edge data structure and its C implementation 15
3.3 Edgebreaker extended half-edge data structure 16
3.4 Edgebreaker compressor 20
3.5 Left:before split, Right:after right traversal 26
3.6 Add hole vertices into vertex sequence 28
3.7 Edgebreaker Operator Sequence with Handle operator N 35
3.8 A triangle fan 38
3.9 Compression with non-manifold pre-processing 40
3.10 fossil mesh with floating small surface 41
3.11 multi-surface compression 42
3.12 multi-surface decompression 44
3.13 Edgebreaker correctness testing flow chart 49
3.14 Uncolored Mesh's Compression Ratio 53
3.15 RGB-colored Mesh's Compression Ratio 54
3.16 RGBA-colored Mesh's Compression Ratio 55

Chapter 1

Introduction

1.1 Motivation

Digitizing artworks and historical records can enable institutions and researchers to simultaneously access the precious objects without any possible physical damage. While digitizing three-dimensional models, the objects' surfaces are usually represented by geometrical approximations, which consist of surface points and surface polygons. The surface points are accurately measured by 3D scanners. The surface polygons are generated by meshing algorithms according to the measured points' locations. Usually, triangles are chosen as surface polygons as they are the simplest forms of polygons. A surface mesh with only triangles is called a triangle mesh.

Triangle meshes have been widely used in many fields such as digital archives, computer-assisted design and in the game industry. A trivial way to store a triangle mesh is to store the vertices and the triangles by their geometrical properties directly: Each vertex can be stored as its three-dimensional coordinates; Each triangle can be stored as an index list of its 1st, 2nd and 3rd vertices. Any additional context-based

property (such as colors, texture,...) will be appended to the corresponding vertex or triangle. This format has been widely implemented in most three-dimensional object formats, such as VRML and PLY. Usually, the above formats are referred to as uncompressed formats.

The uncompressed format is simple and easy for read and write as there is no extra interpretation step needed. A single-pass linear scan is sufficient to retrieve all the geometrical information. However, as measurement tools and applications have been greatly improved, scanning tools can return accurate results to the submillimeter level. Therefore, the problem of storing large triangle meshes has arise. For example, in the digital Michelangelo project[LPC+00], digitizing 10 statues by Michelangelo occupied 250-gigabytes.

1.2 Problem

While compressing the triangle meshes from the trivial formats, there are mainly 2 areas we can compress: the vertices and the triangles. For the triangle compression, we can easily observe the data redundancy from the multiple appearances of the same vertex in different triangles. Therefore, a compression can be achieved by removing this redundancy. There have been a few algorithms and improvements proposed in triangle compression. For example, Topological Surgery [TR98] and Edgebreaker[Ros99] use mesh traversals to implicitly represent the vertex indices, in which the index redundancy is removed. While different algorithms can all successfully compress and remove the redundancy, their performances vary. This motivates us to take a close look at geometrical properties of those algorithms.

My research problem is to find a non-trivial and easy-to-implement representation

of a triangle mesh which will lower the compression rate compared to general-purpose compression methods. In the following pages, we refer to this problem as lossless triangle mesh compression.

1.2.1 Hypothesis

The test models from our research partner, Research Casting International, are mainly fossil models. In their lab, the data of fossil models are first captured by 3D scanners and later processed by meshing softwares. The common characteristics of these fossil models are:

1. The models are in highly-irregular shapes.
2. The models have missing pieces (holes) in the surfaces.
3. The models have multiple surfaces.
4. The models have background noise generated in the scanning process.

Our research hypothesis is: an efficient lossless mesh compression method for models with the above characteristics can be derived from the Edgebreaker[Ros99] compression algorithm for simple meshes.

1.3 Contributions

The major contribution from this thesis is to improve the original Edgebreaker[Ros99] algorithm. By using a new handle detector, we improved the algorithm's running time.

The second contribution from this thesis is experimental results of a new fixed schema, which is generally more efficient than the original fixed schema Rossignac proposed [Ros99].

1.4 Organization of Thesis

We proceed by a short introduction in Chapter 1. Chapter 2 is a brief review of related literature. Chapter 3 is a detailed description of the algorithm and the implementation. Chapter 4 is summary and suggestions for future research.

Chapter 2

Background

2.1 Lossless Connectivity Compression

The lossless connectivity compression is mostly based on the idea of mesh traversals. The first method was introduced by Deering[Dee95], the algorithm decomposes a triangle mesh into a set of long triangle strips. The algorithm visits those triangle strips separately. A queue stores the last 16 visited vertices. When the next vertex is visited, it's compared to the 16 vertices in the queue, if the vertex is found in the queue, it will not be stored as a full entity with all properties, it will be only stored as a 4-bit reference representing its position in the queue. This could be a sufficient storage saving if the hit rate on the queue is high. However, Deering[Dee95] did not propose a general method to decompose a mesh to achieve the high hit rate.

Taubin[TR98] further developed the idea of mesh traversal by using spanning trees as a general method to decompose a mesh. The algorithm was named Topological Surgery and it runs as follow: First of all, a vertex spanning tree is built. A vertex spanning tree is a standard spanning tree in graph theory with mesh vertices as graph

vertices. And then all the edges in the vertex spanning tree are called cutting edges and used to decompose the meshes. Triangle spanning trees are later built using the above cutting edges. A triangle spanning tree is a standard spanning tree in graph theory with mesh triangles as graph vertices. A run-length encoder is then used to compress the vertex spanning tree and the triangle spanning tree separately. The efficiency of the Topological Surgery algorithm depends on the mesh types and the algorithms to build the vertex spanning trees. Experimentally, the Topological Surgery algorithm can achieve a compression rate of 2.48-7.0 bpv, where bpv stands for bits per vertex which is calculated by the compressed file size divided by the number of vertices. The Topological Surgery method later became a part of MPEG-4 3DMC standard.

Rossignac[Ros99] later introduced another mesh traversal algorithm called Edgebreaker. The details of the Edgebreaker algorithm will be discussed in Chapter 3, but we will have a brief introduction here. The biggest advantage of Edgebreaker is that it has a known theoretical storage cost upper bound of 4 bpv; while Topological Surgery only has the experimental results 2.48 - 7.0 bpv. Also Topological Surgery uses both vertex and triangle spanning trees, while Edgebreaker only uses the triangle spanning tree . Edgebreaker works as follows: The algorithm has a set of pre-defined operation types (CELRS). The mesh traversal starts on a boundary triangle, every operation will travel to an unvisited triangle. Every time a new vertex is visited, the vertex is marked and put into a vertex stack. The final compressed output is a triangle spanning tree marked with a operation sequence and a vetex stack with its visiting order. A Huffman encoder then compresses the triangle spanning tree. The

decompression consists of two passes. The first pass calculates the boundary condition and the offset for each split operator. The second pass rebuilds the same mesh traversal and the triangles are reconstructed on the fly. Edgebreaker is a simple and elegant algorithm, which has good compression rate and is also easy to implement. Also, the known theoretical storage cost upper bound makes it a good candidate for compressing irregular meshes.

There are already some publications regarding improvements to Edgebreaker. Wrap and Zip[RS99] is the first improvement for Edgebreaker decompression. The original Edgebreaker decompression needs two passes. The purpose of the first pass is to calculate the boundary condition and the jump length for each split operation in order to correctly label the vertex index. The Wrap and Zip decompression algorithm delays the labelling process so that the first pass can be completely removed. Spirale Reversi is another improvement for Edgebreaker decompression. Instead of re-building the mesh traversal in the same order, Spirale Reversi[IS00] uses a reversed mesh traversal. Same as Wrap and Zip, Spirale Reversi also removes the need of the first pass for labelling the vertex index and it only requires one pass. Lopes[LRS+02] proposed a modified version of Edgebreaker in order to better deal with handles. That modified algorithm uses two proven properties in handle decomposition and a preprocessing step to mark a pair of separator edges for each handle. This modification makes it easier to implement the handle solution. Jong[JLYT04] proposed another modified version for Edgebreaker. Jong's method focused on simplification of the compression process. He accomplished that by completely removing storage of gate stacks for SE operations. Szymczak[SKR01] proposed another modified version for Edgebreaker. His method added a length-predictor for C operators. This can

further improve the compression rate.

2.2 Lossless Geometry Compression

Lossless geometry compression is mostly based on vertex prediction and numerical encoders. Different combination of vertex predictors and numerical encoders can produce different compression rates.

The purpose of vertex predictors is to increase the entropy of the data set in order to shift the data set to have higher likelihood. There are a few vertex predictors which have been proposed. The first proposed predictor is the delta predictor[Cho97], which is based on the first derivative. Similar to the delta predictor in single processing, the delta predictor in mesh compression stores the difference of the current vertex and the last vertex. The delta predictor works as long as the adjacent vertices in the sequence mostly have a similar distance. On the other hand, as the vertex list in the compressed format is generated by the connectivity encoder, the efficiency of the delta predictor highly depends on the choice of the connectivity encoder. The second proposed predictor is the linear predictor[TR98], which is proposed by Taubin along with Topological Surgery connectivity compression. The linear predictor uses a list, up to a certain number of recent visited vertices, to calculate the difference of the current vertex and the linear combination of previous vertices. The linear predictor works as long as the corresponding connectivity encoder uses a region growing method of mesh traversal. The third proposed predictor is the parallelogram predictor proposed by Touma[CC98]. The predictor stores the difference between a parallelogram opposite and the current vertex. The predictor works as long as the connectivity encoder uses a triangle traversal as mesh traversal method. It can be proven that the scanner

and meshing algorithms tend to return similar triangles, where adjacent triangles have a similar form as a parallelogram. This property makes parallelogram predictor very suitable for mesh compression. An entropy encoder is usually used on top of the vertex predictor. Commonly used encoders include Huffman encoder[Huf52] and context-based arithmetic encoder[WNC87]. For Huffman encoding, a preprocessing runs to build the codebook and the corresponding number distribution. Then a Huffman tree is built in order to determine the bit representation for each number. This method is easy to implement but the codebook of all the occurred numbers could be very large and overwhelm the encoding benefit. For context-based arithmetic encoding, all the number representations are embedded into a probability sequence and a number between 0 and 1. This method is practical as the codebook is much smaller than Huffman encoding.

2.3 Out-of-Core Solution

An out-of-core solution is a design to process data that is too large to fit in the computer memory. It typically sorts and divides data into blocks in hard disk and load blocks into computer memory when needed. An out-of-core solution is needed for large meshes because connectivity compression usually requires a global knowledge in advance to determine the neighboring triangles. The memory requirement of this global knowledge could be impractical for large meshes. Therefore, an out-of-core solution is needed in order to efficiently divide the large meshes into smaller pieces for compression and efficiently merge them into the original large meshes for decompression. Isenburg[IG98] proposed an out-of-core algorithm that can be transparent to

both connectivity encoders and geometry encoders. The method starts with clustering vertices according to their coordinates and bounding boxes, puts one copy of each inner triangle into clusters and multiple copies of each boundary triangle into clusters, and then sorts the triangles in order to enable random access. While Isenburg's algorithm greatly reduces the memory requirement, it also significantly increases the processing time.

Chapter 3

Implementation and Results

The major contribution of this implementation is to improve the original Edgebreaker algorithm [Ros99] designed for simple meshes to work better on our dinosaur fossil dataset. Besides adding support for non-manifold vertices and multiple surfaces, our implementation improves the original Edgebreaker algorithm as follows:

1. Our new handle detector is shown experimentally to be more efficient than the theoretical bounds described in the original Edgebreaker method [Ros99]. The implementation's running time appears to be closer to linear time performance rather than the $O(n^2)$ theoretical upper bound.

2. We propose a new fixed encoding schema based on our observation. It reduces the storage cost from 2 bits/triangle to 1.5∼1.7 bits/triangle. Our observation suggests this new schema will be generally more efficient and a better choice than the original schema.

3.1 Choices of Edgebreaker algorithm

There are two major variants of the Edgebreaker algorithm:

1. The first version is the original Edgebreaker [Ros99] which uses the half-edge data structure and traverses the mesh based on dynamic borders. Each triangle is visited once and only once. For simple meshes with no holes or handles, each triangle only takes a constant number of operations. Therefore, the time complexity on simple meshes is O(n).

   This method has been extended to support holes and handles. Its support for holes is fast and has a good compression ratio (64 bits/hole). However, its support for handles increases the time complexity from O(n) to O(n^2).

2. The second version [LRS+02] was published a few years later, and it uses the corner-table data structure and traverses the mesh based on visiting order of adjacent triangles. It improves the original method on handle processing. Now, instead of O(n^2), it only requires O(n) to process meshes with handles. The only drawback of the second method is that its hole processing has a poor compression ratio.

   Its method to support holes is to simply patch the hole by adding a dummy vertex per hole and linking it to each vertex on the hole. As all these added triangles will be embedded into the compression sequence, this will increase storage cost by $(2h + 32)$ bits per hole, where h is the size of the hole and 32 bits are used for storing each dummy vertex index. If the hole has more than 16 vertices, the new method's compression ratio will be poorer than the original Edgebreaker [Ros99]. For our fossil dataset, many samples have holes larger

than 16 vertices. Therefore, this method is not preferred.

A revised version [LLRL04] was later proposed to avoid using dummy vertices and adding triangles. Instead, it just adds one pseudo face with all the vertices on the hole. This revised version can achieve the same compression ratio as the original Edgebreaker.

In this implementation, we started with the original Edgebreaker algorithm. And then, we extended to support holes, improved its running time by introducing a new handle detector. Furthermore, by observing the patterns discovered from our experiments, we also proposed a new simple fixed encoding schema, which is easy to program and can achieve a better compression ratio compared to the old schema. Our implementation is competitive and can compress and decompress a models with millions of vertices within a few seconds.

3.2 Mesh Input Format

A triangle mesh consists of a point cloud and a triangle set. The trivial method to represent this data is to represent a vertex by its three-dimensional coordinates and represent a triangle by references of its three vertices. The surface orientation of the triangle is stored by imposing the right hand rule on the order of the three vertex references. Besides the above essential attributes of a triangle mesh, a mesh can also contain other non-essential attributes such as color and texture. In our fossil dataset, the RGB color attribute is derived from scanning and is associated with vertices. They are stored as vertex color in our dataset.

PLY is a popular mesh format. Our fossil dataset uses PLY as a storage format.

PLY not only supports triangle meshes, but also supports meshes with arbitrary polygonal structures. PLY uses the same vertex/polygon representation rule as previously described.

A PLY file starts with a mesh attribute list and a mesh statistics descriptor, follows with vertex data, and then polygon data. The vertex data is typically represented by three 32-bit floating-point numbers in the form of "x y z". The polygon data is represented by an 8-bit char which is the number of the edge in a polygon and a list of 32-bit integers as vertex references ordered using the right hand rule: "num_v $index_{v1}$ $index_{v2}$ ... $index_{vn}$". Below is a PLY example with respect to a simple planar triangle mesh.

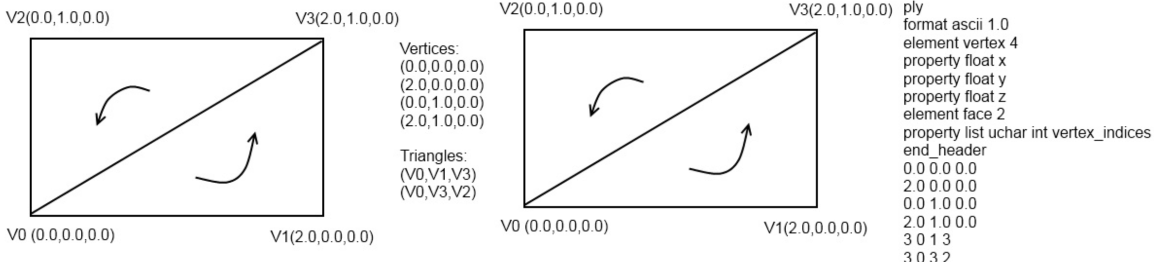


Figure 3.1: Left: Mesh native representation and PLY format

The simplicity of the PLY data representation makes reading and writing PLY files as simple as a linear scan of the files. Some other commonly-used mesh formats, such as VRML, use the same native representation. While the simplicity of this vertex/face representation is obvious, it contains data redundancy. The same vertex is shared among multiple polygons. As a result, it appears multiple times in the representation. Eliminating this redundancy is the major target of mesh compression algorithms.

3.3 Half-edge Data Structure

The Edgebreaker compression algorithm traverses meshes by accessing adjacent triangles. This type of traversal requires querying face adjacency frequently. The native representation doesn't store face adjacency directly. Computing face adjacency directly on each operation by searching all of the vertices in O(n) time is too costly.

More advanced data structures, such as the half-edge data structure solve this problem by pre-computing and storing all adjacency. Rossignac [Ros99] uses the half-edge data structure in his original Edgebreaker paper. The half-edge data structure represents an edge as two line segments directed in opposite directions. Each half-edge will store the pointers of the start vertex, the previous half-edge, the next half-edge and the opposite half-edge. The direction of the half-edge is determined by the triangle orientation using the right hand rule. Each triangle contains 3 half-edges circling the triangle in counter-clock order. The following is an example of the half-edge structure in a graph and a C structure implementation.

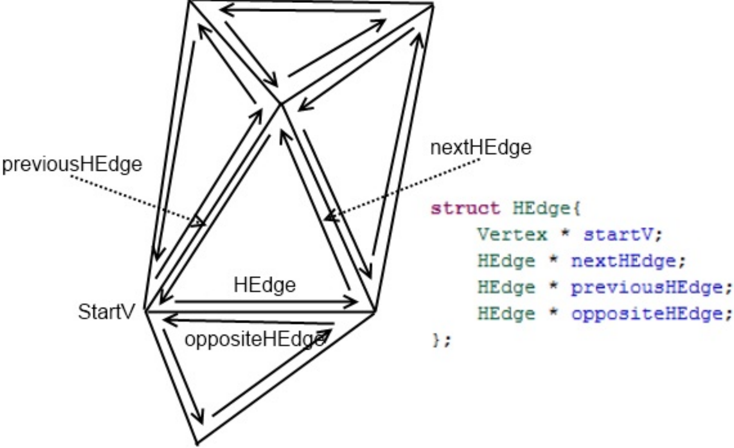


Figure 3.2: Half-edge data structure and its C implementation

Rossignac's Edgebreaker algorithm [Ros99] also needs to store the cut border list

and a flag to identify visited half-edges and vertices. As described in the original Edgebreaker paper, the cut border list is represented as a doubly-linked list embedded into the half-edge data structure. The following is a compact C structure for Edgebreaker.

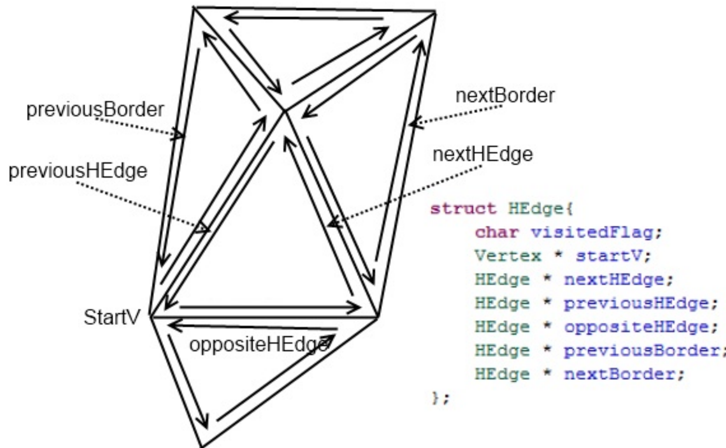


Figure 3.3: Edgebreaker extended half-edge data structure

Once the half-edge data structures are built from the native representation in PLY, querying adjacent triangles can be done in constant time. For example, moving to the right adjacent triangle can be done by using pointers in constant time as shown below.

$$activeBorder = activeBorder- > nextHEdge- > oppositeHEdge. \tag{3.1}$$

3.4 Basic Edgebreaker for simple meshes

3.4.1 Basic Compression

The Edgebreaker algorithm[Ros99] uses 5 basic types of operations to traverse the mesh: Cut(C), Left(L),Right(R), Split(S) and Eliminate(E). The meaning of these symbols and pseudo-code are described as follows:

Table 3.1: Five Basic Edgebreaker operators

Type	Conditions	Meaning	Action	After
Cut(C)		Reach a new vertex in unvisited area	1.Push the new vertex to output sequence. 2.Update border 3.Move to right triangle	
Right(R)		Reach local rightmost triangle	1.Update border 2.Move to left triangle	
Left(L)		Reach local leftmost triangle	1.Update border 2.Move to right triangle	
Split(S)		Split mesh into two disjoint portions	1. Push left portions to the stack 2.Update border 3.Move to right triangle	
Eliminate(E)		Reach last triangle in local area	1.Update border 2.Pop a left portion from the stack to continue processing	N/A

In each operation step, the entry edge of the current processing triangle is called "gate". At the end of each operation, the process will move to the next unvisited triangle and the gate(g) will move accordingly.

Algorithm 1 Basic Edgebreaker compression for solid mesh

Input: A triangle mesh m in half-edge data structure
Output: Edgebreaker compressed file

```
1: leftStack ← emptyStack
2: vertexSequence ← emptyList
3: operatorSequence ← emptyList
4: for each v in getFirstTriangleforSolidMesh(m) do
5:     Push v into vertexSequence
6: end for
7: gate ← getFirstGate(m)
8: while gate ≠ null do
9:     operatorType ← determineOperatorType(gate)
10:    Push operatorType to operatorSequence
11:    if operatorType = C then
12:        Push 3rd vertex to vertexSequence
13:        Update border
14:        gate ← getOpp(getNextHE(gate))
15:    else if operatorType = R then
16:        Update border
17:        gate ← getOpp(getPrevHE(gate))
18:    else if operatorType = L then
19:        Update border
20:        gate ← getOpp(getNextHE(gate))
21:    else if operatorType = S then
22:        Update border
23:        gate ← getOpp(getNextHE(gate))
24:        Push getOpp(getPrevHE(gate)) into leftStack
25:    else if operatorType = E then
26:        Update border
27:        gate ← leftStack.top()
28:        leftStack.pop()
29:    end if
30: end while
31: EncodedOperatorSequence ← EntropyEncoder(operatorSequence)
32: return vertexSequence + EncodedOperatorSequence
```

For simple meshes with no handles, C,R,L,E operations are processed in constant time; an S operation for a vertex v iterates through the neighbours of v. The average vertex degree is 6 in a mesh with no handles so the average cost of the S operation is also constant. Therefore, compression of simple meshes with no handles can be done in linear time.

As the Edgebreaker traversal visits every triangle once and only once, it transforms the 2-dimensional input space (mesh surface) into a one-dimensional input space (Edgebreaker operator sequence). After this traversal, the one-dimensional Edgebreaker operator sequence can be further compressed by a common entropy encoder, such as Huffman Encoding[Huf52], Arithmetic Encoding[WNC87] or LZW[Wel84].

At the end, the output of the Edgebreaker compression will be the compressed Edgebreaker operator sequence and the corresponding vertex sequence. The order of the vertex sequence is determined by the visiting order of each vertex in Edgebreaker traversal. That means if a vertex appear as the Nth vertex in Edgebreaker traversal, it will be placed in the Nth place in the output vertex sequence. This ordering of vertices is critical information for Edgebreaker decompression as we will see in the next section.

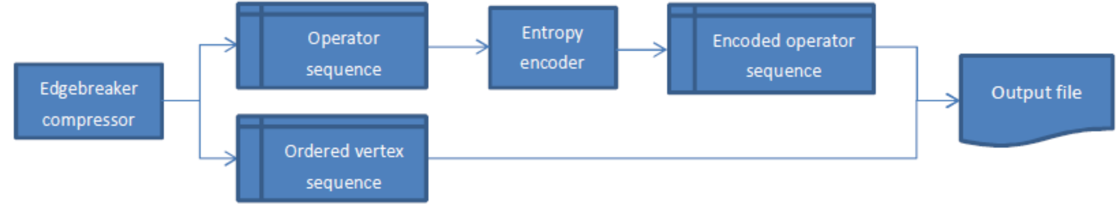


Figure 3.4: Edgebreaker compressor

Please note that the double branch in figure 3.4 is shown on purpose. The operator sequence and ordered vertex sequence are appended alternately inside the compressor. But only operator sequence will be further processed by an entropy encoder. Ordered vertex sequence will be stored as raw data in the output file.

3.4.2 Basic Decompression

Edgebreaker decompression is basically the inverse process of Edgebreaker compression. While compression produces the Edgebreaker operator sequence from mesh connectivity, decompression rebuilds the mesh connectivity from the Edgebreaker operator sequence.

In PLY files, mesh connectivity is stored in the form of the indices of triangle vertices. Therefore, the decompression for PLY files is to compute the triangle vertex indices from the Edgebreaker operator sequence. This is done by an operator sequence walk-through while rebuilding triangles and updating the border vertex list on the fly.

Given a mesh with no holes or handles, the first decompressed triangle is formed by the first 3 vertices in the vertex sequence. At the same time, the initial border vertex list is formed by the opposite order of these 3 vertices.

In the next step, the Edgebreaker operator sequence is read in a one-symbol-at-a-time fashion. For each input operator, we compute the triangle vertices' indices and update the border vertex list accordingly.

The rule of rebuilding triangles and updating the border vertex list is similar to the action described in Edgebreaker compression. Details are described in the following tables:

Table 3.2: Decompression for the five basic Edgebreaker operators

Type	Border (before)	Triangle (decompressed)	Border (after)
Cut(C)		B_0 B_1 B_{n+1}	
Right(R)		B_0 B_1 B_2	
Left(L)		B_0 B_1 B_n	
Split(S)		B_0 B_1 B_{offset}	
Eliminate(E)		B_0 B_1 B_2	N/A

As shown in the above table, the method of computing vertex indices is trivial for Cut(C), Left(L), Right(R) and Eliminate(E):

For Left(L), Right(R) and Eliminate(E) operators, all 3 vertices are either on or adjacent to the current gate in the border vertex list.

For Cut(C) operator, the first 2 vertices are on the current gate in the border

vertex list, while the 3rd vertex is just the next read vertex in the ordered vertex sequence.

For the Split(S) operator, the method of computing vertex indices is more complicated. Unlike C, L, R and E, the 3rd vertex of Split(S) is neither on the current gate nor in constant offset to the current gate in the border vertex list. Therefore, Split(S) decompression requires an extra step to determine the 3rd vertex's location, which is also the offset from the current gate in the border vertex list.

A trivial method to obtain the offset is to record it during compression and to store it in the compressed file. However, this trivial method will increase the storage cost of compressed files.

Rossignac[Ros99] proposed a more elegant method. Instead of storing Split(S) offset in the compressed file, all the offsets can be computed directly from one linear scan of the operator sequence. The details are as follows:

During the linear scan, when the Split(S) operator is read, push the current border length to a stack; when an Eliminate(E) operator is read, pop a length from the stack. Let the popped border length from stack be L_{before} and the current border length when Eliminate(E) is read be L_{after}. The offset can be computed as:

$$offset = L_{before} - L_{after} \tag{3.2}$$

The above formula can be illustrated by figure and the following reasoning:

For simple mesh with no handles and holes, the Split(S) operator will separate the mesh into 2 disjoint portions. After the Split(S), the Edgebreaker traversal will first traverse the right portion completely, and then go back to continue the left portion. Using the Edgebreaker traversal rule, the last operator of the right portion is always

Algorithm 2 Basic Edgebreaker decompression for solid mesh

Input: A Edgebreaker-compressed triangle mesh m

Output: A triangle mesh in PLY format

1: $vertexSequence \leftarrow emptyList$
2: **while** $v < v_num$ **do**
3: vertexSequence.add(readVertex(m))
4: **end while**
5: $operatorSequence \leftarrow entropyDecoder(m)$
6: $l \leftarrow 0$
7: $splitDepth \leftarrow 0$
8: $offsetPlaceHolderList \leftarrow emptyArrayList$
9: $offsetStack \leftarrow emptyArrayList$
10: **for** each $operatorType$ in $operatorSequence$ **do**
11: **if** $operatorType = C$ **then**
12: $l = l + 1$
13: **else if** $operatorType = R$ **then**
14: $l = l - 1$
15: **else if** $operatorType = L$ **then**
16: $l = l - 1$
17: **else if** $operatorType = S$ **then**
18: $l = l + 1$
19: offsetPlaceHolderList.push(0)
20: splitStack.push(Pair(length(offsetArraryList), l))
21: **else if** $operatorType = E$ **then**
22: $l = l - 3$
23: $(offsetIndex, l_before) \leftarrow splitStack.top()$
24: $splitStack.pop()$
25: $offsetPlaceHolderList[offsetIndex] \leftarrow l_before - l$
26: **end if**
27: **end for**
28: $triangleSequence \leftarrow emptyList$
29: $borderList \leftarrow emptyList$
30: $LeftborderStack \leftarrow emptyStack$
31: $readVertexIndex \leftarrow 2$
32: $offsetIndex \leftarrow 0$
33: **for** each $operatorType$ in $operatorSequence$ **do**
34: **if** $operatorType = C$ **then**
35: $readVertexIndex \leftarrow readVertexIndex + 1$
36: $T \leftarrow (borderList[0], borderList[1], readVertexIndex)$
37: $triangleSequence.push(T)$
38: Update border

```
39:     else if operatorType = R then
40:         T ← (borderList[0], borderList[1], borderList[2])
41:         triangleSequence.push(T)
42:         Update border
43:     else if operatorType = L then
44:         T ← (borderList[0], borderList[1], borderList.last())
45:         triangleSequence.push(T)
46:         Update border
47:     else if operatorType = S then
48:         offset ← offsetPlaceHolderList[offsetIndex]
49:         T ← (borderList[0], borderList[1], borderList[offset])
50:         triangleSequence.push(T)
51:         Update border and split border to left and right portion
52:         LeftBorderStack.push(left border)
53:         borderList ← rightborder
54:     else if operatorType = E then
55:         T ← (borderList[0], borderList[1], borderList[2])
56:         triangleSequence.push(T)
57:         borderList ← LeftBorderStack.top()
58:         leftBorderStack.pop()
59:     end if
60: end for
61: Return vertexSequence + triangleSequence
```

Eliminate(E). This forms an operator sequence zone headed by Split(S) and tailed by Eliminate(E) in the operator sequence. We define the following symbols for the formula:

1. L_{before}: the whole length when Split(S) happens.
2. L_{after}: the whole length when the Eliminate(E) happens.
3. L_{left}: the length of the Split(S) left portion.
4. L_{right}: the length of the Split(S) right portion.

Based on the above reasoning:

$$L_{before} = L_{left} + L_{right} \tag{3.3}$$

$$L_{after} = L_{left} \tag{3.4}$$

$$offset = L_{right} \tag{3.5}$$

Therefore, offset = L_{right} =(L_{left}+L_{right})-L_{left}= L_{before} - L_{after}

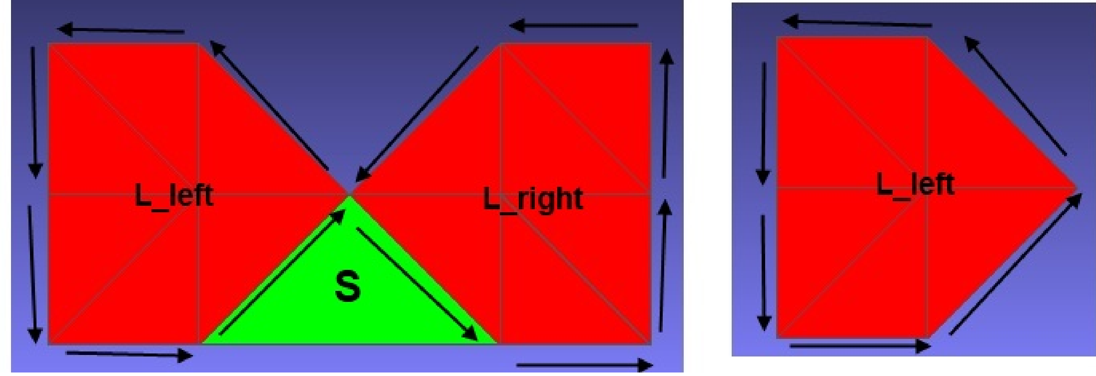


Figure 3.5: Left:before split, Right:after right traversal

Once the offset O is obtained, to compute the 3rd vertex indices for Split(S) is to

find the Oth right element from the current gate in the border list. Using a double-linked list data structure, it can be done in O(offset). In worst case, the offset will be as large as the full border list length and can make the overall decompression to be a $O(n^2)$ process. But as we will see in our fossil dataset, the split is usually unbalanced and the right side is typically very small. This makes the whole decompression closer to linear performance.

3.5 Supporting Holes

The basic Edgebreaker algorithm described above works for simple meshes with no holes or handles. The samples in our fossil dataset usually have holes. To faithfully represent the data, we need to extend the algorithm to support holes. We use the method proposed by Rossignac [Ros99] on supporting holes, which is more cost-efficient than patching holes before compression and then removing the patches in decompression.

On compression:

1. Do a linear scan of the half-edge list and identify all holes by detecting the half-edges which have no opposite half-edge.

2. Pick one hole as the starting point of the compression, record the number of vertices surrounding the hole in the output stream.

3. Later in the compression, if reaching a vertex in another hole, merge that hole's vertices into the current border vertex list and record that hole's length in the output stream. Use a new Edgebreaker operator type M to represent this merge operation and place it in the Edgebreaker operator sequence.

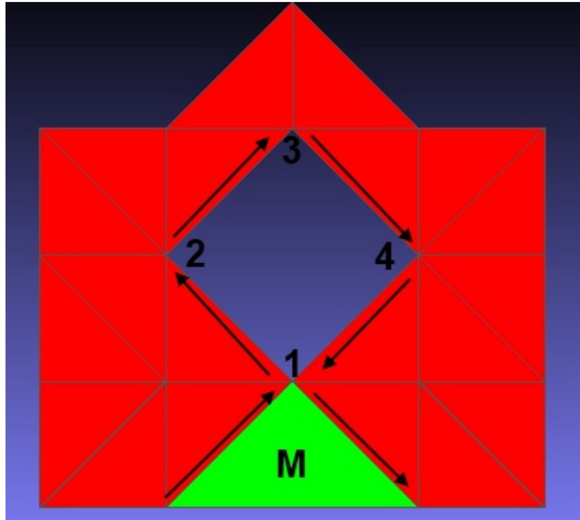


Figure 3.6: Add hole vertices into vertex sequence

On decompression, every time when we reach a Merge(M) operator:

1. We rebuild the triangle using the vertices of current gate and the next vertex in the input stream.

2. We read the corresponding hole length(H) from the input stream.

3. We read the next H vertices in the input stream and merge them into the active border list.

This will rebuild the triangle and update the border list correctly.

This method will extend Edgebreaker to support holes without increasing time complexity and storage cost. We analyze the time complexity and storage cost of this extension and the result is as follows:

Time complexity:

1. In step 1, It takes O(n) time to perform a linear scan to identify and mark all the holes.

2. In step 4, merging a hole will use the following steps. We remove the marker and output all the hole vertices along the way. It will take O(H) time, where H is the number of vertices surrounding the hole.

As the analysis above shows, supporting holes does not increase the computational complexity of the algorithm.

Storage cost:

In the compressed file, the only extra storage cost is a 32-bit integer representing the hole length. Compared to patching holes, which will add triangles to the operator sequence, this method has much better storage cost compared to patching holes when the hole is large.

3.6 Supporting Handles

Like holes, handles add another level of complexity to the simple Edgebreaker mesh traversal. The complexity comes from the fact that adding handles will make the Split(S) operators and Elimiate(E) operators behave differently. As we previously discussed, without handles, a Split(S) operator will fully separate the left and right portions. This property no longer holds when the mesh contains handles. This brings the following 3 problems in Edgebreaker:

1. On compression, not every Split(S) operator has a corresponding Eliminate(E) operator. We need to detect the handle-related Split(S) operators.

2. On compression, we need to merge the handle-related Split(S) operators' left portions into the active border.

3. On decompression, we need to restore the Split(S) offsets which are affected by merging handles.

The above problems raise the 3 aspects we need to consider in order to support handles.

1. Detecting handles.

2. Merging handles.

3. Computing the affected Split(S) offsets.

3.6.1 Handle Detector

Rossignac [Ros99] proposed his method to detect handles, however, his method will increase overall time complexity to O(n^2). His method to detect handles is described in the paper as follows:

"*We modify the S operation as follows. When the current loop is split into the left and right sub-loops, we mark the vertices and edges of the left sub-loop with a 3...During compression, when we later reach a vertex marked with a 3, we perform a different merging operation*"

This method will need O($L_{leftloop}$) marking operations per S operation. Because both the size of the loop and the number of S operation is O(n), the time complexity will increase to O(n^2).

Compared to the original method, I developed a new handle detector as follows:

New handle detector: *On the split(S) operation, there is no need to mark the left loop. Instead, traverse the current loop counterclockwise, and search for the joint half-edge. If this is a split(S) operation, the search will find the joint half-edge after*

Algorithm 3 Old handle detection

```
1: ....
2: if operatorType = S then
3:     ...
4:     for each e in leftBorder do
5:         e.vertex.marker ← 3
6:     end for
7: else if operatorType = E then
8:     ...
9:     for each e in currentBorder do
10:        e.vertex.marker ← 1
11:    end for
12: end if
13: ....
14: thirdVertex ← getPrevHE(gate).vertex
15: if previousHEdgeisnotonborder and nextHEdgeisnotonborder then
16:    if thirdVertex.marker == 1 then
17:        operatorType ← S
18:    else if thirdVertex.marker == 3 then
19:        operatorType ← N
20:    end if
21: end if
22: ....
```

$L_{rightloop}$ *operations; if this is in fact a handle operation, the search will not find the joint half-edge and will stop after traversing the full loop. The handle operator is confirmed when the joint half-edge is not found in the full-loop search.*

Algorithm 4 New handle detection

```
1: ....
2: if operatorType = S then
3:     jointHEdge ← getOpp(getPrveHe(gate))
4:     while jointHEdgeisnotonborder do
5:         jointHEdge ← getOpp(getPrveHe(jointHEdge))
6:     end while
7:     currentHEdge ← gate.nextBorder
8:     while currentHEdge! = jointHEdge and currentHEdge! = gate do
9:         currentHEdge ← gate.nextBorder
10:    end while
11:    if currentHEdge = gate then
12:        operatorType ← N
13:        restart this triangle processing as operator N
14:    end if
15:    ...
16: end if
17: ....
```

The new handle detector is more efficient than the original one. The conclusion comes from the following two observations:

Observation 1. *The split(S) operation is usually unbalanced. The average ratio of* $L_{leftloop}$ *and* $L_{rightloop}$ *is usually between 50:1 to 20:1.*

Table 3.3: left:right ratio of the split(S) operations

Model	Triangles	split(S)	$\sum L_{left}$	$\sum L_{right}$	Avg_l	Avg_r	$Avg_l : Avg_r$
bunny	69451	897	226830	5526	252.88	6.16	41.05
horse	96965	1369	34862	6525	25.47	4.77	5.34
paleop	644628	20779	20499145	382293	986.53	18.40	53.62
Skull	723799	26856	33459065	845008	1245.87	31.46	39.60
dragon	871414	44877	30784929	1587932	685.98	35.38	19.38
mummy	1426014	41156	77874167	2709024	1892.17	65.82	28.74
Plaque	1555485	57253	126034417	2036060	2201.36	35.56	61.90
Axis 5	1824337	65351	131790646	2673750	2016.65	40.91	49.29
coryth	1899550	91810	179871693	9200906	1959.17	100.21	19.54
Barosa	2098167	83488	125759071	2601095	1506.31	31.16	48.35
T_rex	2349117	112022	371877550	17409393	3319.68	155.41	21.36

Observation 2. *The right loops are usually extremely small. Usually, more than 70% of the right loop consist of only 3 edges.*

Table 3.4: ratio of three-edge right loop

Model	Split	L_{right}=3	%
bunny	897	800	89.18%
horse	1369	1229	89.77%
paleop	20779	17181	82.68%
Skull	26856	20974	78.10%
dragon	44877	30710	68.43%
mummy	41156	30664	74.51%
Plaque	57253	46924	81.96%
Axis 5	65351	52905	80.96%
coryth	91810	63776	69.47%
Barosa	83488	66064	79.13%
T_rex	112022	77748	69.40%

All our test models match the above observation and the average ratio between left and right portions are typically from 20:1 to 50:1.

While the old detector will take $\sum L_{left}$ operations, our new detector will only $\sum L_{right}$ plus the number of left loops associated with the same number of handle

operations. A typical model, such as the models in our test samples, will have less than a few dozens of handles. That means our new handle detector can be 20-50 times more efficient than the old detector.

3.6.2 Merging handles

After handle detection, we need to merge handle borders. The process is in fact very similar to merging holes. The only difference is that the search and merge is in the inactive border list, not the hole border list. For more details, we need to do the following:

1. Using half-edge operations, traverse up along triangles in the right fan of disjoint vertex until it reaches a half-edge on the other side of a border.

2. Search that half-edge from the inactive border list

3. Merge the border containing that half-edge into the current active border and remove it from the inactive border list

3.6.3 New formula for Split Offset Calculation

On decompression, a pre-processing algorithm is needed to compute the Split(S) offset. An offset calculation method has been proposed by Rossignac [Ros99]. The formula is $offset = L_{before} - L_{after}$. However, if the mesh has handles, since the left and right portions of Split(S) operations are not always disjoint now, the original formula is no longer valid. Rossignac [Ros99] did not propose a way to calculate the split offset under this new condition. I independently developed the following two methods to solve the Split(S) offset calculation problem.

The first method is as follows:

1. A split(S) operator is considered to be affected by handle operator if the handle operator is between the split(S) operator and its corresponding Eliminate(E) operator.

2. On compression, explicitly store the affected split(S) offset in the compressed file.

3. On decompression, directly use the stored offset info for affected split(S) operator.

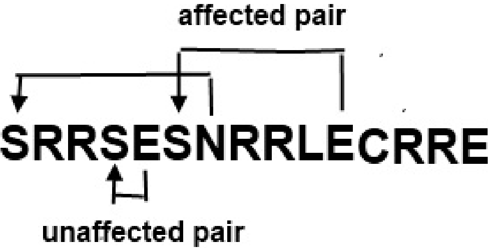


Figure 3.7: Edgebreaker Operator Sequence with Handle operator N

The method comes from our observations in the fossil dataset: the majority of Split(S) operations are unbalanced and the right portions of Split(S) operations are typically very small. In this case, most Split(S) operators will not have handle operators between them and their corresponding Eliminate(E) operators. Those split(S) offsets won't be affected by handle operators. As a result, only a small percentage of Split(S) offsets need to be stored in compressed file. Therefore, using this method won't increase the storage cost noticeably.

The second method is as follows:

1. On decompression, every time a handle operator is reached, for every Split(S) operator that is still in the Split(S) stack, if it is also between handle operator and handle's corresponding Split(S) operator, its L_{before} needs to be deducted by the handle's corresponding Split(S) left portion's length $L_{splitleft}$.

2. The deduction accumulates until the Split(S) is popped from Split(S) stack when its corresponding Eliminate(E) operator is reached.

3. After the Split(s) is popped from Split(S) stack, its offset is now calculated as offset=$(L_{before} - L_{splitleft1} - L_{splitleft2}...... - L_{splitleftN}) - L_{after}$.

This method is more elegant as it does not need to explicitly store offsets in the compressed file. The new formula is derived from Rossignac's [Ros99] original offset calculation formula for simple meshes without handles. Rossignac's formula $offset = L_{before} - L_{after}$ uses 2 lengths (before and after) for the offset calculation. My formula uses (2+N) lengths, where N is the number of handle-merges which across the Split(S).

The new offset calculation formula:

Offset=L_{right}=$(L_{before} - L_{splitleft1} - L_{splitleft2}...... - L_{splitleftN}) - L_{after}$

where N is the number of handle merge across the split(S).

The correctness of the second method is tested in our fossil dataset and can also be proved by the following lemma and theorem:

Lemma 1. *For each split between the handle and handle's corresponding split, its L_{before} reduces by the left portion's length of handle's corresponding split.*

Proof. Without handles, offset=L_{right}=$(L_{right} + L_{left}) - L_{left}$=$L_{before} - L_{after}$. With a handle, $L_{after} \neq L_{left}$ because some left portion of the border has been merged

into the active border when merging the handle. The taken-out portion is the left portion of handle's corresponding split(S), so now $L_{after} = L_{left} - L_{splitleft}$. To re-balance the formula, L_{before} should also subtract $L_{splitleft}$. Under the new condition, offset=L_{right}=$(L_{right} + L_{left}) - L_{left}$=$(L_{right} + L_{left} - L_{splitleft}) - (L_{left} - L_{splitleft})$=$(L_{before} - L_{splitleft}) - L_{after}$. □

Theorem 1. *The offset of Split(S) paired with Eliminate(E) across multiple handles will be adjusted by the total left length of the corresponding handles' merged splits.*

Proof. This is equivalent to proving that each handle adjustment is independent from other handle adjustments. We prove that the left border of one split is totally different from the left border of another split because the split is always on the active border. When a split happens, this left border will be pushed onto the inactive border list; a later split, which will work on the active border list, won't touch the inactive border list. All the affected split(S) by handle merges won't reach the corresponding Eliminate(E) when the merge happens, so their left borders are totally independent. Therefore, their adjustments are independent and their affect can be accumulated by simple addition. □

The 2nd method I develop is the method I use in my final implementation.

3.7 Supporting non-manifold vertices

The samples in our fossil dataset also have non-manifold vertices. The Edgebreaker algorithm described above only works for manifold meshes. The non-manifold vertices break the Edgebreaker algorithm by possibly sending incorrect termination signal while reaching Eliminate(E).

In order to extend the algorithm to support non-manifold vertices, we follow the guidelines in Rossignac's [Ros99] paper to convert a non-manifold mesh to a manifold mesh. The basic idea is to create one dummy vertex for each triangle fan of a non-manifold vertex. A triangle fan is a sequence of adjacent triangles connected to a common vertex as shown in figure 3.8. The dummy vertex has the same coordinates and properties as non-manifold by adding one dummy vertex for each triangle fan. We split the joint non-manifold vertex into a few disjoint triangle fans. Although each dummy vertex has same coordinates and properties, this converts the mesh to a manifold from mesh connectivity point of view.

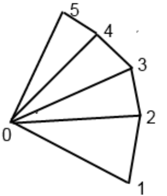
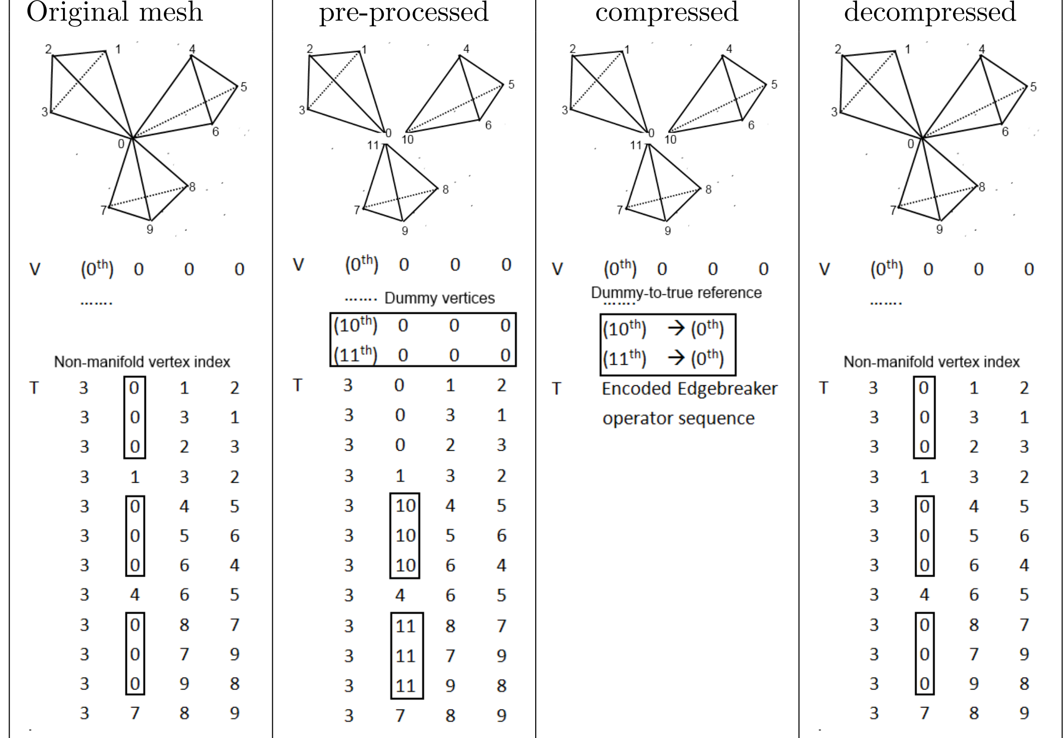


Figure 3.8: A triangle fan

Table 3.5: non-manifold vertices conversion



Original mesh	pre-processed	compressed	decompressed
V (0th) 0 0 0 …….	V (0th) 0 0 0 ……. Dummy vertices (10th) 0 0 0 (11th) 0 0 0	V (0th) 0 0 0 ……. Dummy-to-true reference (10th) → (0th) (11th) → (0th)	V (0th) 0 0 0 …….
Non-manifold vertex index			Non-manifold vertex index
T 3 0 1 2	T 3 0 1 2	T Encoded Edgebreaker operator sequence	T 3 0 1 2
3 0 3 1	3 0 3 1		3 0 3 1
3 0 2 3	3 0 2 3		3 0 2 3
3 1 3 2	3 1 3 2		3 1 3 2
3 0 4 5	3 10 4 5		3 0 4 5
3 0 5 6	3 10 5 6		3 0 5 6
3 0 6 4	3 10 6 4		3 0 6 4
3 4 6 5	3 4 6 5		3 4 6 5
3 0 8 7	3 11 8 7		3 0 8 7
3 0 7 9	3 11 7 9		3 0 7 9
3 0 9 8	3 11 9 8		3 0 9 8
3 7 8 9	3 7 8 9		3 7 8 9

In our application, the above idea is implemented in the following steps in compression and decompression:

1. On compression, in the pre-processing step, identify all non-manifold vertices, create a dummy vertex for each triangle fan, and group them for each non-manifold vertex in the data structure.

2. On compression, when reaching a dummy vertex, if it is the first dummy vertex be reached among its dummy vertex group for that non-manifold vertex, push it to the ordered vertex sequence. Otherwise, only push the vertex index to the output stream as a recorded duplicate vertex location and don't push it to ordered vertex sequence.

3. On decompression, read all recorded duplicate vertex locations in the file header. Later on, when reaching a duplicate vertex reference on decompressing a triangle, we will replace it with the true vertex index.

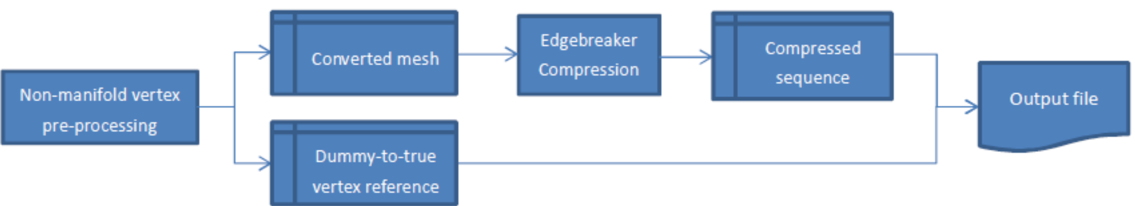


Figure 3.9: Compression with non-manifold pre-processing

Please note that the double branch in figure 3.9 is shown on purpose. The converted mesh and the dummy-to-true vertex reference are generated alternately inside the pre-processing step. But only converted mesh will be further processed by Edgebreaker compression. The dummy-to-true vertex reference will be stored as raw data in the output file.

3.8 Supporting multiple surfaces and isolated vertices

The samples in our fossil dataset also have some small floating surfaces and isolated vertices. These floating surfaces and isolated vertices could be background noise from scanning or smaller objects of the fossil samples. Depending on the objects and applications, this multiple-surface structure could contain valuable information for the end-user. A lossless compression should faithfully represent the input file. Therefore, modification and truncation of data should not be done in lossless compression.

Whether those floating surfaces are valuable is up to the end-user decision.

Figure 3.10: fossil mesh with floating small surface

Based on the above argument and our fossil dataset, we extend Edgebreaker to support multiple surfaces and isolated vertices, we add the following steps in the compression to accomplish that:

1. Keep a complete list of holes/borders after pre-processing.

2. Mark all borders as unvisited borders at the beginning. Each time we merge a border, mark it as visited, and remove it from the unvisited border list.

3. While the Edgebreaker traversal claims to be finished (which means it reaches an Eliminate(E), and there are no remaining Split(S) gate in the stack), check the size of the unvisited border list. If there are still unvisited borders, take one, that must be a border for another surface with holes/borders, and run Edgebreaker from that border to traverse the new surface with holes/borders.

4. While the size of unvisited border list becomes zero, all the surfaces with holes/borders have been visited.

5. Do a linear scan of the half-edge list. If there are still unvisited half-edges, that must be in an unvisited surface with no holes. Run Edgebreaker again from that half-edge and its triangle to traverse the new surface with no holes.

6. While all half-edges are visited, all the closed and open surfaces are visited.

7. Do a linear scan of the vertex list, any unvisited vertex in this step must be an isolated vertex. Push all these isolated vertice to the output vertex sequence.

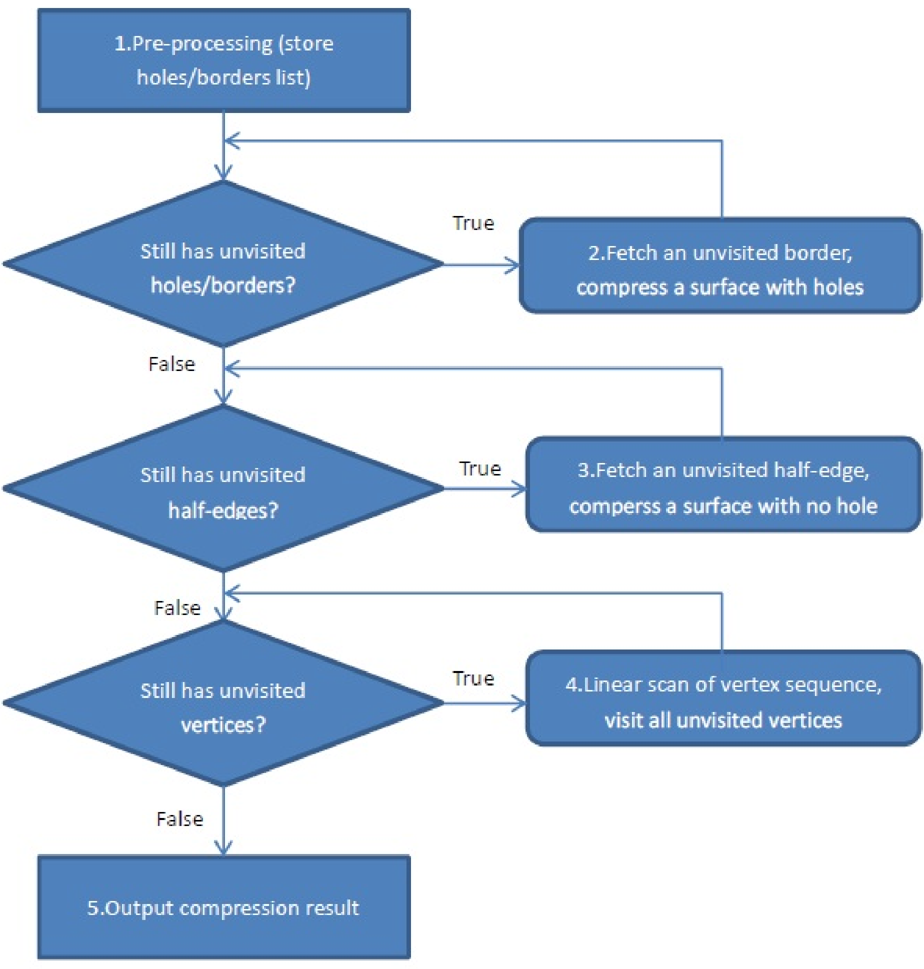


Figure 3.11: multi-surface compression

The above extension ensures we visit all surfaces and vertices in the compression. Record the number of surfaces we reached and the border size of each surface in the compressed file.

On decompression, we utilized surface information we stored in the compressed file to correctly decompress all surfaces:

When Edgebreaker finishes (that means it reaches an Eliminate(E) and with no Split(S) gate in the stack), check whether there are still unvisited surfaces by comparing the reached surfaces at this moment to the stored number of surface information in the compressed file. If there are still other surfaces, read the next surface border size from the stored compressed file, read the next |S| vertex in vertex sequence as starting border and run Edgebreaker again on the new surface. Stop when we reach the same number of surface as the stored number of surface information in the compressed file.There are no extra steps for isolated vertices as they have been placed in the ordered vertex sequence already.

3.9 Encoding Schema

The operation sequence can be encoded using common operations. Static and Dynamic encoding methods are both suitable for this encoding, depending on the need. A mathematical proof can show that a static method with a simple fixed schema can guarantee to compress the data into 2 bits/triangles. Experimentally, dynamic methods can further compress the data into 0.9∼1.3 bits/triangles. Our contribution here is to propose a simple fixed schema which can compress the data into 1.5∼1.7 bits without increasing any programming complexity.

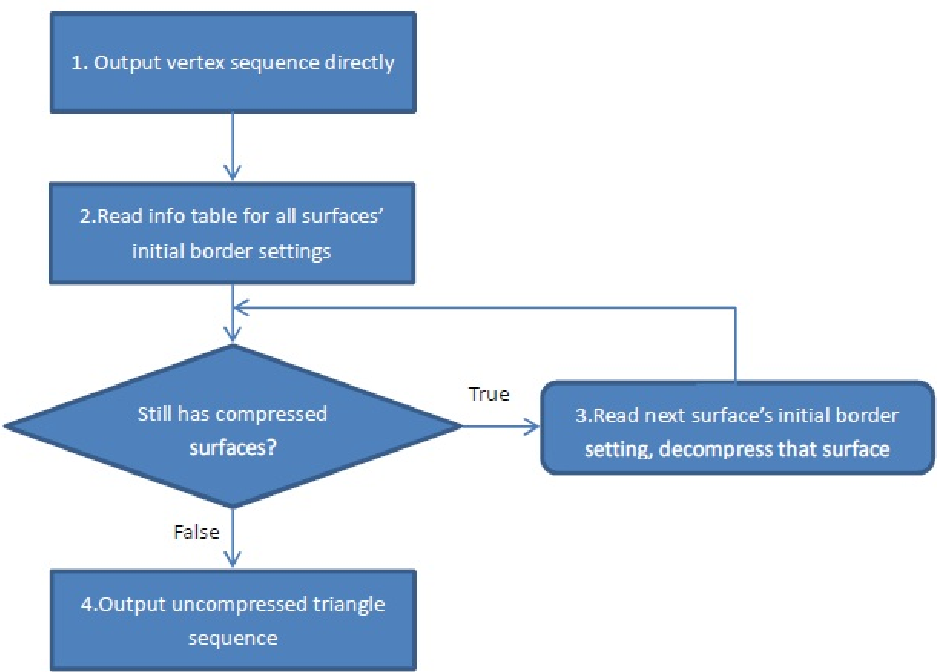


Figure 3.12: multi-surface decompression

3.9.1 New Fixed Encoding Schema

Edgebreaker uses an alphabet with only 5 symbols(C,L,R,S,E), which makes Huffman encoding a suitable choice for compression. A disadvantage is that the encoding has to be done after reading all the elements. This increases the memory storage cost for large meshes. On the other hand, if we can prove that all the meshes have a similar probability distribution on the 5-symbol alphabet, we could use a fixed schema and compress the data on-the-fly.

It's very easy to design a fixed encoding schema based on the following theorem and corollary.

Theorem 2. *For large meshes with small amount of holes and handles, $F \approx 2V$ (where F is the number of faces in a mesh and V is the number of vertices in a mesh).*

Proof. For a large triangle mesh with small amount of holes and handles, any non-boundary edge is shared by 2 triangle faces, and any triangle face consists of 3 edges. This leads to the conclusion that we can express the approximation of the total number of triangles' edges in 2 different ways:

$$2E \approx 3F \Rightarrow E \approx \frac{3}{2}F \tag{3.6}$$

According to Euler's polyhedron formula, for any polyhedron with handles:

$$V - E + F = 2 - 2H \tag{3.7}$$

Substitute equation 3.6 to equation 3.7:

$$V - \frac{3}{2}F + F \approx V - E + F = 2 - 2H \tag{3.8}$$

$$\Rightarrow V - \frac{1}{2}F \approx 2 - 2H \tag{3.9}$$

$$\Rightarrow F \approx 2V - 2(2 - 2H) \tag{3.10}$$

Since it's a large triangle mesh with small amount of holes and handles, the number of handles, H, is negligible compared to F and V.

$$F \approx 2V \tag{3.11}$$

□

Based on the above theorem, we can find the following important property of

Edgebreaker.

Corollary 3. *For a large triangle mesh with small amount of holes and handles, the number of cut(C) operators is about half of the size of Edgebreaker operator sequence.*

Proof. Since each Edgebreaker step visits one unvisited triangle and generates one Edgebreaker operator, F = $|operator_sequence|$.

Since every cut(C) step visits one unvisited vertex, V = $|C|$.

Based on theorem 2:

$$F \approx 2V \tag{3.12}$$

$$|C| = V \approx \frac{1}{2}F = \frac{1}{2}|operator_sequence| \tag{3.13}$$

□

Without using any other probability distribution information, the above corollary yields the fixed schema Rossignac[Ros99] proposed:

Table 3.6: Original fixed schema

Operator	bit representation
Cut(C)	0
Left(L)	100
Right(R)	101
Split(S)	110
Eliminate(E)	111

According to information theory, the above schema will be a good approximation to the optimized encoding only if the probability of L,R,S and E are the same. However, the pattern observed from our experiments shows that L,R,S,E are in fact not uniformly distributed. In fact, the probability distribution obeys another pattern.

Observation 3. *The probability distribution of the five Edgebreaker operators has the following pattern: [P(C),P(R)] $\gg$ [P(S),P(E)] $\gg$ P(L), where P(R) is 35%$\sim$47%, P(S),P(E) is 3%$\sim$5% and P(L) is 1%$\sim$2%.*

Using the above observation, we can use the following alternative schema.

Table 3.7: New fixed schema

Operator	bit representation
Cut(C)	0
Right(R)	10
Split(S)	110
Eliminate(E)	1110
Left(L)	1111

3.10 Correctness Measurement and Testing

The mesh is considered unchanged after compression if all the following three criteria met:

1. All the vertices' coordinates remain the same
2. All the triangles' vertices remain the same
3. All the triangles' edge orientations remain the same

Edgebreaker is a lossless compression algorithm. So in our testing, we need to make sure all 3 criteria above are met. However, in Edgebreaker, it is critical to reorder the vertex sequence to be the same as mesh traversal order. This re-ordering won't change mesh geometrical structure and coordinates' accuracy, but this re-ordering will alter the bit order in the output file.

Table 3.8: vertex and triangle sequence comparison

Before compression	After compression
V 0 0 0 1 1 0 2 0 0 1 0 1	V 0 0 0 2 0 0 1 1 0 1 0 1
T 3 0 2 1 3 0 3 2 3 0 1 3 3 1 2 3	T 3 0 1 2 3 1 0 3 3 3 0 2 3 3 2 1

In this case, a bitwise comparison between native input and output files is not suitable for measuring correctness of Edgebreaker compression. In my experiment, I develop the following method to make sure the mesh geometrical structure remain unchanged based on the above 3 criteria:

The basic idea of our testing method is to sort input and output files before a bitwise comparison.

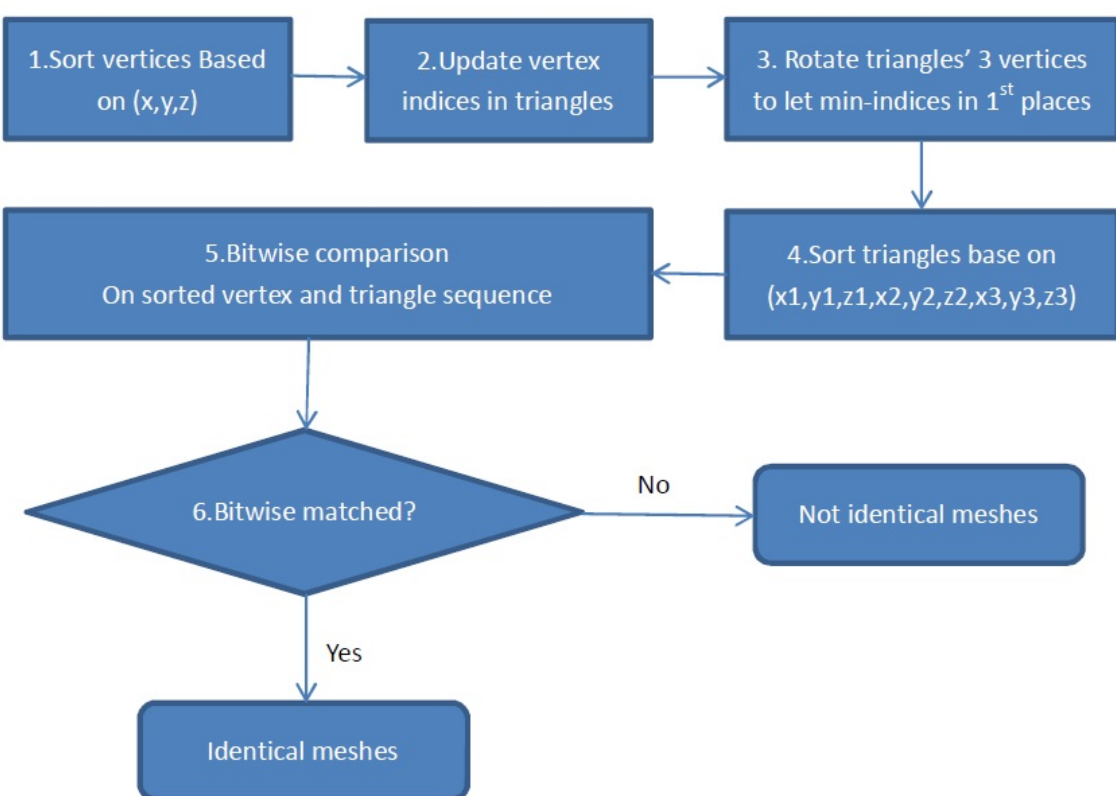


Figure 3.13: Edgebreaker correctness testing flow chart

Table 3.9: vertex and triangle sequence comparison (sorted)

Before compression	After compression
V 0 0 0 1 1 0 2 0 0 1 0 1 T 3 0 2 1 3 0 3 2 3 0 1 3 3 1 2 3	V 0 0 0 2 0 0 1 1 0 1 0 1 T 3 0 1 2 3 1 0 3 3 3 0 2 3 3 2 1
Before compression (sorted)	**After compression (sorted)**
V 0 0 0 1 0 1 1 1 0 2 0 0 T 3 0 1 3 3 0 2 1 3 0 3 2 3 1 2 3	V 0 0 0 1 0 1 1 1 0 2 0 0 T 3 0 1 3 3 0 2 1 3 0 3 2 3 1 2 3

3.11 Experimental Results

3.11.1 Experiment setup

To ensure our implementation work in generalized scenarios, the experiment is run in meshes with various properties. The details of property settings are as follows:

Table 3.10: Test Dataset's Property Matrix

Model	Hole	Handle	Non-manifold vertices	Multiple surfaces	RGB	Alpha
horse						
bunny	X					
dragon	X	X		X		
plaque					X	
Axis 5		X	X	X	X	
Barosa		X	X	X	X	
paleop	X	X	X	X	X	
mummy	X	X	X	X	X	
coryth	X	X	X	X	X	
skull		X	X	X	X	X
T_rex	X	X	X	X	X	X

The test models cover all the properties we discussed in the previous sections, including simple meshes, holes,handles, non-manifold vertices and multiple surfaces. We also choose meshes with different vertex color properties, including uncolored meshes, RGB-colored meshes and RGBA-colored meshes.

The experiment is run in consumer-class PCs in both our lab and our research partner's lab. The test PC in our lab has an Intel Quad-core processor with 4GB RAM. Our implementation runs very fast in this hardware testing. It can compress the largest test models in 14 seconds.

3.11.2 Compression Ratio

To ensure the efficiency of our implementation, a comparative experiment on compression ratio is performed on the 3 compression methods: Zip, Edgebreaker and Edgebreaker+Zip. The compression ratio is size comparison between the compressed file and the original file. In this thesis, the exact meaning of compression ratio is compressed file size divides uncompressed file size.

The charts below shows that our Edgebreaker implementation outperforms Zip in mesh compression with respect to compression ratio.

As Edgebreaker only compresses connectivity data and leaves the vertex data in raw format, it also leads to the following two characteristics in our comparative experiment:

1. Edgebreaker+Zip always produces the best compression ratio.
2. Edgebreaker produces different compression ratio on meshes with different size of vertex property.

In our test dataset, there are 3 mesh categories with respect to vertex property: 1. Uncolored meshes; 2. RGB-colored meshes; 3. RGBA-colored meshes.

Table 3.11: Uncolored Mesh's Compression Ratio

Model	Original	Zip	Edgebreaker	Edgebreaker+Zip
bunny	1304KB	62.65% (817KB)	33.44% (436KB)	30.06% (392KB)
horse	1800KB	56.28% (1013KB)	32.67% (588KB)	27.89% (502KB)
dragon	16192KB	45.20% (7318KB)	32.82% (5315KB)	19.71% (3191KB)

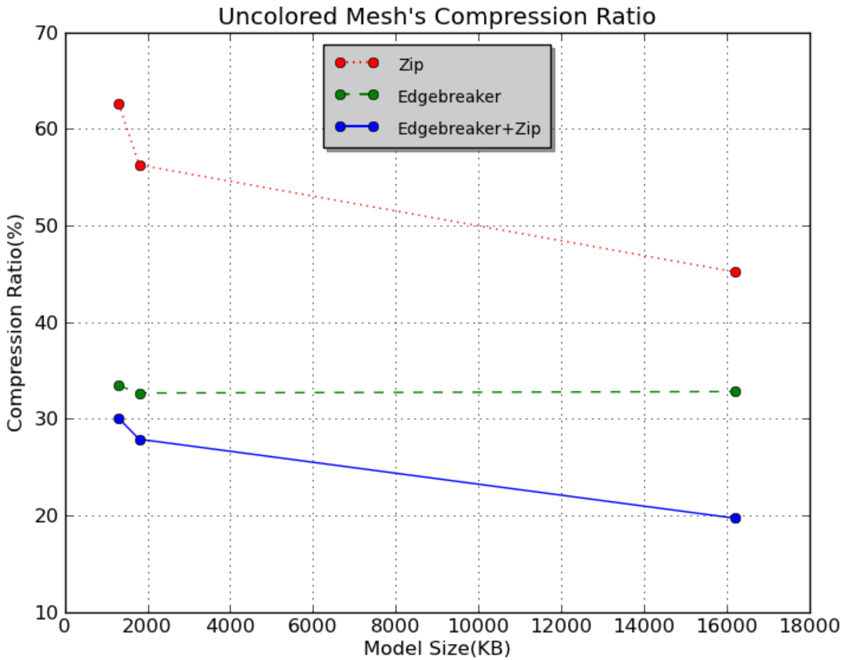


Figure 3.14: Uncolored Mesh's Compression Ratio

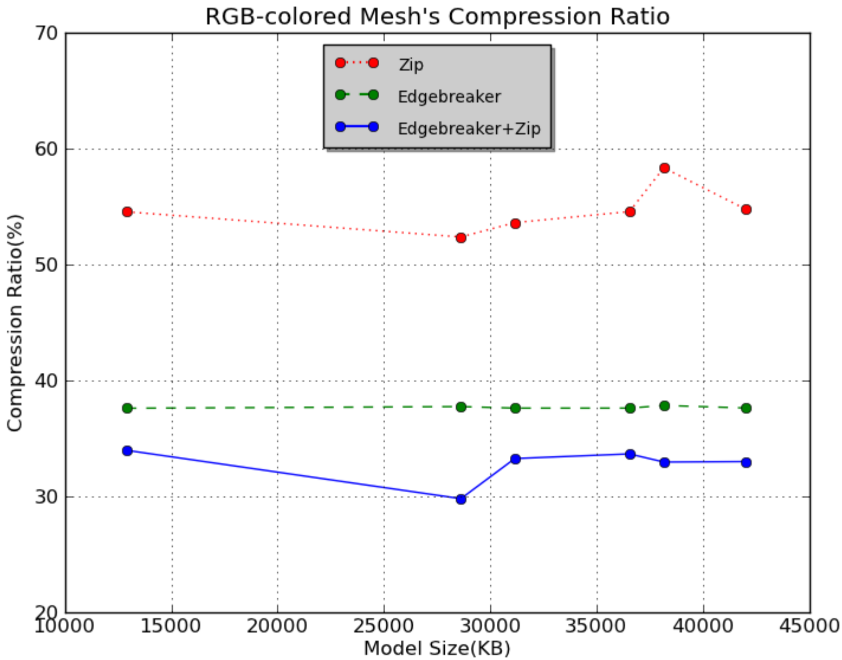


Figure 3.15: RGB-colored Mesh's Compression Ratio

Table 3.12: RGB-colored Mesh's Compression Ratio

Model	Original	Zip	Edgebreaker	Edgebreaker+Zip
paleop	12906KB	54.52% (7036KB)	37.58% (4850KB)	33.95% (4382KB)
mummy	28609KB	52.35% (14977KB)	37.72% (10790KB)	29.78% (8521KB)
plaque	31141KB	53.59% (16689KB)	37.59% (11705KB)	33.24% (10351KB)
Axis 5	36523KB	54.54% (19918KB)	37.59% (13728KB)	33.64% (12288KB)
coryth	38137KB	58.32% (22240KB)	37.82% (14423KB)	32.94% (12561KB)
Barosa	42005KB	54.72% (22984KB)	37.60% (15793KB)	32.98% (13855KB)

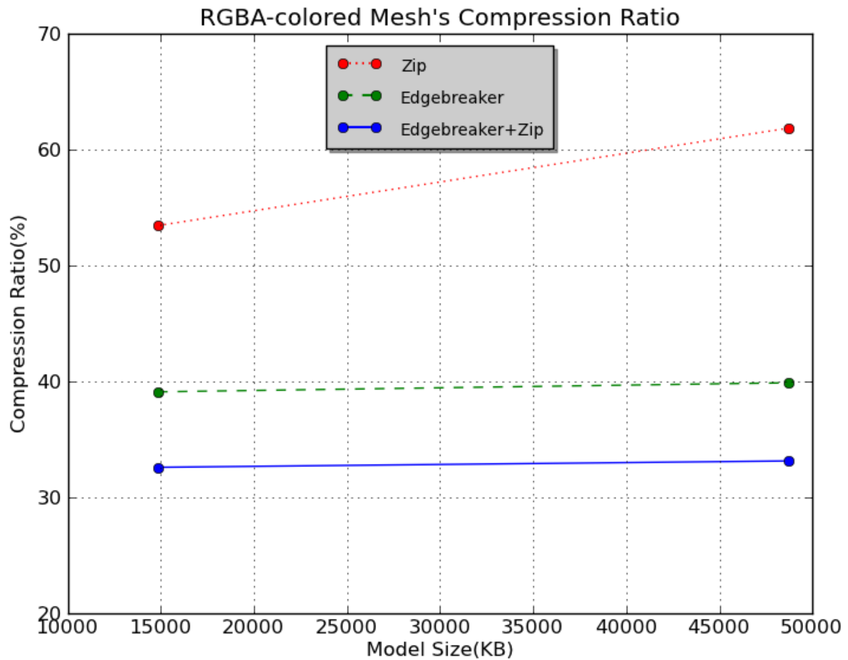


Figure 3.16: RGBA-colored Mesh's Compression Ratio

Table 3.13: RGBA-colored Mesh's Compression Ratio

Model	Original	Zip	Edgebreaker	Edgebreaker+Zip
Skull	14845KB	53.42% (7930KB)	39.08% (5802KB)	32.57% (4835KB)
T_rex	48734KB	61.83% (30134KB)	39.86% (19424KB)	33.12% (16139KB)

The experiments show the compression ratio remains the same level regardless the additional geometrical properties we deal with. This result is consistent to what we

expect as in our design, the storage cost for holes, handles, non-manifold vertices and multiple surface are O(1) to the number of the mesh's total vertices.

3.11.3 Decompression Correctness Verification

As a lossless compression implementation, it's very important to ensure the decompression produces a lossless copy of compressed data. In our experiment, we use the verification method described in section 3.10 to verify the mesh equivalence.

All our test meshes' decompressions pass the test.

Chapter 4

Summary and Conclusions

4.1 Summary

This thesis has provided an in-depth study on Edgebreaker compression algorithm for fossil models. By extending the simple Edgebreaker algorithms, the implementation can efficiently support holes and handles, non-manifold vertices and multiple faces. A test framework for measuring whether meshes are identical after reordering is also provided for verification purposes.

4.2 Future Work

There are a few areas where further enhancement could take place:

1. Lossless color compression is possible to be added in this framework. The Edgebreaker traversal visits new vertices mostly by adjacency. If the color on the mesh is caused by light illumination, even a simple delta predictor can losslessly reduce color info's storage cost sufficiently.

2. Error-resistant decompression is a good extension to the Edgebreaker algorithm. Each step in Edgebreaker traversal is bounded by previous steps. In case of error recovery, the data may offset a lot even if just a few operators in the record have been compromised. An easy solution is to add CRC shell on top of the compressed file. More advanced solution can be developed by adding inter-frame in the compressed sequence.

3. Alternative data structure, half-edge data structure is a very flexible data structure for mesh manipulation. This flexibility comes with the high cost of memory consumption. For lossless mesh compression, the need for mesh manipulation is very small. Therefore, alternative data structure should be used to reduce memory consumption.

4.3 Conclusion

This thesis presents a working application for fossil model compression. The application is very practical. It works for models with various geometrical properties, including holes, handles, non-manifold vertices and multiple surfaces. It also runs very fast and can compress all our test models within 14 seconds. It significantly out-performs the general-purpose compressor Zip. It has a compression ratio upper bound in each of 3 mesh test categories with respect to the vertex property. Combing the application and Zip can further improve the compression ratio, which takes advantage on both Edgebreaker compression on triangle data and Zip compression on vertex data.

There are also potential to further improve this application. Lossless mesh color

compression should be an easy-to-implement extension. Error-resistant decompression can enable error recovery in the decoder. Alternative data structure can reduce the memory consumption.

Besides fossil model compression, the application could also be extended to use in other areas where mesh size is a general concern. For example, it could be used to compress landscape meshes in geographic information system.

Bibliography

[CC98] Touma C. and Gotsman C. Triangle mesh compression. In *Proceedings of Graphics Interface*, 1998.

[Cho97] Mike M. Chow. Optimized geometry compression for real-time rendering. In *Proceedings of the 8th conference on Visualization*, pages 347 – 354. IEEE Computer Society Press, 1997.

[Dee95] Michael Deering. Geometry compression. In *SIGGRAPH '95 Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20. ACM Press, 1995.

[Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098 – 1101, September 1952.

[IG98] Martin Isenburg and Stefan Gumhold. Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics*, July 1998.

[IS00] Martin Isenburg and Jack Snoeyink. Spirale reversi: Reverse decoding of the edgebreaker encoding. In *Canadian Conference on Computational Geometry*, pages 247–256, 2000.

[JLYT04] Bin-Shyan JONG, Tsong-Wuu LIN, Wen-Hao YANG, and Juin-Ling TSENG. Improved edge-based compression for the connectivity of 3d models. *IEICE TRANSACTIONS on Information and Systems*, E87-D(12):2845–2854, December 2004.

[LLRL04] Thomas Lewiner, Hlio Lopes, Jarek Rossignac, and T. Lewiner H. Lopes. Efficient edgebreaker for surfaces of arbitrary topology. In *in Proceedings of 17th Brazilian Symposium on Computer Graphics and Image Processing*, pages 218–225. IEEE, 2004.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 131–144. ACM Press, 2000.

[LRS+02] Helio Lopes, Jarek Rossignac, Alla Safanova, Andrzej Szymczak, and Geovan Tavares. Edgebreaker: A simple compression for surfaces with handles. In *In Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 289–296. ACM Press, 2002.

[Ros99] Jarek Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47 – 61, January 1999.

[RS99] Jarek Rossignac and Andrzej Szymczak. Wrap&zip decompression of the connectivity of triangle meshes compressed with edgebreaker. *Computational Geometry: Theory and Applications*, 14(1-3):119 – 135, November 1999.

[SKR01] Andrzej Szymczak, Davis King, and Jarek Rossignac. An edgebreaker-based efficient compression scheme for regular meshes. *Computational Geometry: Theory and Applications*, 20(1-2):53 – 68, October 2001.

[TR98] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84 – 115, April 1998.

[Wel84] T.A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8 – 19, June 1984.

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520 – 540, June 1987.