Edgebreaker Based Triangle Mesh-Coding Method

by

Yue Tang
B.Sc., Queen Mary University of London, 2013
B.Sc., Beijing University of Posts and Telecommunications, 2013

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

Edgebreaker Based Triangle Mesh-Coding Method

by

Yue Tang
B.Sc., Queen Mary University of London, 2013
B.Sc., Beijing University of Posts and Telecommunications, 2013

ABSTRACT

The Edgebreaker triangle mesh-coding method is presented along with a software implementation of the method developed by the author. The software consists of two programs. The first program performs the mesh compression, and the second program performs the mesh decompression. Various aspects of the method's performance are studied through experiments, such as coding efficiency and the time and memory complexity. In terms of coding efficiency, our Edgebreaker method outperform the gzip text-based compression technique on an average 4.19 times.

Contents

List of Tables

List of Figures

List of Algorithms

ACKNOWLEDGEMENTS

I would like to thank:

My supervisor Dr. Michael Adams. Thank you for your guidance and encouragement throughout my graduate study. Thank you for always being so nice and patient to me. I truly appreciate all the time and advice you gave me throughout my graduate study at the University of Victoria.

Professor Craig Gotsman from Technion - Israel Institute of Technology. Thank y-ou for sending me the triangle meshes that were used in your previous research. Without your generous help, I could not have produced such good comparison with other mesh-coding methods in my project.

My supervisory committee member Dr. Wu-Sheng Lu. Thank you for being on my supervisory committee and spending time reviewing my report. Thank you for delivering those great courses such as Digital Signal Processing and Engineering Optimization.

Ms. Gillian Saunders from the Centre for Academic Communication. Thank y-ou for suggesting me to participate in the Learning Plan and helping me improve my academic writing skills. Your hard work and dedication to teaching have motivated me in my studies. It has been such a pleasure working with you.

My friends. I would like to express my sincere gratitude to Dan Han, Yue Fang, Xiao Feng, Xiao Ma, and all other friends. Thank you for being my friends and the three years in Victoria is so wonderful and unforgettable.

My dearest mother Quan Tang. Thank you for being such a wonderful parent. Thank you for your endless love, support and encouragement. Without you I would never experience so much happiness and joy in my life.

DEDICATION

To my family.

Chapter 1

Introduction

1.1 3-D Triangle Mesh Compression

In recent years, three-dimensional (3-D) animation, modeling, and special visual effects have been widely used and had a significant impact on a number of markets. These applications also play an important role in a variety of disciplines such as computer science, engineering, planetary science, medicine, and architecture. The polygon mesh, a group of polygons stitched together to represent the surface of 3-D objects, is considered to be the most popular representation of 3-D models. More specifically, triangle meshes with all their polygons as triangles, are favored by most graphic applications. In order to help readers better understand triangle meshes, we include several examples in Figure 1.1. The triangle meshes in Figures 1.1(a), (b), and (c) represent the surface of a sphere, a parabolic cylinder, and the surface of a twisted torus, respectively.

Although meshes shown in Figure 1.1 only contain hundreds or thousands of vertices, the mesh datasets used in real world applications can be huge, containing billions of polygons. In many applications that use meshes, the model must often be very accurate. In order

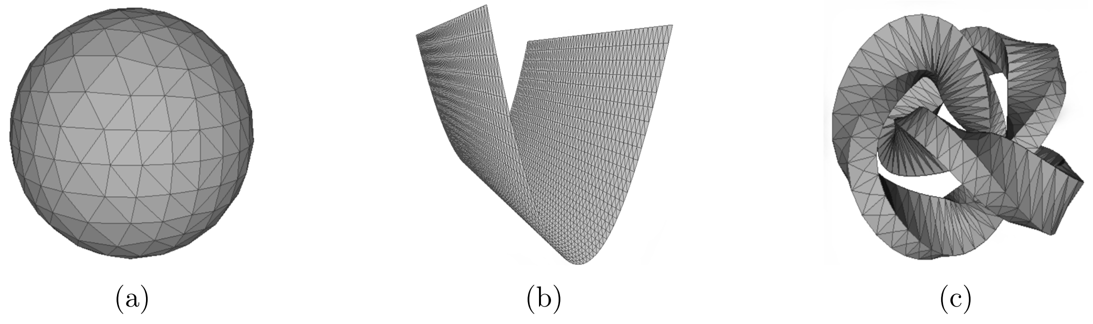

(a) (b) (c)

Figure 1.1: Examples of triangle meshes. (a) Sphere. (b) Parabolic cylinder. (c) Twisted torus.

to include more details in the mesh and achieve a higher resolution, the amount of storage space required by the meshes has rapidly grown. Moreover, many mesh applications require high-speed data transmission through the Internet, or remote access to the mesh datasets. This has led to the requirement of efficient representation of the mesh models, which has given rise to the development of various mesh-compression methods.

From the mesh examples in Figure 1.1, we know that triangle meshes consist of vertices, edges, polygon facets, and their incidence relationships. These incidence relationships are called the connectivity information. When compressing a triangle mesh, some methods encode the connectivity information first. This kind of mesh compression method is said to be connectivity driven. The main contribution of this project is the development of a software implementation of one of the well-known connectivity-driven mesh-coding methods, namely the Edgebreaker method proposed by Rossignac in 1999 [1].

1.2 Historical Perspective

In order to efficiently represent the raw data of triangle meshes, numerous connectivity-driven mesh-coding methods have been proposed [1–13]. Based on how the connectivity information is coded, the most popular methods can be classified into four categories: the triangle-strip approach, the spanning-tree approach, the valence-driven approach, and the triangle-traversal approach.

The main idea of the triangle-strip approach is to split the 3-D triangle mesh into long triangle strips, and then encode these strips. A triangle strip is a sequence of vertices, where each vertex is combined with the previous two vertices in the strip to form a new triangle. In Figure 1.2, we give an example of a tetrahedral mesh and its corresponding triangle strip. Figure 1.2(a) shows the tetrahedral mesh, and Figure 1.2(b) shows the corresponding triangle strip. The compression process of the triangle-strip approach is as follows. The first triangle is coded by its three vertices. After the first triangle is successfully coded, a new vertex index in the strip codes the connectivity of a new triangle. One of the pioneering methods using the triangle-strip approach was proposed by Deering in 1995 [2]. This method combines generalized triangle strips with a vertex buffer. The coded vertices are pushed to the buffer and can be referred to by their index in the buffer. In this way, the number of times each vertex is transmitted and processed by the graphics system is reduced. Other mesh-coding methods based on the triangle-strip approach can be found in [5, 6].

Next, we talk about the spanning-tree approach to mesh coding. A 3-D triangle mesh can be converted to a planar polygon by cutting along a selected set of edges. Therefore, the mesh's connectivity information can be represented by a planar graph and the corresponding

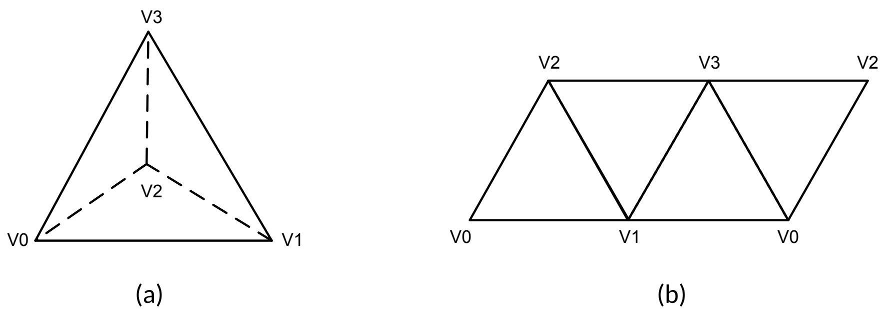


Figure 1.2: Tetrahedral mesh and its triangle strip. (a) Tetrahedral mesh. (b) Triangle strip of the tetrahedral mesh.

structure of cut edges. Based on this theory, in 1998, Taubin and Rossignac proposed the topological-surgery method [3]. The topological-surgery method encodes the original mesh by using two spanning trees: a vertex spanning tree and a triangle spanning tree. The vertex spanning tree is used to predict the vertex position. The difference between the predicted position and the actual position is then encoded by the method. Moreover, the triangle spanning tree is used to encode the connectivity information of the mesh. Some other mesh-coding methods employing the spanning-tree approach can be found in [12, 13].

Another approach to encoding triangle meshes is the valence-driven approach. For most triangle meshes in practice, the number of triangles is approximately twice the number of vertices. This observation has led to the valence-driven mesh-coding approach, which focuses on a vertex's local connectivity. The valence of a vertex is the number of edges incident on that vertex. The algorithm proposed by Touma and Gotsman in 1998 [4] is one of the well-known methods employing this approach. The main idea of this method is to expand the mesh's boundary. Starting from an arbitrary vertex of the initial triangle, the algorithm adds adjacent vertices. The valence of the inserted vertex is encoded, and the output of this method is a stream of vertex valences. Other mesh-coding methods using the valence-driven approach can be found in [7, 8].

The last mesh-coding approach to be discussed is the triangle-traversal approach. Some mesh-coding methods encode the connectivity of meshes by iteratively visiting their triangles. One advantage of this approach is simplicity. The Edgebreaker method, proposed in 1999 by Rossignac [1], is a pioneering work employing the triangle-traversal approach. At each step, the method encodes the relationship between the current triangle and the remaining mesh's boundary, and then traverses to an adjacent triangle. Some other mesh-coding methods based on the triangle-traversal approach can be found in [9–11].

1.3 Overview and Organization of Report

This project focuses on the 3-D triangle mesh coding. The project first studies the Edgebreaker mesh-coding method thoroughly. This is presented along with a software implementation of the method developed by the author. The software consists of two programs to perform the mesh encoding and decoding. Various aspects of the method's performance are measured through experiments. The reminder of this report is organized into five chapters and one appendix.

Chapter 2 introduces some essential background information to facilitate a better understanding of the work in this project. First, polygon and triangle meshes are introduced. This is then followed by the description of the halfedge data structure and the object-file format (OFF). Finally, some fundamentals of data compression are discussed.

Chapter 3 presents the Edgebreaker mesh-coding method implemented in this project. To begin, an overview of the mesh-coding method is presented. Then, we describe the compression and decompression methods in detail along with the parallelogram-prediction scheme used in the mesh-coding method.

Chapter 4 introduces the Edgebreaker mesh-coding software developed by the author. This chapter starts with an overview of the software, including a description of its constituent programs. This is then followed by instructions on how to build the software. Further, we present a detailed introduction to the programs' command-line interfaces. In addition, the file format used to store the compressed triangle-mesh data is described. Finally, several examples are provided to illustrate the use of the software.

Chapter 5 evaluates the performance of the Edgebreaker method. To begin, an overview of the test datasets is presented. Next, we discuss the coding efficiency. This is then followed by the time complexity analysis of the mesh-coding method through experiments. Further, the memory complexity is analyzed.

Chapter 6 summarizes the key points in this report and gives some suggestions for potential future work.

During the course of her work, the author found some typographical errors and missing technical details from the original Edgebreaker paper [1]. As supplemental information, corrections to these errors are provided in Appendix A.

Chapter 2

Background

2.1 Introduction

In this chapter, some necessary background information is provided to help readers better understand the work presented in this report. This includes polygon meshes, halfedge data structures, OFF format, and data compression. In what follows, we start by introducing the concept of a polygon mesh.

2.2 Polygon Mesh

A **polygon** is a closed two-dimensional (2-D) shape that is formed by a finite chain of straight line segments. Those line segments are called **edges**, and the points where two edges join together are called **vertices**. The simplest polygon is a triangle. A **polygon mesh** defines the surface of a polyhedral object usually in three-dimensions, and is a collection of vertices, edges, and facets. The **facets** are the polygons that are stitched together to form the mesh. A **triangle mesh** is a polygon mesh with all its facets being triangles. This project is concerned with the encoding and decoding of triangle meshes. In Figure 2.1, we give an example of a triangle mesh and its elements. Figure 2.1(a) shows a triangle mesh, and Figures 2.1(b), (c) and (d) show the vertices, edges, and facets of the mesh, respectively.

A polygon mesh consists of two types of information: geometry information and connectivity information. The geometry information specifies the position of each vertex in 3-D space. The connectivity information, or topological information, describes the incidence relationship among the mesh's vertices, edges, and facets. An edge that neighbors two facets is called an interior edge, while an edge that is incident on only one facet is known as a boundary edge. The **boundary** of the mesh is the union of all of the boundary edges. A polygon

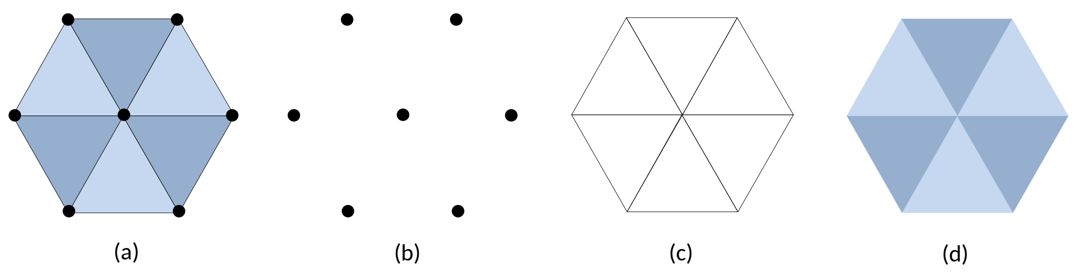


Figure 2.1: Triangle mesh and its elements. (a) Hexagon mesh. (b) to (d) Vertices, edges, and facets of the mesh, respectively.

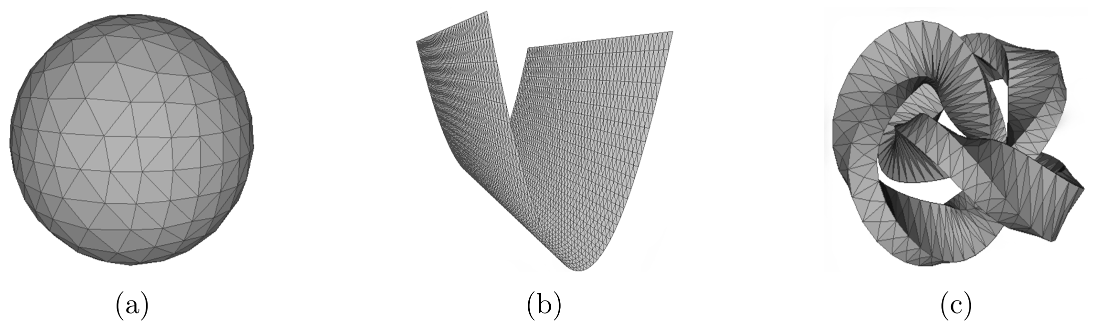


Figure 2.2: Example of closed meshes and a mesh with boundary. (a) and (c) Closed meshes. (b) A mesh with boundary.

mesh is said to be **closed** if it does not have any boundary edges. To better illustrate the concept presented above, three triangle meshes are shown in Figure 2.2. The triangle meshes in Figures 2.2(a) and (c) are closed meshes, while the mesh in Figure 2.2(b) is a mesh with boundary.

In order to define a manifold, we need to first introduce the concept of homeomorphism. A **homeomorphism** is essentially an elastic deformation. Elastic deformation means one shape can be transformed into another by stretching, compressing, bending, and twisting, but not by cutting, tearing, splitting, or joining the original shape. If one shape can be elastically deformed into another shape, then these two shapes are said to be **homeomorphic**. To better illustrate the concept of homeomorphism, we provide an example in Figure 2.3. Figures 2.3(a) and (b) show the surfaces of the coffee cup and the donut, respectively. Since the coffee cup can be transformed into the donut (or vice versa) by an elastic deformation, the surfaces of the coffee cup and the donut are homeomorphic.

If all vertices of a mesh have a neighborhood that is homeomorphic to a disc or half-disc, the mesh is known as a **manifold**. Otherwise, the mesh is a non-manifold. Examples of manifold and non-manifold meshes can be found in Figure 2.4. The meshes in Figures 2.4(a) and (b) are closed manifolds, the mesh in Figure 2.4(c) is a manifold with boundary, and the mesh in Figure 2.4(d) is non-manifold.

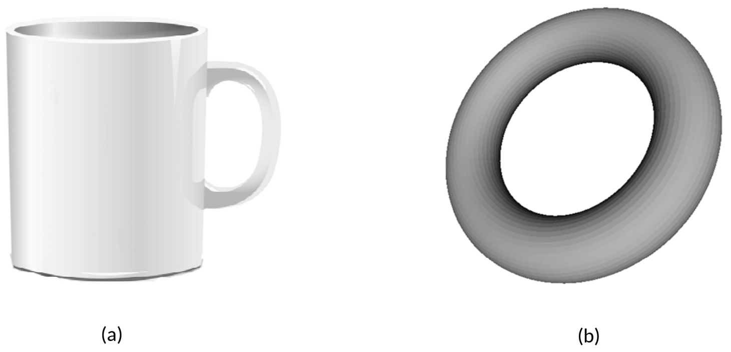


Figure 2.3: Example of homeomorphism. (a) Coffee cup. (b) Donut.

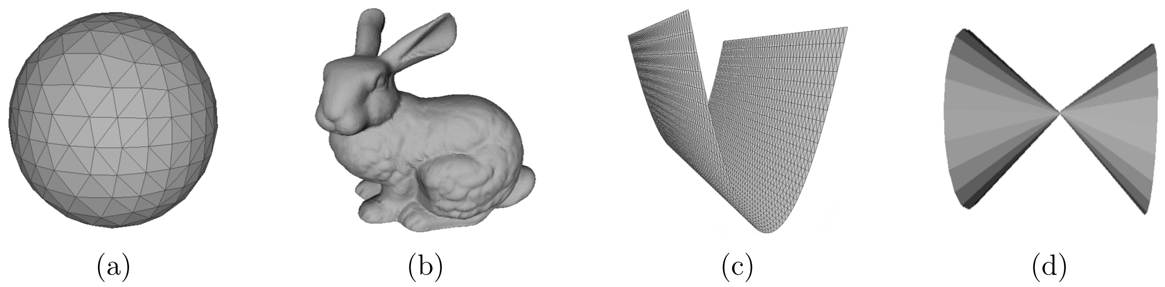


Figure 2.4: Examples of manifolds and non-manifolds. (a) and (b) Closed manifolds. (c) A manifold with boundary. (d) A non-manifold.

A manifold mesh is said to be **orientable** if one can specify a consistent orientation (clockwise or counterclockwise) for all closed paths in the manifold. Figure 2.5 illustrates the concept of orientability. Figure 2.5(a) is an orientable surface, while Figure 2.5(b) is a non-orientable surface.

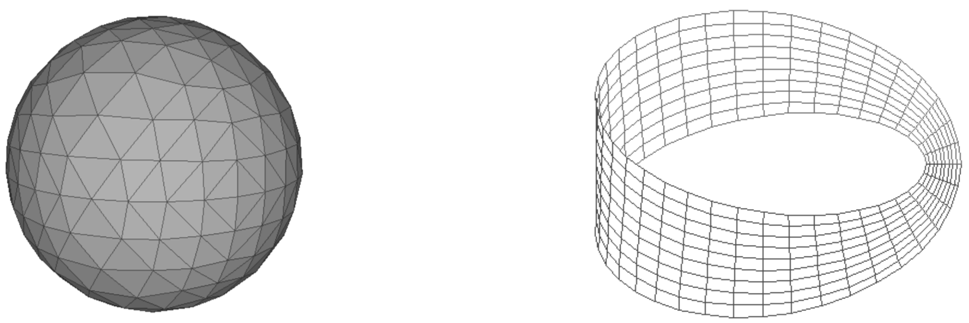
The **genus** of a closed orientable manifold is defined as the number of handles [14]. More generally, the mesh's genus g can be calculated as

$$g = 1 - \frac{1}{2}b - \frac{1}{2}(V - E + F), \tag{2.1}$$

where V, E, F, and b are the numbers of vertices, edges, facets, and bounding loops of the mesh, respectively. Examples of meshes with different genus are shown in Figure 2.6. The sphere in Figure 2.6(a) has genus zero. The torus in Figure 2.6(b) contains one handle, so its genus is one. The mesh in Figure 2.6(c) has nine handles, so its genus is nine.

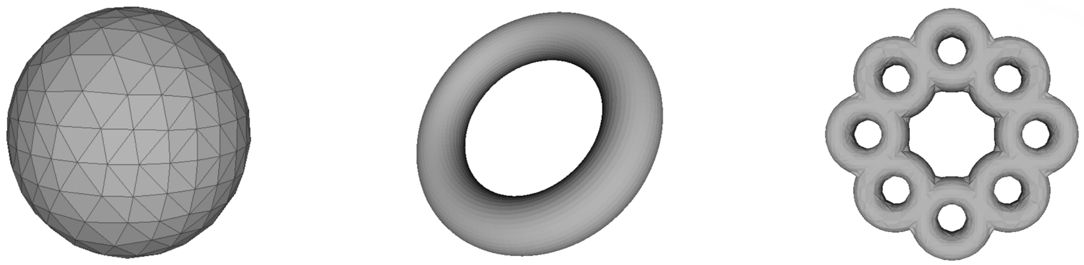
2.3 Halfedge Data Structure

Having introduced polygon meshes, we now consider how such meshes are represented in memory. There are lots of way to represent meshes in memory. The **halfedge data struc-**



(a) (b)

Figure 2.5: Examples of orientable and non-orientable surfaces. (a) A sphere, which is orientable. (c) A Mobius strip, which is not orientable.



(a) (b) (c)

Figure 2.6: Triangle meshes with different genus. (a) Sphere, which has genus 0. (b) Torus, which has genus 1. (c) Multiple torus, which has genus 9.

```
struct Halfedge {
  Vertex* vertex;       // The incident vertex
  Halfedge* prev;       // The previous halfedge around the facet
  Halfedge* next;       // The next halfedge around the facet
  Halfedge* opp;        // The opposite halfedge
  Facet* facet;         // The incident facet
};
```

Figure 2.7: Definition of the halfedge data structure.

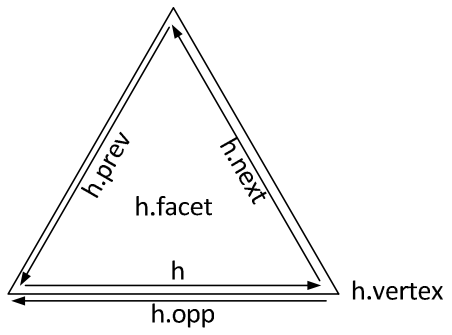


Figure 2.8: Pictorial view of the halfedge data structure.

ture is one of the well-known representations. Each edge in mesh is represented as a pair of directed edges that are oriented in opposite directions. Each of these directed edges is called a halfedge. The pseudocode of the halfedge data structure is shown in Figure 2.7, and the pictorial view of the halfedge data structure can be found in Figure 2.8. As can be seen from Figure 2.8, the end vertex is the incident vertex of the halfedge, and the left side facet is the incident facet of the halfedge. As shown in Figure 2.7, each halfedge stores the pointers to its incident vertex, the previous and next halfedges around the facet, the opposite halfedge, and the incident facet.

The halfedge data structure has two significant benefits: 1) since halfedges are oriented, this orientation property can be used to simplify algorithms that must navigate around meshes; and 2) a halfedge can also be used to identify a particular vertex and facet in the mesh. Since the halfedge data structure has these two obvious advantages, it is heavily used in practice. For example, the well known Computational Geometry Algorithms Library (CGAL) [15] utilizes this data structure.

As has been noted, the orientation property of halfedge can be used to simplify algorithms that must navigate around meshes. In the following example, we illustrate such ideas. Figure 2.9 shows the current triangle `cur.facet` and its two adjacent triangles `left_tri.facet` and `right_tri.facet`. Suppose that the user wants to move from the halfedge `cur` (which points to the current triangle `cur.facet`) to the halfedge `left_tri`

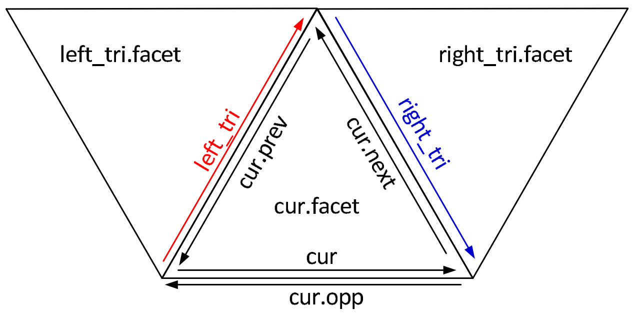


Figure 2.9: Pictorial view of halfedge data structures for a triangle and its two adjacent triangles.

(which points to the left adjacent triangle `left_tri.facet`). This can be achieved by code like:

```
left_tri = cur.prev.opp;
```

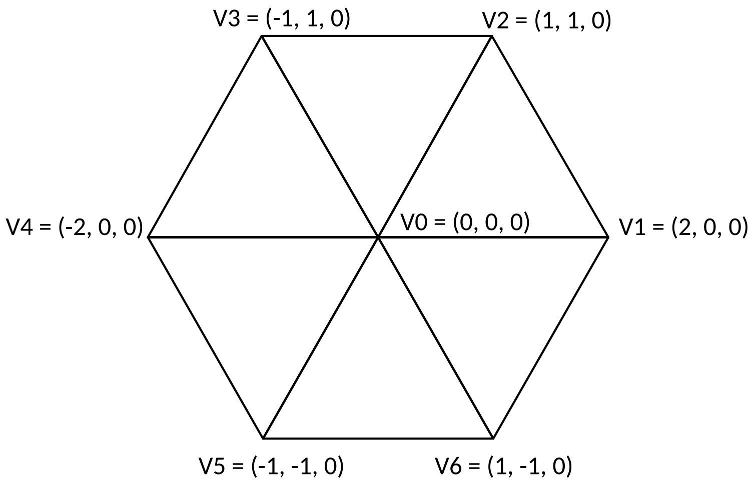
Suppose that the user wants to move from the halfedge `cur` (which points to the current triangle `cur.facet`) to the halfedge `right_tri` (which points to the right adjacent triangle `right_tri.facet`). This can be achieved by code like:

```
left_tri = cur.next.opp;
```

2.4 Object-File Format

The **object-file format** (OFF) is a popular way for storing the 3-D polygon meshes on disk. Data in the OFF format consists of: a header, and information for the vertices, facets, and edges in order. The edge data is optional.

The header contains two parts. The first is the file signature "OFF". The second is the numbers of vertices, facets, and edges in the mesh. If the number of edges equals zero, it means the edge data is omitted. The mesh's vertices are listed after the header part. For each vertex, its three coordinates are listed. The facet data is presented after the vertex list. For each facet, the number of vertices is specified first. This is then followed by the indices of the facet's vertices in the previous vertex list. The vertices in the vertex list are indexed from zero. If the edge data is included, its data is presented after the facet data. An OFF file example with respect to a simple triangle mesh is given in Figure 2.10. Figure 2.10(a) shows the mesh, and Figure 2.10(b) gives its corresponding OFF data. The edge data is omitted in this example.



(a)

```
OFF
7    6    0
 0       0    0
 2       0    0
 1       1    0
-1       1    0
-2       0    0
-1      -1    0
 1      -1    0
3    0    1    2
3    0    2    3
3    0    3    4
3    0    4    5
3    0    5    6
3    0    6    1
```

(b)

Figure 2.10: An OFF file example. (a) The hexagon mesh. (b) The corresponding OFF data for the hexagon mesh. The edge information is omitted in the example.

2.5 Data Compression for Meshes

As explained earlier, the raw data of the 3-D triangle mesh usually needs to be compressed in practice. Data compression is a technique that reduces the number of bits needed to store or transmit the original information. The coding can be divided into two types: lossless and lossy. If information is lost during the coding process, the coder is said to be lossy; otherwise, it is said to be lossless. Quantization is a technique to remove less important information from the original data. In the case of lossy coding, quantization can help achieve a more compact representation. Therefore, the decoded data is only an approximation to the original. In the Edgebreaker mesh-coding method, an arithmetic coder is employed, which is a type of lossless coding technique. In what follows, we present the details of the arithmetic coding.

2.5.1 Arithmetic Coding

Arithmetic coding is a widely-used lossless coding scheme. It represents the entire message as a number in the interval $[0, 1)$. As the source message becomes longer, the interval needed to represent the message becomes smaller, and the number of bits needed to specify that interval grows [16]. The binary arithmetic coding uses only two symbols (i.e., 0 and 1) in the given model. To encode a message using a binary arithmetic coder, the initial range is specified as $[0, 1)$. As each symbol in the source message is coded, the interval is narrowed

Table 2.1: Probability distribution for the symbols $\{0, 1\}$

Symbol	Probability	Interval
0	0.6	$[0.0, 0.6)$
1	0.4	$[0.6, 1.0)$

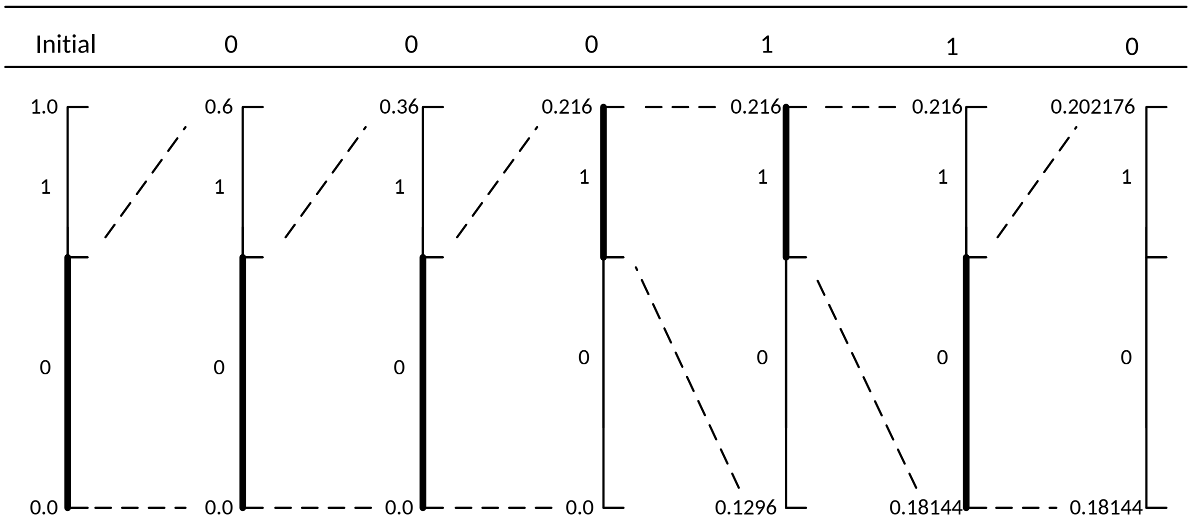


Figure 2.11: Graphic representation of the arithmetic encoding process.

in accordance with the symbol and its probability. If the probability distribution is selected based on the contextual information, rather than always using the same set of probabilities, the arithmetic coder is said to be **context based**. Moreover, the coder is **adaptive** if it updates the probability for each symbol during the coding process.

In what follows, we present a short example illustrating the (binary) arithmetic coding process for a particular message chosen from a binary alphabet {0, 1}, namely the message "000110". The probability distribution for the symbols $\{0, 1\}$ is given in Table 2.1.

The encoding process is shown in Figure 2.11. Initially, the interval is set to $[0.0, 1.0)$. The first symbol narrows the interval to $[0.0, 0.6)$, which corresponds to the interval of symbol 0 in the initial range. Similarly, the second and third symbols narrow the interval to $[0.0, 0.216)$. When the encoder sees the fourth symbol (i.e., symbol 1) from the source message, the new interval $[0.1296, 0.216)$ is obtained. Proceeding in this way, the final interval $[0.18144, 0.202176)$ for the given source message is generated. Since the decoder does not need to know both ends of the final interval, a single number within the range is sufficient to decode the message. Therefore, the lower bound of the final interval (i.e., the number 0.18144) is transmitted to the decoder side. In addition to the transmitted value, the decoder also needs to know the number of symbols encoded.

Now we consider the case of decoding for the above example. The decoding process is

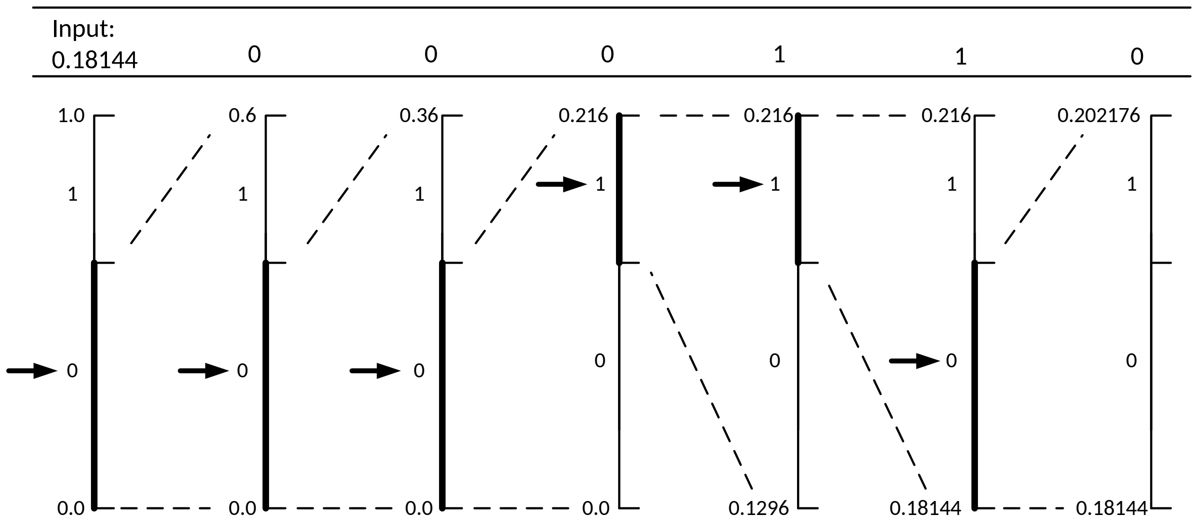


Figure 2.12: Graphic representation of the arithmetic decoding process.

shown in Figure 2.12. Initially, the interval for decoding is set to $[0.0, 1.0)$. To begin, the decoder receives the transmitted value 0.18144 and the number of symbols that are coded by the encoder side. The number 0.18144 is located in the range $[0.0, 0.6)$, which corresponds to the interval that is allocated to the symbol 0 in the initial range. Hence, the first symbol decoded is 0. Based on the transmitted value 0.18144, as seen from Figure 2.12, the next two symbols decoded are also 0. The new interval obtained after the decoder decodes the third symbol is $[0.0, 0.216)$. Proceeding in this way, the decoder identifies the whole message "000110" and generates the final interval $[0.18144, 0.202176)$. Six symbols are coded by the encoder side. Therefore, after the sixth symbol is successfully deciphered, the decoding process terminates.

A binary arithmetic coder can only code binary symbols. If a non-binary symbol needs to be coded with the binary coder, the symbol must first be binarized before the coding process. **Binarization** is the process of converting a non-binary symbol to a sequence binary ones. An example is the UI-binarization scheme in [17], which is also used in the Edgebreaker mesh-coding method as seen later.

2.5.2 Quantization

As explained earlier, quantization can be used in lossy coding to discard less important information. Quantization helps reduce the number of bits needed to represent the information. Therefore, a quantizer can be described by two rules: the classification rule and the reconstruction rule. The **classification rule** maps a real number x to the integer quantization index k, and the **reconstruction rule** maps the quantization index k to the reconstruction value y.

A basic type of quantizer is the **uniform scalar quantization**, which rounds a real number x to the nearest integer $Q(x)$. The classification rule of a typical midtread uniform scalar quantizer is given by

$$k = (\operatorname{sgn} x) \left\lfloor \frac{|x|}{\Delta} + \frac{1}{2} \right\rfloor, \tag{2.2}$$

where Δ denotes the quantization step size. If $\Delta = 1$, the quantizer rounds x to the nearest integer. The reconstruction rule for this quantizer is simply

$$y = Q(x) = k \cdot \Delta. \tag{2.3}$$

Chapter 3

The Edgebreaker Mesh-Coding Method

3.1 Introduction

The connectivity-driven mesh-coding method of interest herein is the Edgebreaker method. We begin this chapter by introducing the details of the encoding method. This is then followed by a detailed description of the decoding process.

3.2 Encoding Method

To begin, we introduce the encoding method in general terms. The basic idea of the Edgebreaker method is triangle traversal. At each step, the encoding method produces an op-code to describe the topological relationship between the current triangle and the remaining mesh's boundary, and the position of any newly encountered vertex is also coded. Since all vertex coordinates are real numbers and the binary arithmetic coder is used, the first step is to quantize all vertex coordinates to produce integer quantizer indices. The method visits each triangle in the mesh until all triangles are visited. For each visited triangle, the following steps are performed. First, the encoder produces an op-code describing the triangle's type and adds this type to the op-code sequence. Second, the encoder predicts the position of any newly encountered vertex by using the parallelogram-prediction scheme. The difference between the predicted and actual positions of the vertex (i.e., the prediction residual) is then binarized (using the UI-binarization scheme in [17]) and encoded by the arithmetic coder. Third, the encoder moves to a particular adjacent triangle. The pseudocode for mesh encoding can be found in Algorithm 1.

Having introduced the Edgebreaker encoder in general terms, we now present the encoding method in detail. The main state information for the encoder consists of the following:

Algorithm 1 Mesh encoding process.

```
1: Quantize all mesh vertices to produce integer quantization indices.
2: while Not all the triangles are processed by the encoding method. do
3:     Find the current triangle's type and add it to the op-code sequence.
4:     Predict the position of newly encountered vertex in the current triangle.
5:     Calculate, binarize, and encode the prediction residuals.
6:     Move to a particular adjacent triangle.
7: end while
```

- The input triangle mesh, which is represented by the halfedge data structure.
- The op-code sequence, which is used to store the triangle types.
- The prediction residuals sequence, which is used to store the coded vertices.
- The halfedge stack, which is used to store the representative halfedges of the boundaries that are not currently being processed.
- The M table, which is used to store the information related to holes.
- The M' table, which is used to store the information related to handles.
- The offset table, which is only used for meshes that contain handles.
- The S counter, which is used to count how many S-type triangles have been encountered so far.

Now, we introduce the data structures used to represent some of the preceding state. As explained earlier, the halfedge data structure is used by the encoding method. In this context, some extra information is added to the data structure as originally introduced in Section 2.3 (on page 7). The pseudocode of the extended halfedge data structure is shown in Figure 3.1, and the pictorial view of the extended halfedge data structure can be found in Figure 3.2. As seen from the pseudocode in Figure 3.1, an integer mark and two extra pointers are added to the data structure. The mark `mark` is used to indicate the halfedge's category. The two extra pointers `prev_bor` and `next_bor` are used to find the adjacent halfedges in the bounding loop. These two pointers refer to the current halfedge's previous and next halfedges along the bounding loop.

Some extra information is also added to the vertex data structure. The pseudocode for the vertex data structure is shown in Figure 3.3. As seen from Figure 3.3, an integer mark and a boolean type flag have been added to the data structure. The mark `mark` is used to distinguish the vertex's category, and the flag `flag` indicates whether the vertex is coded or not. The vertex flag is used as follows. At the beginning of the encoding process, the

```
struct Halfedge {
  Vertex* vertex;      // The incident vertex
  Halfedge* prev;      // The previous halfedge around the facet
  Halfedge* next;      // The next halfedge around the facet
  Halfedge* opp;       // The opposite halfedge
  Halfedge* prev_bor;  // The previous halfedge around the border
  Halfedge* next_bor;  // The next halfedge around the border
  Facet* facet;        // The incident facet
  int mark;            // The halfedge mark
};
```

Figure 3.1: Definition of the extend halfedge data structure.

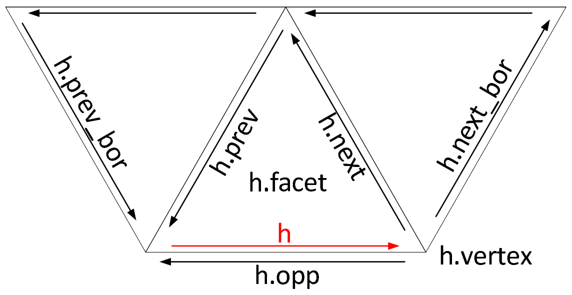


Figure 3.2: Pictorial view of the extended halfedge data structure.

encoder sets all the vertex flags to false. This means none of the vertices have been coded. During the encoding process, when a vertex is encountered, the encoder first checks its flag. If the vertex flag equals false, the encoder encodes the vertex and sets the vertex flag to true.

Having presented the major data structures, we now introduce some terminology and notation used by the encoding process. The *active gate* is a special halfedge of the mesh, and the initial active gate is the starting point of the encoding process. At each step, the topological relationship of the triangle incident upon the active gate is detected by the encoder. For a closed mesh, an arbitrary halfedge can be chosen as the initial active gate.

```
struct Vertex {
  double x_coordinate; // The x coordinate
  double y_coordinate; // The y coordinate
  double z_coordinate; // The z coordinate
  int mark;            // The vertex mark
  bool flag;           // The vertex flag
  Halfedge halfedge;   // The corresponding halfedge
};
```

Figure 3.3: Definition of the vertex data structure in encoder.

For a mesh with boundary, an arbitrary halfedge of boundary edge opposite to boundary can be chosen.

During the encoding process, the triangle mesh may split into two separate parts: the left and right submeshes. The bounding loop is the union of all the halfedges that are located on the remaining mesh's boundary. The bounding loop is active if it belongs to the submesh that is currently being processed. Otherwise, the loop is inactive. An *inactive gate* is a specific halfedge located on an inactive bounding loop. For all the inactive bounding loops, the inactive gate from each loop is temporarily stored in the halfedge stack. The following notation is used in our explanation of encoding method:

- The symbol `g` denotes the active gate.
- The symbol `B` denotes the active bounding loop. Note that `B` contains `g`.
- The symbol `h.e` denotes the ending vertex of the halfedge `h`.
- The symbol `h.s` denotes the starting vertex of the halfedge `h`.
- The symbol `h.v` denotes the vertex which locates opposite of the halfedge `h` in the triangle incident upon `h`.
- The symbol `h.m` denotes the halfedge mark of the halfedge `h`.
- The symbol `h.v.m` denotes the vertex mark of the vertex `h.v`.
- The symbol `h.p` denotes the previous halfedge of the halfedge `h` in the triangle incident upon `h`.
- The symbol `h.n` denotes the next halfedge of the halfedge `h` in the triangle incident upon `h`.
- The symbol `h.o` denotes the opposite halfedge of the halfedge `h`.
- The symbol `h.P` denotes the previous halfedge of the halfedge `h` on `B`.
- The symbol `h.N` denotes the next halfedge of the halfedge `h` on `B`.

As has been noted, the connectivity-coding algorithm in the Edgebreaker method classifies the triangles into different types. The triangle type describes the topological relationship between the vertex opposite the active gate and the mesh's boundary. Seven different triangle types (C, L, R, E, S, M, and M') are identified by the Edgebreaker method. The pictorial view of these seven types can be found in Figure 3.4. Shortly, we will explain how to distinguish between these seven triangle types.

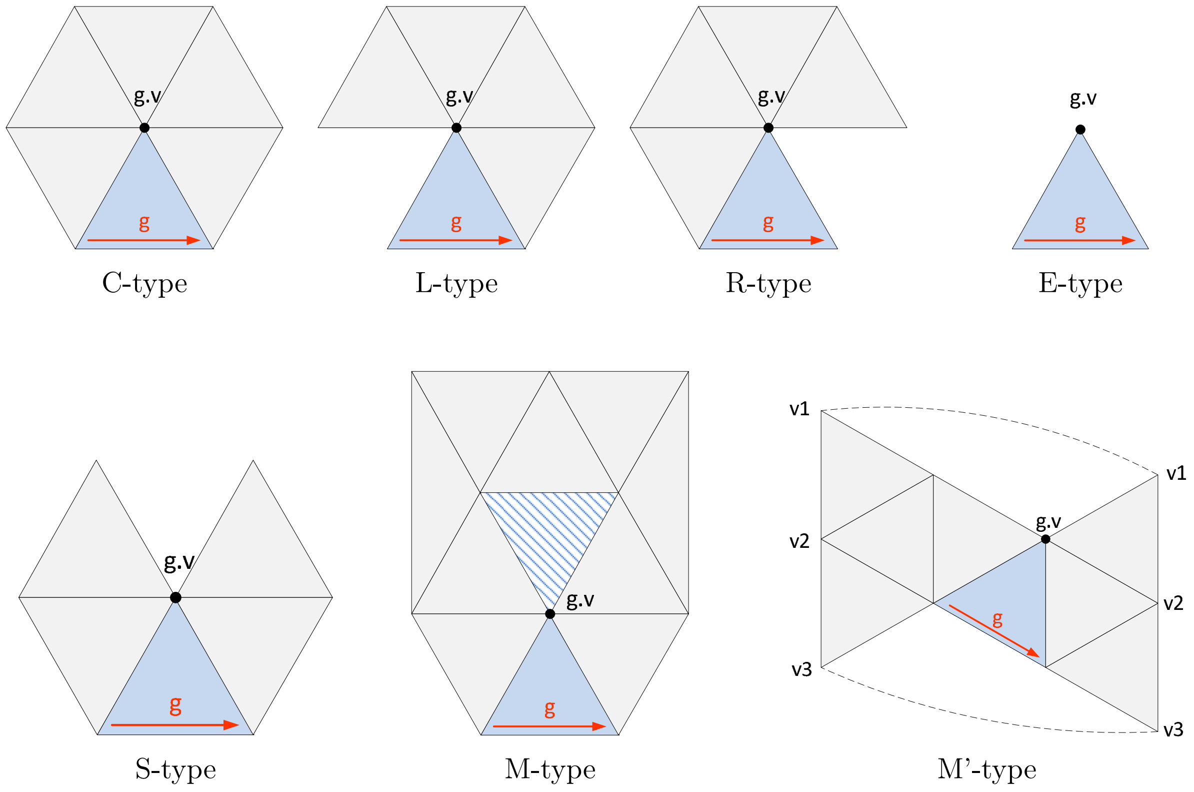


Figure 3.4: Seven triangle types.

Before proceeding further, we need to first introduce the vertex and halfedge marks and their initialization process. This digression is necessary due to the fact that vertex marks are used to distinguish triangle types. Four marks are used to indicate the vertex and halfedge categories.

- Zero. A mark value of zero indicates that the halfedge or vertex is located inside the mesh, or all of the processing for the halfedge is completed.
- One. A mark value of one indicates that the halfedge or vertex is located on the active bounding loop, or all of the processing for the vertex is completed.
- Two. A mark value of two indicates that the halfedge or vertex is located on a hole's boundary.
- Three. A mark value of three indicates that the halfedge or vertex is located on an inactive bounding loop.

Moreover, the mark initialization process is as follows. At the beginning of the encoding process, the encoder sets all the interior vertex and halfedge marks to zero. For all the vertices and halfedges located on mesh's boundary, their marks are initialized to one. For

Algorithm 2 Triangle type distinction.

```
if g.v.m = 0 then
  C-type triangle. {g.v is not belong to B.}
else if g.v.m = 2 then
  M-type triangle. {g.v belongs to hole's boundary.}
else
  if g.p = g.P then
    if g.n = g.N then
      E-type triangle. {g.v is immediately precedes and follows g.}
    else
      L-type triangle. {g.v is immediately precedes g.}
    end if
  else
    if g.n = g.N then
      R-type triangle. {g.v is immediately follows g.}
    else
      if g.v.m = 3 then
        M'-type triangle. {g.v is neither immediately precedes nor follows g.}
      else
        S-type triangle. {vg.v is neither immediately precedes nor follows g.}
      end if
    end if
  end if
end if
```

all the vertices and halfedges located on the holes' boundary, their marks are initialized to two.

With the help of vertex marks, the seven triangle types can be easily identified. If the vertex opposite the active gate's mark (i.e., `g.v.m`) equals zero, the triangle is C-type. If the vertex opposite the active gate's mark (i.e., `g.v.m`) is two, triangle is M-type. For all the other five types, the distinction between those types can be achieved by the relationships between `g.v` and `B`. Since the S-type and M'-type triangles share the same `g.v` and `B` relationship, they can be distinguished as follows. If the vertex opposite the active gate's mark (i.e., `g.v.m`) is three, the current triangle is M'-type. Otherwise, the triangle is S-type. The pseudocode for triangle type distinction is presented in Algorithm 2.

Having presented the triangle-type distinction rule, we can now introduce the encoding operations for each triangle type. The triangle types are introduced in the given order: C, L, R, E, S, M, and M'. For each type, we first introduce the purpose of the triangle type, and then present how the triangle type is processed.

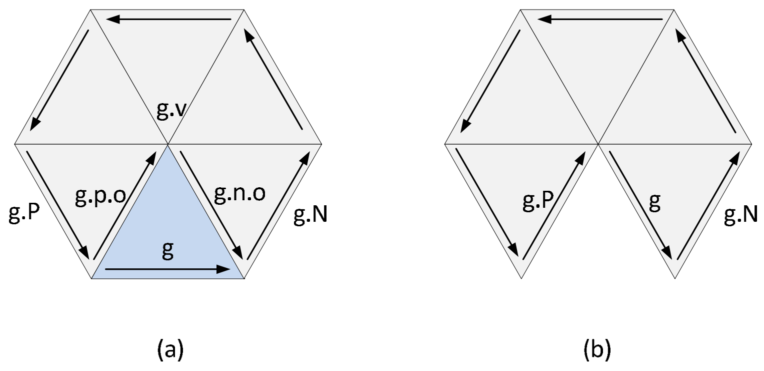


Figure 3.5: An example illustrating the C-type operation. (a) The initial mesh corresponds to a C-type triangle. (b) The resulting mesh obtained after the C-type operation.

3.2.1 C-type Triangle

The first type of triangle to be considered is C-type. For typical meshes, this is the most commonly occurring triangle type. A C-type triangle arises when the active gate `g` has an opposing vertex `g.v` that has not yet been encountered previously in the coding process (as determined by its mark value). An example of this situation is illustrated in Figure 3.5(a), where the facet contains the active gate `g` shown. During the processing of the C-type triangle, the active gate is moved to the halfedge `g.n.o`, and various marks are updated. Then, the incident facet of `g` (before `g` was moved) is deleted, and the bounding loop is updated accordingly. This yields the updated bounding loop with the mesh shown in Figure 3.5(b). The processing steps described above are shown in more detail in Algorithm 3, including exactly how the various halfedges and marks should be updated.

3.2.2 L-type Triangle

The second type of triangle to be considered is L-type. An L-type triangle arises when the active gate `g` has an opposing vertex `g.v` that belongs to `B` and immediately precedes `g` on `B`. An example of this situation is illustrated in Figure 3.6(a), where the facet contains the active gate `g` shown. As seen from Figure 3.6(a), an L-type triangle represents the local leftmost triangle. This means that the halfedges `g.p` and `g.P` are the same halfedge. During the processing of the L-type triangle, the active gate is moved to the halfedge `g.n.o`, and various marks are updated. Then, the incident facet of `g` (before `g` was moved) is deleted, and the bounding loop is updated accordingly. This yields the updated bounding loop with the mesh shown in Figure 3.6(b). The processing steps described above are shown in more detail in Algorithm 4, including exactly how the various halfedges and marks should be updated.

Algorithm 3 C-type encode operation.

```
1: {append C to op-code sequence.}
2: O_seq = O_seq|C;
3: {update marks.}
4: g.m = 0;
5: g.p.o.m = 1;
6: g.n.o.m = 1;
7: g.v.m = 1;
8: {connect the halfedges g.P and g.p.o.}
9: g.P.N = g.p.o;
10: g.p.o.P = g.P;
11: {connect the halfedges g.p.o and g.n.o.}
12: g.p.o.N = g.n.o;
13: g.n.o.P = g.p.o;
14: {connect the halfedges g.n.o and g.N.}
15: g.n.o.N = g.N;
16: g.N.P = g.n.o;
17: {update the active gate g.}
18: g = g.n.o;
```

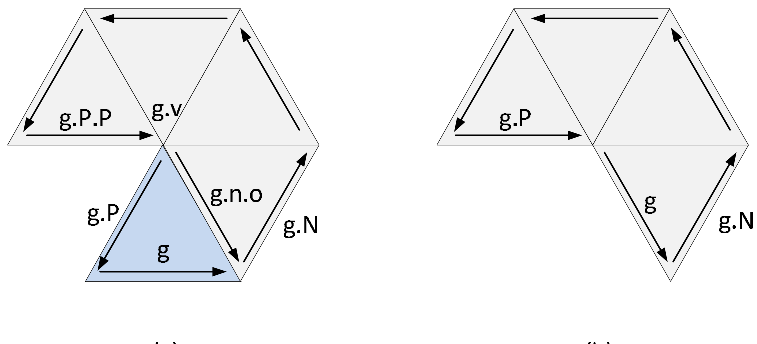


Figure 3.6: An example illustrating the L-type operation. (a) The initial mesh corresponds to an L-type triangle. (b) The resulting mesh obtained after the L-type operation.

Algorithm 4 L-type encoding operation.

```
1: {append L to op-code sequence.}
2: O_seq = O_seq|L;
3: {update marks.}
4: g.m = 0;
5: g.P.m = 0;
6: g.n.o.m = 1;
7: {connect the halfedges g.P.P and g.p.o.}
8: g.P.P.N = g.n.o;
9: g.n.o.P = g.P.P;
10: {connect the halfedges g.n.o and g.N.}
11: g.n.o.N = g.N;
12: g.N.P = g.n.o;
13: {update the active gate g}
14: g = g.n.o;
```

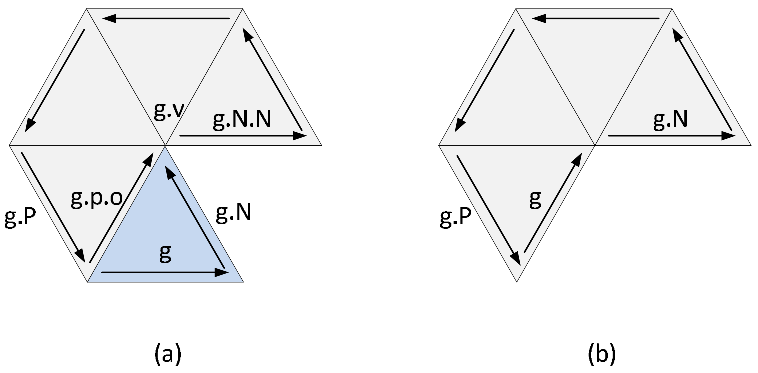


Figure 3.7: An example illustrating the R-type operation. (a) The initial mesh corresponds to an R-type triangle. (b) The resulting mesh obtained after the R-type operation.

3.2.3 R-type Triangle

The third type of triangle to be considered is R-type. An R-type triangle arises when the active gate g has an opposing vertex g.v that belongs to B and immediately follows g on B. An example of this situation is illustrated in Figure 3.7(a), where the facet contains the active gate g shown. As seen from Figure 3.7(a), an R-type triangle represents the local rightmost triangle. This means that the halfedges g.n and g.N are the same halfedge. During the processing of the R-type triangle, the active gate is moved to the halfedge g.p.o, and various marks are updated. Then, the incident facet of g (before g was moved) is deleted, and the bounding loop is updated accordingly. This yields the updated bounding loop with the mesh shown in Figure 3.7(b). The processing steps described above are shown in more detail in Algorithm 5, including exactly how the various halfedges and marks should be updated.

Algorithm 5 R-type encoding operation.

```
1: {append R to op-code sequence.}
2: O_seq = O_seq|R;
3: {update marks.}
4: g.m = 0;
5: g.N.m = 0;
6: g.p.o.m = 1;
7: {connect the halfedges g.p.o and g.N.N.}
8: g.N.N.P = g.p.o;
9: g.p.o.N = g.N.N;
10: {connect the halfedges g.P and g.p.o.}
11: g.P.N = g.p.o;
12: g.p.o.P = g.P;
13: {update the active gate g.}
14: g = g.p.o;
```

3.2.4 S-type Triangle

The next type of triangle to be considered is S-type. An S-type triangle arises when the active gate g has an opposing vertex g.v that belongs to B and neither immediately precedes nor follows g on B. An example of this situation is illustrated in Figure 3.8(a), where the facet contains the active gate g shown. During the processing of the S-type triangle, the current mesh is split into two separate parts: the left and right submeshes. As explained earlier, the halfedge stack is used to store the inactive gates. The halfedge g.p.o from the left submesh is pushed to the halfedge stack. Various marks on the left submesh's bounding loop are updated. The offset table is then updated by the S-type operation. Next, the active gate is moved to the halfedge g.n.o, and various marks are updated. Finally, the incident facet of g (before g was moved) is deleted, and the bounding loop is updated accordingly. This yields the updated bounding loop with the mesh shown in Figure 3.8(b). The processing steps described above are shown in more detail in Algorithm 6, including exactly how the various halfedges and marks should be updated. The encoder moves to the right submesh after processing the S-type triangle. The offset table related operation in step 39 will be explained later in Section 3.2.8.

3.2.5 E-type Triangle

The next type of triangle to be considered is E-type. An E-type triangle arises when the active gate g has an opposing vertex g.v that belongs to B and both immediately precedes and follows g on B. An example of this situation is illustrated in Figure 3.9, where the facet

Algorithm 6 S-type encoding operation.

```
1: {append S to op-code sequence.}
2: O_seq = O_seq|S;
3: {update marks.}
4: g.m = 0;
5: g.p.o.m = 1;
6: g.n.o.m = 1;
7: b = g.n; {initial candidate for the halfedge b.}
8: while b.m $\neq$ 1 do
9:   b = b.o.p; {turn around vertex g.v.}
10: end while
11: {connect the halfedges g.P and g.p.o.}
12: g.P.N = g.p.o;
13: g.p.o.P = g.P;
14: {connect the halfedges g.p.o and b.N.}
15: g.p.o.N = b.N;
16: b.N.P = g.p.o;
17: edge_stack.push(g.p.o); {update the halfedge stack.}
18: g_n = g.p.o.N; {initial candidate for the halfedge g_n.}
19: while g_n.m $\neq$ 3 do
20:   {update marks.}
21:   g_n.m = 3;
22:   g_n.e.m = 3;
23:   g_n = g_n.N; {move to next edge around left submesh.}
24: end while
25: {connect the halfedges b and g.n.o.}
26: b.N = g.n.o;
27: g.n.o.P = b;
28: {connect the halfedges g.n.o and g.N.}
29: g.n.o.N = g.N;
30: g.N.P = g.n.o;
31: offset = 1;
32: o_edge = g.n.o.N; {initial candidate for the halfedge o_edge.}
33: while o_edge $\neq$ g.n.o do
34:   offset = offset + 1;
35:   o_edge = o_edge.N; {move to next edge around right submesh.}
36: end while
37: offset = offset - 2; {calculate the offset value.}
38: s_count = s_count + 1; {update s_count. }
39: offset_table.add(offset, s_count); {update the offset table.}
40: g = g.n.o; {update the active gate g.}
```

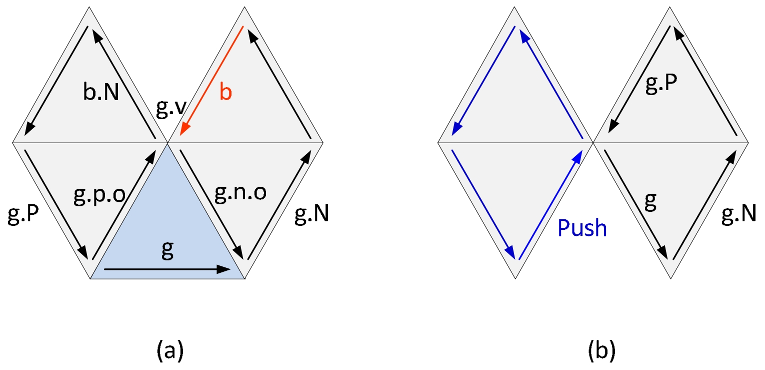


Figure 3.8: An example illustrating the S-type operation. (a) The initial mesh corresponds to an S-type triangle. (b) The resulting mesh obtained after the S-type operation.

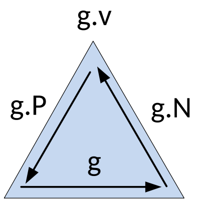


Figure 3.9: An example mesh corresponds to an E-type triangle.

contains the active gate `g` shown. As seen from Figure 3.9, E-type triangle is the last triangle in the current mesh or submesh. This means that the halfedges `g.p` and `g.P` are the same halfedge, and the halfedges `g.n` and `g.N` are the same halfedge. During the processing of the E-type triangle, various marks are updated. If the E-type triangle is the last triangle in the mesh, the entire encoding procedure is finished. Otherwise, a halfedge is popped from the halfedge stack and used as the active gate `g` for subsequent processing. The processing steps described above are shown in more detail in Algorithm 7, including exactly how the various halfedges and marks should be updated. The offset table related operation in step 7 will be explained later in Section 3.2.8.

3.2.6 M-type Triangle

The next type of triangle to be considered is M-type. An M-type triangle arises when the active gate `g` belongs to a hole's boundary (as determined by its mark value). An example of this situation is illustrated in Figure 3.10(a), where the facet contains the active gate `g` shown. During the processing of the M-type triangle, the hole's boundary is merged into the active bounding loop. As explained earlier, the M table is used to store the information related to holes. A pair of values is stored for each table entry. The first value is the number of S-type triangles encountered since the previous M-type triangle or since the beginning of the encoding process for the first M-type triangle. The second value is the number of vertices

Algorithm 7 E-type encoding operation.

```
{append E to op-code sequence.}
O_seq = O_seq|E;
{update marks.}
g.m = 0;
g.p.o.m = 0;
g.n.o.m = 0;
if size(offset_table) > handle_s then
  offset_table.pop_back(); {update the offset table.}
end if
if edge_stack $\neq \emptyset$ then
  g = edge_stack.pop(); {Update the halfedge stack.}
  b = g.N; {initial candidate for the halfedge b.}
  while b.m $\neq 1$ do
    {update marks.}
    b.m = 1;
    b.e.m = 1;
    b = b.N; {move the halfedge b.}
  end while
else
  Encoding process end.
end if
```

located on the hole's boundary. Next, the active gate is moved to the halfedge `g.n.o`, and various marks are updated. Then, the incident facet of `g` (before `g` was moved) is deleted, and the bounding loop is updated accordingly. This yields the updated bounding loop with the mesh shown in Figure 3.10(b). The processing steps described above are shown in more detail in Algorithm 8, including exactly how the various halfedges and marks should be updated.

3.2.7 M'-type Triangle

The last type of triangle to be considered is M'-type. An M'-type triangle arises when the active gate `g` has an opposing vertex `g.v` that belongs to an inactive bounding loop (as determined by its mark value). An example of this situation is illustrated in Figure 3.11(a), where the facet contains the active gate `g` shown. During the processing of the M'-type triangle, the inactive bounding loop (i.e., a bounding loop associated with one of the inactive gates in the halfedge stack) is merged into the active bounding loop. As explained earlier, the M' table is used to store the information related to handles. Three values are stored in each table entry. The first is the position of the halfedge (i.e., the inactive gate of the inactive

Algorithm 8 M-type encode operation.

```
{append M to op-code sequence.}
O_seq = O_seq|M;
{update marks.}
g.m = 0;
g.p.o.m = 1;
g.n.o.m = 1;
b = g.n; {initial candidate for the halfedge b.}
while b.m $\neq$ 2 do
  b = b.o.p; {turn around the vertex g.v.}
end while
len = 0; {initial hole length count.}
while b.m $\neq$ 1 do
  {update marks.}
  b.m = 1;
  b.e.m = 1;
  len = len + 1; {update the hole length count.}
  b = b.N {move to next edge around hole.}
end while
m_table.add(len, s_count); {update M table.}
{connect the halfedges g.P and g.p.o.}
g.P.N = g.p.o;
g.p.o.P = g.P;
{connect the halfedges g.p.o and b.N.}
g.p.o.N = b.N;
b.N.P = g.p.o;
{connect the halfedges b and g.n.o.}
b.N = g.n.o;
g.n.o.P = b;
{connect the halfedges g.n.o and g.N.}
g.n.o.N = g.N;
g.N.P = g.n.o;
{update the active gate g.}
g = g.n.o;
```

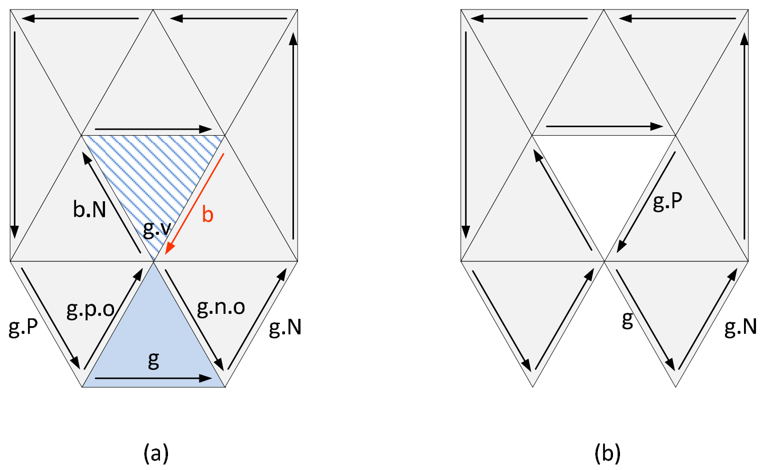


Figure 3.10: An example illustrating the M-type operation. (a) The initial mesh corresponds to an M-type triangle. (b) The resulting mesh obtained after the M-type operations.

loop that needs to be merged) in the halfedge stack. The second is the offset value for the M'-type triangle. The offset is the distance between the halfedge in the stack and the vertex opposite the active gate (i.e., `g.v`). The third is the number of S-type triangles encountered since the previous M'-type triangle or since the beginning of the encoding process for the first M'-type triangle. Next, the active gate is moved to the halfedge `g.n.o`, and various marks are updated. Then, the incident facet of `g` (before `g` was moved) is deleted, and the bounding loop is updated accordingly. This yields the updated bounding loop with the mesh shown in Figure 3.11(b). The detailed processing steps in the M'type operation are shown in Algorithm 9, including exactly how the various halfedges and marks should be updated. The offset table related operation in step 34 will be explained later in Section 3.2.8.

3.2.8 Remarks on Handles

In what follows, we introduce the additional coding complexity produced by the handles. Rossignac's original paper on Edgebreaker in [1] did not fully specify all details of how to process handles. Therefore, some ideas from Zhu's thesis in [18] were used to provide missing details.

As explained earlier, the S-type operation splits the mesh into the left and right submeshes. An example of this situation is illustrated in Figure 3.12(a), where the facet contains the active gate `g` shown an S-type triangle. At the encoder side, the two split points for the S-type operation are always the active gate's incident vertex and the vertex opposite the active gate (i.e., vertices V_1 and V_5 in the figure). After the split operation, the encoder generates the left and right submeshes, as shown in Figures 3.12(b) and (c), respectively. In the decoder, the method performs the same split operation when processing the S-type

Algorithm 9 M'-type encoding operation.

```
1: {append M' to op-code sequence.}
2: O_Seq = O_seq|M';
3: {update marks.}
4: g.m = 0;
5: g.p.o.m = 1;
6: g.n.o.m = 1;
7: b = g.n; {initial candidate for the halfedge b.}
8: while b.m ≠ 3 do
9:     b = b.o.p; {turn around the vertex g.v.}
10: end while
11: b_n = b.N; {initial candidate for the halfedge b_n.}
12: while b_n.m ≠ 1 do
13:     {update marks.}
14:     b_n.m = 1;
15:     b_n.e.m = 1;
16:     b_n = b_n.N; {move to next edge around handle.}
17: end while
18: {connect the halfedges g.P and g.p.o.}
19: g.P.N = g.p.o;
20: g.p.o.P = g.P;
21: {connect the halfedges g.p.o and b.N.}
22: g.p.o.N = b.N;
23: b.N.P = g.p.o;
24: {connect the halfedges b and g.n.o.}
25: b.N = g.n.o;
26: g.n.o.P = b;
27: {connect the halfedges g.n.o and g.N.}
28: g.n.o.N = g.N;
29: g.N.P = g.n.o;
30: position = edge_stack.find(); {find the halfedge position from the halfedge s-
    tack.}
31: edge_stack.remove(position); {delete the halfedge from the halfedge stack.}
32: offset = distance(position, g.v); {calculate the offset value.}
33: mp_table.add(position, offset, s_count); {update the M' table.}
34: handle_s = size(offset_table); {update the counter handle_s.}
35: g = g.n.o; {update the active gate g.}
```

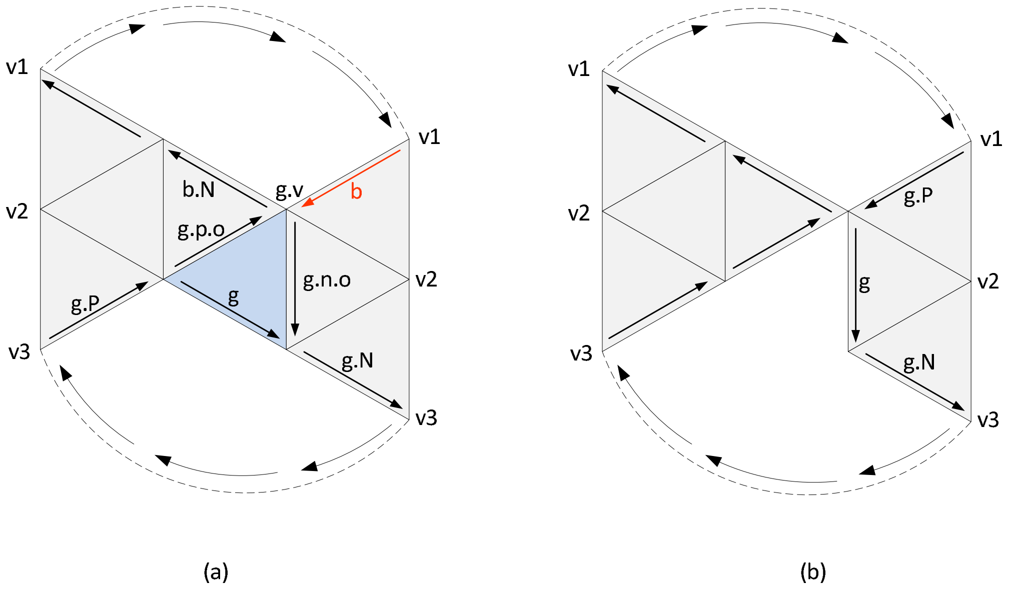


Figure 3.11: An example illustrating the M'-type operation. (a) The initial mesh corresponds to an M'-type triangle. (b) The resulting mesh obtained after the M'-type operation.

triangles. Therefore, the decoder needs to know the two split points for the active bounding loop. If the mesh contains no handles, this information can be determined by the transmitted op-code sequence. When handles are present, however, it is not possible for the decoder to determine one of the split points for some of the S-type triangles. The split point that cannot be determined is the vertex opposite the active gate (i.e., the vertex V_5 in Figure 3.12(a)). To resolve this problem, the information stored in the offset table is used to determine the split point in the decoder. A pair of values is stored for each offset table entry: the first value is the S-type triangle counter, and the second value is the offset value $\mathtt{O_V}$. The offset value $\mathtt{O_V}$ can be calculated by

$$\mathtt{O_V} = \mathtt{R_V} - 2, \tag{3.1}$$

where $\mathtt{R_V}$ denotes the number of vertices located on the right submesh's boundary. The offset value can also viewed as how many steps the method moves from the next halfedge of the active gate on the bounding loop's incident vertex (i.e., the vertex `g.N.e`) to the split point along the bounding loop. For example, as shown in Figure 3.12(a), the method moves three steps following the numbered arrows in order to reach the vertex V_5 (i.e., from V_2 to V_5). Therefore, the offset value for this S-type triangle is three. From Figure 3.12(c), we see that $\mathtt{R_V}$ equals five. By using (3.1), the offset value $\mathtt{O_V}$ is obtained as

$$\mathtt{O_V} = \mathtt{R_V} - 2 = 5 - 2 = 3. \tag{3.2}$$

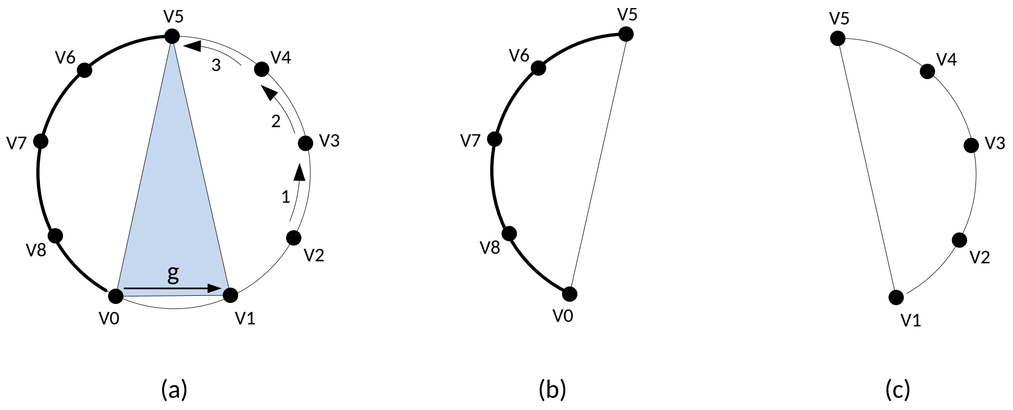


Figure 3.12: An example illustrating the split operation. (a) The initial bounding loop before the split operation. (b) and (c) The resulting left and right submeshes obtained after the split operation, respectively.

The offset value calculated above matches the result we obtained from Figure 3.12(a).

As explained earlier, when mesh contains handles, some of the S-type triangles' split points cannot be determined by the decoder. In what follows, we will explain how to determine whether or not an S-type triangle's split point can be determined at the decoder side.

When the encoder visits an M'-type triangle, there must be one or more halfedges on the halfedge stack. For all the halfedges on the halfedge stack, their corresponding S-type triangles' split point cannot be determined by the decoder. We use a short example to explain the above concept. For example, suppose that the opcode sequence obtained is S_1RS_2E S_3M'RRLERRE. The indices of S-type triangles indicate the order in which they are visited by the encoder. Figure 3.13 shows the halfedge stack's change for the given op-code sequence, and entries in each stack illustrate the halfedge generated by the corresponding S-type triangles. Figures 3.13(a) and (b) show the halfedge stack after encoding the first and second S-type triangles, respectively. Figure 3.13(c) shows the halfedge stack after encoding the first E-type triangle, while Figure 3.13(d) shows the halfedge stack after encoding the third S-type triangle. From Figure 3.13(d), we notice that two halfedges (i.e., halfedges from triangles S_1 and S_3) are on the stack when the encoder encounters the M'-type triangle. This means the split points for S_1 and S_3 triangles cannot be determined by the decoder, and this information needs to be read from the offset table.

From the above example, we notice that the split point of the second S-type triangle (i.e., S_2) can be determined by the decoder. This due to the fact the after encoding the the first E-type triangle, the halfedge produced by S_2 is popped from the halfedge stack, as shown in Figure 3.13(c). An S-type triangle is considered to be affected by a handle (i.e., known as the handle-related S-type triangle) if its split point cannot be determined by

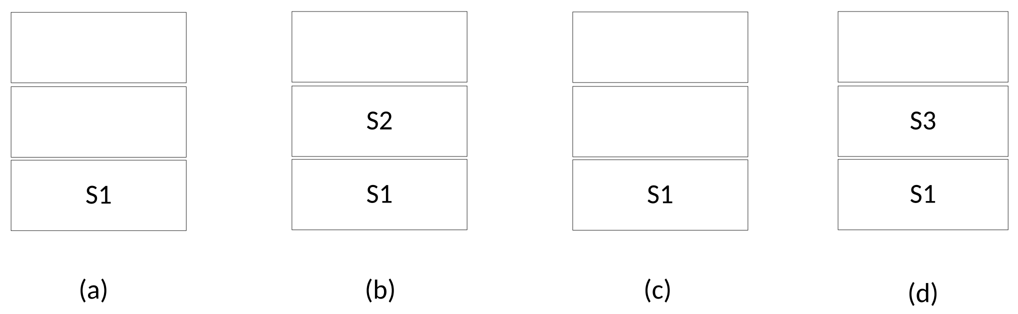


Figure 3.13: The halfedge stack's change for the given op-code sequence. (a) and (b) The resulting halfedge stack after encoding the first and second S-type triangles (i.e., S_1 and S_2), respectively. (c) The resulting halfedge stack after encoding the first E-type triangle. (d) The resulting halfedge stack after encoding the third S-type triangle (i.e., S_3).

the decoder. At the point when the encoder meets an S-type triangle, the method cannot determine whether the S-type triangle is affected by a handle or not. This identification, however, can only be facilitated at the E-type operation and with the help of the counter `handle_s`. The counter `handle_s` is initialized to zero at the beginning of the encoding process, and its value is updated to the offset table's size in each M'-type operation. The operation to update counter `handle_s` can be found in step 34 of Algorithm 9 (on page 30).

The detailed offset table update operations are as follows. In the S-type operation, an entry is added to the offset table, as shown in step 39 of Algorithm 6 (on page 25). In the E-type operation, as shown in step 7 of Algorithm 7 (on page 27), the encoder first compares the counter `handle_s` with the offset table's size. If the offset table's size is larger than the value of `handle_s`, this means the last table entry should be removed from the table. Otherwise, all entries in the offset table belong to the handle-related S-type triangles and should be kept in the table.

3.2.9 Transmitted Data

Having presented the encoding details for each triangle type, we can now introduce the variables and data structures that are transmitted to the decoder side. Since the number of handles and holes are typically small in practical applications, the method uses the S symbol in the op-code sequence that is transmitted to the decoder side to represent S-, M- and M'-types triangles. Moreover, the information stored in both M table and M' table is sufficient to separate these three types in the decoder side. Therefore, the transmitted op-code sequence only contains five symbols: C, L, R, E, and S. The op-code sequence, the offset table, the M table, the M' table, and the coded prediction residuals are transmitted to the decoder side. For triangle meshes without holes, the M table is empty. For triangle meshes without handles, the M' table and offset table are empty. The encoding program gathers all the data structures in the given order, codes them into a binary file, and transmits the file to the decoder side. The coded binary file is stored in EB format. A detailed description of the EB format can be found in Section 4.5.

3.3 Examples of the Encoding Method

Having fully described the encoding method, we now provide some mesh-coding examples for illustrative purposes. In particular, we consider four examples:

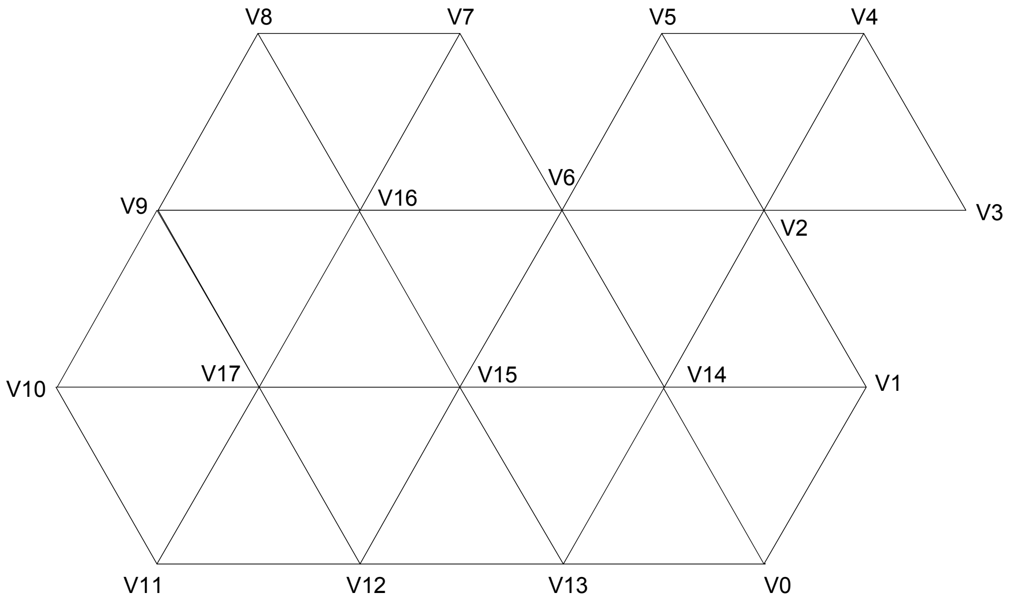
1. A mesh with one bounding loop and no handles.
2. A mesh with zero bounding loops and no handles.
3. A mesh with two bounding loops.
4. A mesh with zero bounding loops and one handle.

3.3.1 Example 1: Mesh with One Bounding Loop and No Handles

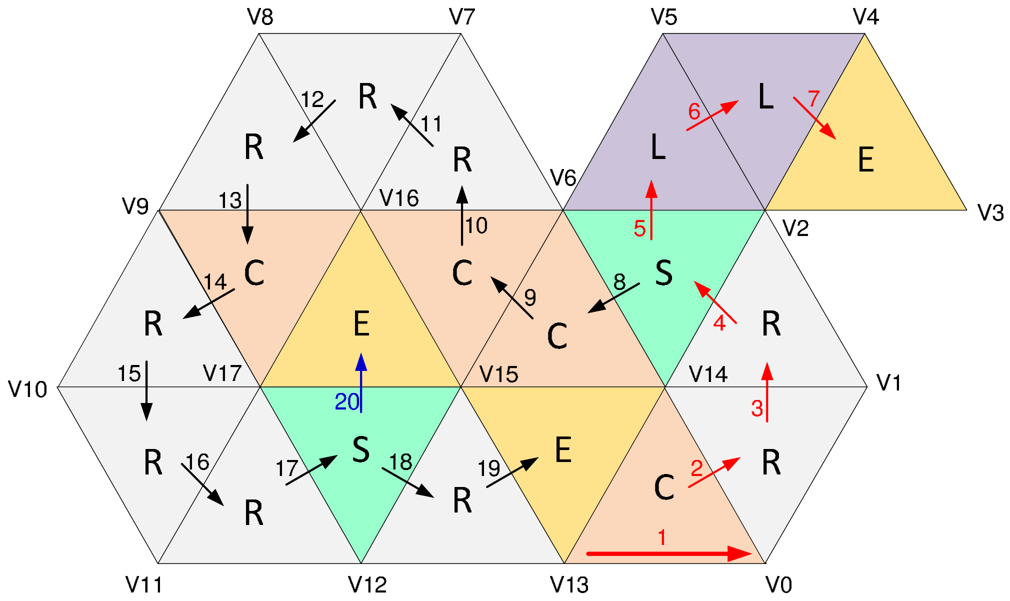
The first mesh-coding example considered is for a mesh with one bounding loop and no handles. Figure 3.14(a) shows the mesh, and Figure 3.14(b) shows the resulting encoding sequence. This example contains all five triangle types (C-, L-, E-, R-, and S-types) such that we can illustrate the encoding process for each type specifically. As shown in Figure 3.14(b), the encoder traverses the mesh by following the arrows in sequence.

The encoding preprocessing steps are as follows. All mesh's vertices are quantized to obtain the integer quantization indices. Then, all vertex flags are set to false meaning no vertex has been coded. Next, all boundary halfedge and vertex marks are set to one, and all interior halfedge and vertex marks are set to zero. Under this circumstance, vertices V_0 to V_{13} in Figure 3.14(b) have a mark of zero, and vertices V_{14} to V_{17} have a mark of one.

Having finished the above preprocessing steps, the encoding process is started. The initial active gate chosen in this example is the halfedge $\overrightarrow{V_{13}V_0}$ (i.e., the halfedge with label 1 in Figure 3.14(b)). Since the vertex V_{14} is located inside the mesh and its mark equals zero, the first triangle is C-type. Then, the encoder updates V_{14}'s mark from zero to one, meaning V_{14} is located on the remaining mesh's boundary. Next, the encoder predicts the positions of the three vertices in the C-type triangle, calculates the differences between the predicted and the actual positions, and binarizes and encodes the prediction residuals. The vertex flag is set to true after coding each vertex. Following the halfedge with label 2 in Figure 3.14(b), the encoder moves to the second triangle, and the new active gate is the halfedge $\overrightarrow{V_{14}V_0}$.



(a)



(b)

Figure 3.14: A triangle mesh with one bounding loop and no handles. (a) The triangle mesh with boundary. (b) Steps in the Edgebreaker encoding process.

Since the vertex V_1 is located on the mesh's boundary and immediately follows the active gate, the second triangle is R-type. The halfedge relationships on the mesh's boundary are updated. According to step 4 of Algorithm 1 (on page 16), the encoder needs to check and encode the newly encountered vertex. The vertex V_1 is coded by the method and its flag is set to true after the vertex encoding. Further, the active gate is updated to the halfedge $\overrightarrow{V_{14}V_1}$. Proceeding in a similar way, the third triangle encoded is also R-type, and the active gate is set to the halfedge $\overrightarrow{V_{14}V_2}$.

The vertex V_6 has three characteristics: 1) it is located on the mesh's boundary, 2) it neither immediately precedes nor follows the active gate, and 3) it has a mark of one. Based on the above information, the fourth triangle is S-type. The S-type operation splits the current mesh into two submeshes. The left submesh's boundary contains the vertices V_6, V_7, ..., V_{14}, and the right submesh's boundary contains the vertices V_2, V_3, ..., V_6. For all the vertices and halfedges located on the left submesh's boundary, their marks are updated to three. The halfedge $\overrightarrow{V_{14}V_6}$ is then pushed to the halfedge stack and will be used as the active gate when processing the left submesh. Next, the encoder updates the offset table, sets the active gate to the halfedge $\overrightarrow{V_6V_2}$, and traverses the right submesh.

The first triangle of the right submesh is L-type. This is due to the fact that the vertex V_5 is located on the right submesh's boundary and immediately precedes the active gate. Similarly, the second triangle of the right submesh is also L-type, and the active gate is set to the halfedge $\overrightarrow{V_4V_2}$. Since the vertex V_3 is located on the submesh's boundary and immediately precedes and follows the active gate, the last triangle is E-type. The counter `handle_s` equals zero and its value is less than the offset table's size (i.e., one). Therefore, an entry is removed from the offset table and the offset table becomes empty. Next, the encoder checks the halfedge stack and pops the halfedge $\overrightarrow{V_{14}V_6}$. Starting from the popped halfedge $\overrightarrow{V_{14}V_6}$, the encoder traverses along the left submesh's boundary and update the vertex and halfedge marks to one. The remaining portion of the mesh is processed similarly. The op-code sequence generated after the entire coding process is CRRSLLECCRRRCRRRSREE.

3.3.2 Example 2: Mesh with Zero Bounding Loops and No Handles

The second mesh-coding example considered is for a tetrahedral mesh. This example illustrates the encoding process for a mesh with zero bounding loops (i.e., closed) and no handles. Figure 3.15(a) shows the original mesh, while Figure 3.15(b) shows a cut and flattened view of the mesh. Note that Figure 3.15(b) is only for visualization purposes as the Edgebreaker method codes the input mesh directly without any cutting or flattening operation. Figure 3.15(c) illustrates the steps in the encoding process.

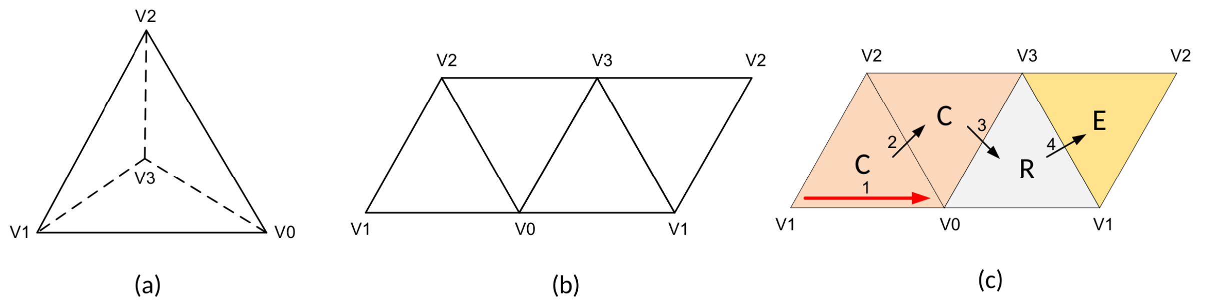


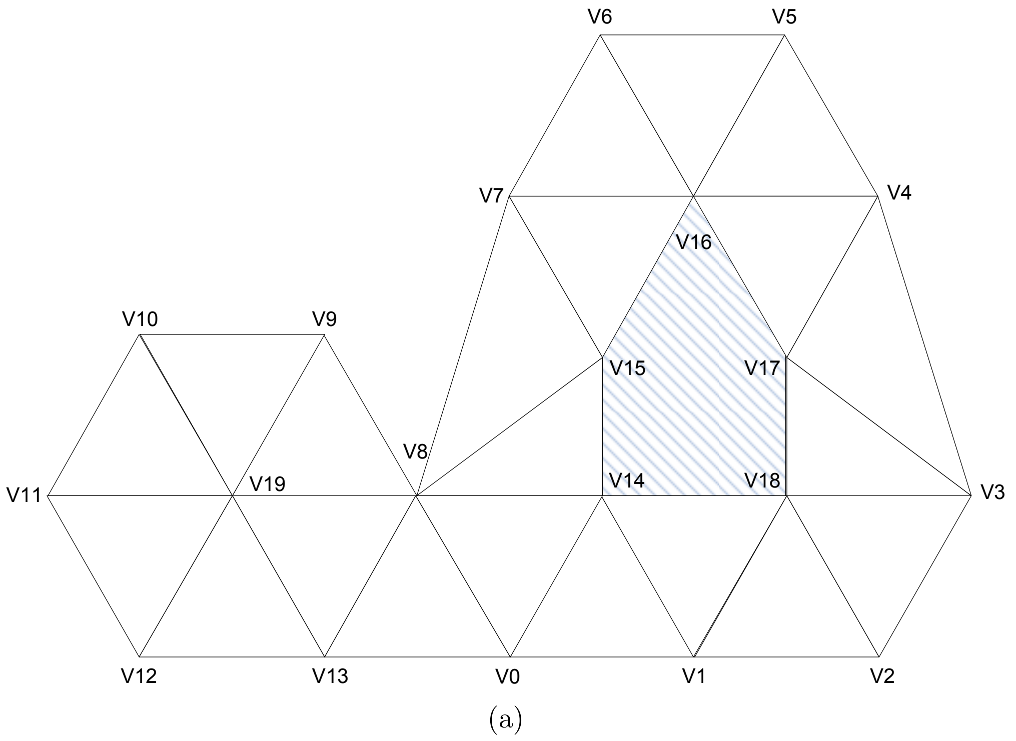
Figure 3.15: A triangle mesh with zero bounding loops (i.e., closed) and no handles. (a) The tetrahedral mesh. (b) A cut and flattened view of the tetrahedral mesh. (c) Steps in the Edgebreaker encoding process.

In the mesh preprocessing, the encoder quantizes all vertex coordinates, initializes all vertex and halfedge marks to zero, and sets all vertex flags to false. Since the original mesh is closed, an arbitrary halfedge can be chosen as the initial active gate. As shown in Figure 3.15(c), the halfedge $\overrightarrow{V_1V_0}$ is chosen. The initial bounding loop is formed by the initial active gate. Therefore, the initial active gate's vertex and halfedge marks need to be updated. The vertices V_1's and V_0's marks are set to one, and the halfedge $\overrightarrow{V_1V_0}$'s mark is also set to one. The encoding process starts after the mark updates. Proceeding in a similar way as the first example, the encoder generates the op-code sequence: CCRE.

3.3.3 Example 3: Mesh with Two Bounding Loops and No Handles

The third mesh-coding example considered is for a mesh with two bounding loops (i.e., one hole) and no handles. The purpose of providing this example is to show how the hole is handled by the encoding method. Figure 3.16(a) shows the mesh, and Figure 3.16(b) shows the steps in the encoding process. Both arrows and vertices in Figure 3.16(b) are labeled with indices. The preprocessing steps of this example is similar to the previous examples. Vertices V_0's to V_{13}'s marks are set to one, vertices V_{14}'s to V_{18}'s marks are set to two, and the vertex V_{19}'s mark is set to zero. The initial active gate is the halfedge $\overrightarrow{V_{13}V_0}$. The first triangle is S-type and the mesh is split into two submeshes. Vertices V_0, V_1, ..., V_8 form the right submesh's boundary, and vertices V_8, V_9, ..., V_{13} form the left submesh's boundary.

As shown in Figure 3.16(b), the active gate for the right submesh is the halfedge $\overrightarrow{V_8V_0}$. Since the vertex V_{14}'s mark is two, the first triangle of the right submesh is M-type. As explained earlier, the M-type operation inserts an entry into the M table, updates the vertex and halfedge marks on hole's boundary, and also merges the hole's boundary into the active bounding loop. By traversing the hole's boundary, the encoder computes the number of vertices located on hole's boundary which is five. The encoder only meets one S-type triangle



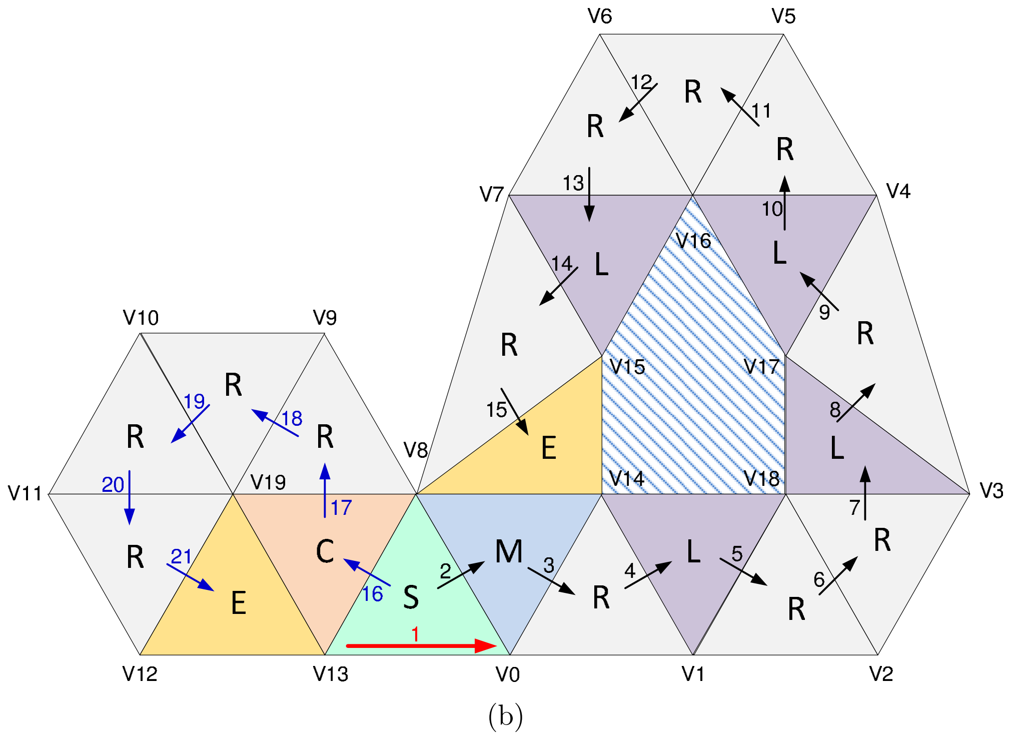


Figure 3.16: A triangle mesh with two bounding loops (i.e., one hole) and no handles. (a) The triangle mesh with a hole. (b) Steps in the Edgebreaker encoding process.

in the previous encoding process. Therefore, two values (i.e., one and five) are added to the M table. Next, the encoder updates the vertex and halfedge marks on hole's boundary and merges the bounding loops. After the halfedge relationship updates, the halfedge $\overrightarrow{V_8V_{14}}$'s next halfedge on the bounding loop is the halfedge $\overrightarrow{V_{14}V_{15}}$, and the halfedge $\overrightarrow{V_{14}V_0}$'s previous halfedge on the bounding loop is the halfedge $\overrightarrow{V_{18}V_{14}}$. The active gate after the M-type operation is updated to the halfedge $\overrightarrow{V_{14}V_0}$. As shown in the figure, the encoder traverses the mesh by following the arrows in sequence. Upon completion of the above process, the op-code sequence SMRLRRLRLRRRLRECRRRRE is obtained. As explained earlier, the transmitted op-code sequence only uses five symbols (i.e., C, L, R, E, and S), and the S symbol is used to represent S-, M-, and M'-types triangles. Therefore, the op-code sequence that is transmitted to the decoder side is SSRLRRLRLRRRLRECRRRRE.

3.3.4 Example 4: Mesh with Zero Bounding Loops and One Handle

The fourth mesh-coding example considered is for a mesh with zero bounding loops and one handle. The simplest such mesh corresponds to a torus. The purpose of providing this example is to show how a handle is processed by the encoding method. For this reason, we focus mainly on the processing of M'-type triangle in the example. Figure 3.17 shows a pictorial view of the torus mesh. Figure 3.17(a) shows the torus mesh in perspective view, while Figures 3.17(b) and (c) show top and bottom views of the mesh, respectively. Figure 3.17(d) shows the connectivity of the torus mesh obtained after cutting and flattening the original mesh. Note that Figure 3.17(d) is only for visualization purposes, since the Edgebreaker method codes the input mesh directly without any cutting or flattening operation.

The mesh preprocessing steps are identical to the coding example in Section 3.3.2 (on page 36). Figure 3.18 shows the internal steps of the Edgebreaker encoding process. Figures 3.18(a) and (b) presents the coding sequence of the first 17 and 20 triangles, respectively. Figure 3.18(c) shows the resulting active bounding loop after the M'-type operation.

The initial active gate is the halfedge $\overrightarrow{V_0V_4}$ in Figure 3.18(a). The triangle traversal sequence follows the numbered arrows in order. The op-code sequence obtained for the first 17 triangles is CCCCCRCCCRCCRCCRS. In Figure 3.18(a), the left submesh's boundary contains the vertices V_9, V_8, V_1, V_6, V_{14}, and V_{10}, and the right submesh's boundary contains the vertices V_{12}, V_8, V_{11}, V_3, and V_7. The thick dotted line in the figure shows the symmetric part of the bounding loop on the other side of the mesh. All vertices and halfedges on the left submesh's boundary have the mark of three. The halfedge $\overrightarrow{V_9V_8}$ in Figure 3.18(a) is pushed to the halfedge stack.

By following the arrows in Figure 3.18(b), the next two triangles encoded are C- and

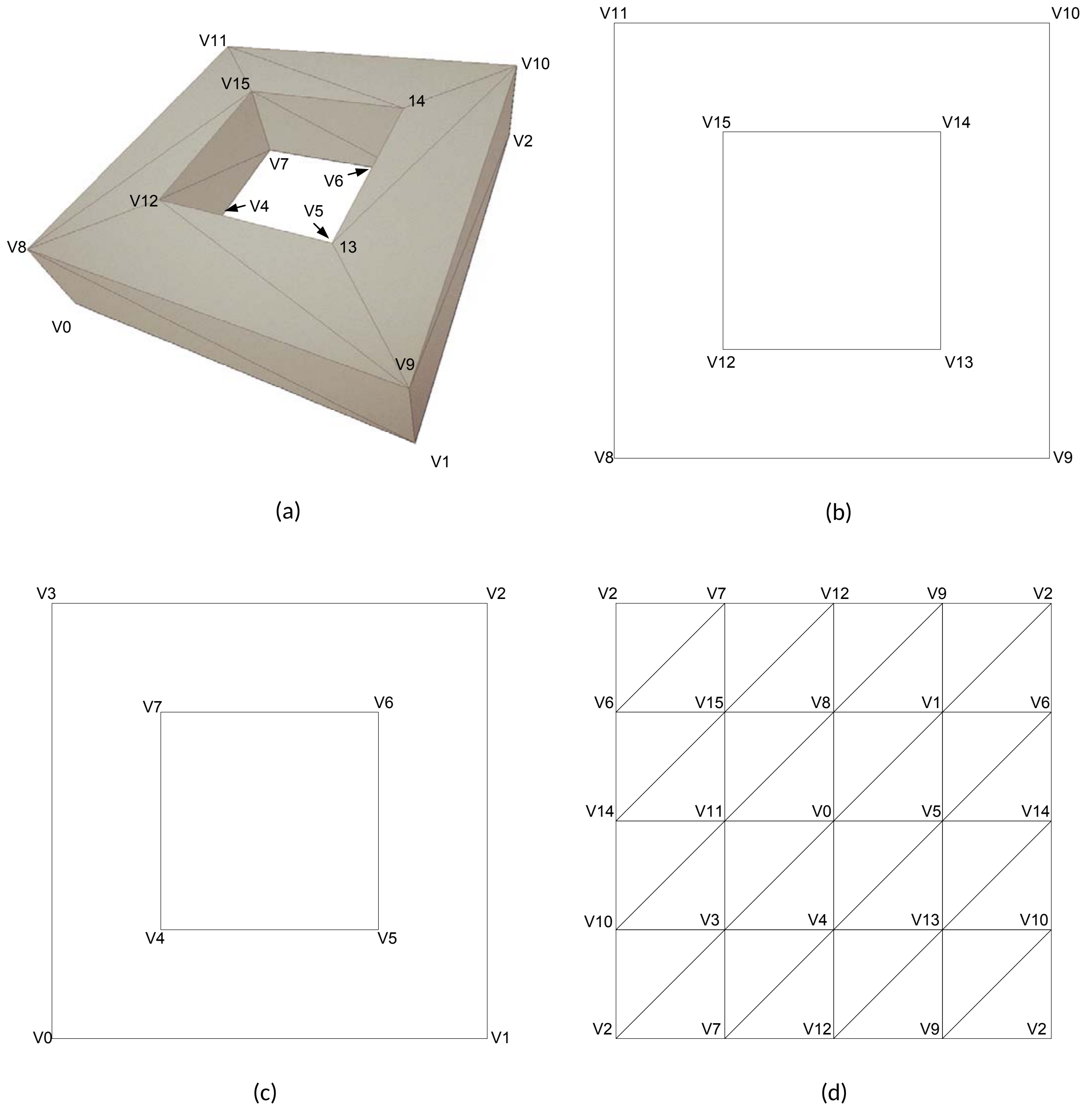


Figure 3.17: A triangle mesh with zero bounding loops and one handle. (a) The torus mesh in perspective view. (b) and (c) The top and bottom views of the mesh. (d) The connectivity of the torus mesh obtained after cutting and flattening the original mesh.

R-types. The third triangle (i.e., the triangle with vertices V_{15}, V_7, V_6) is M'-type. This is due to the fact that the vertex V_6 is located on the mesh's boundary and neither precedes nor follows the active gate, and has the vertex mark of three. For the M'-type triangle, the encoder updates the M' table and also merges an inactive loop into the active bounding loop. The vertex opposite the active gate for the M' triangle is the vertex V_6, and the halfedge sought in the halfedge stack is the halfedge $\overrightarrow{V_9V_8}$ in Figure 3.18(b). Since $\overrightarrow{V_9V_8}$ is the only halfedge located in the stack, the position value for the current M'-type triangle is zero. Starting from $\overrightarrow{V_9V_8}$, after moving two steps along the boundary (i.e., $\overrightarrow{V_8V_1}$ and $\overrightarrow{V_1V_6}$), the encoder reaches the vertex V_6. Therefore, the offset value for the current M'-type triangle is two. The encoder only meets one S-type triangle in the previous encoding process, so the third value of the M' table entry is one. Hence, the position value, the offset value, and the number of S-type triangles encountered previously are zero, two, and one, respectively. These three values are grouped together in the given order and added to the M' table.

Next, the encoder updates the counter `handle_s` to one. Then, the encoder updates the vertex and halfedge marks on the bounding loop which contains the vertices V_9, V_8, V_1, V_6, V_{14}, and V_{10} in Figure 3.18(b) and merges this bounding loop to the active bounding loop. After the halfedge relationship updates, the halfedges $\overrightarrow{V_8V_{15}}$, $\overrightarrow{V_{15}V_6}$, and $\overrightarrow{V_6V_{14}}$ are connected in order, and the halfedges $\overrightarrow{V_1V_6}$, $\overrightarrow{V_6V_7}$, and $\overrightarrow{V_7V_3}$ are connected in order. The bounding loop after the M'-type encoding is shown in Figure 3.18(c). Furthermore, the active gate is set to the halfedge $\overrightarrow{V_6V_7}$.

The rest of the torus mesh can be encoded in a similar way, as shown in Figure 3.19. Upon completion of the above process, we obtain the op-code sequence CCCCCRCCCRC-CRCCRSCRM'CRSRLSEERSEE is obtained. Moreover, the op-code sequence transmitted to the decoder side is CCCCCRCCCRCCRCCRSCRSCRSRLSEERSEE.

3.4 Parallelogram-Prediction Scheme

As explained earlier, the parallelogram-prediction scheme in [4] is used by the Edgebreaker method to predict vertex positions. The integer difference between the predicted and actual positions (i.e., the prediction residual) are binarized and encoded by the arithmetic coder. The scheme's name originates from its geometric interpretation. The parallelogram-prediction scheme predicts one vertex in a parallelogram from zero or more other vertices in that parallelogram. The scheme predicts a vertex r in a parallelogram from zero or more of the other three vertices u, v, and w, as shown in Figure 3.20. In order to combine the parallelogram-prediction scheme with the Edgebreaker method, Rossignac proposed four d-ifferent cases [19]. In some cases, vertices u, v, and w are not all known before the prediction.

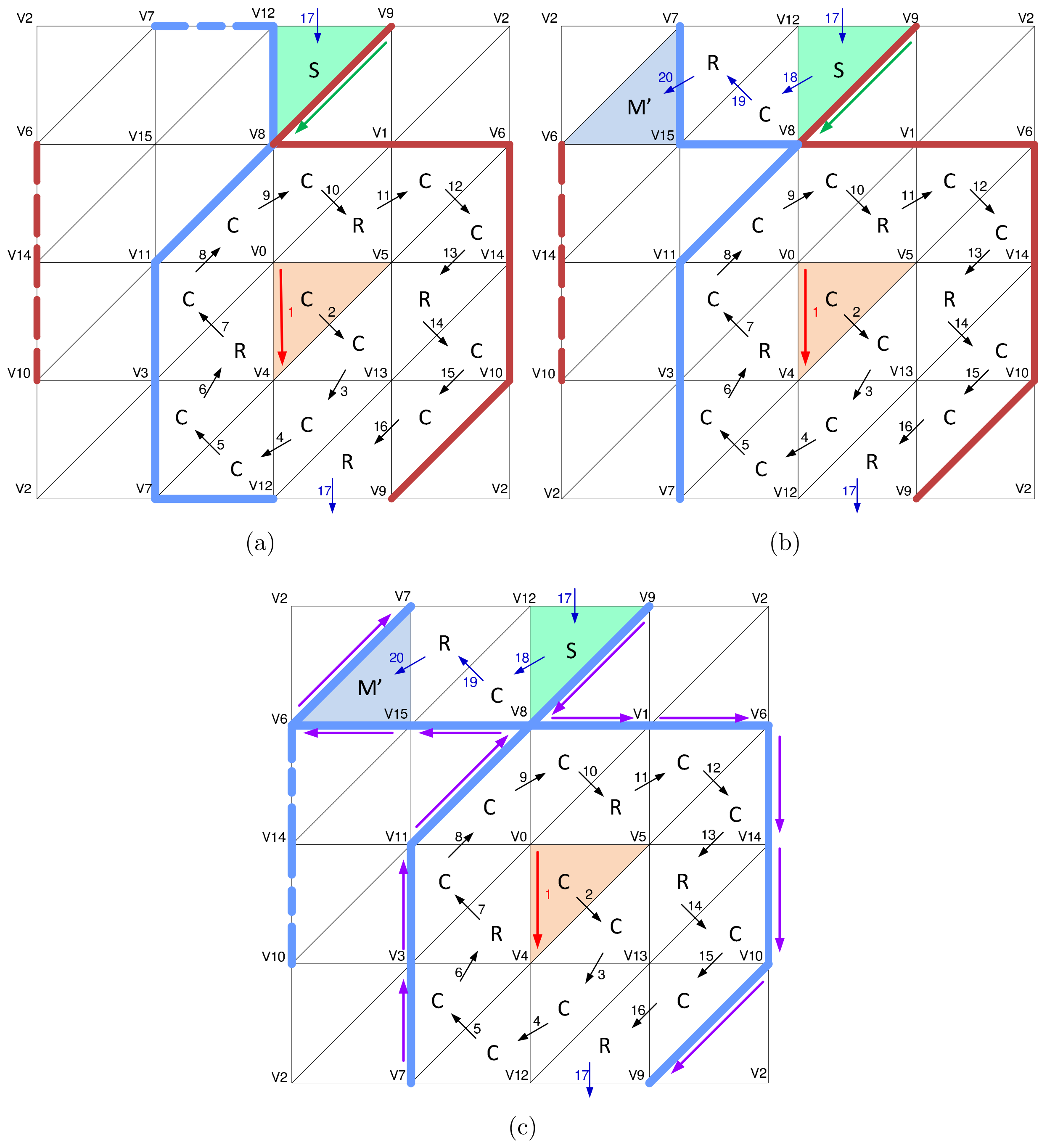


Figure 3.18: The internal steps of the Edgebreaker encoding process. (a) and (b) The encoding sequence of the first 17 and 20 triangles, respectively. (c) The resulting active bounding loop after the M’-type operation.

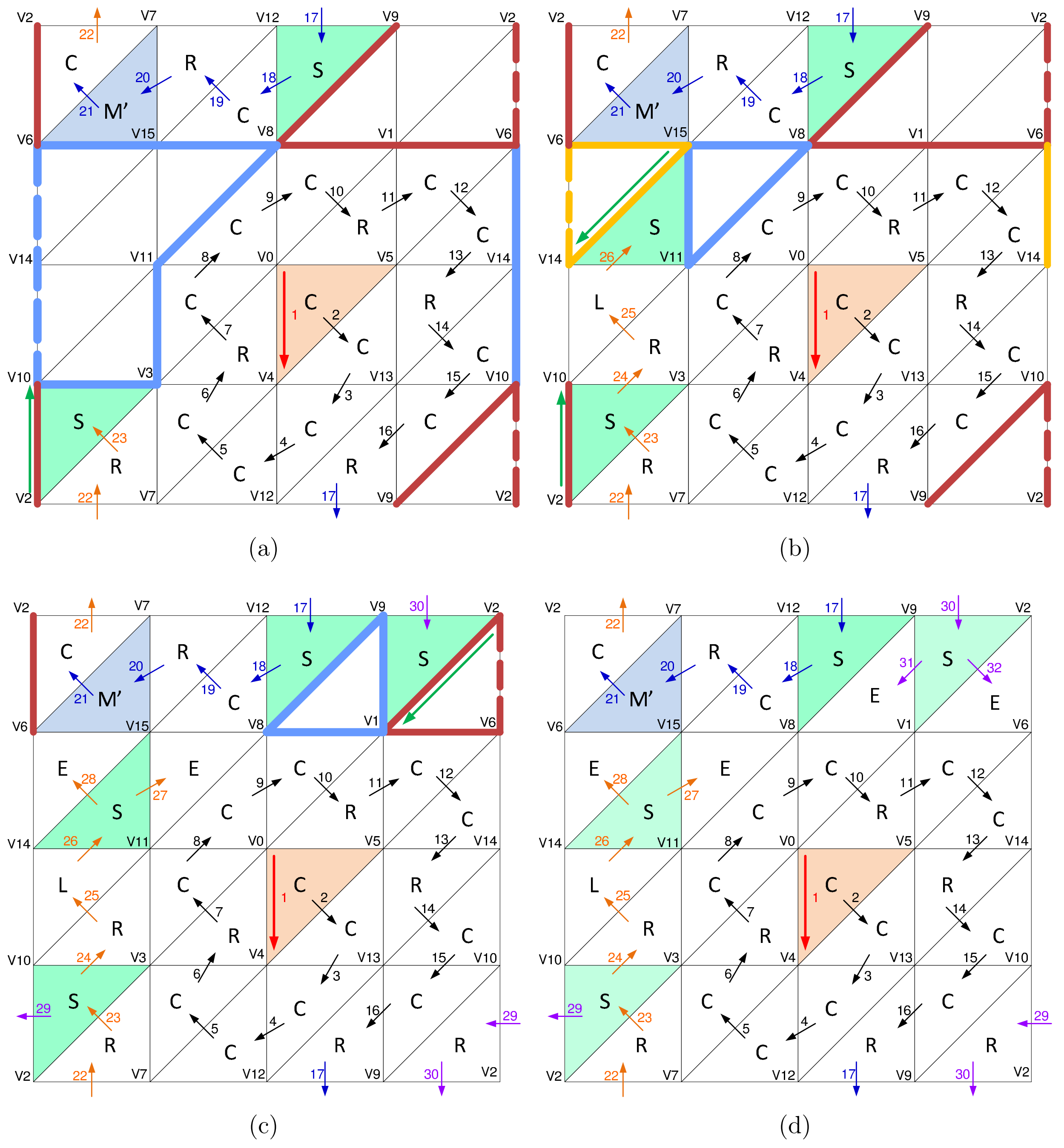


Figure 3.19: The internal steps of the Edgebreaker encoding process. (a) - (c) The coding sequence after processing the second, third, and fourth S-type triangles, respectively. (d) The entire encoding process of the torus example.

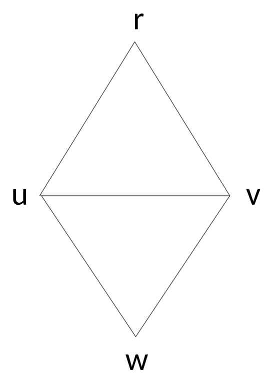


Figure 3.20: Parallelogram-prediction scheme.

Each of these cases is shown in Figure 3.21.

Case 1. In this case, none of the vertices is known before the prediction, as shown in Figure 3.21(a). The vertex r is predicted as the origin (i.e., $\hat{r} = (0, 0, 0)$). This case is used to predict the first vertex of the triangle first processed by the Edgebreaker method.

Case 2. In this case, only the vertex u is known before the prediction, as shown in Figure 3.21(b). The vertex r is predicted as the vertex u (i.e., $\hat{r} = u$). This case is used to predict the second vertex of the triangle first processed by the Edgebreaker method.

Case 3. In this case, two vertices u and v are known before the prediction, as shown in Figure 3.21(c). The vertex r is predicted to the average of the two known vertices as $\hat{r}$, where

$$\hat{r} = \frac{1}{2}(u + v) \tag{3.3}$$

This case is used to predict the third vertex of the triangle first processed by the Edgebreaker method.

Case 4. In this case, all three vertices u, v, and w are all known before the prediction, as shown in Figure 3.21(d). The vertex r can be predicted as $\hat{r}$, where

$$\hat{r} = v + u - w. \tag{3.4}$$

As explained earlier, the difference between the actual and the predicted positions is the data encoded for each vertex. When quantizer indices are used, the prediction is rounded to an integer value to maintain the integer nature of data. Therefore, the prediction residual Δ can be calculated by

$$\Delta = r - \text{round}(\hat{r}). \tag{3.5}$$

In what follows, we provide a short example to illustrate the prediction process. Fig-

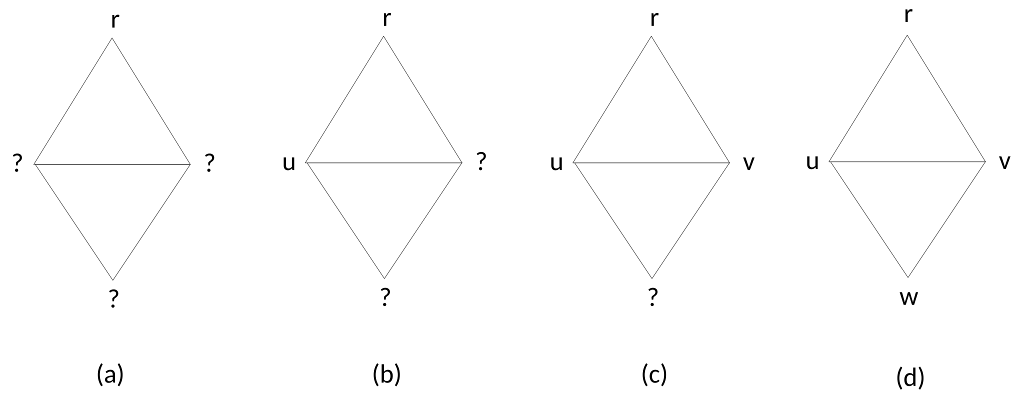


Figure 3.21: Different cases of the parallelogram-prediction scheme. (a) to (d) Pictorial view of cases 1 to 4.

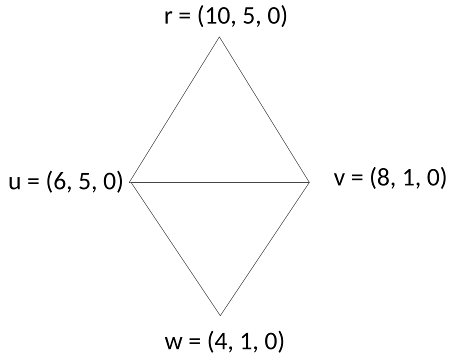


Figure 3.22: A parallelogram to illustrate the prediction residual generation process.

ure 3.22 shows four vertices u, v, w, and r that form a parallelogram, where

$$u = (6,5,0), v = (8,1,0), w = (4,1,0), \text{and}\, r = (10,5,0). \tag{3.6}$$

Since all three vertices u, v and w are known before the prediction, equation in (3.4) is used to predict the vertex $\hat{r}$ as follows

$$\hat{r} = u + v - w = (6,5,0) + (8,1,0) - (4,1,0) = (10,5,0). \tag{3.7}$$

By using (3.5), the prediction residual is obtained as

$$\Delta = r - \hat{r} = (10,5,0) - (10,5,0) = (0,0,0). \tag{3.8}$$

Therefore, the prediction residual encoded for this example is $(0,0,0)$.

Algorithm 10 Mesh decoding process.

```
1: Initialization phase: Compute quantities that are used by the generation phase.
2: Generation phase:
3: while Not all the op-codes are processed by the decoding method. do
4:     Read the op-code and compute the three indices of the triangle.
5:     Decode the prediction residuals of newly encountered vertex in the current triangle.
6:     Predict and reconstruct the vertex position.
7:     Move to the next op-code in the op-code sequence.
8: end while
```

3.5 Decoding Method

Having fully described the encoding process, we now turn our attention to the decoding process. To begin, we introduce the decoding method in general terms. The decoder receives the op-code sequence and coded prediction residuals from the encoder side, and reconstructs the mesh from this. The mesh connectivity is reconstructed from the op-code sequence, and the mesh geometry is reconstructed from the coded prediction residuals. The decoder performs two traversals of the op-code sequence. The first traversal, known as the initialization phase, calculates quantities that are used in the mesh reconstruction. The second traversal, known as the generation phase, creates triangles in the same order as processed by the encoder and also reconstructs the vertex positions. The reconstructed triangles in the generation phase are represented by the indices of their three vertices. For each op-code, the decoder first computes the three vertex indices of the triangle. Then, the prediction residuals of any newly encountered vertex in the triangle is decoded. Similar to the encoding process, the parallelogram-prediction scheme and the UI-binarization scheme are used to reconstruct the integer quantizer indices. Next, the vertex positions are reconstructed from the integer quantizer indices by applying the quantizer reconstruction rule. Finally, the decoder moves to the next op-code in the op-code sequence. The generation process is iterated until all of the op-codes in the op-code sequence are processed. The pseudocode for the mesh decoding can be found in Algorithm 10.

Having introduced the Edgebreaker decoder in general terms, we now present the decoding method in detail. The main state information used by the decoder consists of the following:

- The intermediate mesh's boundary, which is represented by a circular doubly-linked list.
- The op-code sequence, which stores the triangle types.
- The prediction residuals sequence, which stores the coded vertices.

- The facet table, which is used for storing the reconstructed facets.
- The vertex table, which is used for storing the reconstructed vertices.
- The M table, which is used to store the information related to holes.
- The M' table, which is used to store the information related to handles.
- The offset table, which is used for storing the offset values for the S-type operations whose offset value cannot be otherwise determined, due to the presence of handles.
- The S table, which is used for storing the offset values for all S-type triangles.
- The node stack, which is used to store the representative nodes of the boundaries that are not currently being processed.
- The vertex counter, which is used to count how many vertices have been encountered so far.
- The S-type counter, which is used to count the number of S-type triangles that have been encountered by the generation phase.

As explained earlier, the intermediate mesh's bounding loop is represented by a circular doubly-linked list. This means that the main data structure used in the decoding process is a circular doubly-linked list. Each node in the list represents a vertex that is located on the intermediate mesh's boundary. Each node is associated with a vertex index. Therefore, the previous and next nodes on the list represent the previous and next vertices on the intermediate mesh's boundary, respectively. The *active node* is a node that is used to identify the triangle that is currently being reconstructed. The notation used in our explanation of the decoding method is as follows:

- The symbol `G.i` denotes the vertex index of the node `G`.
- The symbol `G.P` denotes the previous node of `G` in the list.
- The symbol `G.N` denotes the next node of `G` in the list.

The pictorial view of the above notation can be found in Figure 3.23.

The vertex flag is also used by the decoder. The pseudocode of the vertex data structure in the decoder can be found in Figure 3.24. The flag `flag` in this data structure indicates whether the vertex is decoded or not. The vertex flag is used as follows. At the beginning of the decoding process, all vertex flags are set to false, meaning none of the vertices has been processed by the decoder. During the decoding process, when a vertex is encountered,

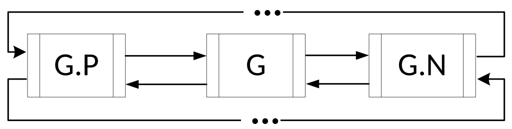


Figure 3.23: Pictorial view of the notations used in the decoding method.

```
struct Vertex {
  double x_coordinate; // The x coordinate
  double y_coordinate; // The y coordinate
  double z_coordinate; // The z coordinate
  bool flag;           // The vertex flag
};
```

Figure 3.24: Definition of the vertex data structure in decoder.

the decoder first checks its flag. If the vertex flag equals false, the decoder reconstructs the vertex position and sets the vertex flag to true. The checking process is necessary due to the fact that the decoder only decodes each vertex once.

As explained in the encoding section, only five codewords (i.e., C, L, R, E, and S) are used to code the seven triangle types. In particular, the S-, M-, and M'-types triangles are all represented by the same codeword. Under this circumstance, the first step of decoding is to preprocess the input op-code sequence to distinguish between the S-, M-, and M'-types triangles which use the same codeword. The triangle type identification is based on the the S-type counter value stored in both M and M' tables. A counter `s_opcode_count` is initialized to zero at the beginning of the preprocessing phase, and its value is updated each time an S codeword is encountered by the decoder. The pseudocode for the op-code sequence preprocessing is shown in Algorithm 11. The symbol `m_count` in Algorithm 11 represents the S-type counter in the M table, and `mp_count` represents the S-type counter in the M' table.

3.5.1 Decompression Initialization Phase

Having fully introduced the op-code sequence preprocessing step, we now present the initialization phase details. The operations in the initialization phase calculate the number of vertices on the mesh's boundary, and also generate the S table. To begin, we introduce the variables and data structures used in the initialization phase. The initialization phase maintains a counter `e`, a counter `s`, an ES-pair stack, and the S table. The detailed descriptions of these variables are listed in Table 3.1. The operations of the initialization phase are

Algorithm 11 Op-code sequence preprocessing.

```
if op_code = S then
    if s_opcode_count = m_count then
        ++s_opcode_count; {M-type triangle.}
        Updates the value of m_count to the S-type counter of the next table entry.
    else if s_opcode_count = mp_count then
        ++s_opcode_count; {M'-type triangle.}
        Updates the value of mp_count to the S-type counter of the next table entry.
    else
        ++s_opcode_count; {S-type triangle.}
    end if
end if
```

Table 3.1: Variables in the decoding initialization phase

Variables	Initialization	Description
e	Zero	The final value of e represents the number of vertices located on the mesh's boundary.
s	Zero	Tracks the number of S-type triangles encountered by the initialization phase.
ES-pair stack	Empty	Saves the (e, s) pairs, helps generate the S table.
S table	Empty	Saves the offset values for all the S-type triangles.

presented in Algorithm 12. Since the decoding initialization details for the M'-type triangle are omitted from [1], with the help of [18], the author of this report independently developed the algorithm for the M'-type triangles. The (ep, sp) in Algorithm 12 denotes the popped ES-pair value, and S[sp] denotes the spth entry in the S table.

3.5.2 Decompression Generation Phase

Having computed all the required quantities, the decoder now enters the generation phase. In the generation phase, the decoder creates triangles in the same order as processed by the encoder and reconstructs the mesh's vertices. Each created triangle is represented by the indices of its three vertices, and the entire generation process is finished by bounding loop adjustments. The initial bounding loop is created as follows. The number of nodes in the initial loop is set to the final value of the external vertex counter e. The vertex index of the first node is assigned zero, and the vertex indices of the remaining nodes are assigned successive increments of one from the previous node's vertex index. For example, if an initial bounding loop contains N nodes, the vertex indices for these N nodes are 0, 1, 2, ..., N - 1.

Before proceeding further, we need to first introduce the variables of the generation phase. The S table, M table, M' table, vertex table and facet table are used in the generation phase.

Algorithm 12 Decoding initialization phase.

```
while Not all the op-code are processed by the initialization phase. do
  if op_code = C then
    e = e - 1;
  else if op_code = L or op_code = R then
    e = e + 1;
  else if op_code = S then
    e = e - 1;
    s = s + 1;
    ES_pair_stack.push(e, s); {push an entry to the ES-pair stack.}
    S.add($\emptyset$) {add an empty entry to the S table.}
  else if op_code = E then
    e = e + 3;
    if ES_pair_stack $\neq \emptyset$ then
      (ep, sp) = ES_pair_stack.pop(); {pop the ES-pair stack.}
      S[sp] = e - ep - 2; {update S table.}
    end if
  else if op_code = M then
    e = e - (len + 1); {len is the length of the hole.}
  else if op_code = M' then
    e = e - 1;
    while ES_pair_stack $\neq \emptyset$ do
      (offset, s_count) = offset_table.begin(); {obtain the beginning entry
      from the offset table.}
      offset_table.erase(offset_table.begin()); {update the offset table.}
      S[s_count] = offset {update S table.}
      ES_pair_stack.pop(); {pop the ES-pair stack.}
    end while
  end if
end while
```

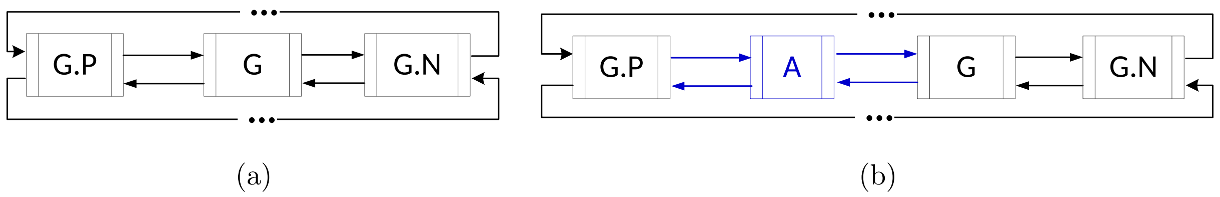


Figure 3.25: An example illustrating the C-type operation. (a) and (b) The bounding loop before and after the C-type operation, respectively.

The counter `c`, initialized to `e - 1`, is introduced to count the number of vertices that have been encountered by the generation phase. Furthermore, the S-type counter `s_cnt`, initialized to zero, is also maintained to count the number of S-type triangles that have been encountered by the generation phase. In what follows, we describe how each of the seven triangle types is handled during the generation operations.

C-type Triangle

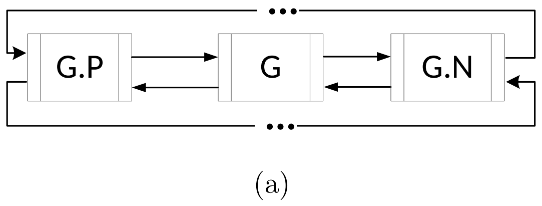
Recall that, in the encoder, a C-type triangle corresponds to the situation when the vertex opposite the active gate is not on the active bounding loop. The processing of such a triangle (in the encoder) results in the active bounding loop being modified by adding a new vertex. Therefore, when a C-type triangle is encountered in the decoder, a new node must be added to the active bounding loop. In particular, a new node `A` with the next available free index (i.e., `c + 1`) is added to the bounding loop just before the current position `G`. The updating of the bounding loop is illustrated in Figure 3.25. Figure 3.25(a) shows the original bounding loop with the active node `G`, and Figure 3.25(b) shows the updated bounding loop with the newly added node `A`. In addition to the above processing, the counter `c` (for vertices) is incremented and a new facet is added to the facet table with vertex indices `G.P.i`, `G.i`, and `A.i`. The above processing steps are shown in detail in Algorithm 13.

L-type Triangle

Recall that, in the encoder, an L-type triangle corresponds to the situation when the vertex opposite the active gate precedes the active gate on the bounding loop. The processing of such a triangle (in the encoder) results in the active bounding loop being modified by deleting a vertex. Therefore, when an L-type triangle is encountered in the decoder, a node must be deleted from the active bounding loop. In particular, the node `G.P` is deleted from the bounding loop, where `G` denotes the active node. The updating of the bounding loop is illustrated in Figure 3.26. Figure 3.26(a) shows the original bounding loop with the active node `G`, and Figure 3.26(b) shows the updated bounding loop (where the node `G.P` has been

Algorithm 13 C-type decoding operation.

```
1: {create new node A.}
2: New node A.i = c + 1;
3: {update facet table.}
4: facet_table.add(G.P.i, G.i, A.i)
5: {connect the G.P and A nodes.}
6: G.P.N = A;
7: A.P = G.P;
8: {connect the A and G nodes.}
9: A.N = G;
10: G.P = A;
11: {update vertex count c.}
12: c = c + 1;
```



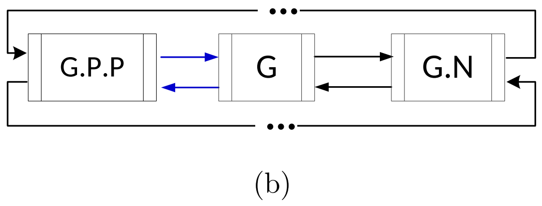


Figure 3.26: An example illustrating the L-type operation. (a) and (b) The bounding loop before and after the L-type operation, respectively.

deleted). In addition to the above processing, a new facet is added to the facet table with vertex indices `G.P.i`, `G.i`, and `G.P.P.i` (before node `G.P` is deleted). The above processing steps are shown in detail in Algorithm 14.

R-type Triangle

Recall that, in the encoder, an R-type triangle corresponds to the situation when the vertex opposite the active gate follows the active gate on the bounding loop. The processing of such a triangle (in the encoder) results in the active bounding loop being modified by deleting a vertex. Therefore, when an R-type triangle is encountered in the decoder, a node must be deleted from the active bounding loop. In particular, the active node `G` is deleted

Algorithm 14 L-type decoding operation.

```
1: {update facet table.}
2: facet_table.add(G.P.i, G.i, G.P.P.i)
3: {connect the G.P.P and G nodes.}
4: G.P.P.N = G;
5: G.P = G.P.P;
```

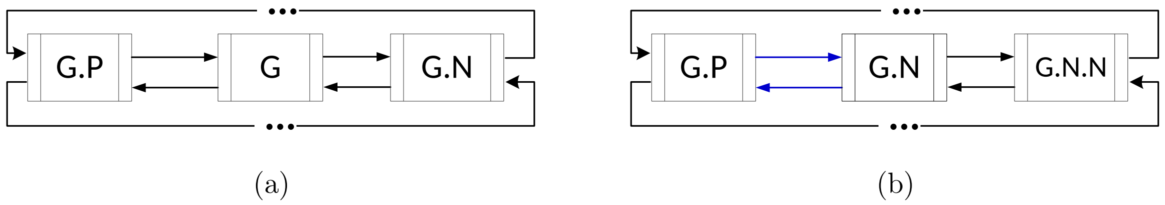


Figure 3.27: An example illustrating the R-type operation. (a) and (b) The bounding loop before and after the R-type operation, respectively.

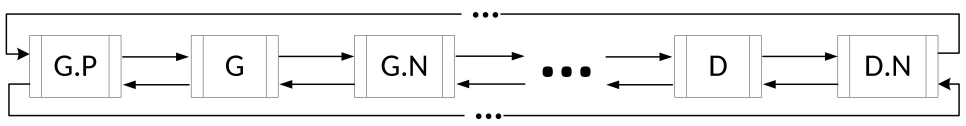
Algorithm 15 R-type decoding operation.

```
1: {update facet table.}
2: facet_table.add(G.P.i, G.i, G.N.i)
3: {connect the G.P and G.N nodes.}
4: G.P.N = G.N;
5: G.N.P = G.P;
6: {update active node.}
7: G = G.N;
```

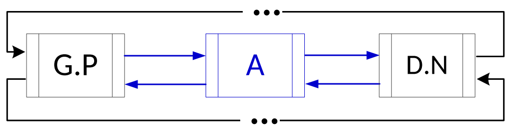
from the bounding loop. The updating of the bounding loop is illustrated in Figure 3.27. Figure 3.27(a) shows the original bounding loop with the active node G, and Figure 3.27(b) shows the updated bounding loop (where the node G has been deleted). A new facet is added to the facet table with vertex indices G.P.i, G.i, and G.N.i (before the node G is deleted). In addition to the above processing, the node G.N (before the node G is deleted from the bounding loop) becomes the active node in the following process. The above processing steps are shown in detail in Algorithm 15.

S-type Triangle

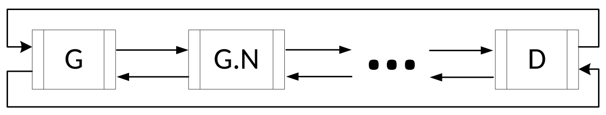
Recall that, in the encoder, an S-type triangle corresponds to the situation when the vertex opposite the active gate neither precedes nor follows the active gate on the bounding loop. The processing of such a triangle (in the encoder) results in the active bounding loop being split into left and right submeshes. Therefore, when an S-type triangle is encountered in the decoder, the active bounding loop is also split into two subloops. The decoder first finds the split node (i.e., the node D) of the current S-type triangle. Next, a new node A with the the same vertex index as the node D is added to the bounding loop just before the current position G. This node A is then pushed to the node stack and will be used as the active node when the decoder processes the left submesh. Next, the bounding loop is split into the two subloops. The updating of the bounding loop is illustrated in Figure 3.28. Figure 3.28(a) shows the original bounding loop with the active node G, while Figures 3.28(b) and (c) show



(a)



(b)



(c)

Figure 3.28: An example illustrating the S-type operation. (a) The bounding loop before the S-type operation. (b) and (c) The left and right submeshes' bounding loops after the S-type operation, respectively.

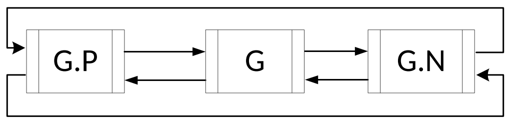


Figure 3.29: The bounding loop before the E-type operation.

the left and right subloops after the S-type operation, respectively. In addition to the above processing, a new facet is added to the facet table with vertex indices `G.P.i`, `G.i`, and `D.i`. The above processing steps are shown in detail in Algorithm 16.

E-type Triangle

Recall that, in the encoder, an E-type triangle corresponds to the situation when the vertex opposite the active gate both precedes and follows the active gate on the bounding loop. The processing of such a triangle (in the encoder) results in the entire active bounding loop being deleted and an inactive bounding loop being popped from the halfedge stack (presuming the stack is not empty). Therefore, when an E-type triangle is encountered in the decoder, the active bounding loop is deleted. Figure 3.29 shows the bounding loop with the active node `G` before the E-type operation. If the current E-type triangle is the last op-code in the op-code sequence, the entire decoding procedure is finished. Otherwise, a node is popped from the node stack and used as the active node for subsequent processing. In addition to the above processing, a new facet is added to the facet table with vertex indices `G.P.i`, `G.i`, and `G.N.i` (before the nodes are deleted). The above processing steps are shown in detail in Algorithm 17.

Algorithm 16 S-type decoding operation.

```
1: {update S-type count count s_cnt.}
2: s_cnt = s_cnt + 1;
3: {initialize candidate for node D.}
4: D = G.N;
5: Repeat D = D.N; for s_cnt times
6: {update facet table.}
7: facet_table.add(G.P.i, G.i, D.i)
8: {create new node A.}
9: New node A.i = D.i;
10: {connect the G.P and A nodes.}
11: G.P.N = A;
12: A.P = G.P;
13: {connect the A and D.N nodes.}
14: A.N = D.N;
15: D.N.P = A;
16: {push the node A to the node stack.}
17: node_stack.push(A)
18: {connect the D and G nodes.}
19: D.N = G;
20: G.P = D;
```

Algorithm 17 E-type decoding operation.

```
1: {update facet table.}
2: facet_table.add(G.P.i, G.i, G.N.i)
3: {delete the G.P, G, G.N nodes from bounding loop.}
4: Delete G.P, G, G.N;
5: if node_stack $\neq \emptyset$ then
6:    G = node_stack.pop(); {pop the active stack.}
7: else
8:    Decoding process end.
9: end if
```

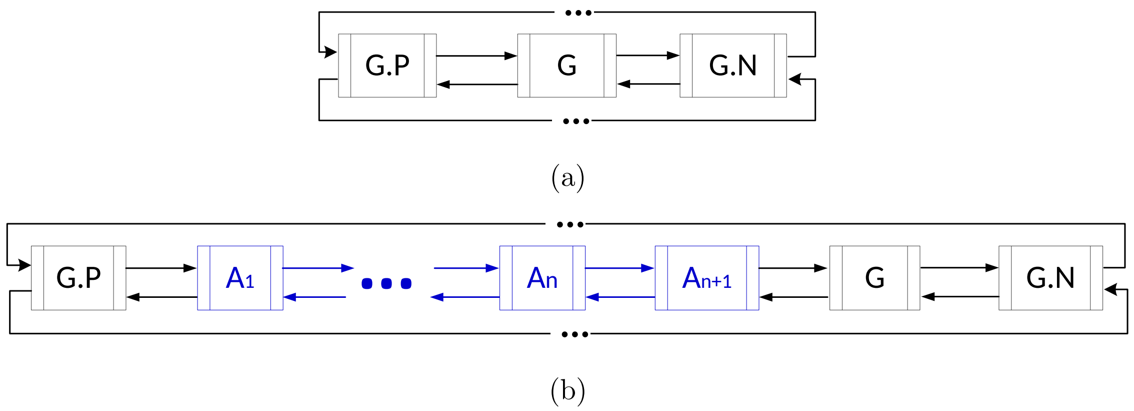


Figure 3.30: An example illustrating the M-type operation. (a) and (b) The bounding loop before and after the M-type operation, respectively.

M-type Triangle

Recall that, in the encoder, an M-type triangle corresponds to the situation when the vertex opposite the active gate belongs to a hole's boundary. The processing of such a triangle (in the encoder) results in the active bounding loop being modified by merging with the hole's boundary. Therefore, when an M-type triangle is encountered in the decoder, a hole's boundary is merged with the active bounding loop. To begin, a new facet is added to the facet table with vertex indices `G.P.i`, `G.i`, and `c + 1`. Next, the hole's length `len` is first read from the M table, and then `len + 1` nodes are inserted into the bounding loop just before the current position `G`. The updating of the bounding loop is illustrated in Figure 3.30. Figure 3.30(a) shows the original bounding loop with the active node `G`, and Figure 3.30(b) shows the updated bounding loop with the newly added nodes $\mathtt{A}_1$ to $\mathtt{A}_{\mathtt{N+1}}$, where `N` equals to the hole's length `len`. The above processing steps are shown in detail in Algorithm 13.

M'-type Triangle

Recall that, in the encoder, an M'-type triangle corresponds to the situation when the vertex opposite the active gate belongs to an inactive bounding loop. The processing of such a triangle (in the encoder) results in the active bounding loop being merged with the inactive bounding loop. Therefore, when an M'-type triangle is encountered in the decoder, an inactive bounding loop is merged with the active bounding loop. To begin, the merging node (i.e., the node `D`) is found from the node stack. Next, a node `A` with the same vertex index as the node `D` is added to the bounding loop just before the current position `G`. Then, the inactive bounding loop is merged into the active bounding loop by various link updates. The updating of the bounding loop is illustrated in Figure 3.31. Figure 3.31(a) shows the

Algorithm 18 M-type decoding operation.

```
{update facet table.}
facet_table.add(G.P.i, G.i, c + 1)
{initialize the node T.}
T = G.P;
repeat
    {update vertex count c.}
    c = c + 1;
    {create new node A.}
    New node A.i = c;
    {connect the T and A nodes.}
    T.N = A;
    A.P = T;
    {update node T.}
    T = A;
until len times;
{create new node A.}
New node A.i = c - len + 1;
{connect the T and A nodes.}
T.N = A;
A.P = T;
{connect the A and G nodes.}
A.N = G;
G.P = A;
```

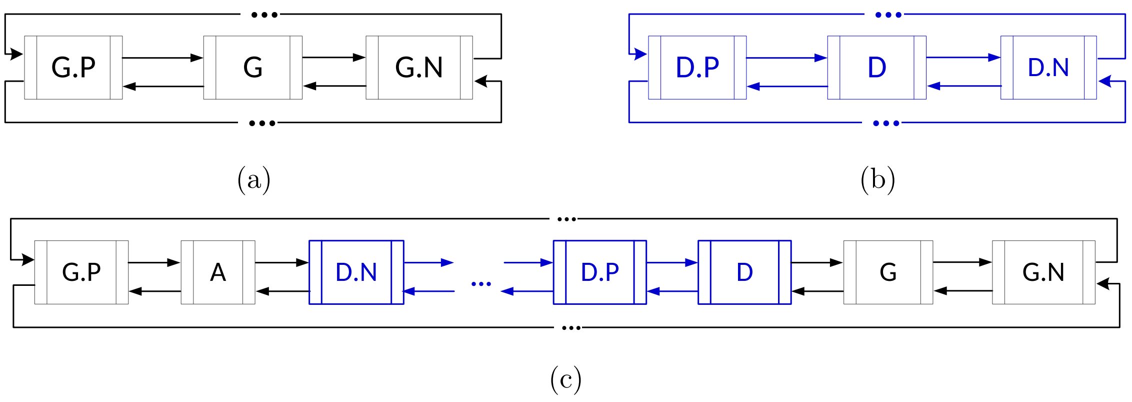


Figure 3.31: An example illustrating the M'-type operation. (a) The bounding loop before the M'-type operation. (b) The bounding loop needs to be merged. (c) The bounding loop after the M'-type operation.

active bounding loop with the active node `G`, and Figure 3.31(b) shows the inactive bounding loop with the node `D`. Figure 3.31(c) shows the bounding loop after updating (i.e., with the merged inactive bounding loop and the newly added node `A`). In addition to the above process, a new facet is added to the facet table with vertex indices `G.P.i`, `G.i`, and `D.i`. The above processing steps are shown in detail in Algorithm 19.

Mesh Generation

Having introduced the decoding operations for each triangle type thoroughly, we can now present the triangle mesh generation process. The decoded triangle mesh is simply generated by the vertex and the facet tables.

3.6 Examples of the Decoding Method

Having fully introduced the decoding process, we now provide some examples to better illustrate the decoding details. These mesh examples are continuations of the encoding examples from Section 3.3 (on page 34). Tables are used to show certain key calculations in the initialization phase. The triangle creation process is also described.

3.6.1 Example 1: Mesh with One Bounding Loop and No Handles

The first coding example is a continuation of the example in Section 3.3.1 (on page 34). Figure 3.32(a) shows the triangle mesh, and Figure 3.32(b) shows the steps in the Edgebreaker encoding process. The op-code sequence CRRSLLECCRRRCRRRSREE is transmitted to

Algorithm 19 M'-type decoding operation.

```
1: {fetch and remove the required node.}
2: D = remove(position);
3: {find the node D.}
4: Repeat D = D.N; for offset_m times
5: {update facet table.}
6: facet_table.add(G.P.i, G.i, D.i)
7: {create new node A.}
8: New node A.i = D.i;
9: {connect the G.P and A nodes.}
10: G.P.N = A;
11: A.P = G.P;
12: {connect the A and D.N nodes.}
13: A.N = D.N;
14: D.N.P = A;
15: {connect the D and G nodes.}
16: D.N = G;
17: G.P = D;
```

Table 3.2: Calculation of the `e`, `s`, and `S` parameters in the initialization phase

	C	R	R	S	L	L	E	C	C	R	R	R	C	R	R	R	S	R	E	E
`e`	-1	0	1	0	1	2	5	4	3	4	5	6	5	6	7	8	7	8	11	14
`s`	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
`S[s]`							3												2	

the decoder side. Since this example is a mesh without handles or holes, the offset table, M table, and M' table are all empty. The offset value in the S table can be calculated directly from the op-code sequence. Moreover, since there are no M- or M'-types triangles in the mesh, the op-code sequence preprocessing step is omitted.

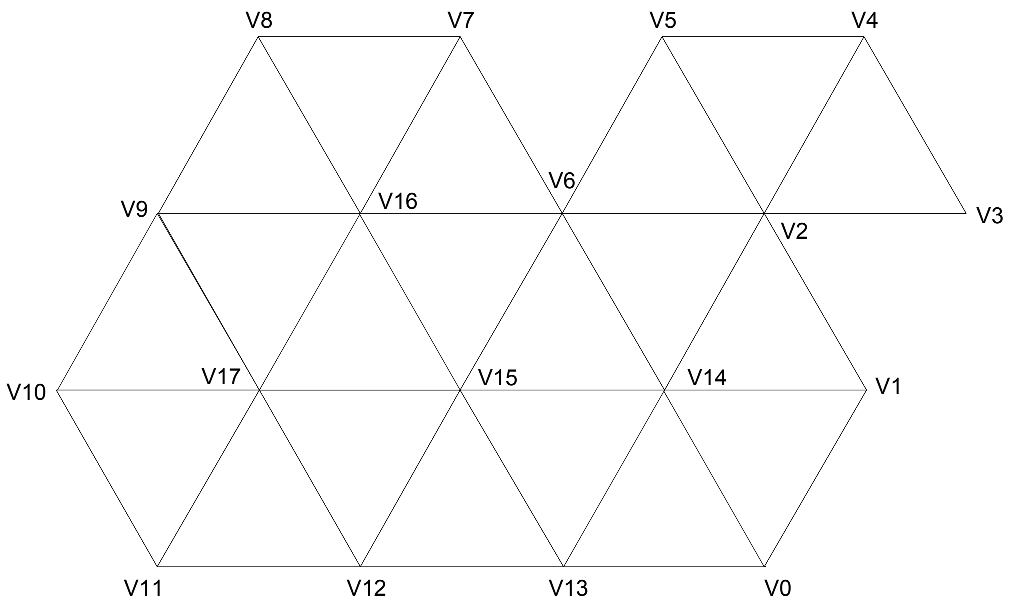
Table 3.2 shows the calculation of the `e`, `s`, and `S` parameters in the initialization phase. Note that the decoder first enters the offset value for the first S-type triangle (i.e., `S[1]`)

$$\texttt{S[1]} = 5 - 0 - 2 = 3, \tag{3.9}$$

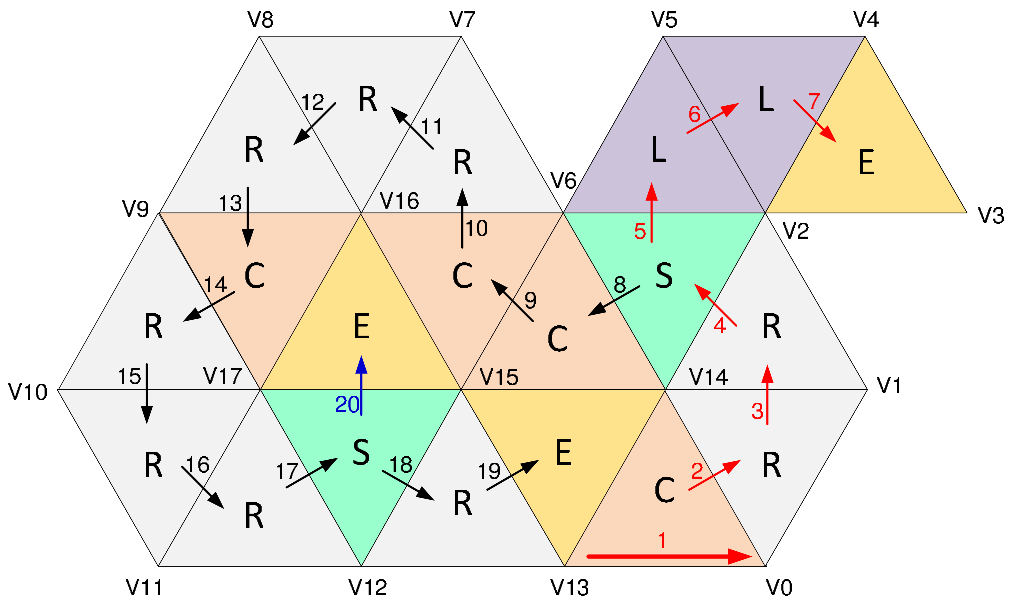
then enters the offset value for the second S-type triangle (i.e., `S[2]`)

$$\texttt{S[2]} = 11 - 7 - 2 = 2. \tag{3.10}$$

The final value for `e` corresponds to the number of vertices located on the mesh's boundary (i.e., 14).



(a)



(b)

Figure 3.32: A triangle mesh with one bounding loop and no handles. (a) The triangle mesh with boundary. (b) Steps in the Edgebreaker encoding process.

Given the external vertex count `e` = 14, the decoder starts the generation phase with a 14 node initial bounding loop with vertex indices 0, 1, 2, ..., 13, as shown in Figure 3.33(a). The counter `c` (for vertices) is initialized to 13. The first C-type operation creates a triangle with vertex indices 13, 0, 14. A node with index 14 is added to the bounding loop, as shown in Figure 3.33(b). The counter `c` (for vertices) is then updated to 14. The active node after the first C-type triangle operation is still the node with index 0. The three vertices in the first C-type triangle are reconstructed and added to the vertex table. Cases 1, 2, and 3 of the parallelogram-prediction scheme in Section 3.4 (on page 41) are used to predict the positions of these vertices.

The second R-type operation creates a triangle with vertex indices 14, 0, 1 and deletes the node with index 0 from the bounding loop, as shown in Figure 3.33(c). The node with index 1 becomes the active node in the subsequent processing. Proceeding in a similar way, the vertex position of the node with index 1 is reconstructed and stored in the vertex table. The third R-type operation creates a triangle with vertex indices 14, 1, 2 and updates the bounding loop. The resulting bounding loop contains 13 nodes with vertex indices 2, 3, 4, ..., 14, as shown in Figure 3.33(d). Since the fourth triangle is S-type and the offset value is `S[1]` = 3, the S-type operation skips the three nodes with indices 3, 4, and 5 in the bounding loop, and creates a triangle with vertex indices 14, 2, 6. Next, the bounding loop is split into two subloops contain nodes with vertex indices 6, 7, 9, ..., 14 and 2, 3, 4, 5, 6, as shown in Figures 3.33(e) and (f), respectively. Next, the node with index 6 is pushed to the node stack, and the node with index 2 becomes the active node during subsequent processing.

The first triangle of the right submesh is L-type. L-type operation creates a triangle with vertex indices 6, 2, 5 and deletes the node with index 6 from the bounding loop, as shown in Figure 3.33(g). The second L-type operation creates a triangle with vertex indices 5, 2, 3 and deletes the node with index 5. The resulting bounding loop after decoding the second L-type triangle is shown in Figure 3.33(h). The seventh triangle is E-type. The E-type operation creates a triangle with vertex indices 4, 2, 3. This E-type triangle the last triangle in the current stream, therefore, the node with index 6 is popped from the node stack. The remaining portion of the op-code sequence can be processed in a similar way.

3.6.2 Example 2: Mesh with Zero Bounding Loops and No Handles

The second coding example is the continuation of the example in Section 3.3.2 (on page 36). Figures 3.34(a) and (b) show the original mesh and a cut and flattened view of the mesh, respectively. Figure 3.34(c) shows the steps in the Edgebreaker encoding process. The op-code sequence transmitted to the decoder side is CCRE. The offset table, M table, and

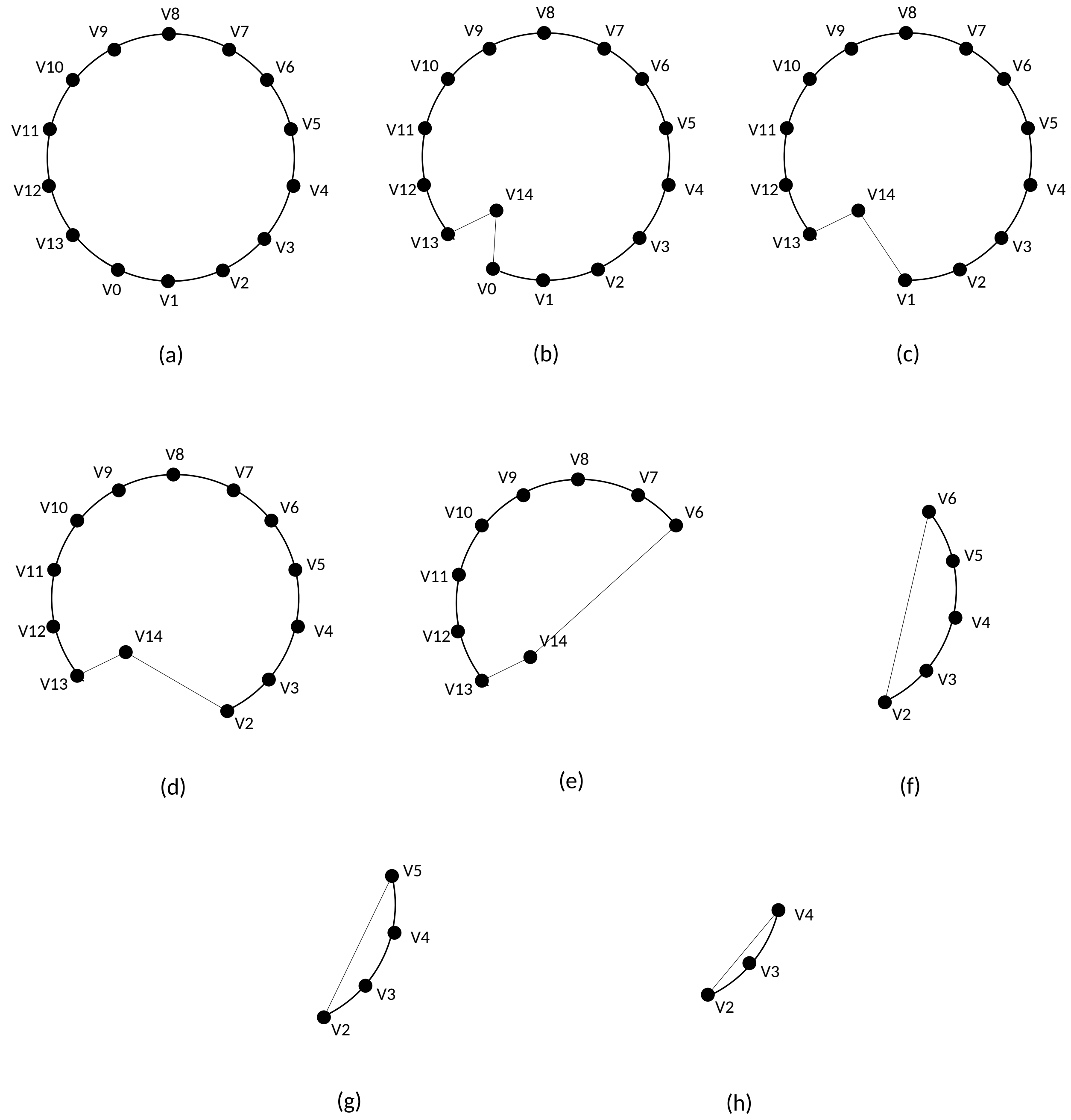


Figure 3.33: The internal steps of the Edgebreaker decoding process. (a) The initial bounding loop. (b) The resulting bounding loop after decoding the first C-type triangle. (c) and (d) The resulting bounding loops after decoding the first and second R-type triangles, respectively. (e) and (f) The resulting left and right submeshes' bounding loops after decoding the S-type triangle, respectively. (g) and (h) The resulting bounding loops after decoding the first and second L-type triangles, respectively.

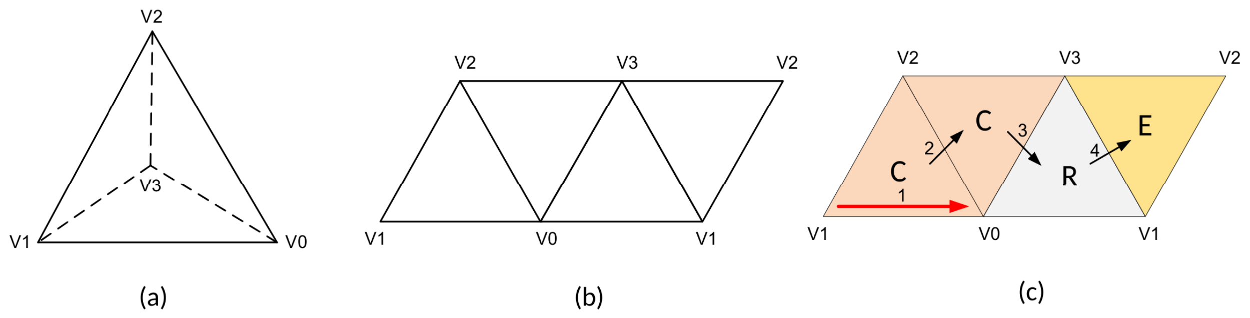


Figure 3.34: A triangle mesh with zero bounding loops (i.e., closed) and no handles. (a) The tetrahedral mesh. (b) A cut and flattened view of the tetrahedral mesh. (c) Steps in the Edgebreaker encoding process.

Table 3.3: Calculation of the e parameter in the initialization phase

	C	C	R	E
e	-1	-2	-1	2

M' table are all empty in this example. Moreover, this example does not have any S-type triangles. Therefore, the S table is also empty.

Table 3.3 shows the calculation of the e parameter in the initialization phase. From Table 3.3, we know that the initial bounding loop in the generation phase only contains two nodes. This observation fits the operations performed in the encoding process. As explained earlier, for a mesh without bounding loops (i.e., closed), an arbitrary halfedge can be chosen as the initial active gate, which forms the initial bounding loop. In this example, the halfedge $\overrightarrow{V_1V_0}$ is chosen as the initial active gate (in the encoder), as shown in Figure 3.34(c). The generation phase starts with a two node initial bounding loop with vertex indices 0, 1, as shown in Figure 3.35(a). The counter c (for vertices) is initialized to 1. The first C-type operation creates a triangle with vertex indices 1, 0, 2 and inserts a node with index 2 to the bounding loop. After the C-type operation, the bounding loop contains three nodes with vertex indices 2, 0, 1, as shown in Figure 3.35(b). The counter c (for vertices) is then set to 2, and the active node is still the node with index 0. Then, the three vertices in the first triangle are reconstructed and added to the vertex table. Proceeding in a similar way, the triangle mesh in Figure 3.34(a) can be easily decoded.

In order to provide readers with a better visualization about the internal stages of the coding process, two additional diagrams are included in Figures 3.35(c) and (d). These two figures show the resulting bounding loops after decoding the second C-type triangle and the R-type triangle, respectively.

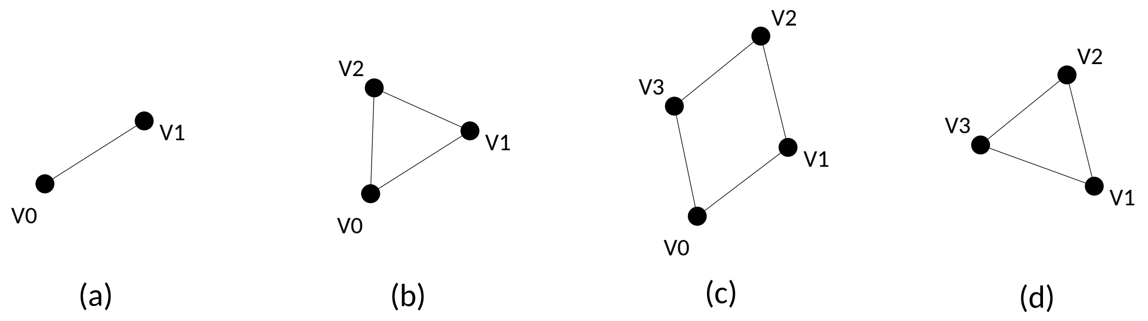


Figure 3.35: The internal steps of the Edgebreaker decoding process. (a) The initial bounding loop. (b) and (c) The resulting bounding loops after decoding the first and second C-type triangles, respectively. (d) The resulting bounding loop after decoding the first R-type triangle.

Table 3.4: Calculation of the `e`, `s`, and `S` parameters in the initialization phase

	S	M	R	L	R	R	L	R	L	R	R	R	L	R	E	C	R	R	R	R	E
`e`	-1	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	8	7	8	9	10	11	14
`s`	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
`S[s]`															7						

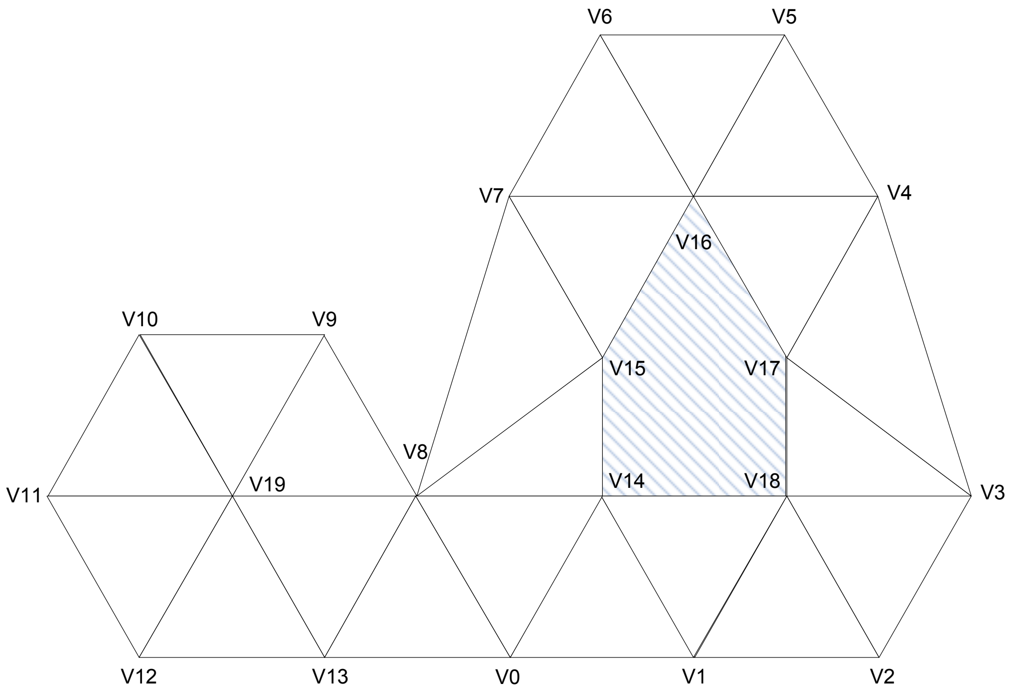
3.6.3 Example 3: Mesh with Two Bounding Loops and No Handles

The third coding example is the continuation of the example in Section 3.3.3 (on page 37). Figure 3.36(a) shows the input mesh, while Figure 3.36(b) shows the steps in the Edgebreaker encoding process. The op-code sequence SSRLRRLRLRRRLRECRRRRE is transmitted to the decoder side. In addition to the op-code sequence and the coded prediction residuals, the (non-empty) M table is also transmitted. Therefore, the op-code sequence preprocessing step is performed to distinguish between the S- and M-types triangles. The op-code sequence obtained after the op-code sequence preprocessing step is SMRLRRLRLRRRLRECRRRRE.

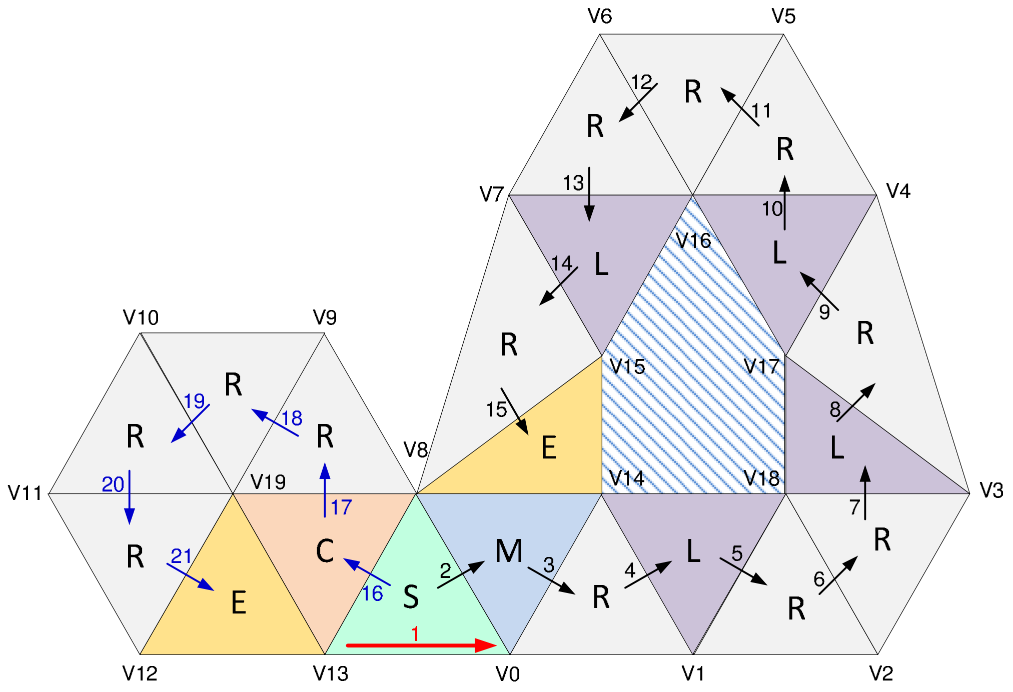
Table 3.4 shows the calculation of the `e`, `s`, and `S` parameters in the initialization phase. The hole's length read from the M table is five. Therefore, the decoder subtracts six (i.e., the hole's length plus one) from the external vertex count `e` when processing the op-code M in the initialization phase. Next, the decoder calculates the offset value for the first S-type triangle (i.e., `S[1]`) as

$$\texttt{S[1]} = 8 - (-1) - 2 = 7. \tag{3.11}$$

The generation phase starts with a 14 node initial bounding loop with vertex indices 0, 1, 2, ..., 13, as shown in Figure 3.37(a). The counter `c` (for vertices) is set to 13. Since the offset value of the first S-type triangle is `S[1]` = 7, seven nodes in the bounding loop are skipped. In particular, the first S-type triangle creates a triangle with vertex indices 13, 0, 8. Then, the bounding loop is split into left and right subloops containing nodes with vertex



(a)



(b)

Figure 3.36: A triangle mesh with two bounding loops (i.e., one hole) and no handles. (a) The triangle mesh with a hole. (b) Steps in the Edgebreaker encoding process.

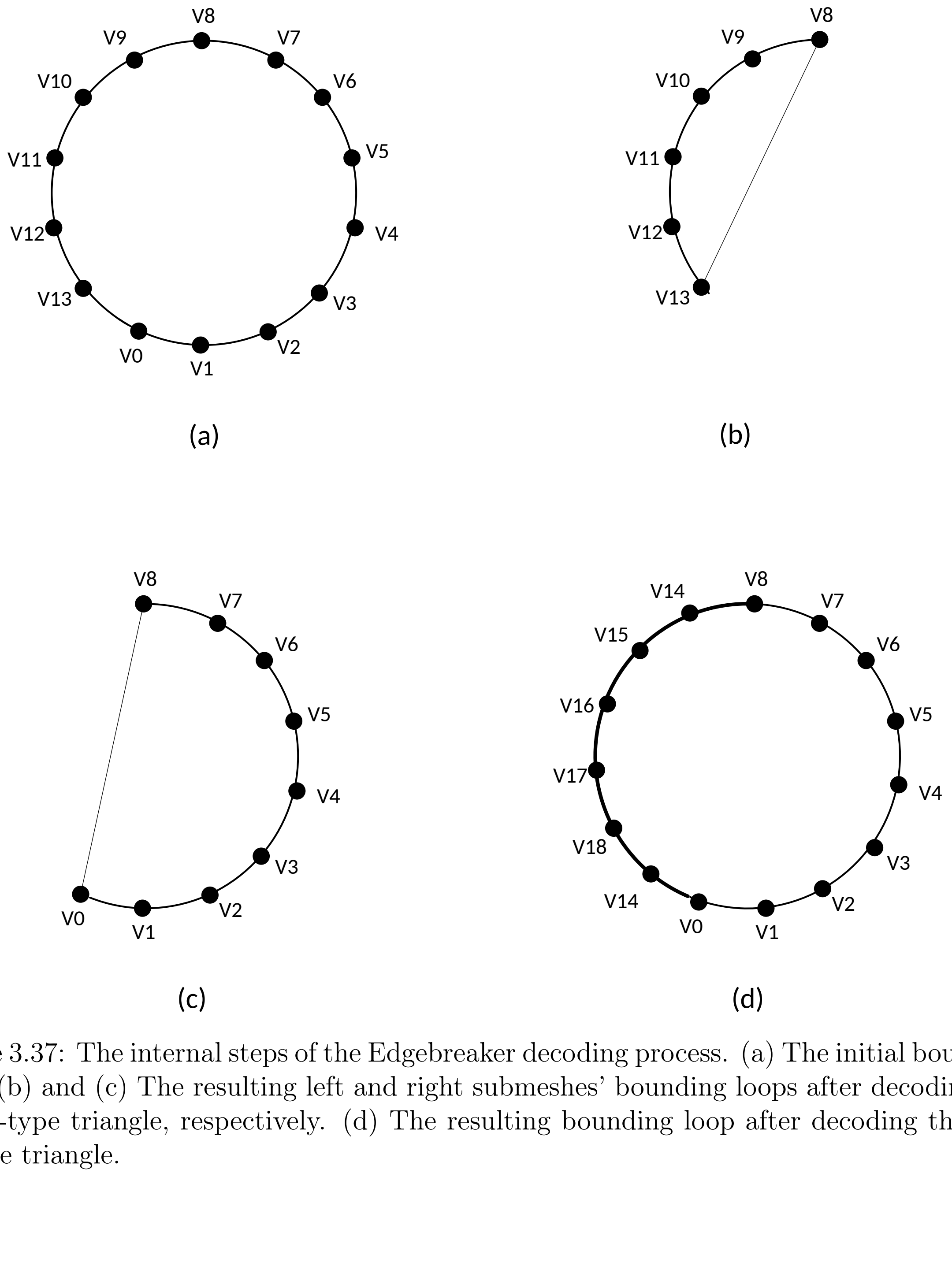


Figure 3.37: The internal steps of the Edgebreaker decoding process. (a) The initial bounding loop. (b) and (c) The resulting left and right submeshes' bounding loops after decoding the first S-type triangle, respectively. (d) The resulting bounding loop after decoding the first M-type triangle.

Table 3.5: Calculation of the `e`, `s`, and `S` parameters in the initialization phase

	C	C	C	C	C	R	C	C	C	R	C	C	R	C	C	R
`e`	-1	-2	-3	-4	-5	-4	-5	-6	-7	-6	-7	-8	-7	-8	-9	-8
`s`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`S[s]`																

	S	C	R	M’	C	R	S	R	L	S	E	E	R	S	E	E
`e`	-9	-10	-9	-10	-11	-10	-11	-10	-9	-10	-7	-4	-3	-4	-1	2
`s`	1	1	1	1	1	1	2	2	2	3	3	3	3	4	4	4
`S[s]`	3										1	5			1	

indices 8, 9, ..., 13 and 0, 1, ..., 8, respectively. The resulting bounding loops after the S-type operation are shown in Figures 3.37(b) and (c). The node with index 8 from the left subloop is pushed to the node stack, and the active node for the right subloop is the node with index 0.

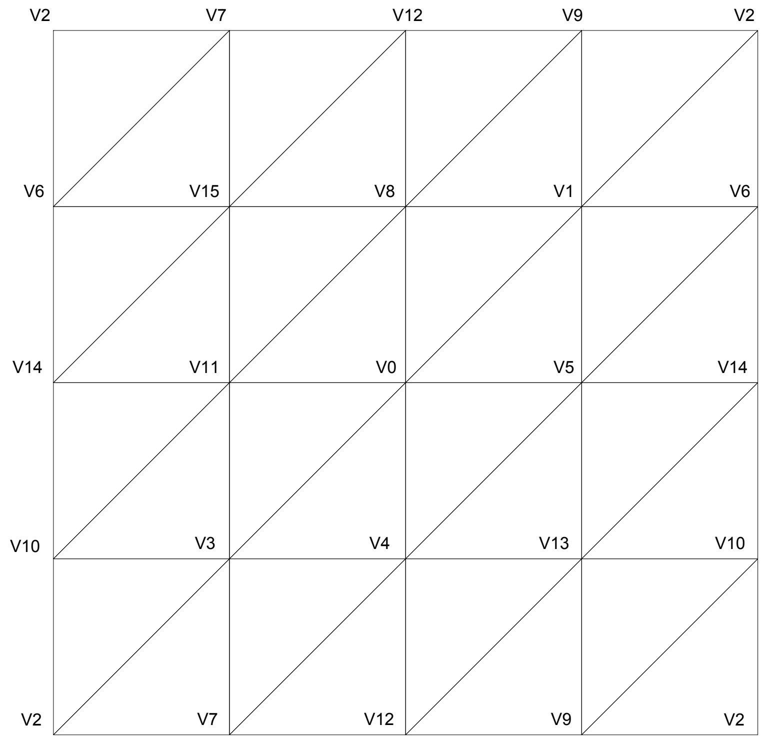
The second M-type triangle creates a triangle with vertex indices 8, 0, 14. Five nodes with indices 14, 15, 16, 17, and 18 are inserted into the bounding loop just before the node with index 0. Next, a node with index 14 is created and added to the bounding loop between the nodes with vertex indices 18 and 0. The bounding loop obtained after the nodes insertion contains nodes with vertex indices 0, 1, ..., 8, 14, 15, ..., 18, 14, as shown in Figure 3.37(d). The counter `e` (for vertices) is then updated to 18. Next, the R-type operation creates a triangle with vertex indices 14, 0, 1 and deletes the node with index 0 from the bounding loop. Proceeding in a similar way, the triangle mesh shown in Figure 3.36(a) can be successfully decoded.

3.6.4 Example 4: Mesh with Zero Bounding Loops and One Handle

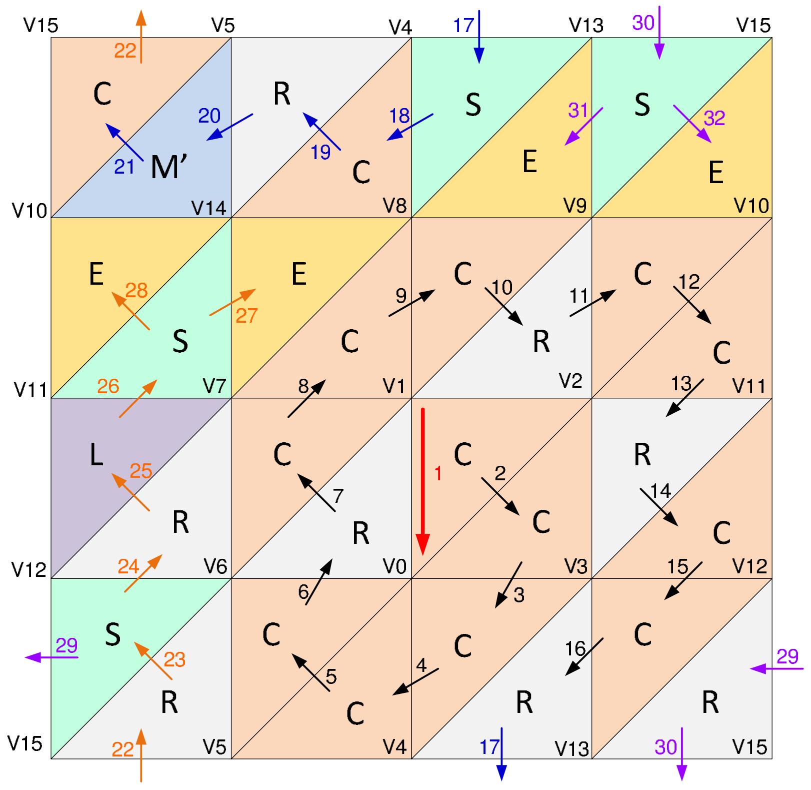
The last coding example is the continuation of the example in Section 3.3.4 (on page 39). Figure 3.38(a) shows a planar graph of the torus mesh, and Figure 3.38(b) shows the steps in the Edgebreaker encoding process. Since the torus mesh contains a handle, the offset table and M’ table in the transmitted file are not empty. The op-code sequence transmitted to the decoder side is CCCCCRCCCRCCRCCRSCRSCRSRLSEERSEE. The op-code sequence obtained after the op-code sequence preprocessing step is CCCCCRCCCRCCRC-CRSCRM’CRSRLSEERSEE.

Table 3.5 shows the calculation of the `e`, `s`, and `S` parameters in the initialization phase. Note that the offset value `S[1]` $= 3$ is directly read from the offset table. The decoder first enters the offset value `S[3]` $= 1$, then enters `S[2]` $= 5$, and then enters `S[4]` $= 1$.

In the generation phase, we focus on the processing of the M’-type triangle. Proceeding



(a)



(b)

Figure 3.38: A triangle mesh with zero bounding loops and one handle. (a) The triangle mesh with a handle. (b) Steps in the Edgebreaker encoding process.

in a similar way as the other examples, the first S-type operation splits the active bounding loop into left and right subloops containing the nodes with vertex indices 8, 9, ..., 13 and 4, 5, ..., 8, respectively. The resulting bounding loops after the S-type operation are shown in Figures 3.39(a) and (b). In addition, the node with index 8 is pushed to the node stack. The next two op-codes, C and R, update the right subloop to contain nodes with vertex indices 5, 6, 7, 8, 14. The resulting bounding loops after decoding the C- and R-types triangles are shown in Figures 3.39(c) and (d), respectively. Then, the decoder encounters the op-code M'. To begin, the `D` node (i.e., the node with index 8) is found from the node stack. The pseudocode `D = D.N;` is then repeated `offset_m` times to obtain the final `D` node (i.e., the node with index 10). As explained earlier, `offset_m` is the offset value read from the M' table (i.e., `offset_m` = 2). Next, a node with same vertex index as the `D` node (i.e., 10) is added to the bounding loop between the nodes with vertex indices 14 and 11. Then, the nodes with vertex indices 10 and 5 are connected. In this way, an active inactive bounding loop is merged with the active bounding loop. The resulting bounding loop contains nodes with vertex indices 14, 10, 11, 12, 13, 8, 9, 10, 5, 6, 7, 8, as shown in Figure 3.39(e). The active node in the subsequent processing is still the node with index 5. The remaining portion of the op-code sequence, which now contains CRSRLSEERSEE, is processed similarly.

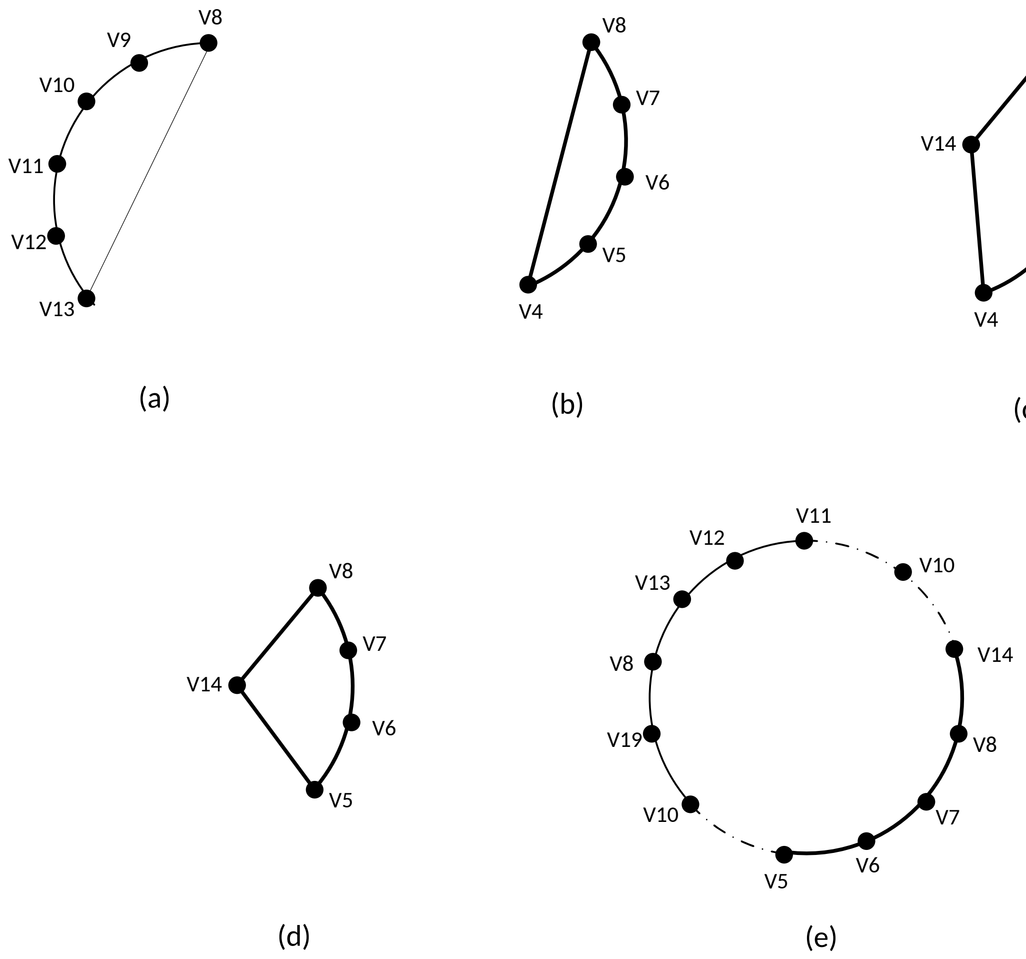


Figure 3.39: The internal steps of the Edgebreaker decoding process. (a) and (b) The resulting left and right submeshes' bounding loops after decoding the S-type triangle, respectively. (c) and (d) The resulting bounding loops after decoding the C-type and R-type triangles, respectively. (e) The resulting bounding loop after decoding the M'-type triangle.

Chapter 4

Software

4.1 Introduction

Having studied the Edgebreaker mesh-coding method thoroughly in the previous chapter, we now introduce the software implementation. The main contribution of this project is the development of a software implementation of the Edgebreaker mesh-coding method. This implementation is written in C++ language and consists of more than 6000 lines of code. In addition to the C++ standard library, the software makes heavy use of CGAL [15] and the Signal Processing Library (SPL) [20].

The software consists of two programs, `encode_mesh` and `decode_mesh`, to provide functionalities of the Edgebreaker mesh-coding method. The `encode_mesh` program performs 3-D triangle mesh encoding, given a triangle mesh in OFF format. The coded triangle mesh is stored in EB format, whose details will be introduced shortly. The `decode_mesh` program performs 3-D triangle mesh decoding. Given a coded triangle mesh in EB format, this program produces the decompressed triangle mesh and outputs the mesh in OFF format.

The remainder of this chapter presents details on how to build and use the software. Several examples are also provided to illustrate the use of the software.

4.2 Building the Software

The author's mesh coding software should build on any Linux system with a C++ compiler compliant with C++14 standard. The compiler version the author has been using in software implementation is GCC 5. 1. 0. The user needs to guarantee that the CGAL and SPL libraries are installed prior to building the software. The following versions of libraries have been verified to work with our software:

- CGAL: 4.2.1
- SPL: 1.1.18.

The Make [21] utility is chosen to build (i.e., compile and link) the code. To build the code, the user needs to change the working directory to the software package folder first. Then, the user needs to delete any old object files and executable files by executing the command:

```
make clean
```

Finally, the user can build the executable programs by invoking the command:

```
make all
```

4.3 The `encode_mesh` Program

The `encode_mesh` program reads a triangle mesh in OFF format from standard input, generates the coded mesh, and writes the compressed triangle mesh in EB format to standard output. The `encode_mesh` program can only encode triangle meshes having a single component. If an invalid mesh is given, the program will terminate with an error. With the command line options, the user can specify the quantization step size for x, y, and z coordinates. If the user specifies the `-d` option as a command line argument, the `encode_mesh` program will check and remove the duplicate vertices that are found in the mesh.

Synopsis

```
encode_mesh [Options]
```

Options

`-x $x_size`	Specifies the x vertex coordinate quantization step size to be `$x_size`. The default step size is $(x_{\max} - x_{\min})/2^{16}$, where $x_{\max}$ and $x_{\min}$ are the maximum and minimum x vertex coordinates in the input mesh.
`-y $y_size`	Specifies the y vertex coordinate quantization step size to be `$y_size`. The default step size is $(y_{\max} - y_{\min})/2^{16}$, where $y_{\max}$ and $y_{\min}$ are the maximum and minimum y vertex coordinates in the input mesh.
`-z $z_size`	Specifies the z vertex coordinate quantization step size to be `$z_size`. The default step size is $(z_{\max} - z_{\min})/2^{16}$, where $z_{\max}$ and $z_{\min}$ are the maximum and minimum z vertex coordinates in the input mesh.
`-b $bits`	Sets the number of bits used to encode each vertex coordinate to be `$bits`. The default value is 16. The quantization step size for each vertex coordinate can be determined by: $(\mathrm{coor}_{\max} - \mathrm{coor}_{\min})/2^{\$\mathtt{bits}}$, where $\mathrm{coor}_{\max}$ and $\mathrm{coor}_{\min}$ are the maximum and minimum vertex coordinates in the input mesh.
`-d`	Enables the "check and remove duplicate vertices function" in the program. Any duplicate vertex that is found by the `encode_mesh` program will be removed.
`-h`	Prints help information and exit.
`-r $results`	Specifies a results file to which the program will store the coded mesh's statistics information. All the statistics are printed on a single line in the file, separated by a single space. Items included in the results file are listed in Table 4.1.

Exit status

The program returns zero for a normal exit and a nonzero value otherwise.

4.4 The `decode_mesh` Program

The `decode_mesh` program performs mesh decoding. In general, the program reads an coded triangle mesh in EB format from standard input, produces a decompressed triangle mesh, and writes the decoded mesh in OFF format to standard output.

Table 4.1: Items included in the encoding result file

Variables	Description
Vertices	Number of vertices in the mesh
Edges	Number of edges in the mesh
Facets	Number of facets in the mesh
Boundaries	Number of bounding loops in the mesh
Genus	Genus of the mesh
Data size	Number of total bytes of coded data
Geometry	Number of bytes of coded geometry data
Connectivity	Number of bytes of coded connectivity data
Time	Time in seconds needed for mesh encoding
Memory	Maximum amount of memory used by the encoding program
Quantizer	The actual quantization step size used by the encoding program

Table 4.2: Items included in the decoding result file

Variables	Description
Vertices	Number of vertices in the mesh
Edges	Number of edges in the mesh
Facets	Number of facets in the mesh
Boundaries	Number of bounding loops in the mesh
Genus	Genus of the mesh
Data size	Number of total bytes of coded data
Geometry	Number of bytes of coded geometry data
Connectivity	Number of bytes of coded connectivity data
Time	Time in seconds needed for mesh decoding
Memory	Maximum amount of memory used by the decoding program
Quantizer	The actual quantization step size used by the decoding program

Synopsis

```
decode_mesh [Options]
```

Options

`-h`	Prints help information and exit.
`-r $results`	Specifies a results file to which the program will store the coded mesh's statistics information. All the statistics are printed on a single line in the file, separated by a single space. Items included in the results file are listed in Table 4.2.

Exit status

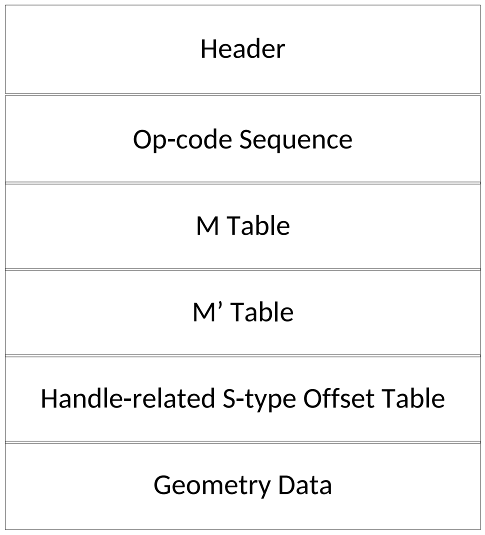


Figure 4.1: Structure of the EB file.

The program returns zero for a normal exit and a nonzero value otherwise.

4.5 EB File Format

The file format used in software for the coded triangle mesh data is known as the EB format. EB files contain six main parts, as shown in Figure 4.1. In what follows, we present the EB file structure in more detail.

Header. The header specifies the triangle mesh’s basic information. Eight elements are contained in the header part:

- The EB file signature.
- A single integer number that represents the code series adopted by the current EB file. Three code series are used by the EB format, and the code series details will be introduced shortly.
- The number of bytes used to store the binary op-code sequence.
- The number of holes in the mesh.
- The number of handles in the mesh.
- The number of handle-related S-type triangles in the mesh.
- The number of bits used to encode the x, y, and z vertex coordinates.

Table 4.3: Codewords for three binary code series

Op-code	Code 1	Code 2	Code 3
C_A	0	0	0
S_A	10	10	10
R_A	11	11	11
C_N	0	00	00
S_N	100	111	010
R_N	101	10	011
L	110	110	10
E	111	01	11

- The quantization step size for the x, y, and z vertex coordinates.

Op-code Sequence. The op-code sequence specifies the mesh's connectivity information. This part contains the binary op-code sequence generated by the `encode_mesh` program. According to King and Rossignac's paper in [22], the Edgebreaker method adopts three alternative prefix binary code series. One of the three code series is used by the current EB file. The code series is determined by the mesh's connectivity characteristics. The codewords for each code series are listed in Table 4.3. For each code series, the notation C_A denotes the situation where a C op-code immediately follows another C op-code, and C_N denotes otherwise. The similar notation is used for S and R op-codes.

M Table. The M table stores the hole information. If the triangle mesh contains no holes, the M table is empty. Two elements are included for each hole. The first is the number of S-type triangles encountered since the previous M-type triangle or since the beginning of the op-code sequence for the first M-type triangle. The second is the number of vertices located on the hole's boundary.

M' Table. The M' table specifies the handle information. If the triangle mesh contains no handles, the M' table is empty. As explained in the description of the M'-type encoding operation (on page 27), for each handle, three elements are included: 1) the position of the representative halfedge in the halfedge stack, 2) the offset value for the M'-type triangle, and 3) the number of S-type triangles encountered since the previous M'-type triangle or since the beginning of the op-code sequence for the first M'-type triangle.

Handle-related S-type Offset Table. The handle-related S-type offset table stores the handle-related S-type triangle's offset values. As explained earlier, the handle-related S-type triangles are related to handles (i.e., the M'-type triangles). If the coded mesh contains no handles, the offset table is empty. Two elements are included for each offset table entry. The first is the S-type triangle's counter, and the second is the offset value.

Geometry Data. The geometry data specifies the mesh's vertices information. For each vertex, the prediction residual between the actual and the predicted vertex positions is binarized and encoded by the arithmetic coder.

4.6 Software Usage Examples

The programs in our software package have been introduced thoroughly in the previous sections. In what follows, several examples are provided to illustrate the use of our software.

Example 1A. Suppose that the user wants to encode a triangle mesh stored in a file named `bunny.off` with following requirements:

- The number of bits used to encode each coordinate is set to 8.
- The coded triangle mesh's result information is stored in the file `enc_result.txt`.
- The coded triangle mesh is stored in the file `bunny.eb`.

The above task can be accomplished by running the `encode_mesh` program as follows:

```
encode_mesh -b8 -r enc_result.txt < bunny.off > bunny.eb
```

Example 1B. Suppose that the user wants to decode the mesh that is generated in Example 1A with following requirements:

- The decoded triangle mesh's statistic information is stored in the file `dec_result.txt`.
- The decoded triangle mesh is stored in the file `bunny_dec.off`.

The above task can be accomplished by running the `decode_mesh` program as follows:

```
decode_mesh -r dec_result.txt < bunny.eb > bunny_dec.off
```

Example 2. Suppose that the user wants to both encode and decode a triangle mesh stored in the file named `hand.off` with following requirements:

- The duplicate vertices are checked and removed from the input mesh;
- The encoded triangle mesh's result information is stored in the file `enc_result.txt`.
- The decoded triangle mesh's result information is stored in the file `dec_result.txt`.
- The decoded triangle mesh is stored in the file `hand_dec.off`.

This can be accomplished by invoking the command:

```
encode_mesh -d -r enc_result.txt < hand.off | decode_mesh \
-r dec_result.txt > hand_dec.off
```

Example 3. Suppose that the user wants to both encode and decode a triangle mesh stored in the file named `lena.off` with following requirements:

- The quantization step size for x coordinates is set to 1.
- The quantization step size for y coordinates is set to 1.
- The quantization step size for z coordinates is set to 1.
- The decoded triangle mesh is stored in the file `lena_dec.off`.

This task can be accomplished by executing the command:

```
encode_mesh -x1 -y1 -z1 < lena.off | decode_mesh > lena_dec.off
```

Chapter 5

Results and Analysis

5.1 Introduction

In this chapter, we evaluate the performance of the Edgebreaker mesh-coding method using the software implementation introduced in Chapter 4. We also study some performance characteristics of this implementation. We start this chapter by introducing the test datasets and methodology that is used in our experiments.

5.2 Methodology

Before discussing our various experiments, we briefly introduce the datasets used. The 20 triangle meshes we used in our experiments have been widely used in the research literature. In Table 5.1, basic information for each mesh is given, including the number of vertices, edges, facets, bounding loops, genus, and the source/origin of the mesh. These particular datasets were chosen to cover a wide variety of mesh types, including meshes having: zero bounding loops and zero handles, one or more bounding loops and zero handles, and zero or more bounding loops and one or more handles.

In our test datasets, we found some meshes contain duplicate vertices. These duplicate vertices are likely caused by an insufficient number of significant digits being stored in the mesh data file. All duplicate vertices must be removed before the coding process. If we do not remove the duplicate vertices, errors will arise during the mesh compression since the mesh is not valid. In our experiments, unless otherwise noted, the mesh vertex coordinates are quantized to 16 bits. The 16 bit quantization is adequate for our datasets.

Table 5.1: Basic information of the test meshes

Name	Vertices	Edges	Facets	Boundaries	Genus	Source
`9handle_torus`	9392	28224	18816	0	9	[23]
`animal`	44382	132971	88590	1	0	[24]
`beethoven`	2258	6686	4429	1	0	[4, 25]
`blob`	8036	24102	16068	0	0	[4, 25]
`bunny_hole`	34835	104310	69473	4	0	[26]
`casting`	5096	15336	10224	0	9	[26]
`dragon`	50000	150000	100000	0	1	[26]
`eight`	766	2304	1536	0	2	[4, 25]
`fandisk`	6475	19419	12946	0	0	[27]
`globe_west`	199065	597189	398126	0	0	[28]
`hand`	36616	109554	72937	3	0	[26]
`heart`	1280	3782	2494	8	1	[27]
`heptoroid`	286678	860160	573440	0	22	[29]
`horse`	112642	337920	225280	0	0	[26]
`hypersheet`	487	1407	917	3	1	[27]
`lena`	7864	23432	15569	1	0	[24]
`ramesses`	826266	2478792	1652528	0	0	[26]
`shape`	2562	7680	5120	0	0	[4, 25]
`tre_twist`	800	2400	1600	0	1	[27]
`triceratops`	2832	8490	5660	0	0	[4, 25]

5.3 Coding Efficiency

We begin our evaluation by analyzing coding efficiency of the Edgebreaker method. We first present the coding efficiency results that are produced by our software. This is then followed by the coding rate comparison. The coding performance of the Edgebreaker method is compared with the topological-surgery (TS) method in [3] and the Touma-Gotsman (TG) method in [4].

In the first experiment, we compare the coding rate of the Edgebreaker method with the gzip text-based compression technique. For all 20 triangle meshes in our test datasets, we first ran the encoding program to generate the coded triangle mesh and measured the coded bitstream size in bits per vertex. Then, we used the gzip program to compress the input triangle mesh files individually and measured the file size. Moreover, we calculated the median value of the Edgebreaker and the gzip results. The results are presented in Table 5.2.

Examining the results in Table 5.2, we see that the Edgebreaker mesh-coding method is roughly 4.19 times better than the gzip method in terms of the coding rate. The median coding rate for the Edgebreaker method and the gzip method are 38.58 bits/vertex and 161.71 bits/vertex, respectively. We also found that the `horse` mesh achieves the lowest coding rate among the 20 meshes in the datasets with 3.02 bits/vertex for connectivity coding and 20.39 bits/vertex for geometry coding. Furthermore, we notice that in the case of all ten meshes without handles or holes, the Edgebreaker method requires 3.02 to 3.52 bits/vertex for connectivity coding. These values lie within the theoretical worst-case guaranteed bound (i.e., 3.67 bits/vertex) that was stated in [22].

Next, we wanted to know how the Edgebreaker method compared to other popular mesh-coding methods in terms of the coding efficiency. Therefore, we compared the Edgebreaker method with the TS and TG methods. The coding rates of the TS and TG methods are taken from [4] since we do not have the access to the programs that implement the TS and TG methods, while Edgebreaker results are obtained with our software. The paper [4] included coding results for eight meshes, but we only have access to five of the meshes. So, our experiment is limited to these meshes. In particular, the five meshes used in this experiment are `beethoven`, `blob`, `eight`, `shape`, and `triceratops`. The quantization of vertex coordinates used in [4] was 8 bits. To allow a fair comparison, we also quantized the vertex coordinates to 8 bits. In this experiment, we first ran the Edgebreaker encoder to generate the coded triangle mesh and measured the coded bitstream size in bits per vertex. Next, we compared the Edgebreaker, TS, and TG methods in terms of connectivity coding. Then, we compared geometry coding rate of the Edgebreaker and TS methods. The results can be found in Table 5.3.

Table 5.2: Individual coding efficiency results

Name	Vertices	Geometry (bits/vertex)	Connectivity (bits/vertex)	Total (bits/vertex)	Gzipped (bits/vertex)	Gzipped/Edgebreaker Ratio
9handle_torus	9392	39.20	3.81	43.06	141.60	3.29
animal	44382	35.25	3.52	38.78	163.04	4.20
beethoven	2258	39.10	3.36	42.69	145.80	3.42
blob	8036	35.28	3.33	38.68	142.34	3.68
bunny_hole	34835	26.93	3.24	30.19	199.70	6.61
casting	5096	30.63	3.58	34.31	178.43	5.20
dragon	50000	33.38	3.52	36.91	220.12	5.96
eight	766	36.22	3.69	40.57	128.60	3.17
fandisk	6475	26.64	3.23	29.95	153.27	5.12
globe_west	199065	29.58	3.40	32.98	191.06	5.79
hand	36616	29.54	3.17	32.72	162.83	4.98
heart	1280	35.11	3.95	39.46	118.01	2.99
heptoroid	286678	20.41	3.06	23.47	158.59	6.76
horse	112642	20.39	3.02	23.41	169.80	7.25
hypersheet	487	41.68	4.16	46.88	169.28	3.61
lena	7864	40.94	3.51	44.51	149.78	3.37
ramesses	826266	27.18	3.43	30.61	188.96	6.17
shape	2562	38.53	3.08	41.81	115.18	2.75
tre_twist	800	43.89	3.70	48.23	157.67	3.27
triceratops	2832	34.92	3.38	38.49	174.22	4.53
median value	—	—	—	38.58	161.71	4.19

Table 5.3: Coding efficiency comparison of the Edgebreaker and other mesh-coding methods

Name	Vertices	Size: bits/vertex					
		Edgebreaker		TS method		TG method	
		Geometry	Connectivity	Geometry	Connectivity	Geometry	Connectivity
beethoven	2258	10.4	3.4	15.0	4.8	10.8	2.4
blob	8036	7.7	3.3	10.3	3.4	7.9	1.7
eight	766	9.2	3.7	12.0	3.8	7.1	0.6
shape	2562	9.1	3.1	14.3	2.2	9.3	0.2
triceratops	2832	9.8	3.4	10.3	4.3	8.3	2.2

According to Table 5.3, the TG method achieves the lowest connectivity coding rate among these three methods in all five cases. The TG method beats the Edgebreaker and the TS methods in terms of connectivity coding in all five test cases by a margin of 1.0 to 3.1 bits/vertex and 1.7 to 3.2 bits/vertex, respectively. According to the survey in [14], the TG method is considered to be a state-of-the-art technique for single-rate 3-D mesh compression. This conclusion fits with the connectivity coding results above.

In what follows, we compared the geometry coding rate of the Edgebreaker and TS methods. From Table 5.3, we see that the Edgebreaker method is better than the TS method in terms of the geometry coding. The Edgebreaker method beats the TS method in all five test cases by a margin of 0.5 to 5.2 bits/vertex. This is due to the fact that the TS method uses a simpler prediction scheme that predicts the vertex position as the previously coded vertex. This shows the effectiveness of the parallelogram-prediction scheme.

5.4 Time Complexity

Next, we consider the computational complexity of the Edgebreaker method as measured by execution time. Before proceeding further, a brief digression is necessary to introduce the hardware that was employed during the experiments. The experimental results were collected on a computer with a 3.16 GHz Intel Core2 Duo CPU and 4.0 GB of RAM. The version of GCC used was 5.1.0, and all code was compiled with full optimization enabled.

In what follows, we consider the time complexity of the encoding and decoding programs. For the 20 triangle meshes in our test datasets, we measured the execution time required by both programs. The median execution time over 30 runs of each program is given in Table 5.4.

To begin, we explore the relationship between the encoding time and the number of handles and holes in the mesh. When two meshes contain the similar number of vertices, we expect that the mesh with more handles or holes will take a longer encoding time. Our expectation is confirmed by the time complexity results in Table 5.4. As can be seen from Table 5.4, the `eight` and `tre_twist` meshes, and the `lena` and `blob` meshes, contain the similar number of vertices. By comparing the time results of these two pairs of meshes, we find that the triangle mesh with more handles or holes requires 20% to 30% more time to encode.

In the above results, we notice that the `ramesses` and `heptoroid` meshes take the largest execution time. Therefore, we performed code profiling on both programs. The goal of the code profiling is to find the bottleneck in our code and provide suggestions for software improvement. The encoder profiling details for the `ramesses` and `heptoroid` meshes are

Table 5.4: Individual time complexity results. Time listed in table is the median execution time over 30 runs for each program.

Name	Vertices	Encode Time (seconds)	Decode Time (seconds)
`9handle_torus`	9392	0.209	0.261
`animal`	44382	1.213	1.177
`beethoven`	2258	0.073	0.083
`blob`	8036	0.138	0.271
`bunny_hole`	34835	0.626	0.807
`casting`	5096	0.110	0.169
`dragon`	50000	1.053	1.560
`eight`	766	0.031	0.032
`fandisk`	6475	0.131	0.232
`globe_west`	199065	4.389	4.895
`hand`	36616	0.646	1.033
`heart`	1280	0.040	0.057
`heptoroid`	286678	5.111	6.762
`horse`	112642	2.223	2.951
`hypersheet`	487	0.025	0.022
`lena`	7864	0.210	0.260
`ramesses`	826266	25.174	17.625
`shape`	2562	0.074	0.123
`tre_twist`	800	0.024	0.037
`triceratops`	2832	0.068	0.106

Table 5.5: Encoding time complexity analysis with profiling. Results showing in table are the accumulated time from eleven runs.

(a) `ramesses` mesh

Percentage time (%)	Cumulative time (seconds)	Self time (seconds)	Function description
37.59	79.34	79.34	Process S-type triangle
7.23	94.59	15.25	Find halfedge's incident vertex
6.09[†]	107.45	12.86	Encode coordinates in regular mode[*]
5.47	118.99	11.54	Find number of connected components in the mesh
5.00[†]	129.55	10.56	Entropy coding[*]
4.26	138.54	8.99	Updates adjacent halfedges information
3.87[†]	146.70	8.16	Arithmetic encoding function[*]
2.83	152.68	5.98	Output encoded bits[*]
2.76	158.50	5.82	Lookup halfedges in the mesh
2.48	163.74	5.24	Predict the vertex position

(b) `heptoroid` mesh

Percentage time (%)	Cumulative time (seconds)	Self time (seconds)	Function description
14.51[†]	5.84	5.84	Encode coordinates in regular mode[*]
8.84[†]	9.40	3.56	Entropy coding[*]
8.49	12.82	3.42	Find halfedge's incident vertex
7.45[†]	15.82	3.00	Arithmetic encoding function[*]
7.10	18.68	2.86	Updates adjacent halfedges information
5.14	20.75	2.07	Find number of connected components in the mesh
4.07	22.39	1.64	Output encoded bits[*]
3.97	23.99	1.60	Lookup halfedges in the mesh
2.86	25.14	1.15	Read mesh from OFF file
2.86	26.29	1.15	Update the probability distribution for the arithmetic coder[*]

[*]Arithmetic coding related function from SPL library.
[†]Items used to calculate the time consumption in the arithmetic coder.

included in Tables 5.5(a) and (b), respectively. In these tables, we provide the profiling results for the ten functions that had the largest execution times accumulated over eleven runs. In what follows, we examine the result tables and give the corresponding explanations.

First, we study the time complexity in the encoding program based on the profiling details. Examining the `ramesses` mesh profiling details in Table 5.5(a), we notice that the function which encodes the S-type triangles takes the largest execution time (i.e., 37.59% of the encoding time). We explain the reasons for this phenomenon as follows. As presented in Chapter 3, the S-type encoding operation splits the current mesh into two submeshes and traverses the left submesh's boundary to update the vertex and halfedge marks. The bounding loop traversal is a very time consuming operation, since the encoder needs to find the adjacent boundary halfedge for the current halfedge and also move along the entire bounding loop. By examining the encoding details of all meshes in our test datasets, we find that the encoder generates the largest number of S-type triangles when processing the `ramesses` mesh (i.e., 85970 S-type triangles which is 5.2% of all triangles in the mesh). This explains why the encoding program spends 37.59% of the time to process the S-type triangles in the `ramesses` mesh. Next, we consider the influence of the bounding loop traversal on the other functions. From the profiling results in Tables 5.5(a) and (b), we see that 5% of the time is spent in finding the number of connected components in the mesh. Since the function to detect the number of connected components requires bounding loop traversal, this explains why this function took a long time to execute in both cases. Therefore, if we could reduce the time that is required by the bounding loop traversal, the execution time of the encoding program would decrease.

The decoder profiling details for the `ramesses` and `heptoroid` meshes are included in Tables 5.6(a) and (b), respectively. In these tables, we provide the profiling results for the ten functions that had the largest execution times accumulated over eleven runs.

To begin, we analyze the execution time that is spent on the arithmetic coder related functions in the decoding program. By summing the profiling results marked with a dagger in Tables 5.6(a) and (b), we notice that the arithmetic-coding functions take a significant amount of time to execute (i.e., approximately 55% of the decoding time). This implies that decoding the binary prediction residuals is the most time consuming operation.

Next, we compare the execution time that is spent on the arithmetic coder related functions in both encoding and decoding programs. The reason we want to conduct this comparison is as follows. We do not observe a significant time percentage taken by the arithmetic-coding functions in the encoder profiling results and we want to find a explanation for this phenomenon. Since the functions used to measure the arithmetic coding time are shown in both the encoding and decoding results, a fair comparison between the encoder and decoder can be made. For the arithmetic-coding functions, the encoder and decoder take approximately 20% and 55% of time to execute, respectively. If we measure the arithmetic coder related time consumptions in seconds, we notice that the execution time are similar in both programs. The encoder and decoder require 31.58 and 30.11 seconds for the `ramesses` mesh, and 12.40 and 11.31 seconds for the `heptoroid` mesh to execute, respectively. The above results imply that the actual time spent on the arithmetic-coding functions in both programs are approximately the same.

Table 5.6: Decoding time complexity analysis with profiling. Results showing in table are the accumulated time from eleven runs.

(a) `ramesses` mesh

Percentage time (%)	Cumulative time (seconds)	Self time (seconds)	Function description
28.93†	15.81	15.81	Decode coordinates in regular mode*
18.99†	26.19	10.38	Entropy coding*
13.98	33.83	7.64	Arithmetic coder probability adjustment*
7.73	38.06	4.23	Update the probability distribution for the arithmetic coder*
7.17†	41.98	3.92	Arithmetic decoding function*
5.10	44.77	2.79	Read encoded bits*
4.61	47.29	2.52	Arithmetic coder interval check*
3.93	49.44	2.15	Read EB file from input stream
2.56	50.84	1.40	Decode vertex coordinates in bypass mode*
1.57	51.70	0.86	Predict the vertex position

(b) `heptoroid` mesh

Percentage time (%)	Cumulative time (seconds)	Self time (seconds)	Function description
32.32†	6.46	6.46	Decode coordinates in regular mode*
17.08†	9.88	3.42	Entropy coding*
14.21	12.72	2.84	Arithmetic coder probability adjustment*
9.08	14.53	1.82	Update the probability distribution for the arithmetic coder*
7.15†	15.96	1.43	Arithmetic decoding function*
5.50	17.06	1.10	Arithmetic coder interval check*
3.85	17.83	0.77	Read EB file from input stream
3.20	18.47	0.64	Read encoded bits*
2.10	18.89	0.42	Predict the vertex position
1.25	19.14	0.25	Decode vertex coordinates in bypass mode*

*Arithmetic coding related function from SPL library.
†Items used to calculate the time consumption in the arithmetic coder.

Table 5.7: Memory complexity analysis based on the major data structures

(a) Encoding memory usage

Data structure	Memory usage (bytes)
vertex	28
op-code sequence	12
offset table	12
halfedge stack	32

(b) Decoding memory usage

Data structure	Memory usage (bytes)
vertex	12
circular doubly-linked list	8
op-code sequence	12
facet table	12
vertex table	12
offset table	12
node stack	32

5.5 Memory Complexity

Next, we consider the memory complexity of the Edgebreaker method. We first analyze the memory complexity by studying the major data structures' size in each program. Since the analysis is based on the memory size required by the major data structures, it cannot reflect the practical memory usage. This analysis, however, still can provide users with the information about the basic memory requirement of both programs.

The major data structures' memory requirement in the encoding program are listed in Table 5.7(a). According to the Euler's formula, the number of triangles is approximately twice the number of vertices, and the number of edges is approximately three times the number of vertices. By checking the encoding details of all meshes in our test datasets, we find that the S-type triangles take approximately 3.2% of the total triangle types. Hence, for a triangle mesh with N vertices, the memory required by the encoding program in bytes is

$$\begin{aligned} &28 \cdot (V + E + F) + 12 \cdot F + (12 + 32) \cdot 3.2\% \cdot F \\ &= 28 \cdot 6N + 12 \cdot 2N + (12 + 32) \cdot 3.2\% \cdot 2N \approx 195N \end{aligned} \tag{5.1}$$

where V, E, F denote the number of vertices, edges, and facets in the mesh, respectively. The major data structures' memory requirement in the decoding program are listed in Table 5.7(b). As has been noted, the circular doubly-linked list contains approximately the same number of nodes as the number of vertices in the mesh. Hence, for a triangle mesh with N vertices, the memory required by the decoding program in bytes is

$$\begin{aligned} & 12 \cdot V + 8 \cdot V + 12 \cdot F + 12 \cdot (V + F) + (12 + 32) \cdot 3.2\% \cdot F \\ = {} & 12N + 8N + 12 \cdot 2N + 12 \cdot 3N + (12 + 32) \cdot 3.2\% \cdot 2N \approx 83N \end{aligned} \tag{5.2}$$

where V, E, F denote the number of vertices, edges, and facets in the mesh, respectively.

In what follows, we consider the actual peak memory that are used by our software. For the 20 triangle meshes in our datasets, we measured the peak memory required by both programs. The memory usage results can be found in Table 5.8.

To begin, we analyze the actual peak memory used in both programs. Examining the individual results in Table 5.8, we observe that the encoding and decoding programs require 0.78 MB to 154.35 MB and 0.69 MB to 22.06 MB memory, respectively. By comparing the actual peak memory consumed in both programs to the memory analysis based on the major data structures, we find that both programs comsume approximately 1.4 times memory compare to the value we calculated. We explain the reasons for this phenomenon as follows. First, the libraries we used in the code may not be memory efficient. The linking process may included some extra functions and codes from the library to our software, and this linking process can increase the memory usage. Second, some other data structures that are not tightly related to the Edgebreaker method but used in both programs are not included in Tables 5.7(a) and (b). For example, the sets and vectors that used in the functions to find the duplicate vertices and number of connected components are not counted in Tables 5.7.

Table 5.8: Individual memory complexity results

Name	Vertices	Encode peak memory (bytes)	Decode peak memory (bytes)
`9handle_torus`	9392	2476032	949248
`animal`	44382	9227264	1867264
`beethoven`	2258	1123328	745984
`blob`	8036	2275328	885248
`bunny_hole`	34835	7389696	1671168
`casting`	5096	1645568	825344
`dragon`	50000	10046464	1985536
`eight`	766	826880	707072
`fandisk`	6475	1900032	854016
`globe_west`	199065	38011392	5870592
`hand`	36616	7647232	1703936
`heart`	1280	917504	723456
`heptoroid`	286678	55522304	8670208
`horse`	112642	21260288	3550720
`hypersheet`	487	783360	689664
`lena`	7864	2243072	883712
`ramesses`	826266	154351616	22059008
`shape`	2562	1171968	760832
`tre_twist`	800	832000	707584
`triceratops`	2832	1239040	763392

Chapter 6

Conclusions

In this project, the Edgebreaker method for 3-D triangle mesh coding has been studied. As part of the work, the author has implemented the method in software. The performance of the Edgebreaker method is evaluated by analyzing the experimental results produced by the mesh-coding software. The coding efficiency of the Edgebreaker method was studied. In terms of coding rate, the Edgebreaker method outperforms the gzip text-based compression technique on average by a factor of 4.19 times. We also compared the coding performance with other well-known mesh-coding methods. Moreover, the time complexity of the Edgebreaker method was analyzed by evaluating in terms of the execution time. Furthermore, the memory complexity for both programs was studied.

The Edgebreaker mesh-coding method has been shown to be effective by our software implementation and the experimental results obtained with it. Although the work presented in this report has achieved the desired level of performance, there is still some additional work that is worth exploring in the future. As explained earlier in Section 5.4, when a triangle mesh results in a large number of S-type triangles, it takes a fairly long time to encode. If we could reduce the time that is required for the bounding loop traversal, it is highly likely that we would be able to obtain even better time complexity results than the current ones. Earlier we showed the memory complexity results (in Section 5.5). Some potential research on data structure improvement could also be done to reduce the amount of memory required by both programs.

Appendix A

Supplementary Material

In this project, we have studied the Edgebreaker mesh-coding method thoroughly. We found some typographical errors in the original Edgebreaker paper [1]. All the error corrections are included in this appendix.

Several subtraction signs are missing from the pseudocode of the decoding initialization phase in [1] (on page 56). For the C and S-type triangles, the decoder needs to subtract one from the counter `e`. For the E-type triangle, the decoder needs to subtract one from the counter `d`. The accurate decoding initialization phase pseudocode is given in Algorithm 20.

For the M-type encoding, the operation sequence for link updates and halfedges traversal is reversed in the pseudocode in [1] (on page 58). The algorithm needs to traverse the hole's boundary first, and then updates the link relationships on the active bounding loop. The accurate pseudocode for M-type compression is given in Algorithm 21.

Algorithm 20 Pseudocode for simple meshes decoding initialization phase.

```
1: Case S: e -= 1; s += 1; push(e, s); d += 1;
2: Case E: e += 3; (e′,s′) = pop; O[s′] = e - e′- 2; d -= 1; if d $\leq$ 0 then
   stop;
3: Case C: e -= 1; c += 1;
4: Case R: e += 1;
5: Case L: e += 1;
```

Algorithm 21 Pseudocode for M-type encoding operation.

```
1: O_seq = O_seq|M; {append M to op-code sequence.}
2: g.m = 0; g.p.o.m = 1; g.n.o.m = 1; {update marks.}
3: b = g.n; {initial candidate for the halfedge b.}
4: while b.m ≠ 2 do
5:    b = b.o.p; {turn around the vertex g.v.}
6: end while
7: l = 0; {initial hole length count.}
8: while b.m ≠ 1 do
9:    b.m = 1; b.e.m = 1; {mark hole.}
10:   l++; {update the hole length count.}
11:   P = P|b.e; {append new vertex reference to P.}
12:   b = b.N {move to next edge around hole.}
13: end while
14: M = M|l; {add the hole length count to M table.}
15: g.P.N = g.p.o; g.p.o.P = g.P; {connect the halfedges g.P and g.p.o.}
16: g.p.o.N = b.N; b.N.P = g.p.o; {connect the halfedges g.p.o and b.N.}
17: b.N = g.n.o; g.n.o.P = b; {connect the halfedges b and g.n.o.}
18: g.n.o.N = g.N; g.N.P = g.n.o; {connect the halfedges g.n.o and g.N.}
19: g = g.n.o; {move the active gate g.}
```

Bibliography

[1] J. Rossignac, "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, pp. 47–61, Jan. 1999.

[2] M. Deering, "Geometry compression," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pp. 13–20, 1995.

[3] G. Taubin and J. Rossignac, "Geometric compression through topological surgery," *ACM Trans. Graph.*, vol. 17, pp. 84–115, Apr. 1998.

[4] C. Touma and C. Gotsman, "Triangle mesh compression," in *Proceedings of the Graphics Interface 1998 Conference, Vancouver, British Columbia, Canada*, pp. 26–34, June 1998.

[5] M. M. Chow, "Optimized geometry compression for real-time rendering," in *Visualization '97., Proceedings*, pp. 347–354, Oct. 1997.

[6] C. L. Bajaj, V. Pascucci, and G. Zhuang, "Single resolution compression of arbitrary triangular meshes with properties," in *Data Compression Conference, DCC 1999, Snowbird, Utah, USA, Mar. 29-31, 1999.*, pp. 247–256, 1999.

[7] P. Alliez and M. Desbrun, "Valence-driven connectivity encoding for 3D meshes," *Comput. Graph. Forum*, vol. 20, no. 3, pp. 480–489, 2001.

[8] P. Diaz-Gutierrez, M. Gopi, and R. Pajarola, "Hierarchyless simplification, stripification and compression of triangulated two-manifolds," *Computer Graphics Forum*, pp. 457–467, 2005.

[9] S. Gumhold and W. Strasser, "Real time compression of triangle mesh connectivity," *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 133–140, 1998.

[10] H. Lopes, G. Tavares, J. Rossignac, A. Szymczak, and A. Safanova, "Edgebreaker: a simple compression for surfaces with handles," in *Proceedings of the seventh ACM symposium on Solid modeling and applications*, pp. 289–296, ACM, 2002.

[11] J. Rossignac and A. Szymczak, "Wrap&zip decompression of the connectivity of triangle meshes compressed with Edgebreaker," *Computational Geometry*, vol. 14, no. 1-3, pp. 119–135, 1999.

[12] G. Turán, "On the succinct representation of graphs," *Discrete Applied Mathematics*, vol. 8, no. 3, pp. 289–294, 1984.

[13] K. Keeler and J. Westbrook, "Short encodings of planar graphs and maps," *Discrete Applied Mathematics*, vol. 58, pp. 239–252, 1993.

[14] J. Peng, C.-S. Kim, and C. C. Jay Kuo, "Technologies for 3D mesh compression: A survey," *Journal of Visual Communication and Image Representation*, vol. 16, pp. 688–733, Dec. 2005.

[15] "CGAL, Computational Geometry Algorithms Library." `http://www.cgal.org`, 2016.

[16] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[17] M. D. Adams, "An efficient progressive coding method for arbitrarily-sampled image data," *IEEE Signal Processing Letters*, vol. 15, pp. 629–632, 2008.

[18] J. Zhu, "Lossless triangle mesh compression," Master's thesis, Queen's University, July 2013.

[19] J. Rossignac, "Estimate function." `http://www.cc.gatech.edu/~jarek/edgebreaker/eb/PredictionScheme.htm`, 2016.

[20] M. D. Adams, "Signal Processing Library." `http://www.ece.uvic.ca/~mdadams/SPL`, 2016.

[21] "Make, make manual." `http://www.gnu.org/software/make/manual`, 2016.

[22] D. King and J. Rossignac, "Guaranteed 3.67v bit encoding of planar triangle graphs," in *Proceedings of the 11th Canadian Conference on Computational Geometry, UBC, Vancouver, British Columbia, Canada*, Aug. 1999.

[23] "Examples of Input and Output files in ASCII format." `http://www.cc.gatech.edu/~jarek/edgebreaker/eb/Examples.html`, 2016.

[24] M. Adams. Personal communication, 2015-12-15.

[25] C. Gotsman. Personal communication, 2016-03-14.

[26] "A Benchmark for 3D Mesh Watermarking." `http://liris.cnrs.fr/meshbenchmark/`, 2016.

[27] "Lutz Kettner: Projects: Selected 3D Example Objects." `https://people.mpi-inf.mpg.de/~kettner/proj/obj3d/`, 2016.

[28] "NASA 3D Resources." `http://nasa3d.arc.nasa.gov/models`, 2016.

[29] "Suggestive Contour Gallery." `http://gfx.cs.princeton.edu/proj/sugcon/models/`, 2016.