University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations Theses, Dissertations, and Major Papers

1-1-2006

A comparative study of quadmesh compression for Touma-Gotsman and Spirale Reversi schemes.

Demin Yin
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Yin, Demin, "A comparative study of quadmesh compression for Touma-Gotsman and Spirale Reversi schemes." (2006). *Electronic Theses and Dissertations*. 7148.
https://scholar.uwindsor.ca/etd/7148

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

A Comparative Study of Quadmesh Compression for Touma-Gotsman and Spirale Reversi Schemes

by

Demin Yin

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2006

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-42339-4
Our file Notre référence
ISBN: 978-0-494-42339-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© 2006 Demin Yin

All Rights Reserved. No Part of this document may be reproduced, stored or otherwise retained in a retreival system or transmitted in any form, on any medium by any means without prior written permission of the author.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

A 3D mesh is made up of polygonal faces. A complete description of such a mesh is provided by its connectivity (adjacencies of polygons) and geometry (vertex coordinates). For most practical work, the meshes used are either triangular (all faces are triangles) or quadrilateral (all faces are quadrilaterals) meshes (quadmeshes, for short). Recently, there has been much work on compressing the connectivity information pertaining to a mesh. For quadmesh compression, existing algorithms triangulate the quadmesh first, and then apply triangle mesh compression techniques as previous researches on mesh compression were mostly focused on triangle meshes. To avoid the additional triangulation step, we propose two direct techniques to compress and decompress the connectivity of quadmeshes in linear time. In this thesis, we will describe how to extend two well-known triangle mesh compression algorithms to quadmesh compression, and how to apply encoding schemes for them. A comparison of the two algorithms for quadmesh compression is also given.

Keyword: quadmesh, quadrilateral mesh, mesh compression, Edgebreaker, Spirale Reversi, connectivity encoding, linear encoding

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To my Mom, Dad

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Asish Mukhopadhyay. Dr. Mukhopadhyay gave me many invaluable helps during the last 3 years. Without Dr. Mukhopadhyay's supervision, this thesis would not have been possible.

I would like to thank Dr. Jagdish Pathak and Dr. Arunita Jaekel for having the patience to read drafts and for their attentive comments, suggestions and relevant feedback.

I would like to thank committee chair Dr. Subir Bandyopadhyay for his interest and time.

I would like to thank some of my friends in Windsor and Canada. Thank James, Angela, Ben/Marlene, Lester, ..., who gave me many valuable helps in the last 3 years.

I would like to thank the faculties and staffs in Computer Science. You gave me many kindly helps, and I really appreciate these helps.

Finally, I would like to thank the Government of Canada and the Canadians. Thank the Government of Canada provided a chance for me to immigrate to Canada, accepted my immigration application and provided supports for my study in Canada; Thank the Canadians made such a great, such a peaceful, such a friendly country so that I could enjoy my life here.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Abstract iv

Dedication v

Acknowledgements vi

Contents vii

List of Figures xi

List of Tables xiv

1 Introduction 1

1.1 Mesh compression 1

1.2 Objectives of the thesis 3

1.3 Overview of the thesis 3

2 Literature review about connectivity compression 4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1 Background and basic concepts 5

2.1.1 Geometric relationship of triangular mesh 7

2.1.2 Geometric relationship of quadrilateral mesh 7

2.2 Connectivity compression 8

2.2.1 Indexed face set 9

2.2.2 Triangle strip 9

2.2.3 Spanning tree 12

2.2.4 Layered decomposition 14

2.2.5 Valence-driven approach 17

2.2.6 Triangle conquest 18

2.3 Summary 25

3 Quadmesh compression and decompression using Touma-Gotsman algorithm 27

3.1 Definitions 28

3.2 Definition of opposite half edge (OHE) data structure 28

3.3 Algorithm description 30

3.3.1 Compression process 39

3.3.2 Decompression process 44

3.4 Illustrate the approach 50

3.4.1 Compression of a torus mesh 51

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4.2 Decompression of a torus mesh 55

3.5 Handling boundaries 60

3.6 Entropy coding of the command sequence 60

3.7 Time complexity analysis 61

3.8 Summary 61

4 Linear time quadmesh decompression using Spirale Reversi 83

4.1 Definition of opposite edge (OE) data structure 84

4.2 Edgebreaker compression algorithm 84

4.3 Spirale Reversi decompression algorithm 88

4.4 Illustrate the approach 93

4.4.1 Compress a mesh with boundary 93

4.4.2 Decompress a mesh with boundary 97

4.5 Quadmesh of higher genus 101

4.6 Apply encoding schemes for the operation sequence 103

4.7 Time complexity analysis 105

4.8 Summary 105

5 Experimental Results 132

5.1 Implementation 132

5.2 Experimental results 133

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3 Summary 138

6 Conclusions and Future Work **140**

6.1 Major contributions 140

6.2 Future work 141

References **143**

VITA AUCTORIS **147**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

2.1 Examples of (A) an orientable manifold mesh, (B) a non-orientable non-manifold mesh, and (C) an orientable non-manifold mesh. 6

2.2 (A) The sphere, (B) the torus, and (C) the eight-shaped mesh. 6

2.3 The indexed face set representation of a mesh: (A) a mesh example and (B) its face array. 10

2.4 (A) The triangle strip, (B) the triangle fan, and (C) the generalized triangle strip. 11

2.5 (A) A set of boundary edges, (B) triangle fans for the first strip, and (C) triangle fans for the second strip, where thick arrows show selected boundary edges and thin arrows show the triangle fans associated with each inner boundary vertex. 12

2.6 (A) An octahedron mesh, (B) its vertex spanning tree, and (C) the cut and flattened mesh with its triangle spanning tree shown by dashed lines. 13

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.7 Fig. 7. Illustration of (A) the layered vertex structure and the branching point depicted by a black dot, (B) a triangle strip, (C) bubble triangles, and (D) a cross-contour triangle fan, where contours are depicted with solid lines and other edges with dashed lines. 15

2.8 A mesh connectivity encoding example by Touma and Gotsman (35), where the active list is shown with thick lines, the focus vertex with the black dot, and the dummy vertex with the gray dot. 19

2.9 Illustration of (A) edge loops and (B) gates and initial edge loops for a mesh without boundary, and (C) gates and initial edge loops for a mesh with boundary, where thick lines depict edge loops, and g denotes the gate. 22

2.10 (A) Five op-codes C, L, R, E, and S, where the gate g is marked with an arrow, and (B) an example of the encoding process in the edgebreaker algorithm where the arrows and the numbers show the traversal order and different filling patterns are used to represent different op-codes. . 24

3.1 OHE data structure 29

3.2 Compression example (A to U) 62

3.3 Decompression example (A to T) 73

4.1 OE data structure 85

4.2 Compression operations 108

4.3 Decompression operations 111

4.4 Compression example (A to S) 112

4.5 Decompression example (A to S) 122

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1 Comparison of compression ratios archived by different encoding schemes and GZip 138

5.2 Comparison of compression ratios archived by different encoding schemes 139

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

2.1 The output of each step in Fig. 2.8 20

3.1 Free vertices found in different cases 36

3.2 Effects on size of current active list 37

3.3 Number of commands created in each case 38

3.4 Huffman coding scheme for sample mesh "Torus" 61

4.1 Gotsman's encoding scheme for Edgebreaker algorithm 103

4.2 Dr. Mukhopadhyay's encoding scheme for Edgebreaker algorithm . . 104

4.3 Huffman coding scheme for sample mesh "Torus" 105

5.1 Sample quadmeshes for test 134

5.2 File size for Touma-Gotsman's algorithm 135

5.3 File size for Edgebreaker algorithm 135

5.4 Compression ratios archived by different encoding schemes for Edgebreaker algorithm 136

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.5 Compression ratios archived by using GZip 136

5.6 Compression ratios archived by different encoding schemes and GZip 137

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

1.1 Mesh compression

Graphics data are more and more widely used in various applications, including video gaming, engineering design, architectural walk through, virtual reality, e-commerce, and scientific visualization. Among various representation tools, triangular meshes provide an effective means to represent 3D mesh models. Typically, connectivity, geometry, and property data are used to represent a 3D polygonal mesh. Connectivity data describe the adjacency relationship between vertices; geometry data specify vertex locations; and property data specify several attributes such as the normal vector, material reflectance, and texture coordinates. We concentrate on the compression of connectivity in this thesis.

To achieve a high level of realism, complex models are required, and they are obtained from various sources such as modeling software and 3D scanning. They

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

usually demand a huge amount of storage space and/or transmission bandwidth in the raw data format. As the number and the complexity of existing 3D meshes increase explosively, higher resource demands are placed on storage space, computing power, and network bandwidth. Among these resources, the network bandwidth is the most severe bottleneck in network-based graphic applications that demand real-time interactivity. Thus, it is essential to compress graphics data efficiently. This research area has received a lot of attention since early 1990s, and there has been a significant amount of progress along this direction over the last decade.

Early research on 3D mesh compression focused on single-rate compression techniques to save the bandwidth between CPU and the graphics card. In a single-rate 3D mesh compression algorithm, all connectivity and geometry data are compressed and decompressed as a whole. The graphics card cannot render the original mesh until the entire bit stream has been wholly received. Later, with the popularity of the Internet, the progressive compression and transmission has been intensively researched. When progressively compressed and transmitted, a 3D mesh can be reconstructed continuously from coarse to fine levels of detail (LODs) by the decoder while the bit stream is being received. Moreover, progressive compression can enhance the interaction capability, since the transmission can be stopped whenever an user finds out that the mesh being downloaded is not what he/she wants or the resolution is already good enough for his/her purposes.

Three-dimensional mesh compression is so important that it has been incorporated into several international standards. VRML (7) established a standard for transmitting 3D models across the Internet. Originally, a 3D mesh was represented in ASCII format without compression in VRML. For efficient transmission, Taubin et al. developed a compressed binary format for VRML (33) based on the topological surgery algorithm (34), which easily achieved a compression ratio of 50:1 over the VRML ASCII format. MPEG-4 (1), which is an ISO/IEC multimedia standard developed by the Moving Picture Experts Group for digital television, interactive graphics,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and interactive multimedia applications, also includes three-dimensional mesh coding (3DMC) algorithm to encode graphics data. The 3DMC algorithm is also based on the topological surgery algorithm, which is basically a single-rate coder for manifold triangular meshes. Furthermore, MPEG-4 3DMC incorporates progressive 3D mesh compression, non-manifold 3D mesh encoding, error resiliency, and quality scalability as optional modes.

1.2 Objectives of the thesis

For quadmesh compression, existing single-rate compression algorithms triangulate the quadmesh first, and then apply triangle mesh compression techniques as previous researches on mesh compression were mostly focused on triangle meshes. To avoid the additional triangulation step, we propose two direct techniques to compress and decompress the connectivity of quadmeshes in linear time.

A comparison of the experimental results of two algorithms will also be discussed.

1.3 Overview of the thesis

The rest of this thesis is organized as follows. Chapter 2 provides a review of the background and introduces some definitions necessary to understand 3D mesh compression techniques for connectivity. Chapter 3 shows how to extend Touma-Gotsman's algorithm for quadmesh, Chapter 4 shows how to extend Edgebreaker/Spirale Reversi algorithm for quadmesh. Chapter 5 discusses the results of experiments. Finally, concluding remarks are given in Chapter 6.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Literature review about connectivity compression

In this chapter, we intend to review various connectivity compression technologies with the main focus on triangular mesh compression. It is worthwhile to point out that there were several survey papers contains this subject. Rossignac (26) briefly summarized prior schemes on vertex data compression and connectivity data compression. Taubin (32) gave a survey on various mesh compression schemes. Although the two schemes in the MPEG-4 standard (i.e., topological surgery and progressive forest split) were described in detail in (32), the review of other schemes was relatively sketchy. Shikhare (29) classified and described mesh compression schemes. However, this work did not treat progressive schemes with enough depth. Gotsman et al. (13) gave a tutorial on techniques for mesh simplification, connectivity compression, and geometry compression. This tutorial gave a detailed treatment on mesh simplification and geometry compression. However, the review on connectivity coding focused

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mostly on single-rate region-growing schemes. Recently, Alliez and Gotsman (3) surveyed techniques for both single-rate and progressive compression of 3D meshes. This survey gave a high-level algorithm classification, but focused only on static polygonal 3D mesh compression.

2.1 Background and basic concepts

Several definitions and concepts needed to understand 3D mesh compression algorithms are presented in this section. More rigorous definitions can be found in (10, 14, 22).

We say that two objects A and B are homeomorphic, if A can be stretched or bent without tearing to B. A 3D mesh is called a manifold if its every point has a neighborhood homeomorphic to an open disk or a half disk. In a manifold, the boundary consists of the points that have no neighborhoods homeomorphic to an open disk but have neighborhoods homeomorphic to a half disk. In 3D mesh compression, a manifold with boundary is often pre-converted into a manifold without boundary by adding a dummy vertex to each boundary loop and then connecting the dummy vertex to every vertex on the boundary loop. Fig. 2.1A is a manifold mesh, while Figs. 1B and C are non-manifold meshes. Fig. 2.1B is non-manifold since each point on the edge (v_1, v_2) has no neighborhood that is homeomorphic to an open disk or a half disk. Similarly, the vertex v_1 in Fig. 2.1C has no neighborhood homeomorphic to a open disk or a half disk.

The orientation of a polygon can be specified by the ordering of its bounding vertices. The orientations of two adjacent polygons are called compatible if they impose opposite directions on their common edges. A 3D mesh is said to be orientable if there exists an arrangement of polygon orientations such that each pair of adjacent polygons are compatible. Figs. 2.1A and C are orientable with the compatible orienta-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tions marked by arrows. In contrast, Fig. 2.1B is not orientable, since three polygons share the same edge (v_1, v_2). Note that, after we make polygon B and C compatible, it is impossible to find an orientation of polygon A such that A is compatible with both B and C.

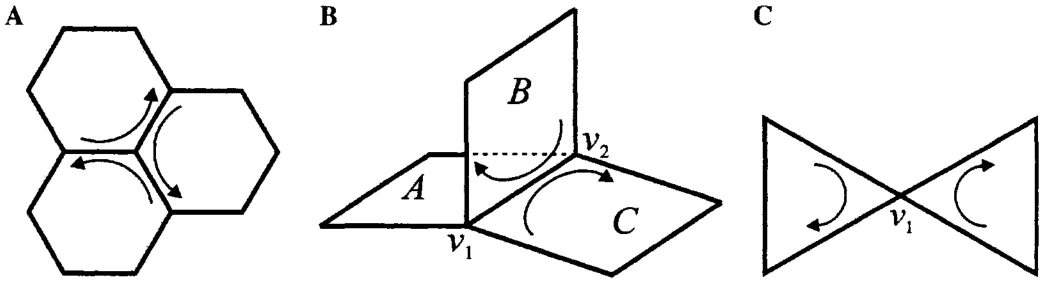


Figure 2.1: Examples of (A) an orientable manifold mesh, (B) a non-orientable non-manifold mesh, and (C) an orientable non-manifold mesh.

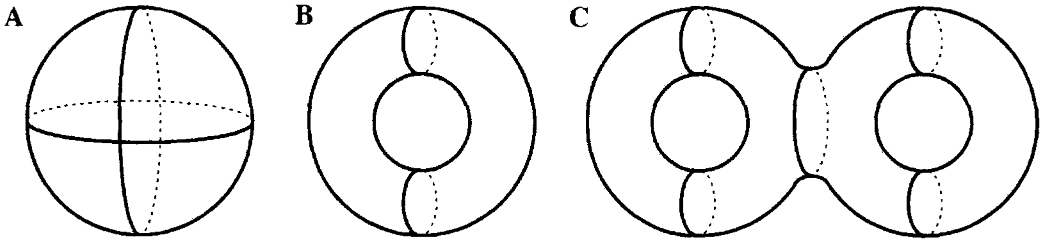


Figure 2.2: (A) The sphere, (B) the torus, and (C) the eight-shaped mesh.

The genus of a connected orientable manifold without boundary is defined as the number of handles. For example, there is no handle in a sphere, one handle in a torus, and two handles in an eight-shaped surface as shown in Fig. 2.2. Thus, their genera are 0, 1, and 2, respectively. A mesh homeomorphic to a sphere is called a simple mesh. For a connected orientable manifold without boundary, Euler's formula is given by

$$v - e + f = 2 - 2g \tag{2.1}$$

where v, e,and f are, respectively, the number of vertices, edges, and faces in the manifold, and g is the genus of the manifold.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.1 Geometric relationship of triangular mesh

Suppose that a triangular manifold mesh contains a sufficiently large number of edges and triangles, and that the ratio of the number of boundary edges to the number of non-boundary edges is negligible. Then, we can approximate the number of edges by

$$e \simeq 3f/2 \tag{2.2}$$

since an edge is shared by two triangles in general. Substituting equation 2.2 into equation 2.1, we have $v \simeq f/2 + 2 - 2g$. Since $f/2$ is much larger than $2 - 2g$, we get

$$v \simeq f/2 \tag{2.3}$$

In other words, a typical triangular mesh has twice as many triangles as vertices.

Also, from equation 2.2 and equation 2.3, we have an approximate relation

$$e \simeq 3v \tag{2.4}$$

The degree (or valence) of a vertex is the number of edges incident on that vertex. It can be shown that the sum of degrees is twice the number of edges (14). Thus, we have

$$\sum degree = 2e \simeq 6v \tag{2.5}$$

Thus, in a typical triangular mesh, the average vertex degree is 6.

2.1.2 Geometric relationship of quadrilateral mesh

Again, suppose that a quadrilateral manifold mesh contains a sufficiently large number of edges and quadrilaterals, and that the ratio of the number of boundary edges to the number of non-boundary edges is negligible. Then, we can approximate the number of edges by

$$e \simeq 2f \tag{2.6}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

since an edge is shared by two quadrilaterals in general. Substituting equation 2.6 into equation 2.1, we have $v \simeq f + 2 - 2g$. Since f is much larger than $2 - 2g$, we get

$$v \simeq f \tag{2.7}$$

In other words, a typical quadrilateral mesh has as many quadrilaterals as vertices.

Also, from equation 2.6 and equation 2.7, we have an approximate relation

$$e \simeq 2v \tag{2.8}$$

It can be shown that the sum of degrees is twice the number of edges (14). Thus, we have

$$\sum degree = 2e \simeq 4v \tag{2.9}$$

It c
hav

$$\sum degree = 2e \simeq 4v \tag{2.9}$$

Thus, in a typical quadrilateral mesh, the average vertex degree is 4.

When reporting the compression performance, some papers employ the measure of bits per quad (bpq) while others use bits per vertex (bpv). For consistency, we adopt the bpv measure exclusively, and convert the bpq metric to the bpv metric by assuming that a mesh has as many quadrilaterals as vertices.

2.2 Connectivity compression

Single-rate compression is a typical mesh compression algorithm encodes connectivity data and geometry data separately. Most early work focused on the connectivity coding. Then, the coding order of geometry data is determined by the underlying connectivity coding. In this thesis, we focus on connectivity data compression only.

We classify existing single-rate connectivity compression algorithms into six classes: the indexed face set, the triangle strip, the spanning tree, the layered decomposition, the valence-driven approach, and the triangle conquest. They are described in detail below.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.1 Indexed face set

In the VRML ASCII format (7), a triangular mesh is represented with an indexed face set that consists of a coordinate array and a face array. The coordinate array lists the coordinates of all vertices, and the face array lists each face by indexing its three vertices in the coordinate array. For instance, Fig. 2.3 shows a mesh and its face array.

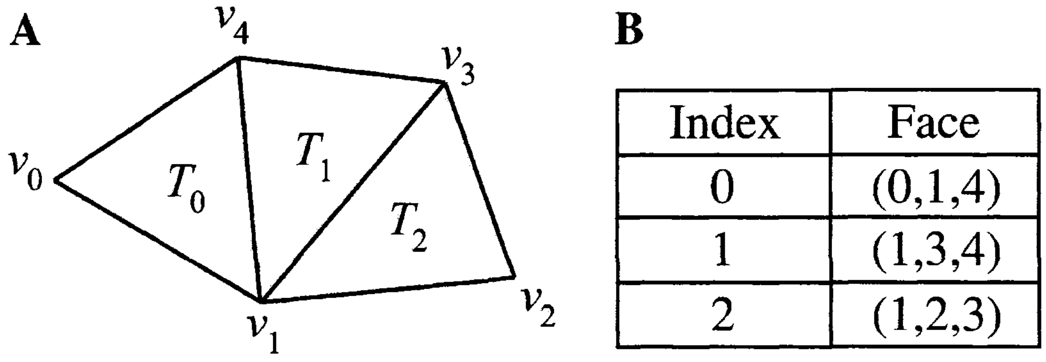
If there are v vertices in a mesh, the index of each vertex requires $log_2 v$ bits. Therefore, a triangular face needs $3log_2 v$ bits for its connectivity information. Since there are about twice triangles as many as vertices in a typical triangular mesh, the connectivity information costs about $6log_2 v$ bpv in the indexed face set method. This method provides a straightforward way for the triangular mesh representation. There is actually no compression involved in this method, but we still list it here to provide a basis of comparison for the following compression schemes.

In this method, each vertex is indexed several times by all its adjacent triangles. Repeated vertex references degrade the efficiency of connectivity coding. In other words, a good connectivity coding scheme should reduce the number of repeated vertex references. This observation leads to the triangle strip method.

2.2.2 Triangle strip

The triangle strip method attempts to divide a 3D mesh into long strips of triangles, and then encode these strips. The primary purpose of this method is to reduce the amount of data transmission between CPU and the graphic card, since triangle strips are well supported by most graphic cards. Although this scheme demands less storage space and transmission bandwidth than the indexed face set representation, it is still not very efficient for the compression purpose.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Index	Face
0	(0,1,4)
1	(1,3,4)
2	(1,2,3)

Figure 2.3: The indexed face set representation of a mesh: (A) a mesh example and (B) its face array.

Fig. 2.4A shows a triangle strip, where each vertex is combined with the previous two vertices in a vertex sequence to form a new triangle. Fig. 2.4B shows a triangle fan, where each vertex after the first two forms a new triangle with the previous vertex and the first vertex. Fig. 2.4C shows a generalized triangle strip that is a mixture of triangle strips and triangle fans. Note that, in a generalized triangle strip, a new triangle is introduced by each vertex after the first two in the vertex sequence. However, in an indexed face set, a new triangle is introduced by three vertices. Therefore, the generalized triangle strip provides a more compact representation than the indexed face set, especially when the strip length is long. In a rather long generalized triangle strip, the ratio of the number of triangles to the number of vertices is very close to 1, meaning that a triangle can be represented by almost exactly 1 vertex index.

However, since there are about twice as many triangles as vertices in a typical mesh, some vertex indices should be repeated in the generalized triangle strip representation of the mesh, which indicates a waste of storage. To alleviate this problem, several schemes have been developed, where a vertex buffer is utilized to store the indices of recently traversed vertices.

Deering (9) first introduced the concept of the generalized triangular mesh. A generalized triangular mesh is formed by combining generalized triangle strips with

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

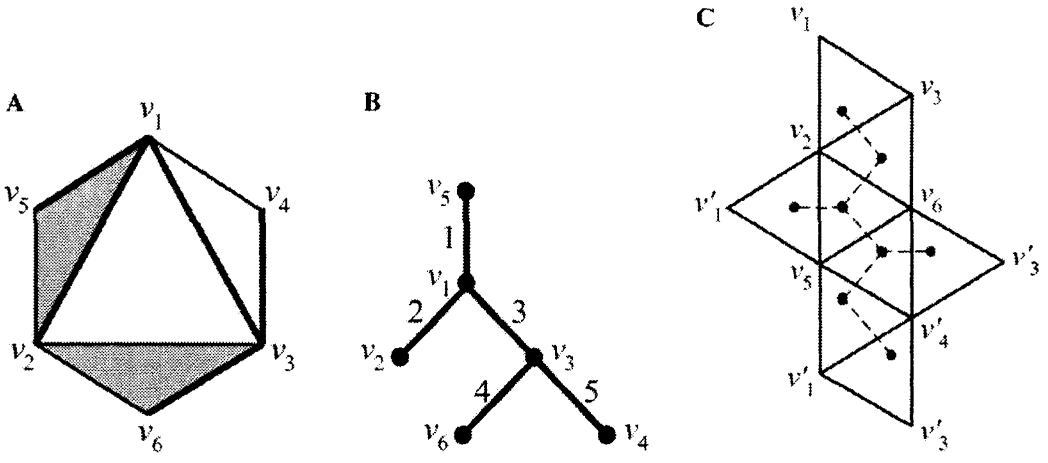


Figure 2.4: (A) The triangle strip, (B) the triangle fan, and (C) the generalized triangle strip.

a vertex buffer. He used a first-in-first-out (FIFO) vertex buffer to store the indices of up to 16 recently visited vertices. If a vertex is saved in the vertex buffer, it can be represented with the buffer index that requires a less number of bits than the global vertex index. Assuming that each vertex is reused by the buffer index only once, Taubin and Rossignac (34) showed that the generalized triangular mesh representation requires approximately 11 bpv to encode the connectivity data for large meshes. Deering, however, did not propose a method to decompose a mesh into triangle strips.

Based on Deering's work, Chow (8) proposed a mesh compression scheme optimized for real-time rendering. He proposed a mesh decomposition method, illustrated in Fig. 2.5. First, it finds a set of boundary edges. Then, it finds a fan of triangles around each vertex incident on two consecutive boundary edges. These triangle fans are combined to form the first generalized triangle strip. The triangles in this strip are marked as discovered, and a new set of boundary edges is generated to separate discovered triangles from undiscovered triangles. The next generalized triangle strip is similarly formed from the new set of boundary edges. With the vertex buffer, the vertices in the previous generalized triangle strip can be reused in the next one. This process continues until all triangles in a mesh are traversed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

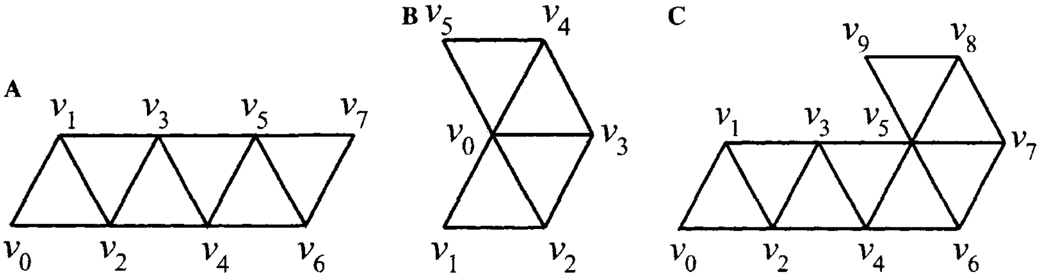


Figure 2.5: (A) A set of boundary edges, (B) triangle fans for the first strip, and (C) triangle fans for the second strip, where thick arrows show selected boundary edges and thin arrows show the triangle fans associated with each inner boundary vertex.

The triangle strip representation can be applied to a triangular mesh of arbitrary topology. However, it is effective only if the triangle mesh is decomposed into long triangle strips. It is a challenging computational geometry problem to obtain an optimal triangle strip decomposition (4, 11). Several heuristics have been proposed to obtain suboptimal decompositions at a moderate computational cost (12, 30, 38).

2.2.3 Spanning tree

Turan (36) observed that the connectivity of a planar graph can be encoded with a constant number of bpv using two spanning trees: a vertex spanning tree and a triangle spanning tree. Based on this observation, Taubin and Rossignac (34) presented a topological surgery approach to encode mesh connectivity. The basic idea is to cut a given mesh along a selected set of cut edges to make a planar polygon. The mesh connectivity is then represented by the structures of cut edges and the polygon. In a simple mesh, any vertex spanning tree can be selected as the set of cut edges.

Fig. 2.6 illustrates the encoding process. Fig. 2.6A is an octahedron mesh. First, the encoder constructs a vertex spanning tree as shown in Fig. 2.6B, where each node corresponds to a vertex in the input mesh. Then, it cuts the mesh along the edges

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the vertex spanning tree. Fig. 2.6C shows the resulting planar polygon and the triangle spanning tree. Each node in the triangle spanning tree corresponds to a triangle in the polygon, and two nodes are connected if and only if the corresponding triangles share an edge.

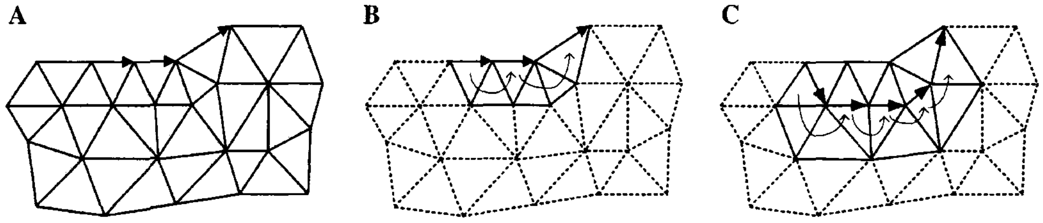


Figure 2.6: (A) An octahedron mesh, (B) its vertex spanning tree, and (C) the cut and flattened mesh with its triangle spanning tree shown by dashed lines.

Then, the two spanning trees are run-length encoded. A run is defined as a tree segment between two nodes with degrees not equal to 2. For each run of the vertex spanning tree, the encoder records its length with two additional flags. The first flag is the branching bit indicating whether a run subsequent to the current run starts at the same branching node, and the second flag is the leaf bit indicating whether the current run ends at a leaf node. For example, let us encode the vertex spanning tree in Fig. 2.6B, where the edges are labeled with their run indices. The first run is represented by (1,0,0), since its length is 1, the next run does not start at the same node, and it does not end at a leaf node. In this way, the vertex spanning tree in Fig. 2.6B is represented by

$$(1,0,0),\ (1,1,1),\ (1,0,0),\ (1,1,1),\ (1,0,1).$$

Similarly, for each run of the triangle spanning tree, the encoder writes its length and the leaf bit. Note that the triangle spanning tree is always binary so that it does not need the branching bit. Furthermore, the encoder records the marching pattern with one bit per triangle to indicate how to triangulate the planar polygon internally. The decoder can reconstruct the original mesh connectivity from this set

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of information.

In both vertex and triangle spanning trees, a run is a basic coding unit. Thus, the coding cost is proportional to the number of runs, which in turn depends on how the vertex spanning tree is constructed. Taubin and Rossignac's algorithm builds the vertex spanning tree based on layered decomposition, which is similar to the way we peel an orange along a spiral path, to maximize the length of each run and minimize the number of runs generated.

Taubin and Rossignac also presented several modifications so that their algorithm can encode general manifold meshes: meshes with arbitrary genus, meshes with boundary, and non-orientable meshes. However, their algorithm cannot directly handle non-manifold meshes. As a preprocessing step, the encoder should split a non-manifold mesh into several manifold components, thereby duplicating non-manifold vertices, edges, and faces. Experimentally, Taubin and Rossignac's algorithm costs 2.48-7.0 bpv for mesh connectivity. It was also shown that the time as well as the space complexities of their algorithm are $O(N)$, where N is the maximum of the vertex number v, the edge number e, and the triangle number f in a mesh. This method demands a large memory buffer due to its global random vertex access at the decompression stage.

2.2.4 Layered decomposition

Bajaj et al. (6) presented a connectivity coding method using a layered structure of vertices. It decomposes a triangular mesh into several concentric layers of vertices, and then constructs triangle layers within each pair of adjacent vertex layers. The mesh connectivity is represented by the total number of vertex layers, the layout of each vertex layer, and the layout of triangles in each triangle layer. Ideally, a vertex layer does not intersect itself and a triangle layer is a generalized triangle strip. In such a case, the connectivity compression is reduced to the coding of the number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertex layers, the number of vertices in each vertex layer, and the generalized triangle strip in each triangle layer. However, in practice, overhead bits are introduced due to the existence of branching points, bubble triangles, and triangle fans.

Branching points are generated when a vertex layer intersects itself. In Fig. 2.7A, the middle layer intersects itself at the branching point depicted by a big dot. Branching points divide a vertex layer into several segments called contours. To encode the layout of a vertex layer, we need to encode the information of both contours and branching points. Also, as shown in Figs. 2.7B-D, each triangle in a triangle layer can be classified into one of three cases.

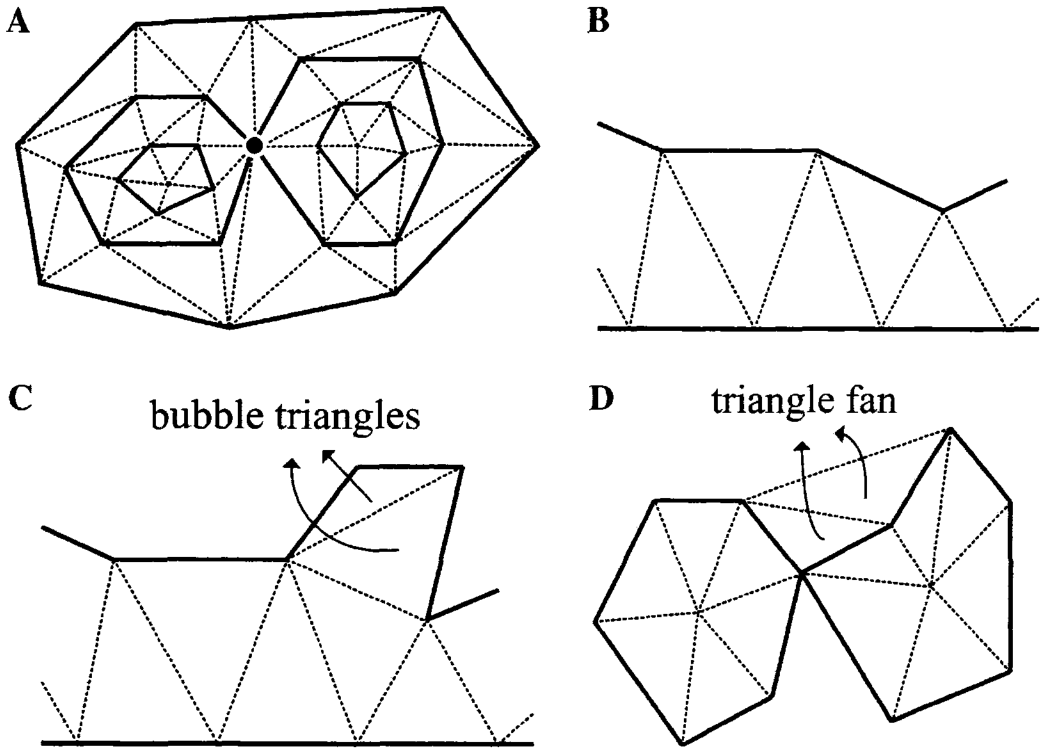


Figure 2.7: Fig. 7. Illustration of (A) the layered vertex structure and the branching point depicted by a black dot, (B) a triangle strip, (C) bubble triangles, and (D) a cross-contour triangle fan, where contours are depicted with solid lines and other edges with dashed lines.

- Its vertices lie on two adjacent vertex layers. A generalized triangle strip is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

composed of a sequence of triangles of this kind.

- All its vertices belong to one contour. It is called a bubble triangle.
- Its vertices lie on two or three contours in one vertex layer. A cross-contour triangle fan consists of a sequence of triangles of this kind.

Therefore, in addition to encoding generalized triangle strips between two adjacent vertex layers, this algorithm requires extra bits to encode bubble triangles and cross-contour triangle fans.

Taubin and Rossignac (34) also employed layered decomposition in the vertex spanning tree construction. However, Bajaj et al.'s algorithm (6) is different from (34) in the following:

- It does not combine vertex layers into the vertex spanning tree.
- Its decoder does not require a large memory buffer, since it accesses only a small portion of vertices at each decompression step.
- It is applicable to any kind of mesh topology, while (34) cannot encode non-manifold meshes directly.

The layered decomposition method encodes the connectivity information using about 1.40-6.08 bpv. Moreover, it has a desirable property. That is, each triangle depends on at most two adjacent vertex layers and each vertex is referenced by at most two triangle layers. This property enables the error-resilient transmission of mesh data, since the effects of transmission errors can be localized by encoding different vertex and triangle layers independently. Based on the layered decomposition method, Bajaj et al. (5) also proposed an algorithm to encode large CAD models. This algorithm extends the layered decomposition method to compress quadrilateral and general polygonal models as well as CAD models with smooth non-uniform rational B-splines (NURBS) patches.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.5 Valence-driven approach

The valence-driven approach starts from a seed triangle whose three edges form the initial borderline. The borderline divides the whole mesh into two parts, i.e., the inner part that has been processed and the outer part that is to be processed. Then, the borderline gradually expands outwards until the whole mesh is processed. The output is a stream of vertex valences, from which the original connectivity can be reconstructed.

In (35), Touma and Gotsman proposed a pioneering algorithm known as the valence-driven approach. It starts from an arbitrary triangle, and pushes its three vertices into a list called the active list. Then, it pops up a vertex from the active list, traverses all untraversed edges connected to that vertex, and pushes the new vertices into the end of the list. For each processed vertex, it outputs the valence. Sometimes, it needs to split the current active list or merge it with another active list. These cases are encoded with special codes. Before encoding, for each boundary loop, a dummy vertex is added and connected to all the vertices in that boundary loop, making the topology closed. Fig. 2.8 shows an example of the encoding process, where the active list is depicted by thick lines, the focus vertex by the black dot, and the dummy vertex by the gray dot. Table 2.1 lists the output of each step in association with Fig. 2.8.

Since vertex valences are compactly distributed around 6 in a typical mesh, arithmetic coding can be adopted to encode the valence information of a vertex effectively (35). The resulting algorithm uses less than 1.5 bpv on average to encode mesh connectivity. This is the state-of-the-art compression ratio which has not been seriously challenged till now. However, their algorithm is only applicable to orientable and manifold meshes.

Alliez and Desbrun (2) proposed a method to further improve the performance of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Touma and Gotsman's algorithm. They observed that split codes, split offsets, and dummy vertices consume a non-trivial portion of coding bits in Touma and Gotsman's algorithm. To reduce the number of split codes, they used a heuristic method that chooses the vertex with the minimal number of free edges as the next focus vertex, instead of choosing the next vertex in the active list. To reduce the number of bits for split offsets, they excluded the two adjacent vertices of the focus vertex in the current active list that are not eligible for the split, and sort the remaining vertices according to their Euclidean distances to the focus vertex. Then, a split offset is represented with an index into this sorted list, which is further added by 6 and encoded in the same way as a normal valence. To reduce the number of dummy vertices, they used one common dummy vertex for all boundaries in the input mesh. In addition, they encoded the output symbols with the range encoder (28), an effective adaptive arithmetic encoder.

Alliez and Desbrun's algorithm is also applicable only to orientable manifold meshes. It performs better than Touma and Gotsman's algorithm, especially for irregular meshes. Alliez and Desbrun proved that if the number of splits is negligible, the performance of their algorithm is upper-bounded by 3.24 bpv, which is exactly the same as the theoretical bpv value computed by enumerating all possible planar graphs (37).

2.2.6 Triangle conquest

Similar to the valence-driven approach, the triangle conquest approach starts from the initial borderline, which divides the whole mesh into conquered and unconquered parts, and inserts triangle by triangle into the conquered parts. The main difference is that the triangle conquest approach outputs the building operations of new triangles, while the valence-driven approach outputs the valences of new vertices.

Gumhold and Straßer (18) first proposed a triangle conquest approach, called the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

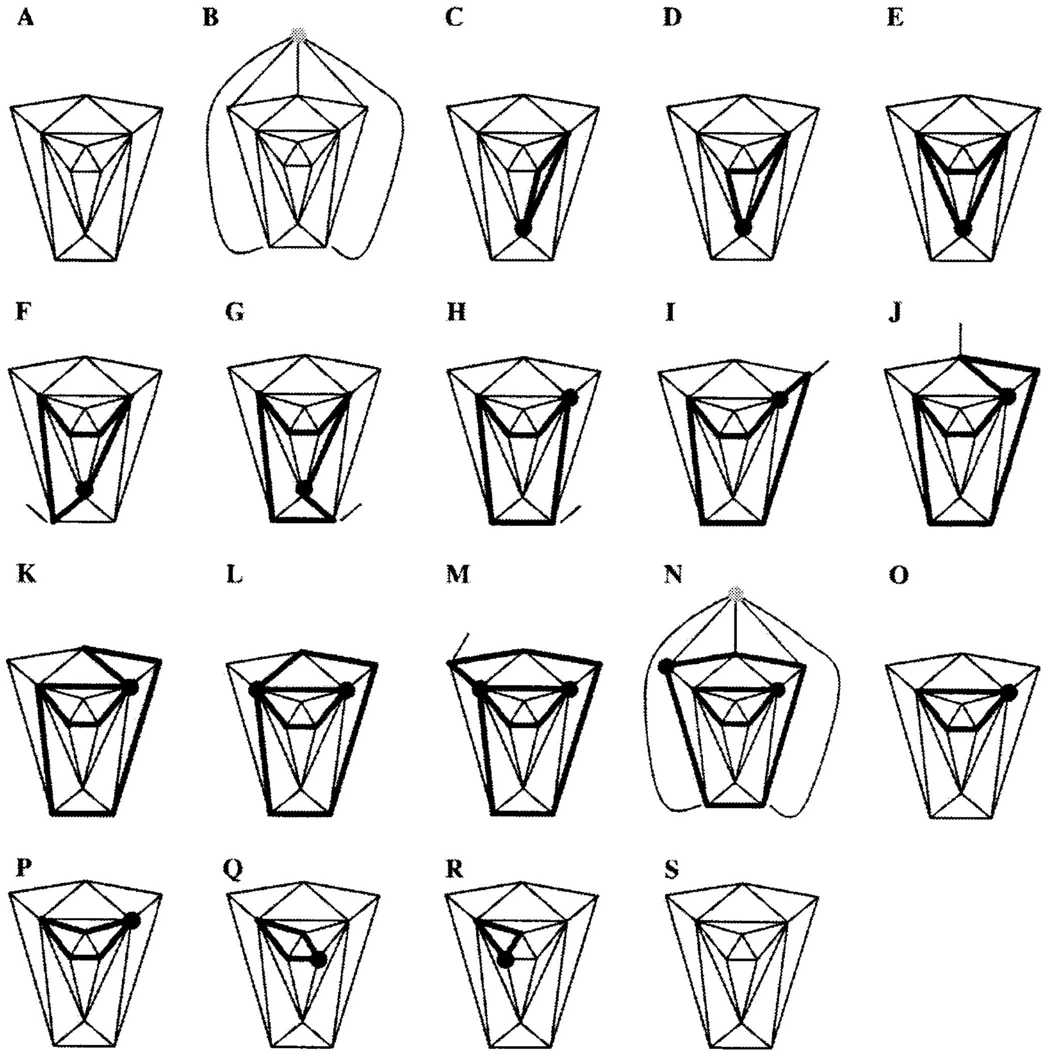


Figure 2.8: A mesh connectivity encoding example by Touma and Gotsman (35), where the active list is shown with thick lines, the focus vertex with the black dot, and the dummy vertex with the gray dot.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure	Output	Comments
(A)		An input mesh is given
(B)		Add a dummy vertex
(C)	Add 6, add 7, add 4	Output the valences of starting vertices
(D)	Add 4	Expand the active list
(E)	Add 7	Expand the active list
(F)	Add 5	Expand the active list
(G)	Add 5	Expand the active list
(H)		Choose the next focus vertex
(I)	Add 4	Expand the active list
(J)	Add 5	Expand the active list
(K)	Split 5	Split the active list, and push the new active list into stack
(L)		Choose the next focus vertex
(M)	Add 4	Expand the active list
(N)	Add dummy 5	Choose the next focus vertex and conquer the dummy vertex
(O)		Pop the new active list from the stack
(P)	Add 4	Expand the active list
(Q)		Choose the next focus vertex
(R)		Choose the next focus vertex
(S)		The whole mesh is conquered

Table 2.1: The output of each step in Fig. 2.8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cut-border machine. At each step, this algorithm inserts a new triangle into the conquered part, closed by the cut-border, using one of the five building operations: 'new vertex', 'forward,' 'backward', 'split', and 'close'. The sequence of building operations is encoded using Huffman codes. This algorithm can encode manifold meshes that are either orientable or non-orientable. Experimentally, its compression performance lies within 3.22-8.94 bpv, mostly around 4 bpv. Its most important feature is that the decompression speed is very fast and the decompression method is easy to implement in hardware. Moreover, compression and decompression operations can be processed in parallel. These properties make the method very attractive in real-time cod-ing applications. In (16), Gumhold further improved the compression performance using an adaptive arithmetic coder and optimizing the border encoding. The experimental compression ratio is within the range of 0.3-2.7 bpv, and on average 1.9 bpv.

Rossignac (26) proposed the edgebreaker algorithm, which is another example of the triangle conquest approach. It is nearly equivalent to the cut-border machine, except that it does not encode the offset data associated with the split operation. The triangle traversal is controlled by edge loops as shown in Fig. 2.9A. Each edge loop bounds a conquered region and contains a gate edge. At each step, this algorithm focuses on one edge loop and its gate edge is called the active gate, while the other edge loops are stored in a stack and will be processed later. Initially, for each connected component, one edge loop is defined. If the component has no physical boundary, two half edges corresponding to one edge are set as the edge loop. For example, in Fig. 2.9B, the mesh has no boundary and the initial edge loop is formed by g and gÆo, where gÆo is the opposite half edge of g. In Fig. 2.9C, the initial edge loop is the mesh boundary.

At each step, this algorithm conquers a triangle incident on the active gate, updates the current loop, and moves the active gate to the next edge in the updated loop. For each conquered triangle, this algorithm outputs an op-code. Assume that the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

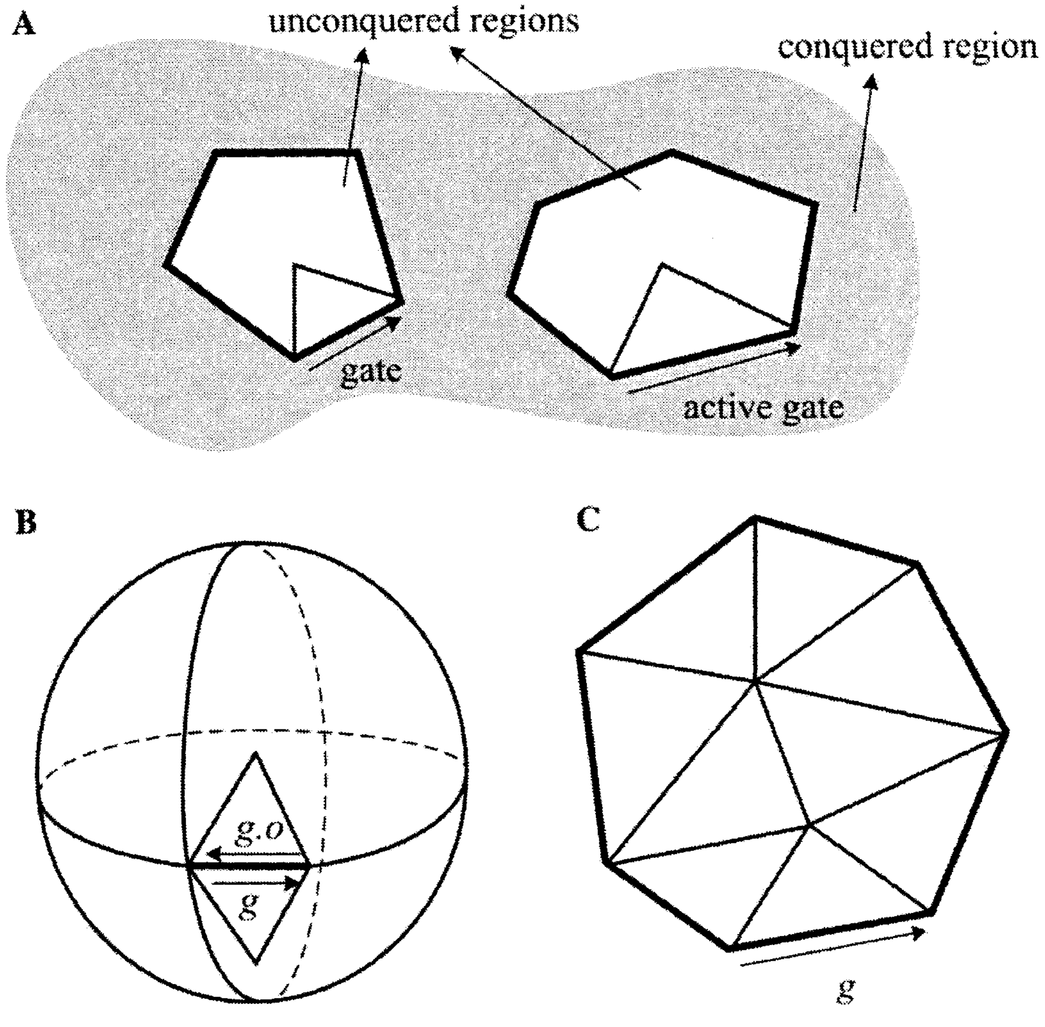


Figure 2.9: Illustration of (A) edge loops and (B) gates and initial edge loops for a mesh without boundary, and (C) gates and initial edge loops for a mesh with boundary, where thick lines depict edge loops, and g denotes the gate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

triangle to be removed is enclosed by active gate g and vertex v, there are five kinds of possible op-codes as shown in Fig. 2.10A:

- C (loop extension), if v is not on the edge loop;
- L (left), if v immediately precedes g in the edge loop;
- R (right), if v immediately follows g;
- E (end), if v precedes and follows g;
- S (split), otherwise.

Essentially, the compression process is a depth-first traversal of the dual graph of the mesh. When the split case is encountered, the current loop is split into two, and one of them is pushed into the stack while the other is further traced. Fig. 2.10B shows an example of the encoding process, where the arrows and the numbers give the order of the triangle conquest. The triangles are filled with different patterns to represent different op-codes, which are generated when they are conquered. For this case, the encoder outputs the series of op-codes as *CCRSRLLRSEERLRE*.

The edgebreaker method can encode the topology data of orientable manifold meshes with multiple boundary loops or with arbitrary genus, and guarantee a worst-case coding cost of 4 bpv for simple meshes. However, it is unsuitable for streaming applications, since it requires a two-pass process for decompression, and the decompression time is $O(v^2)$. Another disadvantage is that, even for regular meshes, it requires about the same bitrate as that for non-regular meshes.

King and Rossignac (23) modified the edgebreaker method to guarantee a worst-case coding cost of 3.67 bpv for simple meshes, and Gumhold (17) further improved this upper bound to 3.522 bpv. The decoding efficiency of the edgebreaker method was also improved to exhibit linear time and space complexities in (20, 23, 27). Furthermore, Szymczak et al. (31) optimized the edgebreaker method for meshes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

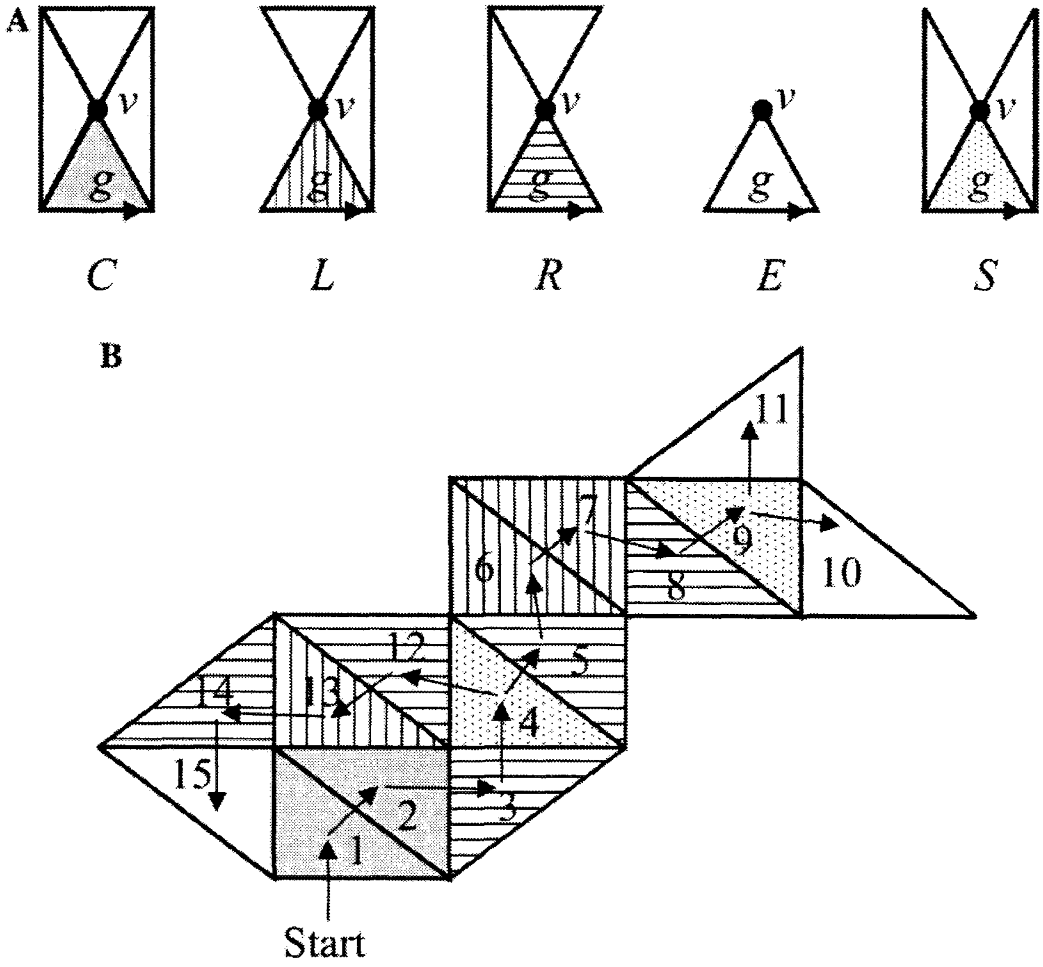


Figure 2.10: (A) Five op-codes C, L, R, E, and S, where the gate g is marked with an arrow, and (B) an example of the encoding process in the edgebreaker algorithm where the arrows and the numbers show the traversal order and different filling patterns are used to represent different op-codes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with high regularity by exploiting dependencies of output symbols. It guarantees a worst-case performance of 1.622 bpv for sufficiently large meshes with high regularity.

As mentioned earlier, we can reduce the amount of data transmission between CPU and the graphic card by decomposing a mesh into long triangle strips, but finding a good decomposition (or stripification) is often computationally intensive. Thus, it is often desirable to generate long strips from a given mesh only once and distribute the stripification information together with the mesh. Based on this observation, Isenburg (19) proposed an approach to encode the mesh connectivity together with its stripification information. It is basically a modification of the edgebreaker method, but its traversal order is guided by strips obtained with the STRIPE algorithm (12). When a new triangle is included, its relation to the underlying triangle strip is encoded with a label. The label sequences are then entropy encoded. The experimental compression performance ranges from 3.0 to 5.0 bpv.

2.3 Summary

Among the various connectivity coding methods discussed in this chapter, Touma and Gotsman's algorithm (35) is considered as the state-of-the-art technique for single-rate 3D mesh compression. With some minor improvements on Touma and Gotsman's algorithm, Alliez and Desbrun's algorithm (2) yields an improved compression ratio.

The indexed face set, triangle strip, and layered decomposition methods can encode meshes with arbitrary topology. In contrast, the other approaches can handle only manifold meshes with additional constraints. For instance, the valence-driven approach (2, 35) require that the manifold is also orientable. Szymczak et al.'s algorithm (31) requires that the manifold has neither boundary nor handles. Note that using these algorithms, a non-manifold mesh can be handled only if it is pre-converted to a manifold mesh by replicating non-manifold vertices, edges, and faces as in (15).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The methods discussed in this chapter focus on triangular mesh compression. In next two chapters, we will propose two direct techniques to compress and decompress the connectivity of quadmeshes, both of which are extended from the algorithms discussed in this chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Quadmesh compression and decompression using Touma-Gotsman algorithm

Touma-Gotsman's algorithm is a valence-driven approach, originally for triangle mesh compression. It starts from a seed triangle whose three edges form the initial borderline. The borderline divides the whole mesh into two regions, i.e., the inner region that has been processed and the outer region that is to be processed. Then, the borderline gradually expands outwards until the whole mesh is processed. The output is a stream of vertex valences, from which the original connectivity can be reconstructed.

The details of this algorithm for triangle mesh compression has been explained in section 2.2.5. In this chapter we show how to expand the algorithm for quadmesh connectivity compression.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1 Definitions

First, we make some definitions relevant to the algorithm. Most of them have been defined in Touma-Gotsman's paper (35).

Vertex cycle A cyclic sequence of vertices along triangle edges in the mesh.

Active List A vertex cycle in the mesh. The active list partitions the mesh into an "outer" part containing edges not yet encoded, and an "inner" part containing edges already encoded. Each vertex in the active list has encoded and unencoded incident edges separated by the edges to the two vertices which are its predecessor and successor in the active list.

Focus One vertex in the active list is designated as the focus vertex. All coding operations are done on the focus vertex.

We define a new term called "focus edge". Focus edge is an opposite half edge of an edge in the active list, and it takes focus vertex as its start point. So, we must be aware that focus edge is not in the active list. Actually, The quad which focus edge belongs to is the one which will be conquered next by the active list.

Free Vertex A vertex not yet encoded.

Full Vertex A vertex with no free edges. Here we introduce another term "almost full" to describe a vertex which has only one unvisited quad left around it.

3.2 Definition of opposite half edge (OHE) data structure

We propose a data structure, called OHE (opposite half edge) for representing the ajacencies of quadmesh.

Fig. 3.1 illustrates the OHE data structure. Each quad in a quad mesh is represented by four integer references for the four vertices and four integer references

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

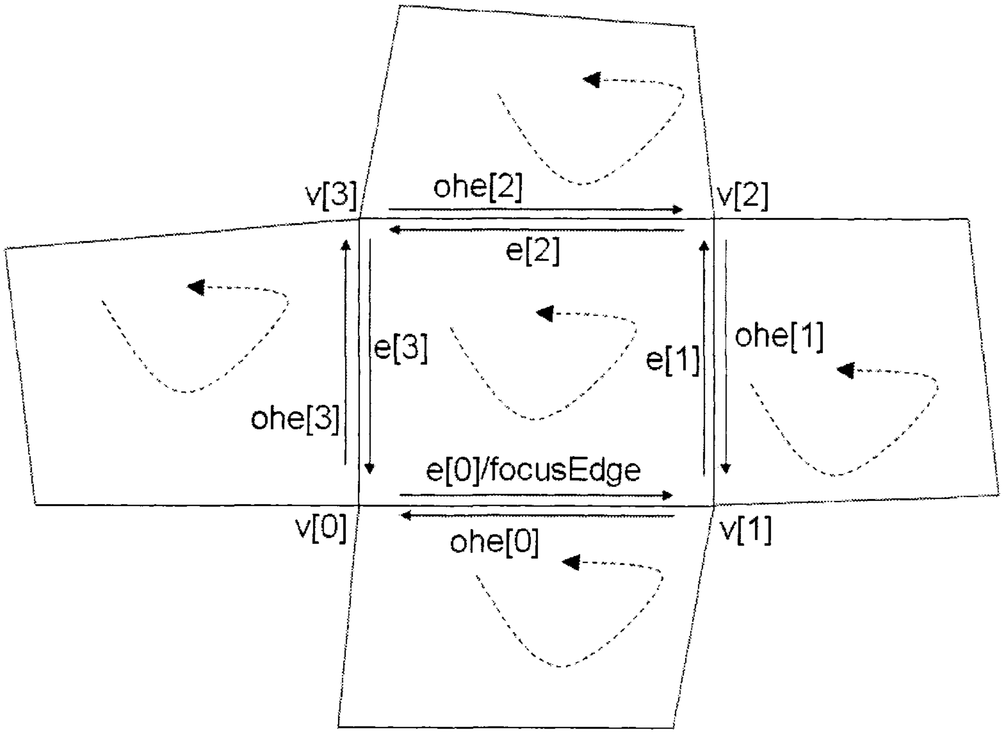


Figure 3.1: OHE data structure

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for the four edges, plus four integer references for the four opposite half edges. The opposite half edge of an edge $e[1]$ in the center quad refers to the left edge in the right quad. If $e[1]$ is a boundary edge, we assign -1 in the opposite-half-edge field $ohe[1]$.

Suppose the quad is the one to be conquered right now, which means that focus edge is one of the four edges in the quad. In our implementation, we re-label the vertices and edges of the quad, so that focus edge will be edge $e[0]$ in the quad, and edge $ohe[0]$ is in the active list. This is a very important step which makes the implementation easier to understand.

3.3 Algorithm description

A simple example demonstrating the operation of the algorithm for triangle mesh appears in Fig 2.8. The input of the algorithm is an orientable manifold quadmesh, and the output is the code for the mesh connectivity.

Our algorithm is slightly different from what Touma-Gotsman explained, but both follow the same idea.

The encoding algorithm starts off with an arbitrary quad in the mesh, defining an active list of four edges. An arbitrary opposite half edge of this quad is designated as the focus edge. The algorithm proceeds by trying to expand the active list by "conquering" an unvisited quad[1] (if exists) in counter-clockwise order around the start of the focus edge, which is the focus vertex. Some commands ("add", "split", or "merge") will be generated when conquering that unvisited quad. Each time we conquer an unvisited quad, we make a decision where the next focus should be, no matter weather current focus vertex is full or not.[2] The new focus vertex could be

---

[1] In Touma-Gotsman's implementation, they conquer edges rather than polygons.

[2] This is quite different from Touma-Gotman's implementation. They insist that when coding operations on the focus vertex are complete, the focus will move to the next vertex in the active list and the previous focus vertex is removed from the active list to become one of the "inner" vertices. We did it in another way.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the current focus vertex, or could be a vertex n adjacent to the current focus vertex, or some other vertex.

The conquering procedure repeats for the new focus, and terminates when all the quads have been traversed.

For the unvisited quad to be conquered, two vertices in the quad are end points of the focus edge, which have already been encoded. The problem is to encode the other two vertices. If any of the other two vertices is free (has not been encoded), we create an “add” command for it. If any of the other two vertices in the unvisited quad has been encoded during encoding, there are two possibilities: either it leads to a vertex in the same active list or to a vertex in another active list. In the first case, the active list is split. The encoding procedure proceeds with one, and push the others onto the stack for future treatment. In the latter case, the active lists are merged to form one active list on which the encoding continues. there will be no merge commands if the object has sphere topology (genus 0), and can only occur in a torus-like topology (non-genus-0).

The compression algorithm is shown in algorithm 1. The input file is a quad PLY file, and the output files are two files, one file “out_commmand.txt” which contains a sequence of commands, the other one is file “out_vertex.txt” which contains vertex coordinates, as shown below.

$$\boxed{quadPLYfile \rightarrow \begin{array}{l} out\_command.txt \\ out\_vertex.txt \end{array}}$$

The decompression algorithm is shown in algorithm 3. The two input files are “out_command.txt” and “out_vertex.txt”, and the output is a quad PLY file.

$$\boxed{\begin{array}{l} out\_command.txt \\ out\_vertex.txt \end{array} \rightarrow quadPLYfile}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

During implementation, we noted that there are 7 different cases that need to be handled for triangle mesh compression. However, there are 18 different cases that need to be handled for quadmesh compression.

In each case, the current active list might expand or decrease by conquering one unvisited quad, or it might be removed and we pop another active list from the list stack for traversing.

Each such case would generate 1 command, or 2 commands, or even 3 commands. Table 3.3 shows the number of commands created for each case. For case 1, 2, 3, 4.2, 4.3, no commands are generated, while case 7.3.2 generates 3 commands. All the other cases generate 2 commands.

The number of "add" commands in the connectivity code is the same as the number of vertices in the quadmesh, since each free vertex introduce one "add" command. Table 3.1 shows the number of free vertices found in different cases.

Table 3.3 shows the effects of size of the current active list in different cases. The column "Effects on Size" shows how the size of the current active list changes. For example, the size will be the same for case 4.1.1, while it will increase by 2 in case 5.1. The symbol "$-\infty$" is used to indicate that the size of the active list decreases to 0, and the active list is deleted. For split and merge commands generated from vertex $v[2]$ or $v[3]$, they either create a new active list, or remove an existing active list from list stack. In Table 3.3 T_1 is the size of the active list involved on $v[2]$, while T_2 is the size of the active list involved on $v[3]$. The exception is, for case 7.3.2 where T_1 is the size of the active list found from the list stack, while T_2 is the size of the active list created by the split command.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
input  : file_in_ply
output: file_out_command, file_out_vertex
init(file_in_ply, 4);
while not listStack.isEmpty() do
    activeList = listStack.pop();
    while not activeList.isEmpty() do
        if activeList.size() $< 3$ or activeList.areAllVerticesFull() then
            activeList.clear();                                   /* case 1, case 2 */
        else
            currentPolygon = getRevisedCurrentPolygon();
            if LastPolygonOutsideActiveList() then activeList.clear(); :     /* case 3 */
            else
                if isLastUnvisitedPolygonAroundFocusEdge() then           /* case 4 */
                    handleLastUnvisitedPolygonAroundFocusEdgeForCompression();
                else if currentPolygon.v[2].isUnvisited() then            /* case 5 */
                    if currentPolygon.v[3].isUnvisited() then handle case 5.1;
                    else if activeList.contains(currentPolygon.v[3]) then handle case 5.2;
                    else handle case 5.3;
                else if activeList.contains(currentPolygon.v[2]) then     /* case 6 */
                    if currentPolygon.v[3].isUnvisited() then handle case 6.1;
                    else if activeList.contains(currentPolygon.v[3]) then handle case 6.2;
                    else handle case 6.3;
                else                                                      /* case 7 */
                    if currentPolygon.v[3].isUnvisited() then handle case 7.1;
                    else if activeList.contains(currentPolygon.v[3]) then handle case 7.2;
                    else                                                  /* case 7.3 */
                        if currentPolygon.v[2] and currentPolygon.v[3] not on same list then
                            handle case 7.3.1;
                        else
                            handle case 7.3.2;        /* one more split command created */
            increaseNumOfVisitedEdgesForPolygon(currentPolygon);
createCommandFile(file_out_command, commandStack);
createVertexFile(file_out_vertex, newV);
```

Algorithm 1: Quadmesh compression algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
getRevisedCurrentPolygon():
switch type do            /* how many edges of currentPolygon are in activeList */
    case 2                                                        /* case 4.1 */
        if currentPolygon.v[2].isUnvisited() then handle case 4.1.1;
        else if activeList.contains(currentPolygon.v[3]) then handle case 4.1.2;
        else handle case 4.1.3;
    case 3                                                        /* case 4.2 */
        handle case 4.2;
    case 4                                                        /* case 4.3 */
        activeList.clear();
```

Function `handleLastUnvisitedPolygonAroundFocusEdgeForCompression`

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
input  : file_in_command, file_in_vertex
output: file_out_ply
init(file_in_command, file_in_vertex, 4);
while not listStack.isEmpty() do
    activeList = listStack.pop();
    while not activeList.isEmpty() do
        if activeList.size() < 3 or activeList.areAllVerticesFull() then
            activeList.clear();                                    /* case 1, case 2 */
        else
            currentCommand1 = commandStack.removeFirst();
            currentCommand2 = commandStack.removeFirst();
            if LastPolygonOutsideActiveList() then activeList.clear(); ;   /* case 3 */
            else
                if isLastUnvisitedPolygonAroundFocusEdge() then             /* case 4 */
                    handleLastUnvisitedPolygonAroundFocusEdgeForDecompression();
                else if currentPolygon.v[2].isUnvisited() then              /* case 5 */
                    if currentPolygon.v[3].isUnvisited() then handle case 5.1;
                    else if activeList.contains(currentPolygon.v[3]) then handle case 5.2;
                    else handle case 5.3;
                else if activeList.contains(currentPolygon.v[2]) then       /* case 6 */
                    if currentPolygon.v[3].isUnvisited() then handle case 6.1;
                    else if activeList.contains(currentPolygon.v[3]) then handle case 6.2;
                    else handle case 6.3;
                else                                                        /* case 7 */
                    if currentPolygon.v[3].isUnvisited() then handle case 7.1;
                    else if activeList.contains(currentPolygon.v[3]) then handle case 7.2;
                    else                                                    /* case 7.3 */
                        if currentPolygon.v[2] and currentPolygon.v[3] not on same list then
                            handle case 7.3.1;
                        else
                            handle case 7.3.2;          /* one more split command popped */
            increaseNumOfVisitedEdgeForPolygon(currentPolygon);
createPLYFile(file_out_ply, V, P);
```

Algorithm 3: Quadmesh decompression algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
getRevisedCurrentPolygon();
switch type do          /* how many edges of currentPolygon are in activeList */
    case 2                                                      /* case 4.1 */
        if currentPolygon.v[2].isUnvisited() then handle case 4.1.1;
        else if activeList.contains(currentPolygon.v[3]) then handle case 4.1.2;
        else handle case 4.1.3;
    case 3                                                      /* case 4.2 */
        handle case 4.2;
    case 4                                                      /* case 4.3 */
        activeList.clear();
```

Function `handleLastUnvisitedPolygonAroundFocusEdgeForDecompression`

Case #	Case Index	# of Free Vertices	Comments
case 1	1	0	activeList.size() < 3
case 2	2	0	all vertices in activeList are full
case 3	3	0	last quad outside activeList
case 4	4.1.1	1	2 edges in activeList & v[2].isUnvisited()
case 5	4.1.2	0	2 edges in activeList & v[2].split()
case 6	4.1.3	0	2 edges in activeList & v[2].merge()
case 7	4.2	0	3 edges in activeList
case 8	4.3	0	4 edges in activeList
case 9	5.1	2	v[2].isUnvisited() & v[3].isUnvisited()
case 10	5.2	1	v[2].isUnvisited() & v[3].split()
case 11	5.3	1	v[2].isUnvisited() & v[3].merge()
case 12	6.1	1	v[2].split() & v[3].isUnvisited()
case 13	6.2	0	v[2].split() & v[3].split()
case 14	6.3	0	v[2].split() & v[3].merge()
case 15	7.1	1	v[2].merge() & v[3].isUnvisited()
case 16	7.2	0	v[2].merge() & v[3].split()
case 17	7.3.1	0	v[2].merge() & v[3].merge() & v[2]/v[3] not in same list
case 18	7.3.2	0	v[2].merge() & v[3].merge() & v[2]/v[3] in same list

Table 3.1: Free vertices found in different cases

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Case Index	Effects on Size	Comments
1	$-\infty$	activeList.size() < 3
2	$-\infty$	all vertices in activeList are full
3	$-\infty$	last quad outside activeList
4.1.1	0	2 edges in activeList & v[2].isUnvisited()
4.1.2	$-\infty$	2 edges in activeList & v[2].split()
4.1.3	T_1	2 edges in activeList & v[2].merge()
4.2	-2	3 edges in activeList
4.3	$-\infty$	4 edges in activeList
5.1	$+2$	v[2].isUnvisited() & v[3].isUnvisited()
5.2	$-T_1+3$	v[2].isUnvisited() & v[3].split()
5.3	T_1+2	v[2].isUnvisited() & v[3].merge()
6.1	$-T_1+3$	v[2].split() & v[3].isUnvisited()
6.2	$-T_1-T_2+4$	v[2].split() & v[3].split()
6.3	$-T_1+T_2+3$	v[2].split() & v[3].merge()
7.1	T_1+2	v[2].merge() & v[3].isUnvisited()
7.2	T_1-T_2+3	v[2].merge() & v[3].split()
7.3.1	T_1+T_2+2	v[2].merge() & v[3].merge() & v[2]/v[3] not in same list
7.3.2	T_1-T_2+2	v[2].merge() & v[3].merge() & v[2]/v[3] in same list

Table 3.2: Effects on size of current active list

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Case Index	# of Commands	Comments
1	0	activeList.size() < 3
2	0	all vertices in activeList are full
3	0	last quad outside activeList
4.1.1	1	2 edges in activeList & v[2].isUnvisited()
4.1.2	2	2 edges in activeList & v[2].split()
4.1.3	1	2 edges in activeList & v[2].merge()
4.2	0	3 edges in activeList
4.3	0	4 edges in activeList
5.1	2	v[2].isUnvisited() & v[3].isUnvisited()
5.2	2	v[2].isUnvisited() & v[3].split()
5.3	2	v[2].isUnvisited() & v[3].merge()
6.1	2	v[2].split() & v[3].isUnvisited()
6.2	2	v[2].split() & v[3].split()
6.3	2	v[2].split() & v[3].merge()
7.1	2	v[2].merge() & v[3].isUnvisited()
7.2	2	v[2].merge() & v[3].split()
7.3.1	2	v[2].merge() & v[3].merge() & v[2]/v[3] not in same list
7.3.2	3	v[2].merge() & v[3].merge() & v[2]/v[3] in same list

Table 3.3: Number of commands created in each case

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.1 Compression process

The 18 cases are handled in the following ways for compression.

1. activeList.size() $<$ 3

- Remove (delete) the active list.

2. All vertices in active list are full

- Remove (delete) the active list.

3. Last unprocessed quad outside

In this case, size of the active list is 4, and all vertices are almost full.

- increase "numOfVisitedEdge" for all vertices of the quad by 1;
- Remove (delete) the active list.

4. Last unvisited polygon around focus edge

There are 3 subcases.

4.1. ohe[0], ohe[3] are in active list

There are 3 subcases.

4.1.1. v[2].isFree

- Set v[2] "visited";
- Push v[2] to "newV" stack (for creating ordered vertex file);
- create an add command for v[2];
- reconstruct the current active list;
- set focus edge for the current acitve list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.1.2. v[2].Split

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- get "offset" of v[2];
- create a split command for v[2];
- create 2nd active list, set focus edge for it and push it to list stack;
- reconstruct the current active list;
- set focus edge for the current acitve list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.1.3. v[2].Merge

- find the 2nd list (intersects at v[2]) from list stack;
- create a merge command for v[2];
- reconstruct the current active list (merge two lists);
- set focus edge for the current acitve list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.2. ohe[0], ohe[3], ohe[2] are in active list

- reconstruct the current active list;
- set focus edge for the current acitve list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.3. ohe[0], ohe[3], ohe[2], ohe[1] are in active list

This case happens only in non-genus-0 mesh. In this case, the size of the current active list is 4, and all vertices in the active list are in the same quad. The difference between the active list in case 3 and in this case is: In case 3, all vertices are "almost full", while in this case, all other vertices are "almost full" except one ($v[1]$).

- increase "numOfVisitedEdge" for all vertices of the quad by 1;
- remove (delete) the active list.

5. v[2].isFree

There are 3 subcases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.1. v[2].isFree and v[3].isFree

- Set v[2] and v[3] "visited";
- Push v[2] and v[3] to "newV" stack (for creating ordered vertex file);
- create add commands for v[2] and v[3];
- reconstruct the current active list;
- set focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

5.2. v[2].isFree and v[3].Split

- Set v[2] "visited";
- Push v[2] to "newV" stack (for creating ordered vertex file);
- create an add command for v[2];
- get "offset" of v[3];
- create a split command for v[3];
- create 2nd active list, set focus edge for it and push it to list stack;
- reconstruct the current active list;
- set focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

5.3. v[2].isFree and v[3].Merge

- Set v[2] "visited";
- Push v[2] to "newV" stack (for creating ordered vertex file);
- create an add command for v[2];
- find the 2nd list (intersects at v[3]) from list stack;
- create a merge command for v[3];
- reconstruct the current active list (merge two lists);
- set focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6. v[2].Split

There are 3 subcases.

6.1. v[2].Split and v[3].isFree

- get “offset” of v[2];
- create a split command for v[2];
- create 2nd active list, set focus edge for it and push it to list stack;
- Set v[3] “visited”;
- Push v[3] to “newV” stack (for creating ordered vertex file);
- create an add command for v[3];
- reconstruct the current active list;
- set focus edge for the current active list;
- increase “numOfVisitedEdge” for all vertices of the quad by 1.

6.2. v[2].Split and v[3].Split

- get “offset” of v[2];
- create a split command for v[2];
- create 2nd active list, set focus edge for it and push it to list stack;
- get “offset” of v[3];
- create a split command for v[3];
- create 3rd active list, set focus edge for it and push it to list stack;
- reconstruct the current active list;
- set focus edge for the current active list;
- increase “numOfVisitedEdge” for all vertices of the quad by 1.

6.3. v[2].Split and v[3].Merge

- get “offset” of v[2];
- create a split command for v[2];

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- create 2nd active list, set focus edge for it and push it to list stack;
- find a 3rd list (intersects at v[3]) from list stack;
- create a merge command for v[3];
- reconstruct the active list (merge the 3rd list and current active list);
- set focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

7. v[2].Merge

There are 3 subcases.

7.1. v[2].Merge and v[3].isFree

- find a 2nd list (intersects at v[2]) from list stack;
- create a merge command for v[2];
- Set v[3] "visited";
- Push v[3] to "newV" stack (for creating ordered vertex file);
- create add commands for v[3];
- reconstruct the current active list;
- set focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

7.2. v[2].Merge and v[3].Split

- find a 2nd list (intersects at v[2]) from list stack;
- create a merge command for v[2];
- get "offset" of v[3];
- create a split command for v[3];
- create 3rd active list, set focus edge for it and push it to list stack;
- reconstruct the current active list;
- set focus edge for the current active list;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- increase “numOfVisitedEdge” for all vertices of the quad by 1.

7.3. v[2].Merge and v[3].Merge There are 2 subcases.

7.3.1. v[2].Merge and v[3].Merge, v[2]/v[3] are not on same list

- find the 2nd list (intersects at v[2]) from list stack;
- create a merge command for v[2];
- find the 3rd list (intersects at v[3]) from list stack;
- create a merge command for v[3];
- reconstruct the active list (merge the three lists);
- set focus edge for the active list;
- increase “numOfVisitedEdge” for all vertices of the quad by 1.

7.3.2. v[2].Merge and v[3].Merge, v[2]/v[3] are on same list

- find the 2nd list (intersects at v[2]) from list stack;
- create a merge command for v[2];
- create a merge command for v[3];
- get distance (offset) between v[2] and v[3] along the 2nd list;
- create a split command for v[3] on the 2nd list;
- create 3rd active list, set focus edge for it and push it to list stack;
- reconstruct the current active list;
- set focus edge for the current active list;
- increase “numOfVisitedEdge” for all vertices of the quad by 1.

3.3.2 Decompression process

The 18 cases are handled in the following ways for decompression.

1. activeList.size() < 3

- Remove (delete) the active list.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. All vertices in active list are full

- Remove (delete) the active list.

3. Last unprocessed quad outside

In this case, size of the active list is 4, and all vertices are almost full.

- create a new quad with all vertices on current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1;
- Remove (delete) the active list.

4. Last unvisited polygon around focus edge

There are 3 subcases.

4.1. ohe[0], ohe[3] are in active list

There are 3 subcases.

4.1.1. v[2].isFree

(read next command, which is an add command)

- set v[2] to be next vertex in vertex list, and set degree infomation for it;
- create a new quad with focusEdge, v[2] and nextOfFocusVertex;
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.1.2. v[2].Split

(read next command, which is a split command)

- set v[2] to be a vertex in current active list;
- create a new quad with focusEdge, v[2] and nextOfFocusVertex;
- reconstruct the active list;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.1.3. v[2].Merge

(read next command, which is a merge command)

- set v[2] to be a vertex in an active list from list stack;
- create a new quad with focusEdge, v[2] and nextOfFocusVertex;
- reconstruct the current active list (merge two lists);
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.2. ohe[0], ohe[3], ohe[2] are in active list

- create a new quad with focusEdge, nextOfFocusVertex and nextnextOfFocusVertex;
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

4.3. ohe[0], ohe[3], ohe[2], ohe[1] are in active list

- create a new quad with focusEdge, nextOfFocusVertex and nextnextOfFocusVertex;
- increase "numOfVisitedEdge" for all vertices of the quad by 1;
- remove (delete) the active list.

5. v[2].isFree

There are 3 subcases.

5.1. v[2].isFree and v[3].isFree

(read next two commands which are two add commands)

- set v[2] to be next vertex in vertex list, and set degree infomation for it;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- set v[3] to be next vertex in vertex list, and set degree infomation for it;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

5.2. v[2].isFree and v[3].Split

(read next two commands, 1st is an add command, next is split)

- set v[2] to be next vertex in vertex list, and set degree infomation for it;
- set v[3] to be a vertex in current active list;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

5.3. v[2].isFree and v[3].Merge

(read next two commands, 1st is an add command, next is merge)

- set v[2] to be next vertex in vertex list, and set degree infomation for it;
- set v[3] to be a vertex in an active list from list stack;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list (merge two lists);
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

6. v[2].Split

There are 3 subcases.

6.1. v[2].Split and v[3].isFree

(read next two commands, 1st is a split command, next is add)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- set v[2] to be a vertex in current active list;
- set v[3] to be next vertex in vertex list, and set degree infomation for it;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

6.2. v[2].Split and v[3].Split

(read next two commands, which are split commands)

- set v[2] to be a vertex in current active list;
- set v[3] to be a vertex in current active list;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

6.3. v[2].Split and v[3].Merge

(read next two commands, 1st is a split command, next is merge)

- set v[2] to be a vertex in current active list;
- set v[3] to be a vertex in an active list from list stack;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

7. v[2].Merge

There are 3 subcases.

7.1. v[2].Merge and v[3].isFree

(read next two commands which are two add commands)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- set v[2] to be a vertex in an active list from list stack;
- set v[3] to be next vertex in vertex list, and set degree infomation for it;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

7.2. v[2].Merge and v[3].Split

(read next two commands, 1st is an add command, next is split)

- set v[2] to be a vertex in an active list from list stack;
- set v[3] to be a vertex in current active list;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

7.3. v[2].Merge and v[3].Merge

There are 2 subcases.

7.3.1. v[2].Merge and v[3].Merge, v[2]/v[3] are not on same list

(read next two commands, both are merge commands)

- set v[2] to be a vertex in an active list from list stack;
- set v[3] to be a vertex in an active list from list stack;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list (merge the three lists);
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

7.3.2. v[2].Merge and v[3].Merge, v[2]/v[3] are on same list

(read next 3 commands, two merge commands, 1 split command)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- set v[2] to be a vertex in an active list from list stack;
- set v[3] to be a vertex in the same active list from list stack;
- create a new quad with focusEdge, v[2] and v[3];
- reconstruct the current active list;
- set virtual focus edge for the current active list;
- increase "numOfVisitedEdge" for all vertices of the quad by 1.

3.4 Illustrate the approach

To illustrate the algorithm, we use a quadmesh of a twelve faced torus shown in Fig. 3.2(A) as example. The surface, shown in Fig. 3.2(B), is obtained from the torus in Fig. 3.2(A) by cutting along its four edges $(v10, v4)$, $(v4, v5)$, $(v5, v11)$ and $(v11, v10)$ and laying it flat on the ground to produce a two dimensional image representation which can be represented on paper.

For each vertex in the mesh, we need to know two properties of it: it's degree (field *degree*), and how many quads/edges around it has been visited (field $numOfVisitedEdge$). In our example, each vertex in the figures is shown in the following format:

$$vertexIndex(degree/numOfVisitedEdge)$$

For example, in Fig. 3.2(C), the degree of vertex $v10$ is 4, and number of visited quads/edges around vertex $v10$ is 0. So, in Fig. 3.2(C), vertex $v10$ is described as:

$$10(4/0)$$

Fig. 3.2(A) to Fig. 3.2(U) shows how the compression algorithm works for the sample mesh, and Fig. 3.3(A) to Fig. 3.3(U) shows how the decompression algorithm works.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.4.1 Compression of a torus mesh

After reading the mesh's information from an input file, we set degree information for each vertex in the mesh, as shown in Fig. 3.2(C). From this step, we can see that the value of field *numOfVisitedEdge* for each vertex is 0, which means that all vertices are free vertices now.

The compression process starts with one quad in the mesh. Here we start the compression with the first quad $(v0, v1, v3, v2)$, defining the first active list $(v0, v1, v3, v2)$ of four edges which are edges of the first quad, as shown in Fig. 3.2(D). The focus edge $(v0, v2)$ is the opposite half edge of the last edge of the quad, and focus vertex $(v0)$ is the first vertex of the quad. We should always keep in mind that: first, focus vertex is the start vertex of the focus edge; secondly, the focus edge is not in the active list, while it's opposite half edge is in the active list; thirdly, each time we create a new active list, we need to assign a focus edge for it.

Next, by enumeration, we can see that the current active list $(v0, v1, v3, v2)$ doesn't match conditions for case 1, case 2, case 3 or case 4. So the current active list must match conditions for case 5, case 6 or case 7, and at least two commands will be created. To clarify which case it matches, we need to check the the other two vertices ($v8$ and $v6$) in quad $(v0, v2, v8, v6)$ which contains the focus edge $(v0, v2)$. Since the value of field *numOfVisitedEdge* for either vertex is 0, which means both of the vertices are free, we conclude that the current active list matches conditions for subcase 5.1. In this case, two add commands are created for vertex $v8$ and $v6$, the current active list is expanded, and the focus edge moves to edge $(v0, v6)$, as shown in Fig. 3.2(E). There are also some other operations should be performed, but here we will not give a detailed description for all the operations. For the operations performed in each case, you can find the details in section 3.3.1. The same rule applies for all the following discussion in this chapter.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now the focus edge $(v0, v6)$ is in quad $(v0, v6, v10, v4)$. From the same logic as in the previous step, we now find that the current active list matches conditions for subcase 5.1 again. So, we create two more add commands for vertex $v10$ and $v4$, expand the current active list, and move the focus edge to edge $(v0, v4)$, as shown in Fig. 3.2(F).

Now, the current active list matches conditions for case 4, since vertex $v0$, one of the two endpoints of the focus edge, is "almost full". By checking neighboring vertices of vertex $v0$ in the current active list, we find that there is only one **adjacent** vertex around vertex $v0$ that is "almost full", which means 3 **consecutive** vertices in the current active list are shared with quad $(v0, v4, v5, v1)$ which contains the focus edge $(v0, v4)$. This matches conditions for subcase 4.1. We also find that vertex $v5$ (which is the only vertex in the quad that is not shared with the current active list) is a free vertex since the value of field $numOfVisitedEdge$ for it is 0. From what we have examined, we finally conclude that the current active list matches conditions for subcase 4.1.1. So, we create one add command for vertex $v5$, and perform some other necessary operations, as shown in Fig. 3.2(G).

Now, the focus edge is $(v1, v5)$, and the quad containing the focus edge is $(v1, v5, v11, v7)$. We can see that the current active list matches conditions for case 5.1. So we create two more add commands for vertex $v11$ and $v7$, expand the current active list, and move the focus edge to edge $(v1, v7)$, as shown in Fig. 3.2(H).

Similar to the case shown in Fig. 3.2(F), we now find that the current active list matches conditions for subcase 4.1.1. So, we create one add command for vertex $v9$, and perform some other necessary operations, as shown in Fig. 3.2(I).

Now, the focus edge is $(v3, v9)$, and the current active list is $(v4, v5, v11, v7, v9, v3, v2, v8, v6, v10)$. By enumeration, we know that the current active list doesn't match conditions for case 1, case 2, case 3 or case 4. Then we check quad $(v3, v9, v11, v5)$, which contains the focus edge $(v3, v9)$. We find that vertex $v11$ is in the current

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

active list, which matches conditions for case 6; vertex $v5$ is also in the current active list, which matches conditions for subcase 6.2. This indicates that all 4 vertices of quad $(v3, v9, v11, v5)$ are in the current active list. We create two split commands for vertex $v11$ and $v5$. The offset of the first split command on vertex $v11$ is the distance between vertex $v11$ and vertex $v9$ (end vertex of the focus edge) along the current active list; the offset of the second split command on vertex $v5$ is the distance between vertex $v3$ (start vertex of the focus edge) and vertex $v5$ along current active list. The current active list is split into 3 active lists, 2 of which are pushed onto list stack, and we proceed with the third one, which is active list $(v11, v5)$, as shown in Fig. 3.2(J) and Fig. 3.2(K).

Since size of the current active list $(v11, v5)$ is less than 3, which matches conditions for case 1, we just remove the current active list, and proceed with the active list $(v3, v2, v8, v6, v10, v4, v5)$ popped from the list stack, as shown in Fig. 3.2(L).

Now, the focus edge is $(v3, v5)$, and the current active list is ($v3$, $v2$, $v8$, $v6$, $v10$, $v4$, $v5$). We can see that the current active list matches conditions for case 4, since at least one of the endpoints of the focus edge is "almost full". By checking the neighboring vertices of that "almost full" vertex in the active list, we find that there are two **consecutive** "almost full" vertices around it, which means 4 **consecutive** vertices in the active list are shared with quad $(v3, v5, v4, v2)$ which contains the focus edge $(v3, v5)$. This matches conditions for subcase 4.2. We perform some operations for this case, and move to the next step, as shown in Fig. 3.2(M).

Now, the focus edge is $(v2, v4)$, and the current active list is $(v2, v8, v6, v10, v4)$. Similar to the previous step, the current active list matches conditions for subcase 4.2. We perform the same operations as in the previous step, as shown in Fig. 3.2(N).

Now, the focus edge is $(v8, v10)$, and the current active list is $(v8, v6, v10)$. By enumeration, we know that the current active list doesn't match conditions for case 1, case 2, case 3 or case 4. So we need to check the other two vertices $v11$ and $v9$ in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

quad $(v8, v10, v11, v9)$ which contains the focus edge $(v8, v10)$. It's clear that vertex $v11$ is in one of the active lists in the list stack, which matches conditions for case 7; meanwhile, vertex $v9$ is also in one of the active lists in the list stack, which matches conditions for subcase 7.3; what's more, vertex $v11$ and vertex $v9$ are in the same active list $(v11, v7, v9)$, which matches conditions for subcase 7.3.2. For this case, we add all the edges outside vertex $v11$ and vertex $v9$ along active list $(v11, v7, v9)$ to the current active list $(v8, v6, v10)$, and create a new active list $(v9, v11)$, which contains edge $(v11, v9)$ and all the edges between vertex $v11$ and vertex $v9$ along the active list $(v11, v7, v9)$. Since the size of the new created active list $(v9, v11)$ is less than 3, we just delete this new active list. Three commands are created for this case, two merge commands for vertex $v11$ and vertex $v9$, and one split command for the new created active list. The index value of the second merge command is -1, which indicates that two merge operations are performed on the same active list. The details are shown in Fig. 3.2(O), Fig. 3.2(P) and Fig. 3.2(Q).

Now, the focus edge is $(v11, v10)$, and the current active list is ($v11$, $v7$, $v9$, $v8$, $v6$, $v10$). We can see that the current active list matches conditions for case 4, since at least one of the endpoints of the focus edge is "almost full". By checking neighboring vertices of that "almost full" vertex in the current active list, we can find that there are two **consecutive** "almost full" vertices around it, which means 4 **consecutive** vertices in the active list are shared with quad $(v11, v10, v6, v7)$ which contains the focus edge $(v11, v10)$. This matches conditions for subcase 4.2. We perform some operations for this case, and move to the next step, as shown in Fig. 3.2(R).

Now, the focus edge is $(v7, v6)$, and the current active list is $(v7, v9, v8, v6)$. By enumeration, we know that the current active list doesn't match conditions for case 1 or case 2. On the other hand, the current active list matches conditions for case 3 since there are only 4 vertices in the current active list, and all of them are "almost full". This indicates that the quad $(v7, v9, v8, v6)$ is the only quad outside the current active list $(v7, v9, v8, v6)$. For this case, we increase the value of field $numOfVisitedEdge$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by 1 for each vertex in the quad $(v7, 9, 8, v6)$, and then delete the current active list, as shown in Fig. 3.2(S).

Now, there are no active lists left, and we claim that the compression process is done.

During the compression process, each time we create an add command for a free vertex, we push this free vertex onto stack $newV$. The output for the compression process are two files. One contains the ordered vertices from stack $newV$, named "out_vertex.txt", the other one contains the sequence of commands, named "out_command.txt". The content of the output files are shown in Fig. 3.2(U). The command file can be encoded using entropy coding, which will be discussed later.

3.4.2 Decompression of a torus mesh

Now we have two input files for decompression, file "out_vertex.txt" contains a sequence of vertex coordinates, and file "out_command.txt" contains a sequence of commands, as shown in Fig. 3.3(B). Let's see how we reconstruct the torus using these two input files.

As shown in Fig. 3.3(C), first, we read file "out_vertex.txt" and put the vertices into a vertex stack; then, we read the command file "out_command.txt", and put the commands into a command stack.

The first 4 commands in command stack must be add commands. We pop the first 4 commands from command stack, and pop the first 4 vertices $(v0, v1, v3, v2)$ from vertex stack. Then we create a quad $(v0, v1, v3, v2)$ with these 4 vertices. We set the current active list to be $(v0, v1, v3, v2)$, and the focus edge to be $(v0, v2)$, which is the opposite half edge of the last edge of the quad, as shown in Fig. 3.3(D).

Next, by enumeration, we can see that the current active list $(v0, v1, v3, v2)$ doesn't

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

match conditions for case 1, case 2, case 3 or case 4. So the current active list must match conditions for case 5, case 6 or case 7. To clarify which case it matches, we pop two commands from command stack, since at least two commands are created by case 5, case 6 or case 7 during compression. The next two commands are add commands, which matches conditions for subcase 5.1. So we pop two vertices ($v8$ and $v6$) from vertex stack, create a new quad with the focus edge and the two new vertices ($v8$ and $v6$). Also we need to expand the active list, move the focus edge to edge $(v0, v6)$, and perform some other necessary operations for case 5.1, as shown in Fig. 3.3(E).

Now the focus edge is $(v0, v6)$, and the current active list is $(v0, v1, v3, v2, v8, v6)$. From the same logic as the previous step, we can find that now the current active list matches conditions for case 5.1 again (after popping two commands from command stack). So we pop another two vertices ($v10$ and $v4$) from vertex stack, create a new quad $(v0, v6, v10, v4)$, and perform some other necessary operations, as shown in Fig. 3.3(F).

Now, the focus edge is $(v0, v4)$, and the current active list is ($v0$, $v1$, $v3$, $v2$, $v8$, $v6$, $v10$, $v4$). You can see that the current active list matches conditions for case 4, since vertex $v0$, one of the two endpoints of the focus edge, is "almost full". By checking neighbor vertices of vertex $v0$ in the current active list, we can find that there is only one **consecutive** vertex around vertex $v0$ is "almost full", which means 3 **consecutive** vertices $(v4, v0, v1)$ in the current active list are shared with the next created quad which contains the focus edge $(v0, v4)$. This matches conditions for subcase sub4.1. To clarify the subcase further more, we pop one command from command stack. The command popped out is an add command, which matches conditions for subcase 4.1.1. So, we pop one vertex ($v5$) from vertex stack, create a new quad $(v0, v4, v5, v1)$, and perform some other necessary operations, as shown in Fig. 3.3(G).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now, the focus edge is $(v1, v5)$. Again, by enumeration, we can find that the current active list $(v4, v5, v1, v3, v2, v8, v6, v10)$ doesn't match conditions for case 1, case 2, case 3 or case 4. So current active list must match conditions for case 5, case 6 or case 7. To clarify which case it matches, we pop two commands out from command stack. The next two commands are add commands, which matches conditions for subcase 5.1. So we pop two vertices ($v11$ and $v7$) from vertex stack, create a new quad with the focus edge and the two new vertices ($v11$ and $v7$), as shown in Fig. 3.3(H).

Now, the focus edge is $(v1, v7)$, and the current active list is ($v4$, $v5$, $v11$, $v7$, $v1$, $v3$, $v2$, $v8$, $v6$, $v10$). Similar to the case shown in Fig. 3.3(F), we can find that now the current active list matches conditions for subcase 4.1 since 3 **consecutive** vertices $(v7, v1, v3)$ in the current active list are shared with the next created quad which contains the focus edge $(v1, v7)$. We pop one command from command stack. The command popped out is an add command, which matches conditions for subcase 4.1.1. So, we pop one vertex ($v9$) from vertex stack, create a new quad $(v3, v1, v7, v9)$, and perform some other necessary operations, as shown in Fig. 3.3(I).

Now, the focus edge is $(v3, v9)$, and the current active list is ($v4$, $v5$, $v11$, $v7$, $v9$, $v3$, $v2$, $v8$, $v6$, $v10$). Again, by enumeration, we know that the current active list doesn't match conditions for case 1, case 2, case 3 or case 4. So the current active list must match conditions for case 5, case 6 or case 7. To clarify which case it matches, we pop two commands from command stack. The next two commands are split commands, which matches conditions for subcase 6.2. This indicates that all vertices of next created quad are from current active list. By checking the $offset$ values of the splits commands, we can find that the other two vertices of the new created quad are $v11$ and $v5$. So we create a new quad $(v3, v9, v11, v5)$, split the current active list into three active lists, push two of them into list stack, and proceed with the third one, which is active list $(v11, v5)$, as shown in Fig. 3.3(J) and Fig. 3.3(K).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now the current active list is $(v11, v5)$. Since size of the current active list $(v11, v5)$ is less than 3, which matches conditions for case 1, we just remove the current active list, and proceed with the active list $(v3, v2, v8, v6, v10, v4, v5)$ popped from list stack. To make the following description easier for understanding, we draw the active list $(v3, v2, v8, v6, v10, v4, v5)$ in an alternative way, as shown in Fig. 3.3(L).

Now, the focus edge is $(v3, v5)$, and the current active list is $(v3,\ v2,\ v8,\ v6,\ v10,\ v4,\ v5)$. We can see that the current active list matches conditions for case 4, since at least one of the endpoints of the focus edge is "almost full". By checking neighbor vertices of that "almost full" vertex in the active list, we can find that there are three **consecutive** "almost full" vertices around it, which means 4 **consecutive** vertices $(v4, v5, v3, v2)$ in the current active list are shared with the new created quad which contains the focus edge $(v3, v5)$. This matches conditions for subcase 4.2. So, we just create a new quad $(v3, v5, v4, v2)$, and perform some other necessary operations, as shown in Fig. 3.3(M).

Now, the focus edge is $(v2, v4)$, and the current active list is $(v2,\ v8,\ v6,\ v10,\ v4)$. Similar to the previous step, the current active list matches conditions for case 4.2, and 4 **consecutive** vertices $(v10, v4, v2, v8)$ in the current active list are shared with the new created quad which contains the focus edge $(v2, v4)$. So, we just create a new quad $(v2, v4, v10, v8)$, and perform some other necessary operations, as shown in Fig. 3.3(N).

Now, the focus edge is $(v8, v10)$, and the current active list is $(v8, v6, v10)$. By enumeration, we know that the current active list doesn't match conditions for case 1, case 2, case 3 or case 4. So current active list must match conditions for case 5, case 6 or case 7. To clarify which case it matches, we pop two commands from command stack. The next two commands are merge commands, which matches conditions for subcase 7.3. What's more, the *index* value for the second merge command is -1, which matches conditions for subcase 7.3.2. In this situation, we need to create a new quad

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

first. Two vertices of the new quad are the endpoints of the focus edge $(v8, v10)$. The other two vertices come from the active list $(v11, v7, v9)$ which is identified by the *index* value of the first popped merge command. We can find the other two vertices from the active list $(v11, v7, v9)$ by checking the $offset$ value of the two merge commands. After the other two vertices $v11$ and $v9$ are found, we create a new quad $(v8, v10, v11, v9)$. Then we add all edges outside vertex $v11$ and vertex $v9$ along active list $(v11, v7, v9)$ to the current active list $(v8, v6, v10)$, and create another new active list $(v9, v11)$, which contains edge $(v11, v9)$ and all the edges between vertex $v11$ and vertex $v9$ along active list $(v11, v7, v9)$. Creating a new active list means splitting current active list into two, so we need to pop a new command (which must be a split command) from command stack and then split. Details are shown in Fig. 3.2(O), Fig. 3.2(P) and Fig. 3.2(Q).

Now, the focus edge is $(v11, v10)$, and the current active list is ($v11$, $v7$, $v9$, $v8$, $v6$, $v10$). We can see that the current active list matches conditions for case 4, since at least one of the endpoints of the focus edge is "almost full". By checking neighbor vertices of that "almost full" vertex in the active list, we can find that there are three **consecutive** "almost full" vertices around it, which means four **consecutive** vertices $(v11, v7, v6, v10)$ in the current active list are shared with the new created quad which contains the focus edge $(v11, v10)$. This matches conditions for subcase 4.2. So, we just create a new quad $(v11, v10, v6, v7)$, and perform some other necessary operations, as shown in Fig. 3.3(R).

Now, the focus edge is $(v7, v6)$, and the current active list is $(v7, v9, v8, v6)$. By enumeration, we know that the current active list doesn't match conditions for case 1 or case 2. On the other hand, current active list matches conditions for case 3 since there are only 4 vertices in the current active list, and all of them are "almost full". This indicates that there should be only one un-created quad outside current active list $(v7, v9, v8, v6)$. For this case, we just create a new quad $(v7, 9, 8, v6)$, increase the value of field $numOfVisitedEdge$ by 1 for each vertex of the quad, and then delete

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the current active list, as shown in Fig. 3.3(S).

Now, there are no any active lists left, and we claim that the decompression process is done.

3.5 Handling boundaries

Our implementation works for quadmesh without boundary. For mesh with boundary, Touma-Gotsman suggested to add one dummy vertex and encode this dummy vertex separately. The same idea won't work for quadmeshes.

For a quadmesh with boundary, consider the polygon formed the by boundary edges. Either you have to split this polygon into quads, which will introduce more vertices/edges, or you have to introduce triangles. We do think the latter is better.

The easiest way might be:

1. Add one dummy vertex 1st, triangulate the polygon;
2. Traverse all triangles around the dummy vertex first.

The result mesh contains quads and some dummy triangles. During the compression process, we conquer those dummy triangles first, and then conquer all rest quads.

3.6 Entropy coding of the command sequence

The three commands appearing in the connectivity code are "add <degree>", "split <offset>" and "merge <index><offset>". Generally, a typical code contains many "add" commands, a few "split" commands, and almost no "merge" commands.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In typical quadmeshes, the average vertex degree is 4, and there is a spread of degrees around this value. Using entropy coding, like Huffman to encode the connectivity code could compress the connectivity further more. Table 3.4 is the Huffman code for a sample mesh "Torus" which is presented in Chapter 5.

add 4	1
split 1	001
split 2	000
split 6	0110
merge -1 2	0111
merge 0 0	010

Table 3.4: Huffman coding scheme for sample mesh "Torus"

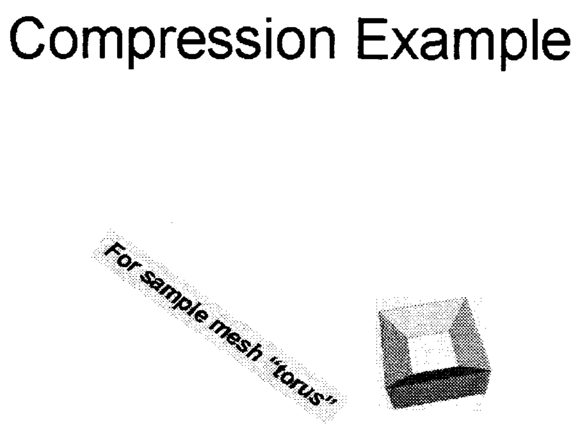
3.7 Time complexity analysis

The most time-consuming operation in the connectivity compression procedure is searching for a given vertex in some active list on the list stack. This is needed only for "merge" operations, which are extremely rare. Apart from that, both the space and time complexity of the compression and decompression algorithms are linear in the number of mesh quads/vertices.

3.8 Summary

In this chapter we discussed how Touma-Gotsman's compression algorithm for triangle meshes is successfully extended for quadrilateral meshes. Time complexity analysis shows that both the compression and decompression algorithms are linear in the mesh size.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Compression Example

For sample mesh "torus"

Figure 3.2: Compression example (A to U)

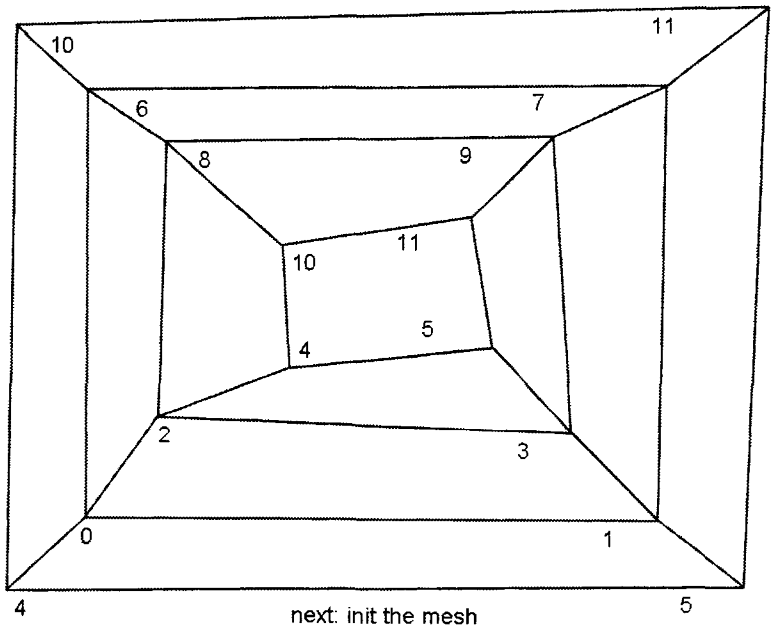
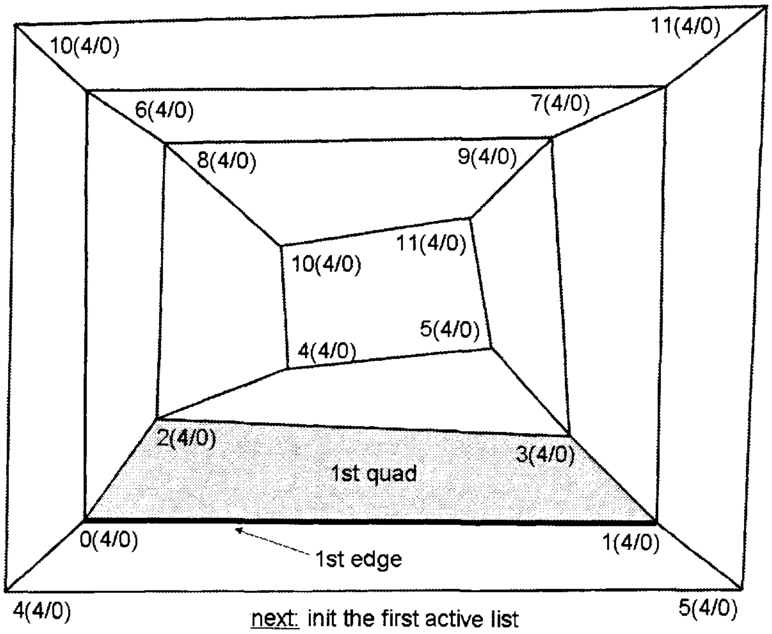


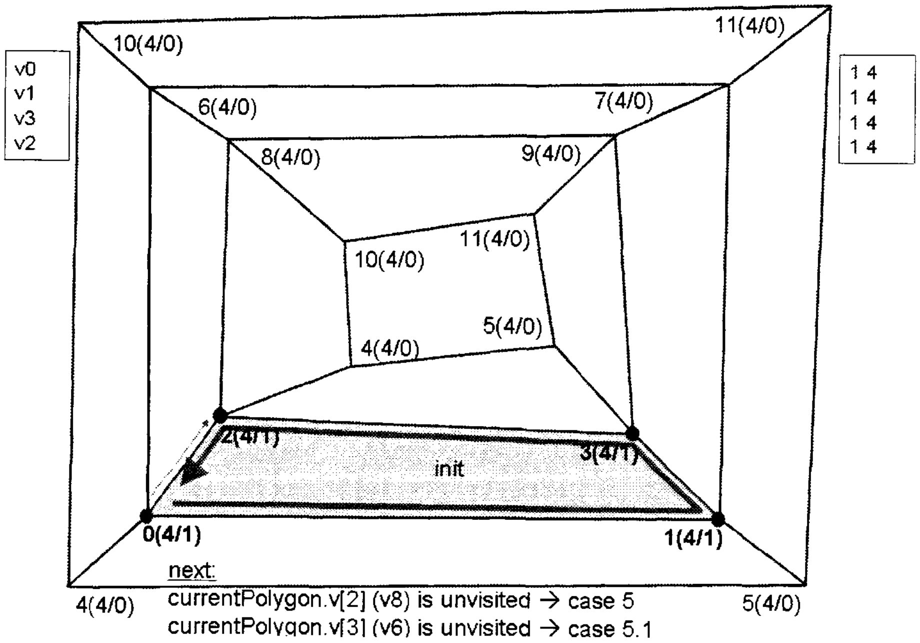
Figure 3.2: Compression example (B)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



next: init the first active list

Figure 3.2: Compression example (C)



next:
currentPolygon.v[2] (v8) is unvisited → case 5
currentPolygon.v[3] (v6) is unvisited → case 5.1

Figure 3.2: Compression example (D)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

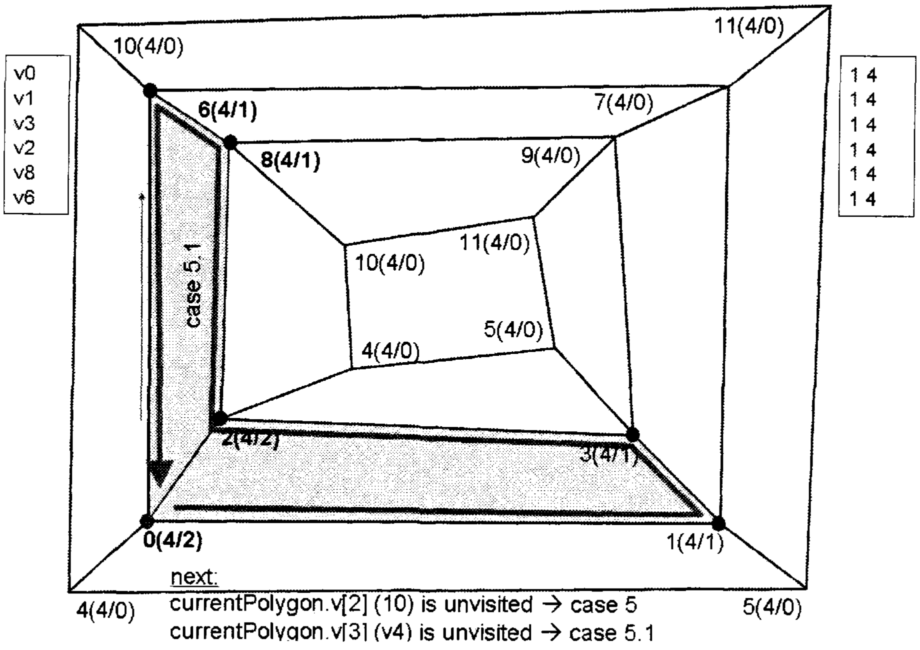


Figure 3.2: Compression example (E)

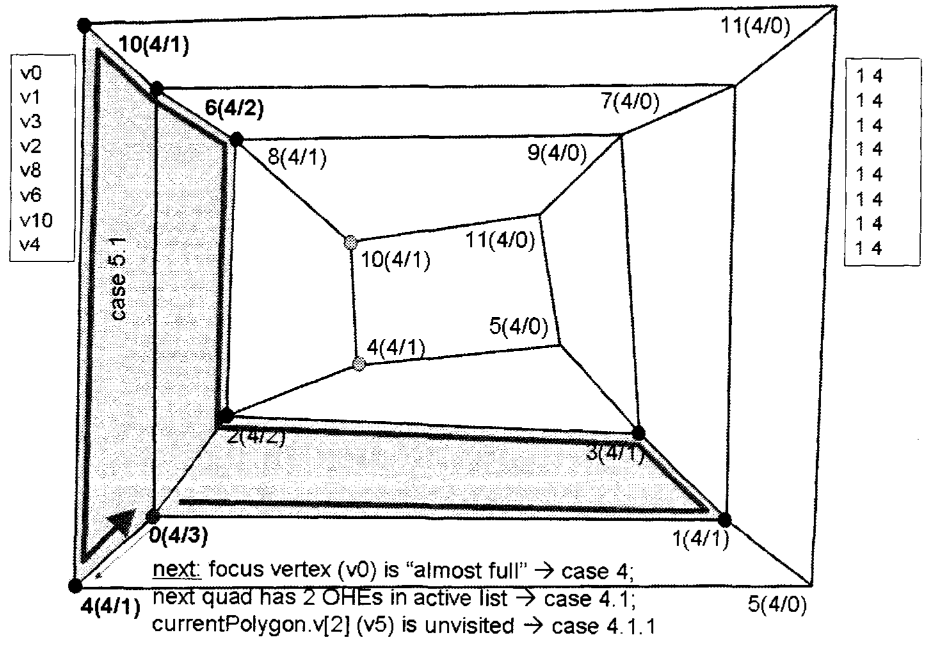
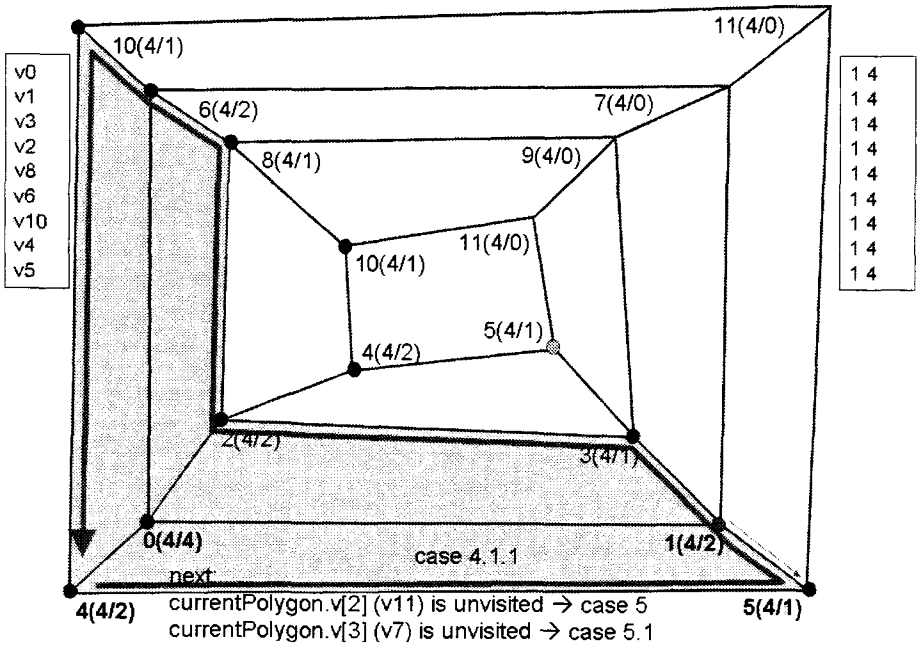


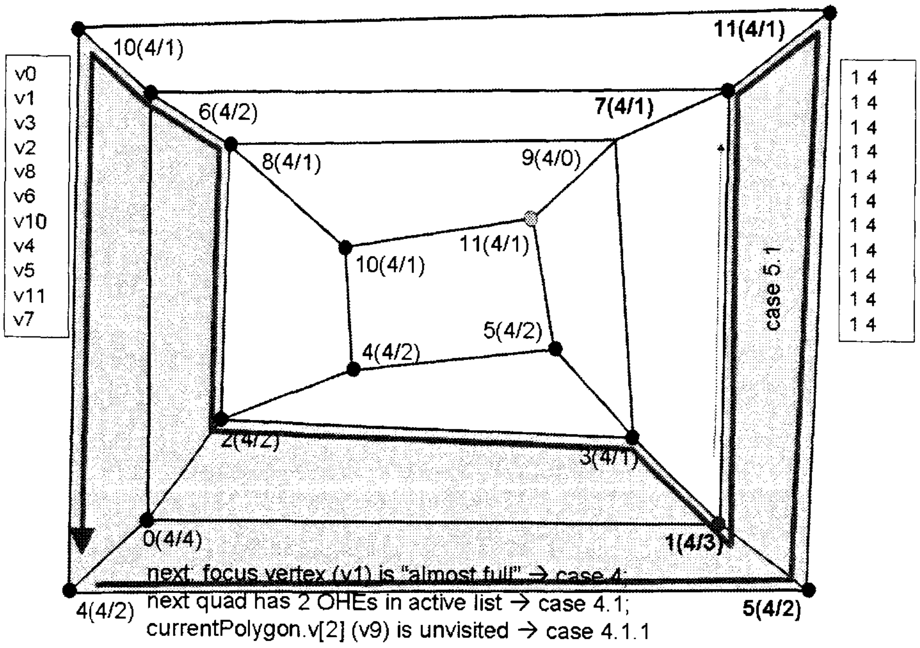
Figure 3.2: Compression example (F)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



currentPolygon.v[2] (v11) is unvisited → case 5
currentPolygon.v[3] (v7) is unvisited → case 5.1

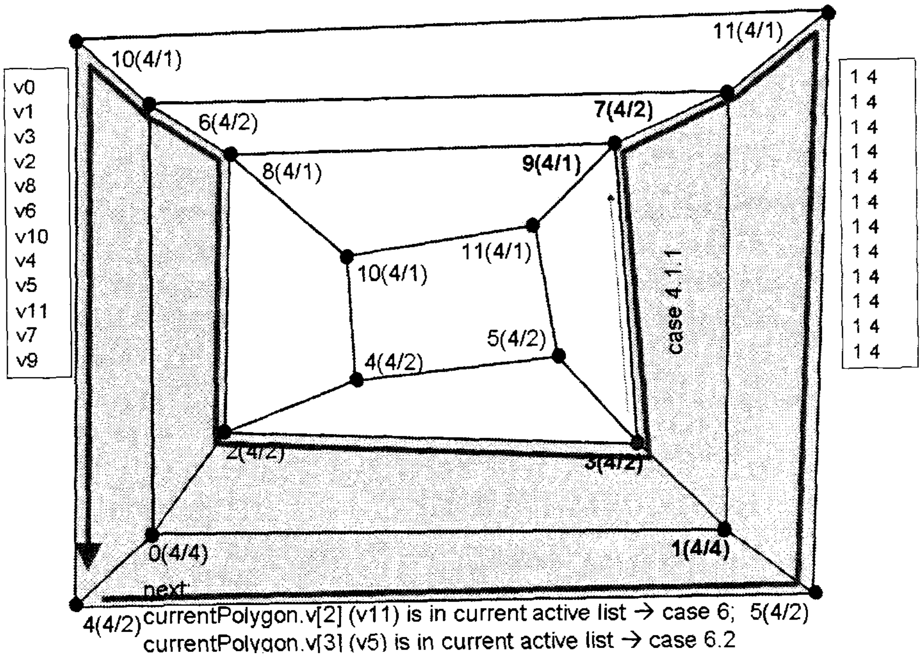
Figure 3.2: Compression example (G)



next: focus vertex (v1) is "almost full" → case 4;
next quad has 2 OHEs in active list → case 4.1;
currentPolygon.v[2] (v9) is unvisited → case 4.1.1

Figure 3.2: Compression example (H)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



currentPolygon.v[2] (v11) is in current active list → case 6;
currentPolygon.v[3] (v5) is in current active list → case 6.2

Figure 3.2: Compression example (I)

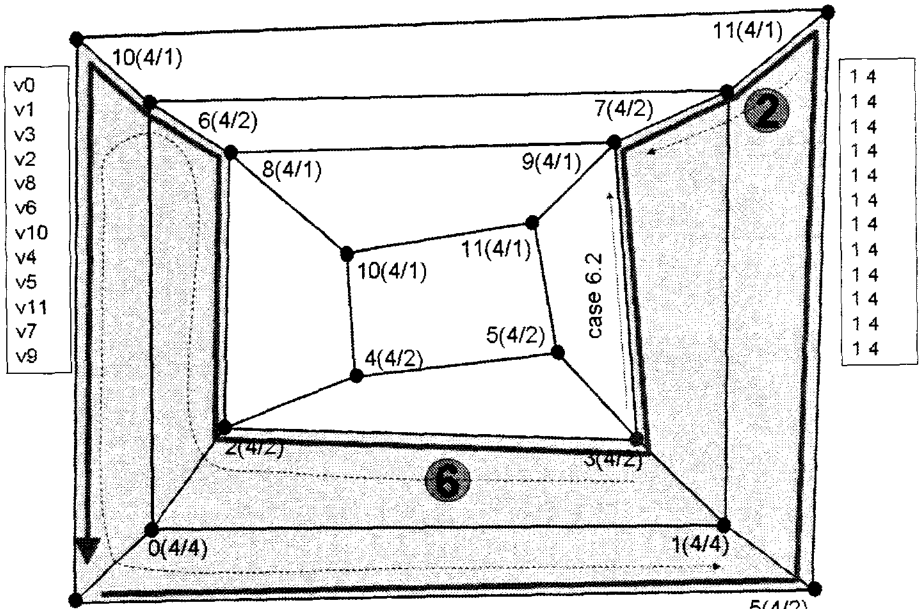
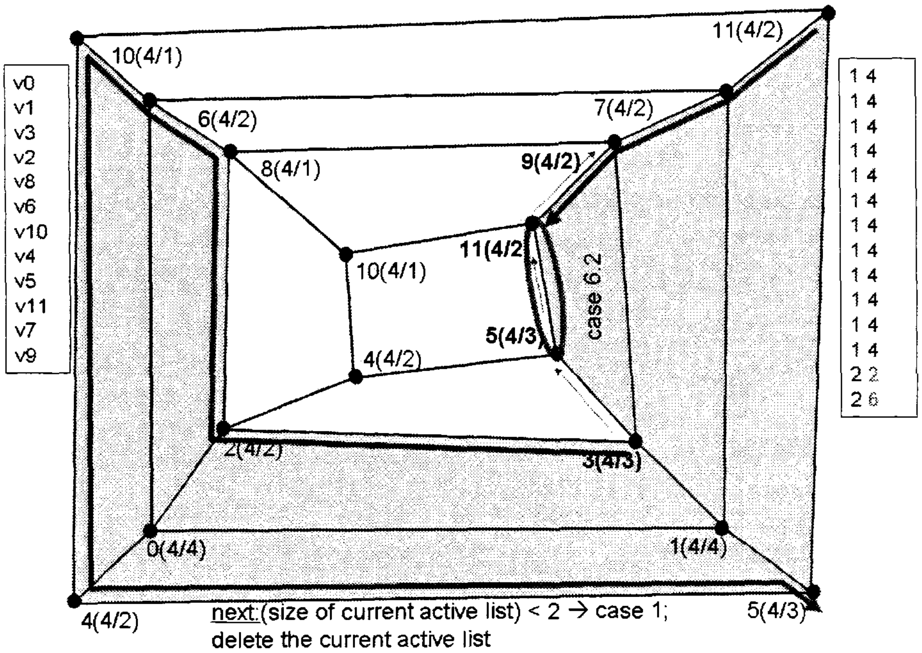


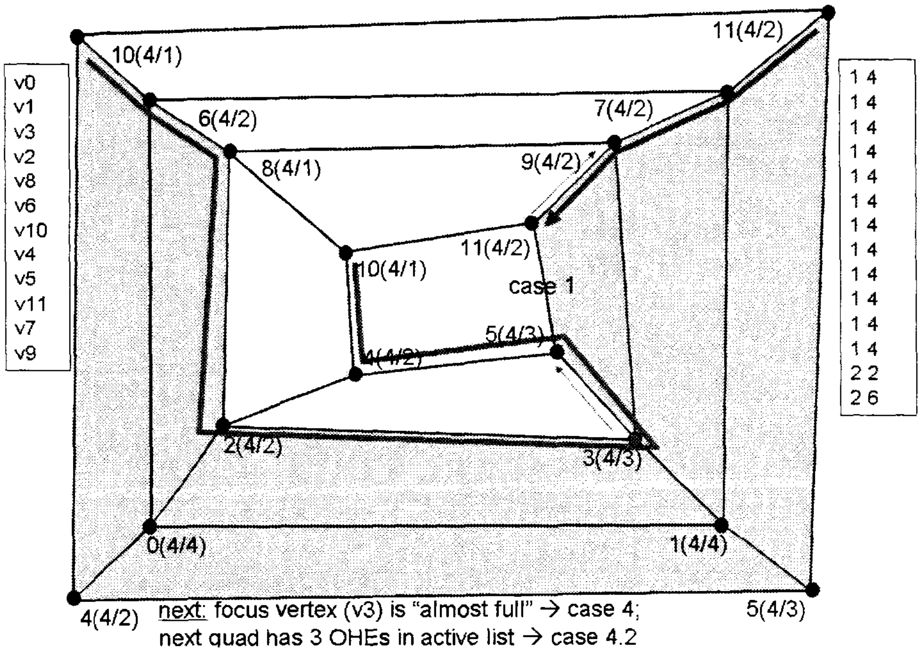
Figure 3.2: Compression example (J)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



next:(size of current active list) < 2 → case 1;
delete the current active list

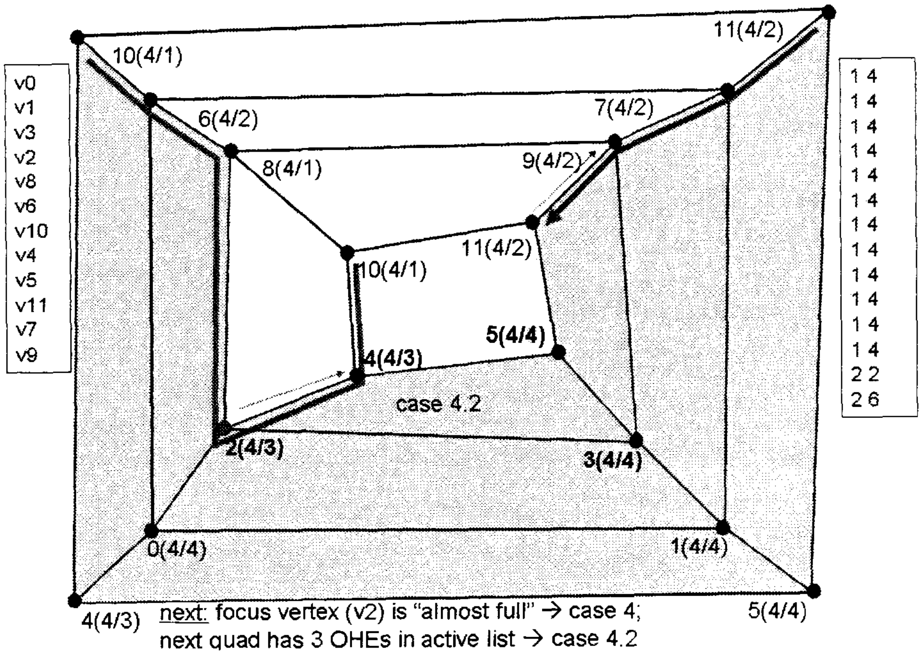
Figure 3.2: Compression example (K)



next: focus vertex (v3) is "almost full" → case 4;
next quad has 3 OHEs in active list → case 4.2

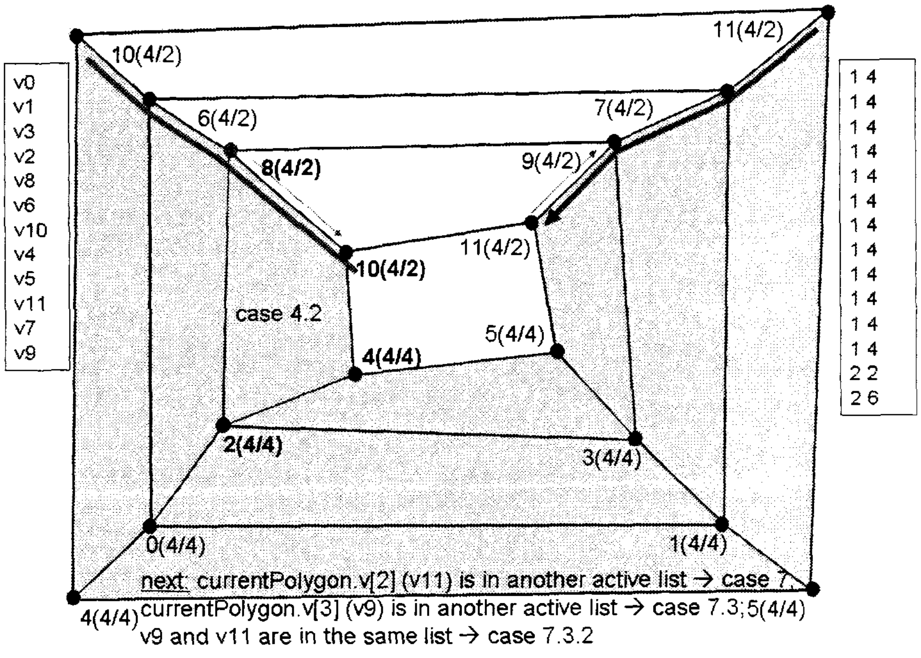
Figure 3.2: Compression example (L)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



next: focus vertex (v2) is "almost full" → case 4;
next quad has 3 OHEs in active list → case 4.2

Figure 3.2: Compression example (M)



next: currentPolygon.v[2] (v11) is in another active list → case 7,
currentPolygon.v[3] (v9) is in another active list → case 7.3;
v9 and v11 are in the same list → case 7.3.2

Figure 3.2: Compression example (N)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

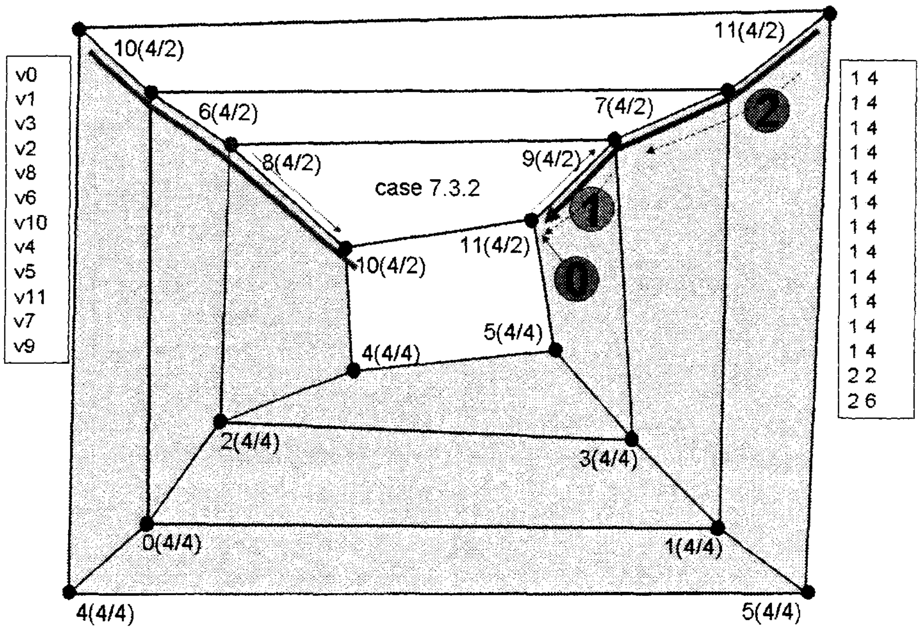
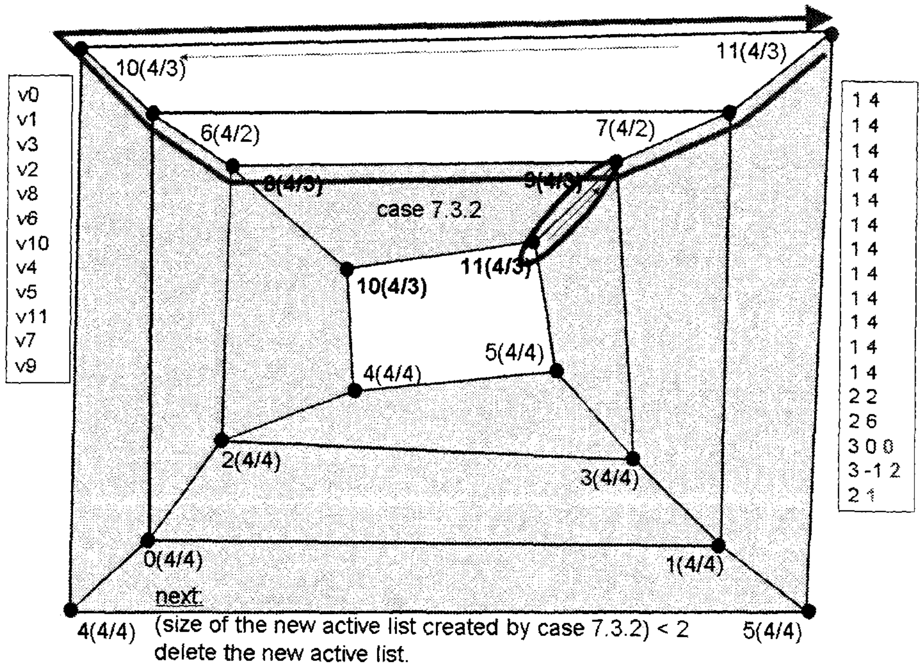


Figure 3.2: Compression example (O)



next:
(size of the new active list created by case 7.3.2) < 2
delete the new active list.

Figure 3.2: Compression example (P)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

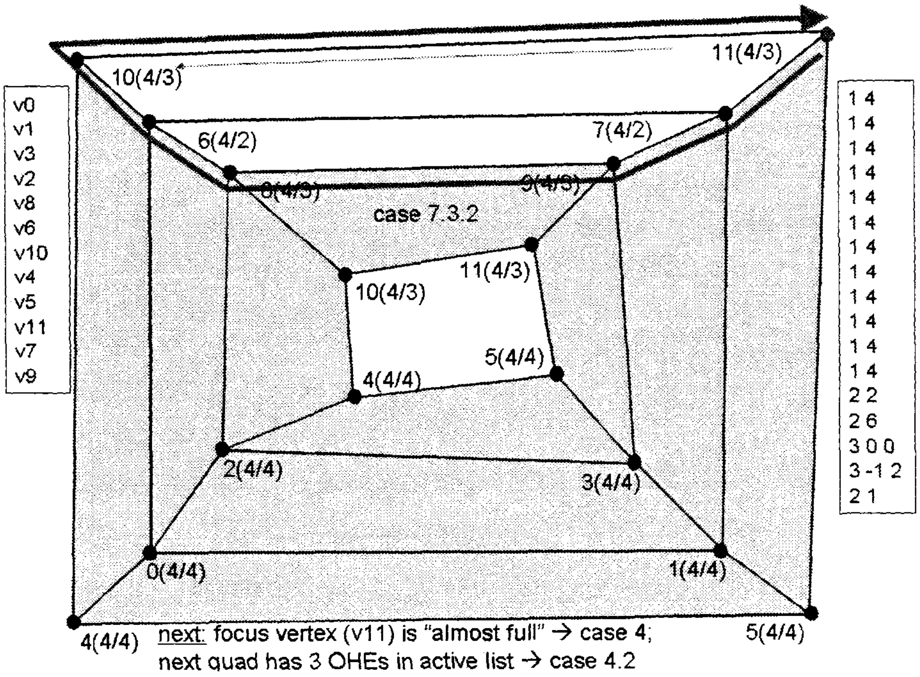


Figure 3.2: Compression example (Q)

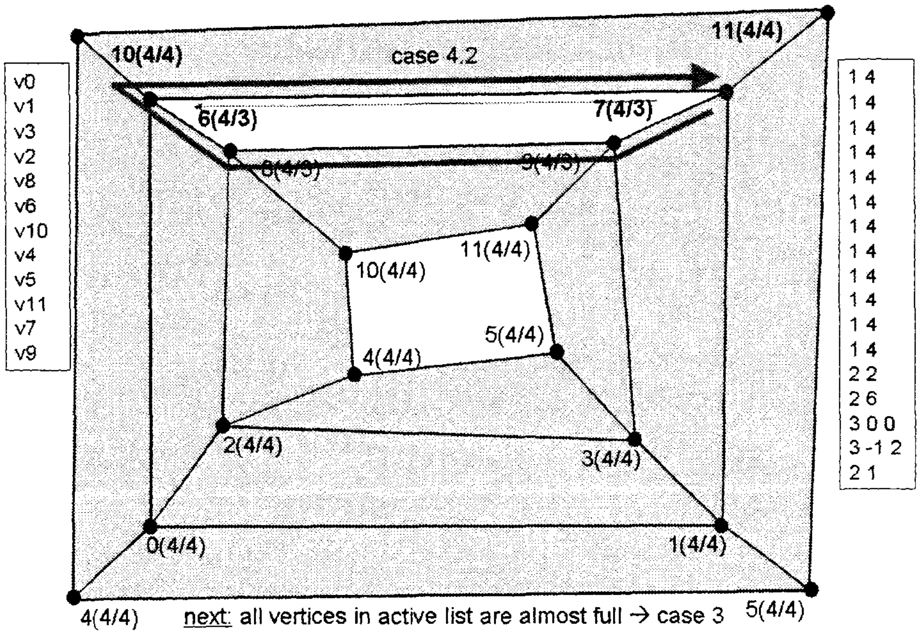


Figure 3.2: Compression example (R)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

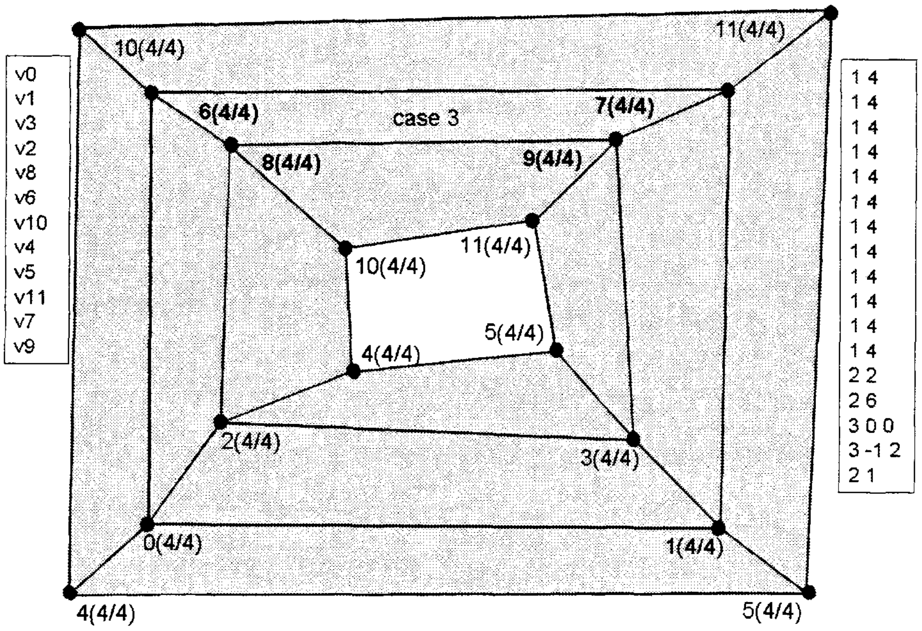


Figure 3.2: Compression example (S)

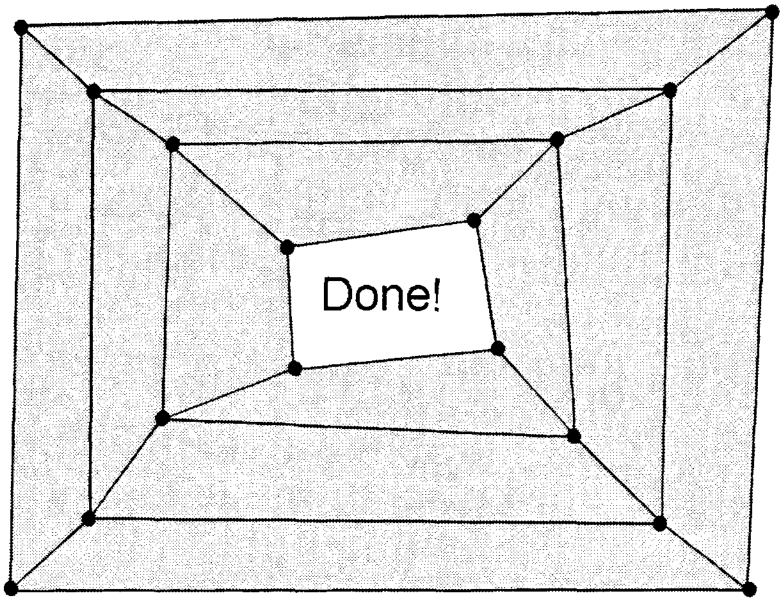


Figure 3.2: Compression example (T)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

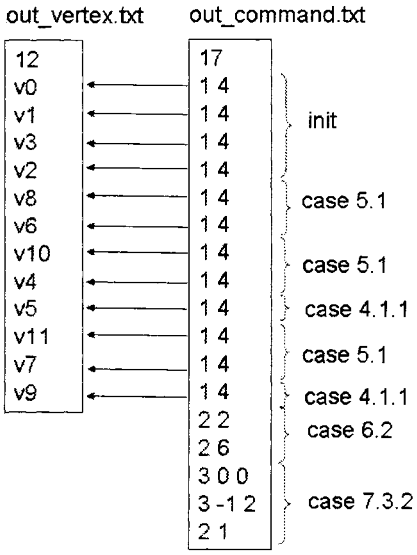


Figure 3.2: Compression example (U)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Decompression Example

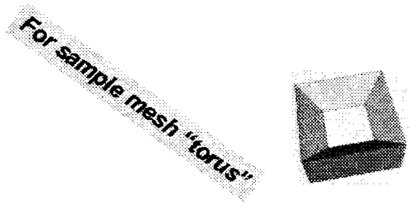
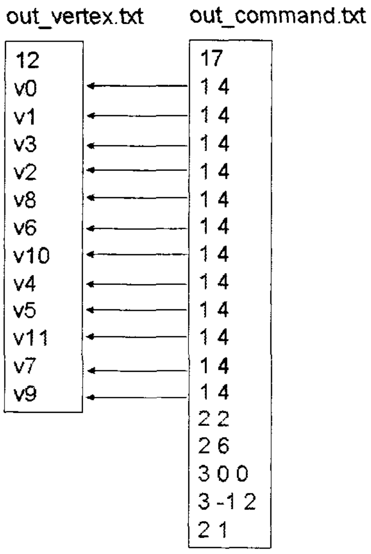


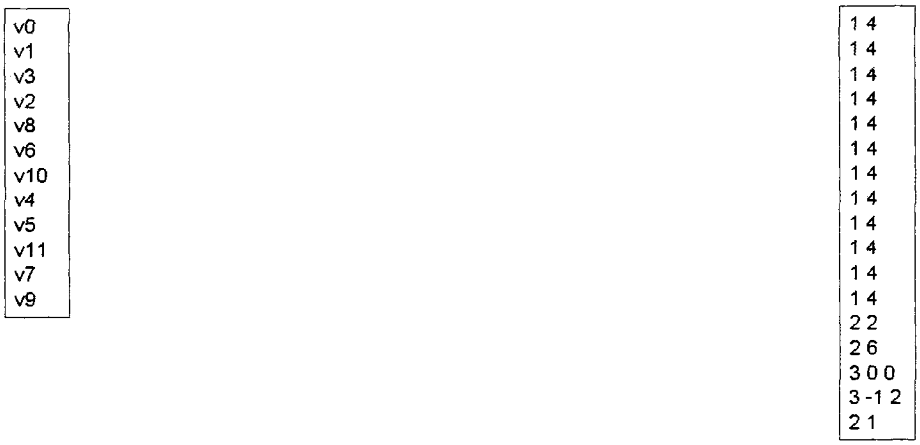
Figure 3.3: Decompression example (A to T)



out_vertex.txt	out_command.txt
12	17
v0	1 4
v1	1 4
v3	1 4
v2	1 4
v8	1 4
v6	1 4
v10	1 4
v4	1 4
v5	1 4
v11	1 4
v7	1 4
v9	1 4
	2 2
	2 6
	3 0 0
	3 -1 2
	2 1

Figure 3.3: Decompression example (B)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

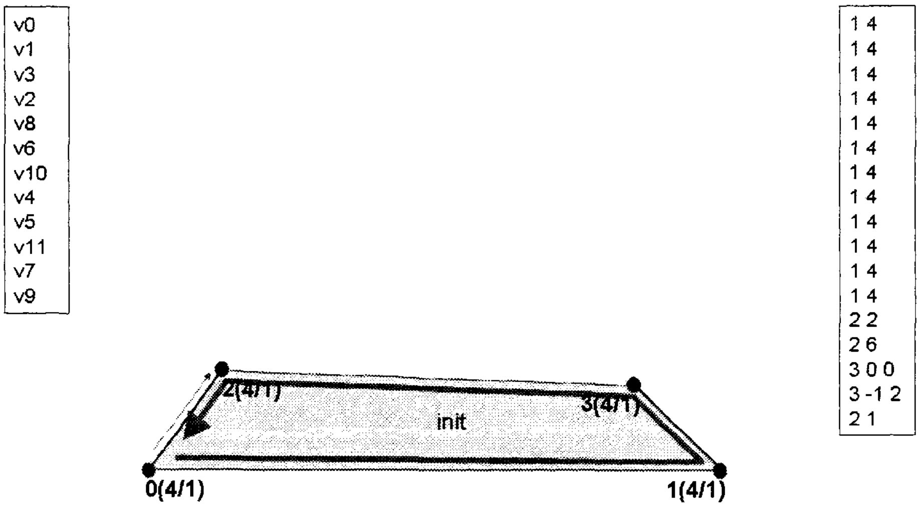


v0
v1
v3
v2
v8
v6
v10
v4
v5
v11
v7
v9

1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
2 2
2 6
3 0 0
3 -1 2
2 1

next: init (read first 4 commands and first 4 vertices)

Figure 3.3: Decompression example (C)



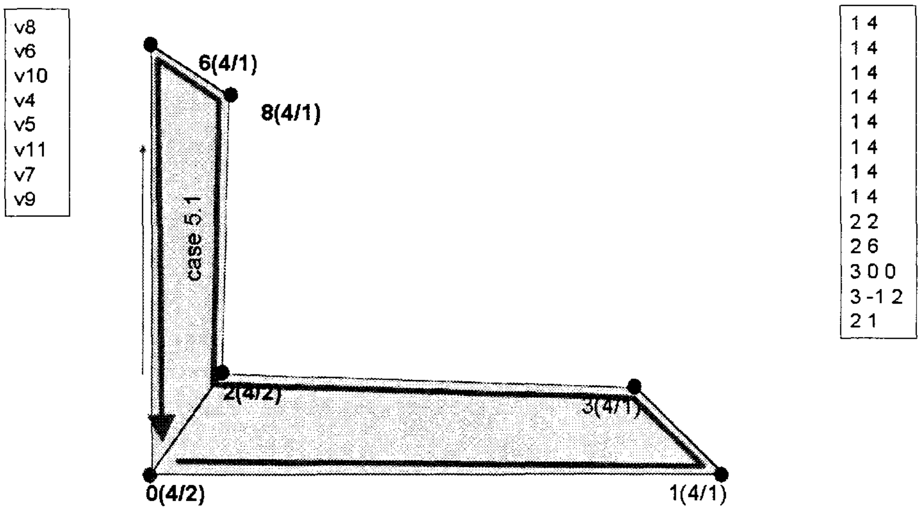
v0
v1
v3
v2
v8
v6
v10
v4
v5
v11
v7
v9

1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
1 4
2 2
2 6
3 0 0
3 -1 2
2 1

next:
read next command → case 5 ("add" command, v8)
read next command → case 5.1 ("add" command, v6)

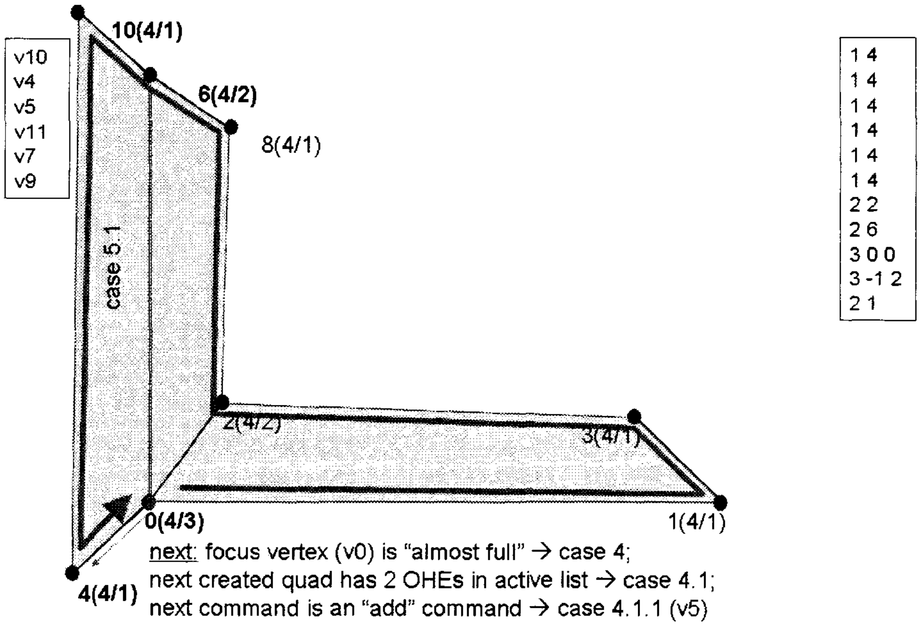
Figure 3.3: Decompression example (D)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



next:
read next command → case 5 ("add" command, v10)
read next command → case 5.1 ("add" command, v4)

Figure 3.3: Decompression example (E)



next: focus vertex (v0) is "almost full" → case 4;
next created quad has 2 OHEs in active list → case 4.1;
next command is an "add" command → case 4.1.1 (v5)

Figure 3.3: Decompression example (F)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

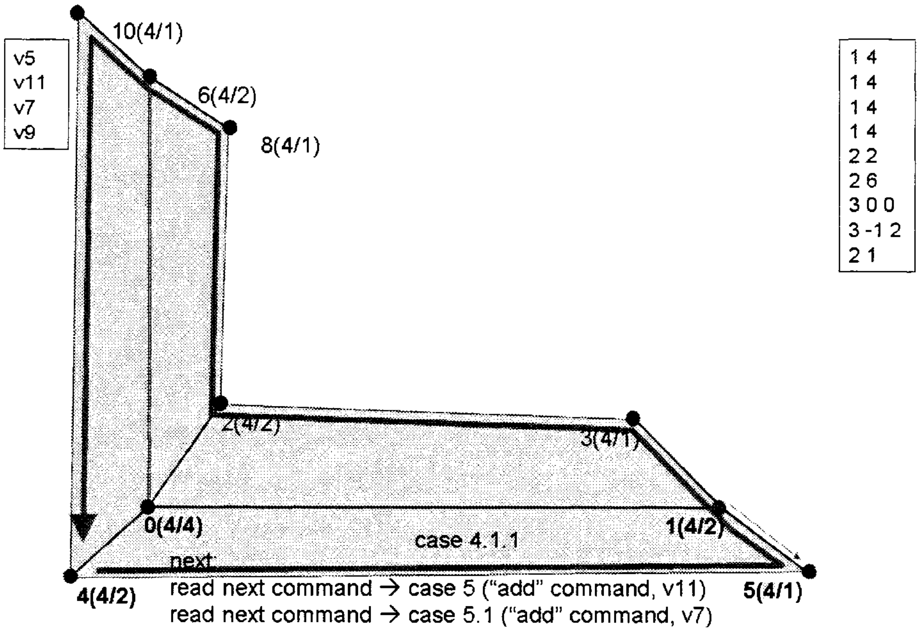


Figure 3.3: Decompression example (G)

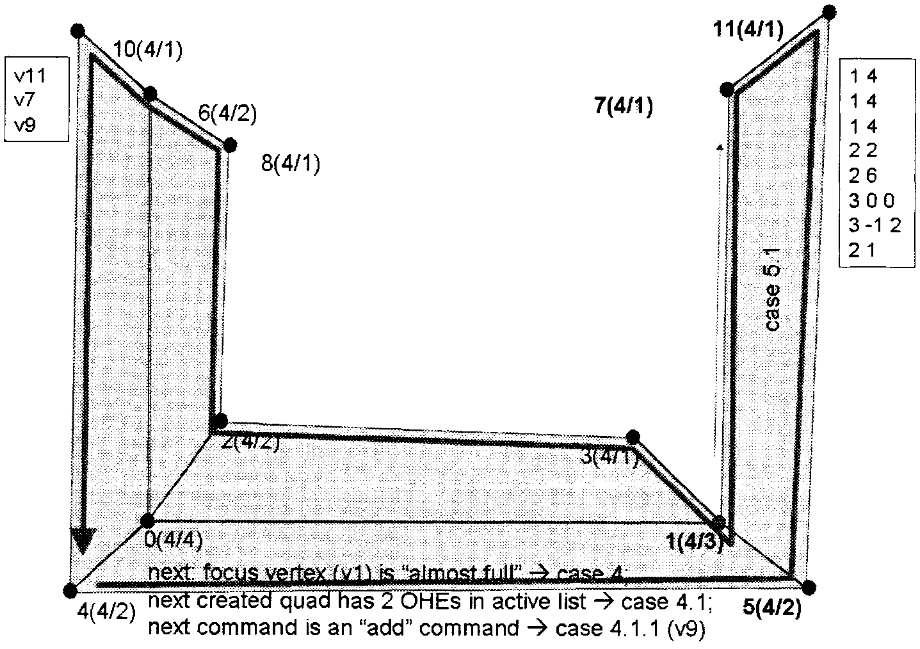
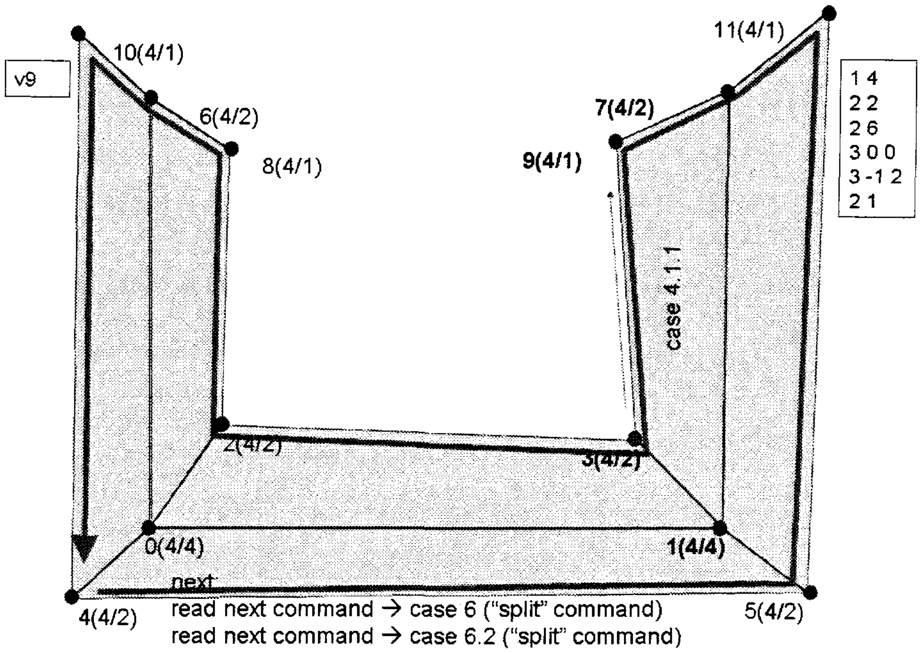

v11
v7
v9
10(4/1)
6(4/2)
8(4/1)
7(4/1)
11(4/1)
1 4
1 4
1 4
2 2
2 6
3 0 0
3 -1 2
2 1
case 5.1
2(4/2)
3(4/1)
0(4/4)
1(4/3)
4(4/2)
5(4/2)
next: focus vertex (v1) is "almost full" → case 4;
next created quad has 2 OHEs in active list → case 4.1;
next command is an "add" command → case 4.1.1 (v9)

Figure 3.3: Decompression example (H)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



read next command → case 6 ("split" command)
read next command → case 6.2 ("split" command)

Figure 3.3: Decompression example (I)

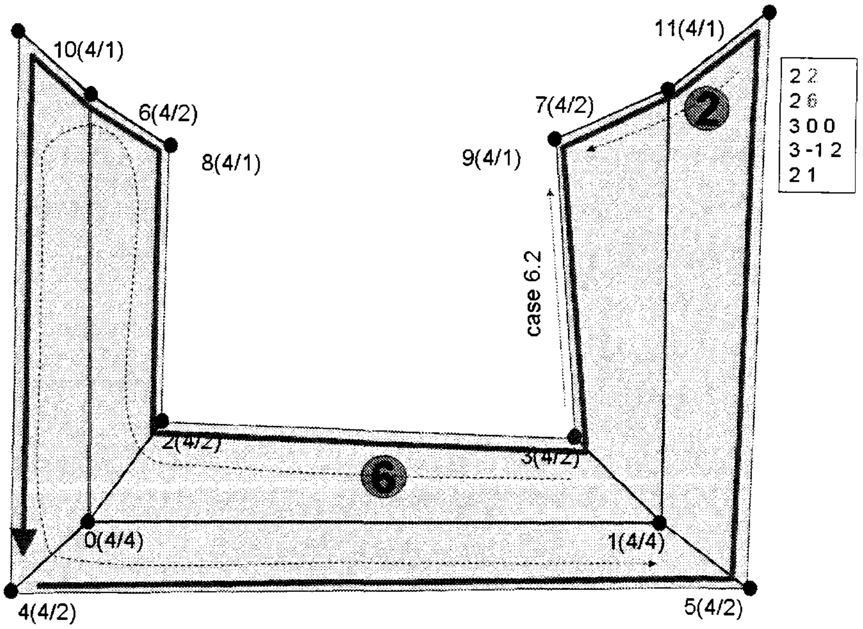
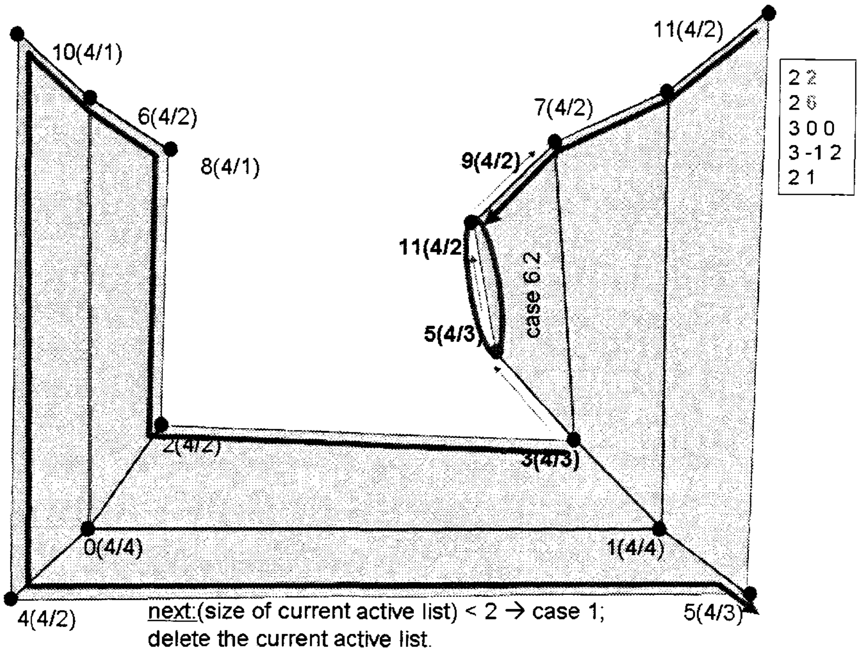


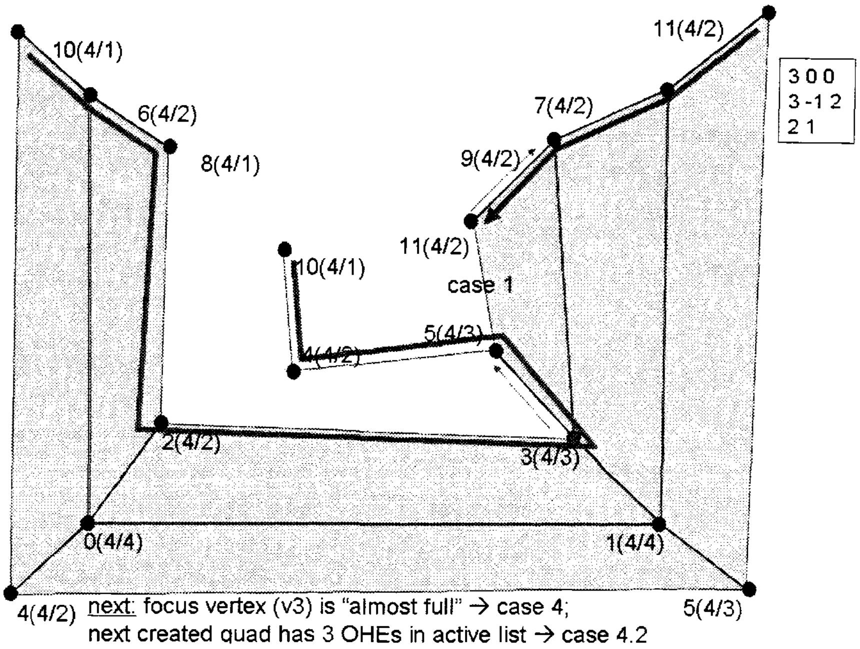
Figure 3.3: Decompression example (J)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



next:(size of current active list) < 2 → case 1; delete the current active list.

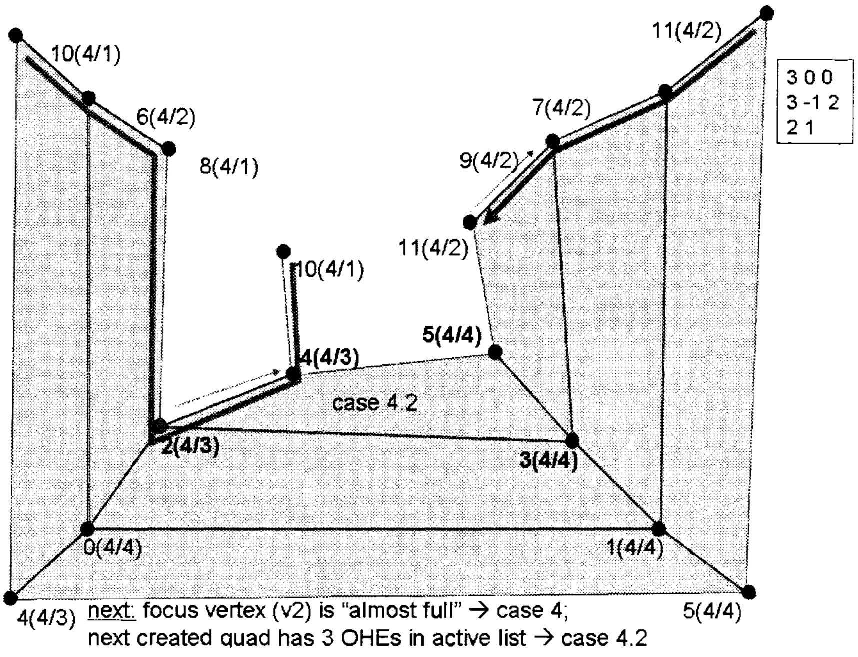
Figure 3.3: Decompression example (K)



next: focus vertex (v3) is "almost full" → case 4; next created quad has 3 OHEs in active list → case 4.2

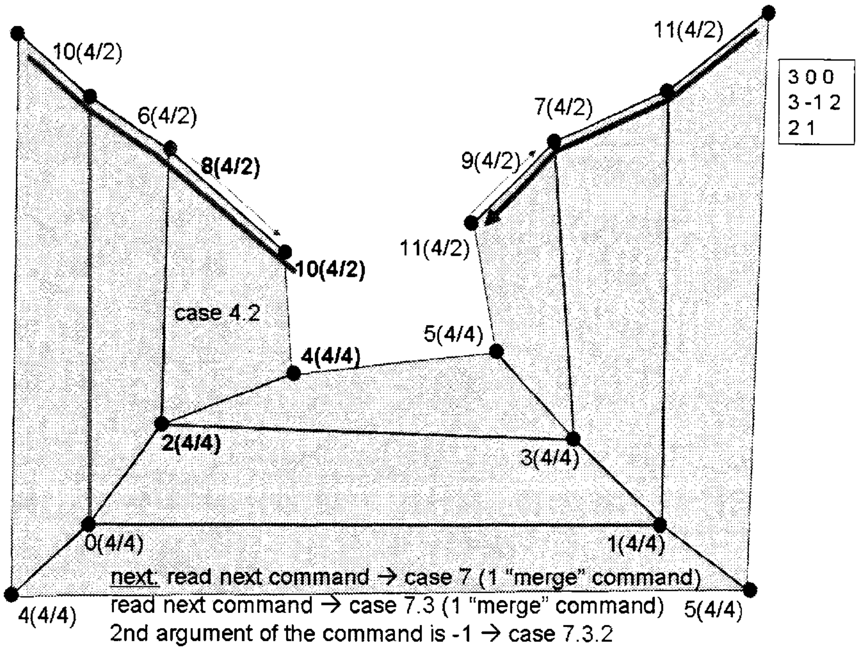
Figure 3.3: Decompression example (L)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



next: focus vertex (v2) is "almost full" → case 4;
next created quad has 3 OHEs in active list → case 4.2

Figure 3.3: Decompression example (M)



next: read next command → case 7 (1 "merge" command)
read next command → case 7.3 (1 "merge" command)
2nd argument of the command is -1 → case 7.3.2

Figure 3.3: Decompression example (N)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

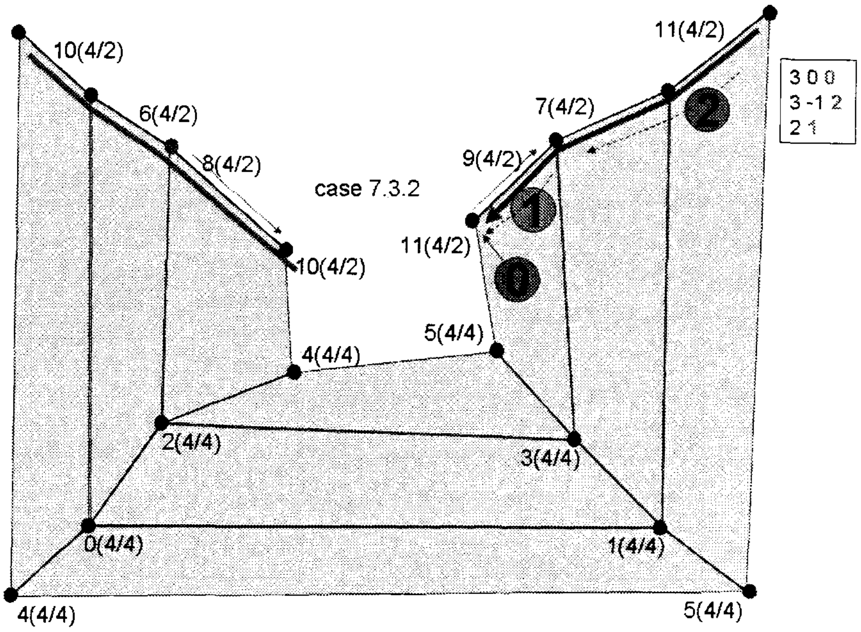
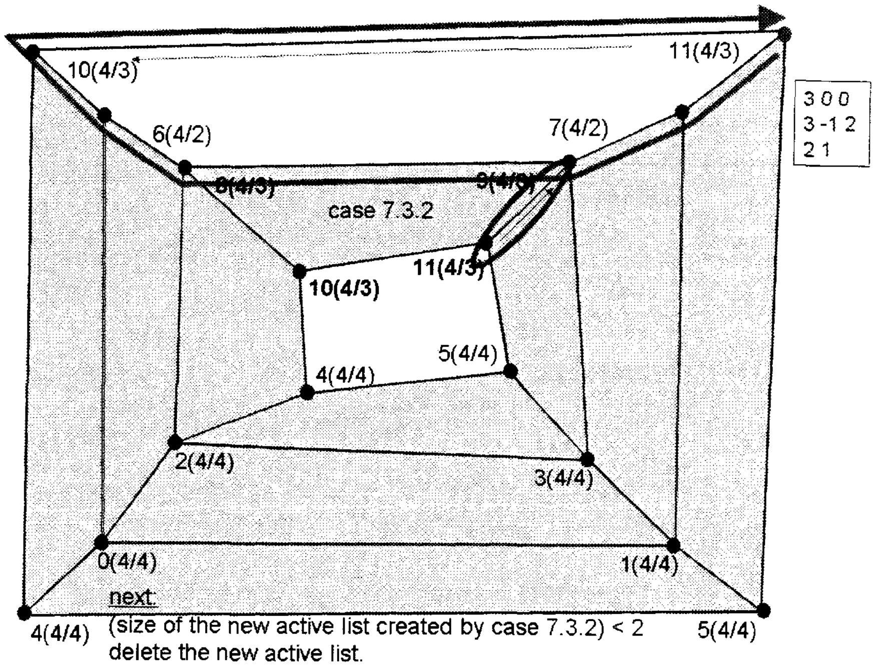


Figure 3.3: Decompression example (O)



next:
(size of the new active list created by case 7.3.2) < 2
delete the new active list.

Figure 3.3: Decompression example (P)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

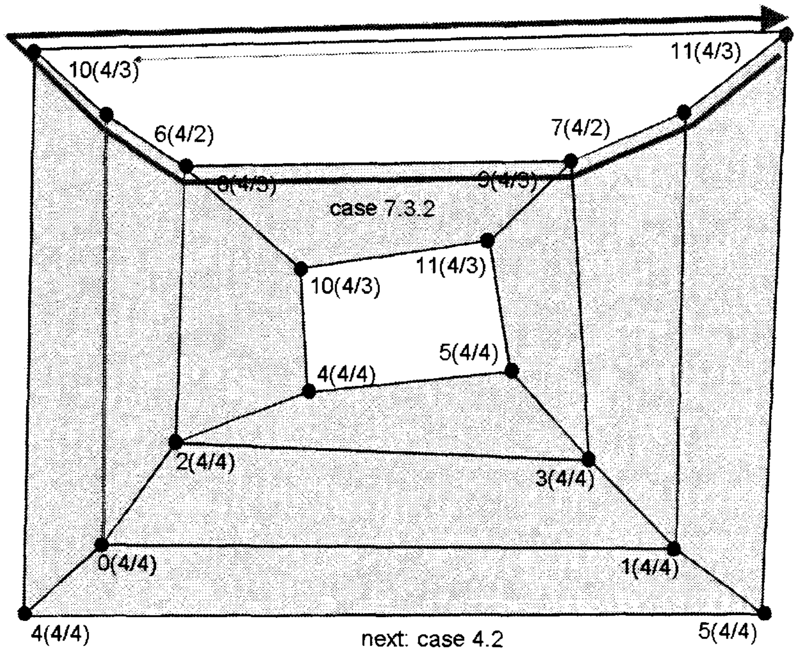


Figure 3.3: Decompression example (Q)

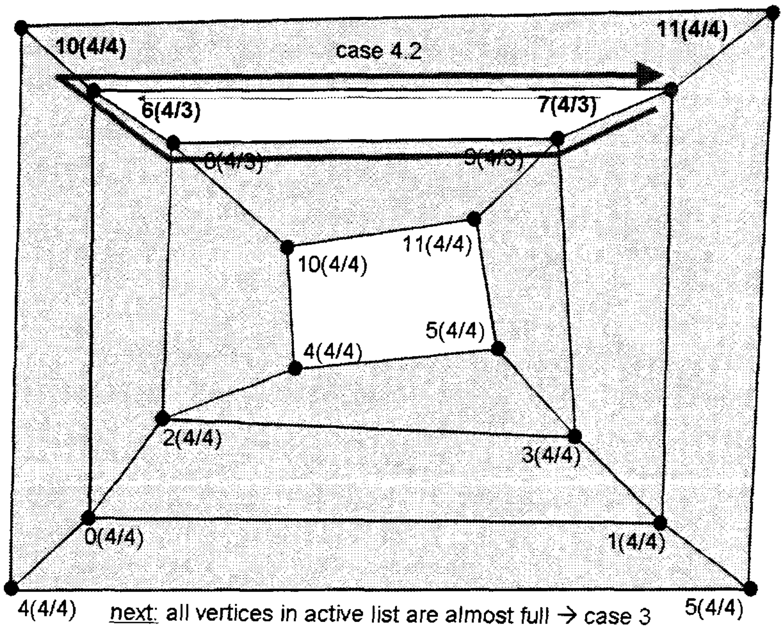


Figure 3.3: Decompression example (R)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

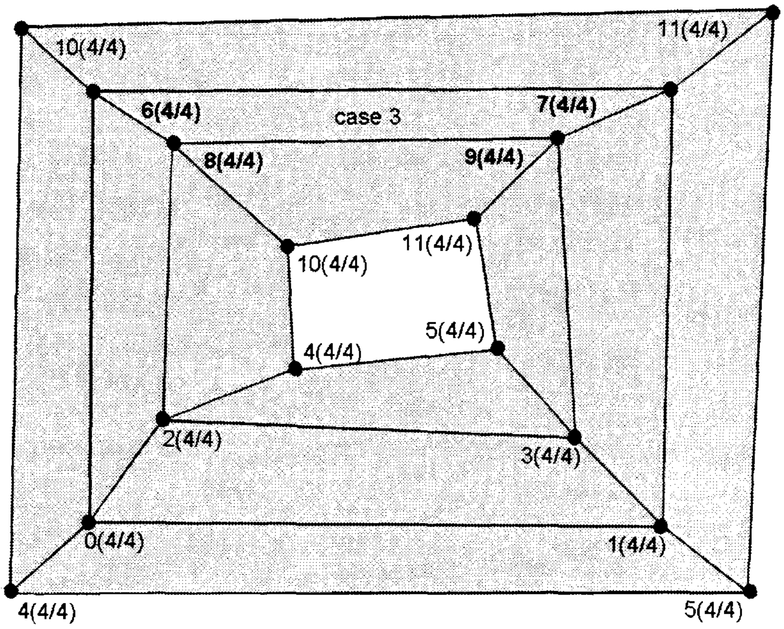


Figure 3.3: Decompression example (S)

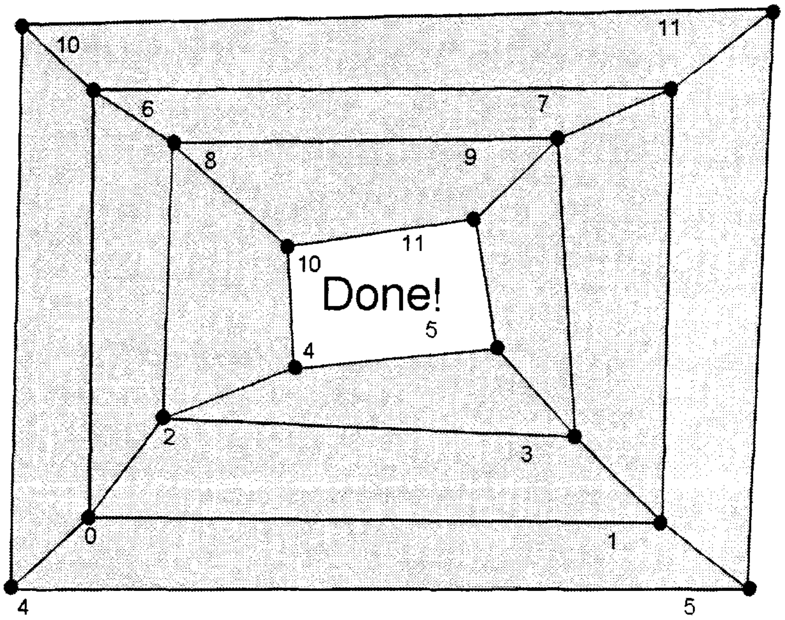


Figure 3.3: Decompression example (T)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Linear time quadmesh decompression using Spirale Reversi

Edgebreaker is a triangle conquest approach for connectivity compression of triangle meshes, which has been explained in section 2.2.6. The approach starts from an initial borderline, which divides the whole mesh into two regions: visited and unvisited regions, and adds one triangle at a time to the visited regions. The main difference between Edgebreaker algorithm and Touma-Gotsman's algorithm is that Edgebreaker outputs the building operations of new triangles, while Touma-Gotsman's algorithm outputs the valences of new vertices.

There are different algorithms that can be used for the decoding process. Among them, Spirale Reversi(20) is the most efficient one. In this chapter we show how to extend the Spirale Reversi algorithm to quadmesh connectivity decompression.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1 Definition of opposite edge (OE) data structure

In this chapter we use a simple data structure called OE (opposite-edge) for representing connectivity of a quad mesh. The data structure is defined by Jing (21).

Fig. 4.1 illustrates the OE data structure. Each quad in a quad mesh is represented by four integer references for the four vertices and four integer references for edges, plus four integer references for the four opposite-edges. The opposite-edge of an edge $e[1]$ in the center quad refers to the edge that is next to next to the opposite half edge of $e[3]$ in the left quad. If $e[3]$ is a boundary edge, we arbitrarily assign -1 in the opposite-edge field for $ohe[1]$. In other words, given an edge of the center quad, its opposite-edge is an edge that belongs to an adjacent quad and is the edge of the center quad that is opposite to the edge that it shares with the adjacent quad. Vertices, edges, and opposite edges are identified using positive integers.

4.2 Edgebreaker compression algorithm

The algorithm for Edgebreaker compression is shown as algorithm 5. The input file is a quad PLY file, and the output files are three files: file "out_operation.txt" contains a sequence of commands, file "out_handles.txt" contains handles information, and file "out_vertex.txt" contains vertex coordinates, as shown below.

$$\boxed{\begin{array}{ll} & out\_operation.txt \\ quadPLYfile \rightarrow & out\_handles.txt \\ & out\_vertex.txt \end{array}}$$

After the compression process, we get the operation code (op-code for short) for the compression operation associated with each quad. We need to encode each operation to convert the operation code to binary code to finalize the compression process. Once

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

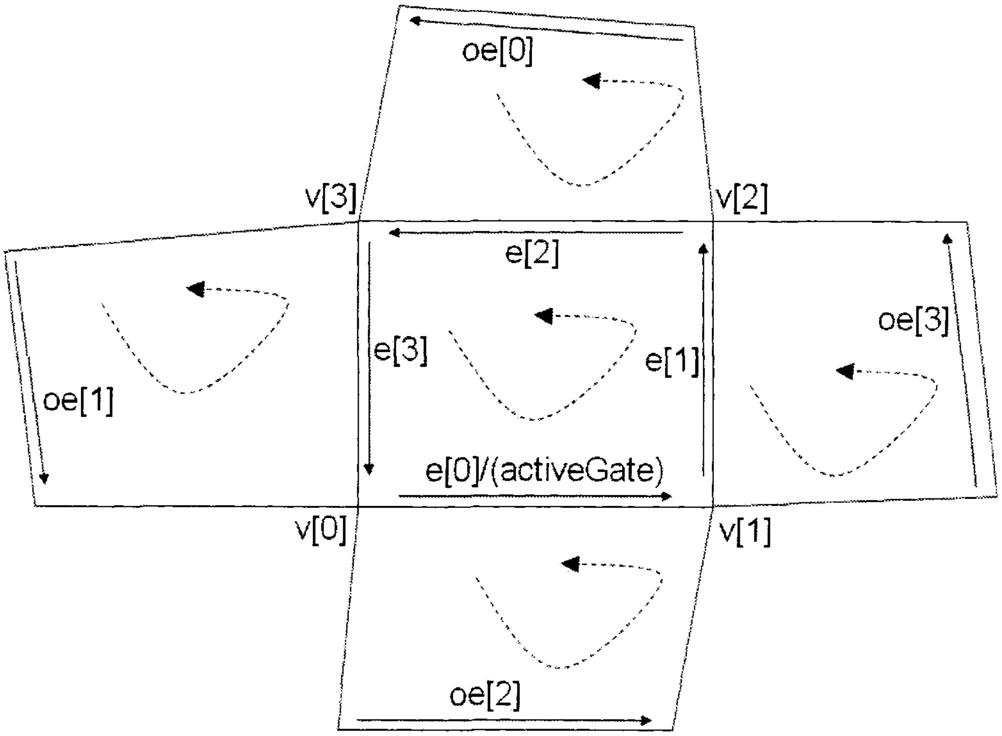


Figure 4.1: OE data structure

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we have the coding scheme and binary coding stream, we can easily extract ASCII operations from the coding stream.

Compression operations are shown in Fig. 4.2. During compression, each operation is processed as follows.

(1). Q1

- set status of all vertices of current quad to visited.

(2). Q2

- v[0].status = v[3].status = visited;
- push opposite half edge at RIGHT to gateStack.

(3). Q3

- v[0].status = visited;
- push opposite half edge at OPPOSITE/RIGHT to gateStack;
- markHandles(e[2]).

(4). Q4

- v[0].status = v[1].status = visited;
- push opposite half edge at OPPOSITE to gateStack.

(5). Q5

- v[0].status = visited;
- mark new inner vertex v[2];
- push opposite half edge at RIGHT to gateStack.

(6). Q6

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- v[1].status = v[2].status = visited;
- push opposite half edge at LEFT to gateStack.

(7). Q7

- push opposite half edge at LEFT/RIGHT to gateStack;
- markHandles(e[3]).

(8). Q8

- push opposite half edge at LEFT/OPPOSITE/RIGHT to gateStack;
- markHandles(e[2], e[3]).

(9). Q9

- v[1].status = visited;
- push opposite half edge at LEFT/OPPOSITE to gateStack;
- markHandles(e[3]).

(10). Q10

- mark new inner vertex v[2];
- push opposite half edge at LEFT/RIGHT to gateStack;
- markHandles(e[3]).

(11). Q11

- mark new inner vertex v[3];
- push opposite half edge at OPPOSITE/RIGHT to gateStack;
- markHandles(e[2]).

(12). Q12

- v[1].status = visited;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- mark new inner vertex v[3];
- push opposite half edge at OPPOSITE to gateStack.

(13). Q13

- mark new inner vertex v[2] and v[3];
- push opposite half edge at RIGHT to gateStack.

4.3 Spirale Reversi decompression algorithm

Following the Spirale Reversi decompression algorithm for triangle mesh, which was described by Isenburg and Snoeyink (20), we define the Spirale Reversi decompression algorithm for quadmesh, which is shown as algorithm 6. The three input files are file "out_operation.txt", "out_handles.txt" and "out_vertex.txt", and the output is a quad PLY file, as shown below.

$$\boxed{\begin{array}{ll} out\_operation.txt & \\ out\_handles.txt & \rightarrow quadPLY file \\ out\_vertex.txt & \end{array}}$$

Decompression operations are shown in Fig. 4.3. For each operation, we process it as follows.

(1). Q1

- set status of all vertices of current quad to visited;
- create a new quad with 4 new vertices;
- set previous on boundary vertex for v[0], v[3], v[2];
- push previous gate to gateStack.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(2). **Q2**

- create a new quad with 2 new vertices;
- set previous on boundary vertex for v[0], v[3].

(3). **Q3**

- create a new quad with 1 new vertex;
- set previous on boundary vertex for v[0];
- remove duplicate created vertices at v[2].

(4). **Q4**

- create a new quad with 2 new vertices;
- set previous on boundary vertex for v[0], v[2].

(5). **Q5**

- create a new quad with 1 new vertex;
- set previous on boundary vertex for v[0];
- mark new inner vertex v[2], assign coordinates for it.

(6). **Q6**

- create a new quad with 2 new vertices;
- set previous on boundary vertex for v[2], v[3].

(7). **Q7**

- pop a gate from gateStack;
- create a new quad;
- set previous on boundary vertex for v[3].

(8). **Q8**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- pop 2 gates from gateStack
- create a new quad ;
- remove duplicate created vertices at v[2], v[3].

(9). Q9

- pop a gate from gateStack
- create a new quad with 1 new vertex;
- set previous on boundary vertex for v[2];
- remove duplicate created vertices at v[3].

(10). Q10

- pop a gate from gateStack;
- create a new quad;
- remove duplicate created vertices at v[3];
- mark new inner vertex v[2], assign coordinates for it.

(11). Q11

- pop a gate from gateStack;
- create a new quad;
- remove duplicate created vertices at v[2];
- mark new inner vertex v[3], assign coordinates for it.

(12). Q12

- create a new quad with 1 new vertex;
- set previous on boundary vertex for v[2];
- mark new inner vertex v[3], assign coordinates for it.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(13). Q13

- create a new quad with 1 new vertex;
- mark new inner vertex v[2] and v[3], assign coordinates for them.

```
input : file_in_ply
output: file_out_operation, file_out_vertex, file_out_handle
Read OE file, and set all vertices and all quads to be unvisited;
Get boundary info by checking OE file, and create stack boundaryVertex;
if boundaryVertex is empty then                    /* no boundary on the mesh */
    Set 1st quad's boundary edges to be the boundary of the mesh;
    Set status of vertices on boundary to be onBoundary;
    Push the opposite edge of the 1st edge on boundary into gateStack;
    Set status of the 1st quad to be visited;
else
    Set status of vertices on boundary to be onBoundary;
    Push the opposite edge of the 1st edge on boundary into gateStack;
while gateStack is not empty do
    activeGate = a gate popped from gateStack;
    activeQuad = the quad containing activeGate;
    if activeQuad is unvisited then
        Get activeQuad's interactionType;
        Update mesh according to interactionType;    /* call function updateMesh() */
        Set status of activeQuad to be visited;
        Push interactionType into operationStack;
        Check possible handles for activeQuad;
Create operation file containing all operations;
Create handle file containing all handles;
Sort vertices according to stacks innerVertex and boundaryVertex;
Create vertex file containing sorted vertex coordinates;
```

Algorithm 5: Quadmesh compression using EdgeBreaker

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
input  : file_in_operation, file_in_vertex, file_in_handle
output: file_out_ply
Read operation file and create stack operationStack;
Read vertex file;
Read handle file, and create stack handleStack;
while operationStack is not empty do
    operation = operationStack.pop();
    processOperation(operation);             /* new vertices/quad created here */
    Set next activeGate to be new quad's 1st edge;
    mark new quad to be unvisited;
Remove duplicate vertices from temporary vertex list;
Assign coordinates for boundary vertices; Create PLY file;
```

Algorithm 6: Quadmesh decompression using Spirale Reversi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4 Illustrate the approach

To illustrate the algorithm, we use a quadmesh with boundary shown in Fig. 4.4(B) as example.

For each vertex and each quad has two possible status: visited or unvisited. We use the status information of the vertices and the quads to identify a quad's quad's type.

There are two kinds of vertices in the mesh: inner vertex and boundary vertex.

We need to create four stacks for the algorithm: an inner vertex stack used to store inner vertices of the mesh, a boundary vertex stack used to store boundary vertices of the mesh, an operation stack used to store quad's types for each quad, and a gate stack used to store gates.

Fig. 4.4(C) to Fig. 4.4(S) shows how the compression algorithm works for the sample mesh, and Fig. 4.5(B) to Fig. 4.5(S) shows how the decompression algorithm works.

4.4.1 Compress a mesh with boundary

After reading the mesh's information from an input file, we can create an image of the mesh, as shown in Fig. 4.4(C).

Next we need to identify the boundary of the quadmesh. All vertices on the boundary are boundary vertices. Boundary vertices are ordered in counter-clockwise order, and then pushed onto the boundary stack. The first active gate $(v0, v1)$, is the opposite half edge of the first edge $(v1, v0)$ on the boundary, as shown in Fig. 4.4(D).

Quad $(v0, v1, v2, v3)$ which contains the active gate is the one we are visiting. We can identify quad $(v0, v1, v2, v3)$ as quad's type "Q5" since its left neighbor quad

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

does not exist or is visited, and the third vertex $v2$ is not on the current boundary now. So, we push vertex $v2$ onto the inner vertex stack, push quad type "Q5" onto the operation stack, and push a new gate $(v2, v1)$ onto the gate stack, as shown in Fig. 4.4(E). What's more, we need to mark $v0$ as not on the current boundary (here we mark $v0$ as *visited*). Although there is no boundary shown in the figures, you still need to keep in mind that there is a boundary (or several boundaries) which enclose the visited region of the mesh during the compression process, and same for the decompression process.

For each step, we need to check if a handle is created or not. Since in this example there are no any handles created, we won't discuss how to identify handles and how to handle them.

Next, we pop a gate $(v2, v1)$ off the gate stack. Now quad $(v2, v1, v4, v5)$ which contains active gate is the one we are visiting, and we identify it as quad's type "Q11". So, we push vertex $v5$ onto the inner vertex stack, push quad's type "Q11" onto the operation stack, and push two new gates $(v4, v1)$ and $(v5, v4)$ onto the gate stack, as shown in Fig. 4.4(F).

Next, we pop a gate $(v4, v1)$ off the gate stack. Now quad $(v4, v1, v6, v7)$ which contains the active gate is the one we are visiting, and we identify it as quad's type "Q1". So, we just mark all vertices of the quad as not being on the current boundary (here we mark the vertices as *visited*), and push quad's type "Q1" onto the operation stack, as shown in Fig. 4.4(G).

Next, we pop a gate $(v5, v4)$ off the gate stack. Now quad $(v5, v4, v8, v9)$ which contains the active gate is the one we are visiting, and we identify it as quad's type "Q6". So, we just mark vertex $v4$ and vertex $v8$ of the quad as not being on the current boundary (here we mark the vertices as *visited*), push one new gate$(v5, v9)$ onto the gate stack, and push quad's type "Q6" onto the operation stack, as shown in Fig. 4.4(H).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Next, we pop a gate $(v5, v9)$ off the gate stack. Now quad $(v5, v9, v10, v11)$ which contains the active gate is the one we are visiting, and we identify it as quad's type "Q11". So, we push vertex $v11$ onto the inner vertex stack, push quad's type "Q11" onto the operation stack, and push two new gates $(v11, v10)$ and $(v10, v9)$ onto the gate stack, as shown in Fig. 4.4(I).

Next, we pop a gate $(v10, v9)$ off gate stack. Now quad $(v10, v9, v12, v13)$ which contains active gate is the one we are visiting, and we identify it as quad's type "Q1". So, we just mark all vertices of the quad as not on the current boundary, and push quad's type "Q1" onto the operation stack, as shown in Fig. 4.4(J).

Next, we pop a gate $(v11, v10)$ off gate stack. Now quad $(v11, v10, v14, v15)$ which contains active gate is the one we are visiting, and we identify it as quad's type "Q6". So, we just mark vertex $v10$ and vertex $v14$ of the quad as not on the current boundary, push one new gate$(v11, v15)$ to gate stack, and push quad's type "Q6" onto the operation stack, as shown in Fig. 4.4(K).

Next, we pop a gate $(v11, v15)$ off gate stack. Now quad $(v11, v15, v16, v17)$ which contains active gate is the one we are visiting, and we identify it as quad's type "Q11". So, we push vertex $v17$ onto the inner vertex stack, push quad's type "Q11" onto the operation stack, and push two new gates $(v16, v15)$ and $(v17, v16)$ onto the gate stack, as shown in Fig. 4.4(L).

Next, we pop a gate $(v16, v15)$ off gate stack. Now quad $(v16, v15, v18, v19)$ which contains active gate is the one we are visiting, and we identify it as quad's type "Q1". So, we just mark all vertices of the quad as not on the current boundary, and push quad's type "Q1" onto the operation stack, as shown in Fig. 4.4(M).

Next, we pop a gate $(v17, v16)$ off gate stack. Now quad $(v17, v16, v20, v21)$ which contains active gate is the one we are visiting, and we identify it as quad's type "Q6". So, we just mark vertex $v16$ and vertex $v20$ of the quad as not on the current

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

boundary, push one new gate($v17, v21$) onto the gate stack, and push quad's type "Q6" onto the operation stack, as shown in Fig. 4.4(N).

Next, we pop a gate ($v17, v21$) off gate stack. Now quad ($v17, v21, v3, v2$) which contains active gate is the one we are visiting, and we identify it as quad's type "Q7". So, we push quad's type "Q7" onto the operation stack, and push two new gates ($v3, v21$) and ($v17, v2$) onto the gate stack, as shown in Fig. 4.4(O).

Next, we pop a gate ($v3, v21$) off gate stack. Now quad ($v3, v21, v22, v23$) which contains active gate is the one we are visiting, and we identify it as quad's type "Q1". So, we just mark all vertices of the quad as not on the current boundary, and push quad's type "Q1" onto the operation stack, as shown in Fig. 4.4(P).

Next, we pop a gate ($v17, v2$) off gate stack. Now quad ($v17, v2, v5, v11$) which contains active gate is the one we are visiting, and we identify it as quad's type "Q1". So, we just mark all vertices of the quad as not on the current boundary, and push quad's type "Q1" onto the operation stack, as shown in Fig. 4.4(Q).

Now, the gate stack is empty, and we claim that the compression process is done. The output are two files, as shown in Fig. 4.4(R). File "out_operation.txt" contains the sequence of quad's types, and file "out_vertex.txt" contains the sequence of vertices, which contains two parts. The first part is the ordered inner vertices, which are identified in sequence during compression process, and the second part is the ordered boundary vertices, which are identified before we traverse any quads in the mesh. We need to keep in mind that, the last two vertices in file "out_vertex.txt" are two boundary vertices which are endpoints of the first active gate. The first active gate is where the compression starts and where the decompression ends.

The operation file "out_operation.txt" can be encoded using some code schemes, which will be discussed later.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.2 Decompress a mesh with boundary

Now we have two input files for decompression, file "out_vertex.txt" contains a sequence of vertex coordinates, and file "out_operation.txt" contains a sequence of operations, as shown in Fig. 4.5(B). Let's see how we reconstruct the mesh using these two input files.

As shown in Fig. 4.5(C), first, we read file "out_vertex.txt" and put the vertices onto a vertex stack; then, we read the file "out_operation.txt", and put the operations onto a operation stack. We also need to create another two more empty stacks, one is the gate stack, the other one is the inner vertex stack.

The decompression process works like this: we pop the operations from the operation stack one by one, and create a quad for each operation, until the operation stack is empty.

The first operation popped out from operation stack is "Q1". So we create four "virtual" vertices, create a quad with them, and set the active gate to be the first edge $g1$ of the quad, as shown in Fig. 4.5(D). For the newly created quad and its four vertices ($v[0], v[1], v[2], v[3]$), we claim that all the four vertices are on the boundary now, the previous on-boundary vertex of vertex $v[0]$ is $v[3]$, the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$. Actually, for each on-boundary vertex, we remember its previous on-boundary vertex; thus we know the exact information about current boundary of the mesh during decompression.

Next operation popped out from operation stack is "Q1". So we create four more "virtual" vertices, create a quad with them, push active gate $g1$ onto gate stack and set the active gate to be the first edge $g2$ of the quad, as shown in Fig. 4.5(E). For the new created quad and its four vertices ($v[0], v[1], v[2], v[3]$), we claim that all the four vertices are on boundary now, the previous on boundary vertex of vertex $v[0]$ is $v[3]$,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q7". So we pop a gate $g1$ off gate stack, create a quad with the four vertices of gate $g1$ and gate $g2$, and set active gate to be the first edge $g3$ of the quad, as shown in Fig. 4.5(F). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that the previous on-boundary vertex of vertex $v[3]$ is $v[2]$.

Next operation popped out from operation stack is "Q6". So we So we create two more "virtual" vertices, create a quad with the two "virtual" vertices and the two vertices of gate $g3$, and set active gate to be the first edge $g4$ of the quad, as shown in Fig. 4.5(G). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q1". So we create four more "virtual" vertices, create a quad with them, push active gate $g4$ to gate stack and set active gate to be the first edge $g5$ of the quad, as shown in Fig. 4.5(H). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that all the four vertices are on boundary now, the previous on boundary vertex of vertex $v[0]$ is $v[3]$, the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q11". We pop a gate $g4$ off gate stack first. The start vertex of active gate $g5$ and the end vertex of gate $g4$ are the same vertex in the mesh, so we remove one of them out from the mesh. Next, we create a quad with the previous on-boundary vertex of the start vertex of gate $g4$, and the three vertices of gate $g4$ and gate $g5$. Next, we set active gate to be the first edge $g6$ of the quad, as shown in Fig. 4.5(I). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we can see that $v[3]$ is an inner vertex, so we pop one

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertex $v17$ out from vertex stack and push it onto inner vertex stack. We claim that $v[3]$ of the new created quad is $v17$.

Next operation popped out from operation stack is "Q6". So we So we create two more "virtual" vertices, create a quad with the two "virtual" vertices and the two vertices of gate $g6$, and set active gate to be the first edge $g7$ of the quad, as shown in Fig. 4.5(J). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q1". So we create four more "virtual" vertices, create a quad with them, push active gate $g7$ onto gate stack and set active gate to be the first edge $g8$ of the quad, as shown in Fig. 4.5(K). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that all the four vertices are on boundary now, the previous on boundary vertex of vertex $v[0]$ is $v[3]$, the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q11". We pop a gate $g7$ off gate stack first. The start vertex of active gate $g8$ and the end vertex of gate $g7$ are the same vertex in the mesh, so we remove one of them out from the mesh. Next, we create a quad with the previous on-boundary vertex of the start vertex of gate $g7$, and the three vertices of gate $g7$ and gate $g8$. Next, we set active gate to be the first edge $g9$ of the quad, as shown in Fig. 4.5(L). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we can see that $v[3]$ is an inner vertex, so we pop one vertex $v11$ out from vertex stack and push it onto inner vertex stack. We claim that $v[3]$ of the new created quad is $v11$.

Next operation popped out from operation stack is "Q6". So we So we create two more "virtual" vertices, create a quad with the two "virtual" vertices and the two vertices of gate $g9$, and set active gate to be the first edge $g10$ of the quad, as shown

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in Fig. 4.5(M). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q1". So we create four more "virtual" vertices, create a quad with them, push active gate $g10$ to gate stack and set active gate to be the first edge $g11$ of the quad, as shown in Fig. 4.5(N). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we claim that all the four vertices are on boundary now, the previous on boundary vertex of vertex $v[0]$ is $v[3]$, the previous on-boundary vertex of vertex $v[3]$ is $v[2]$, and the previous on-boundary vertex of vertex $v[2]$ is $v[1]$.

Next operation popped out from operation stack is "Q11". We pop a gate $g10$ off gate stack first. The start vertex of active gate $g11$ and the end vertex of gate $g10$ are the same vertex in the mesh, so we remove one of them out from the mesh. Next, we create a quad with the previous on-boundary vertex of the start vertex of gate $g10$, and the three vertices of gate $g10$ and gate $g11$. Next, we set active gate to be the first edge $g12$ of the quad, as shown in Fig. 4.5(O). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we can see that $v[3]$ is an inner vertex, so we pop one vertex $v5$ out from vertex stack and push it onto inner vertex stack. We claim that $v[3]$ of the new created quad is $v5$.

Next operation popped out from operation stack is "Q5". So we create one "virtual" vertex, create a quad with the "virtual" vertex, the previous on-boundary vertex of the start vertex of gate $g12$, and the two vertices of gate $g12$. Next, we set active gate to be the first edge $g13$ of the quad, as shown in Fig. 4.5(P). For the new created quad and its four vertices $(v[0], v[1], v[2], v[3])$, we can see that $v[2]$ is an inner vertex, so we pop one vertex $v2$ out from vertex stack and push it onto inner vertex stack. We claim that $v[2]$ of the new created quad is $v2$, and the previous on-boundary vertex of vertex $v[0]$ is $v[3]$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Now there are no any operations left in the operation stack, and we have identified all the inner vertices and labeled them. The problem is: how to label the boundary vertices?

Now the vertex stack only contains boundary vertices which are stored in order. The two vertices ($v1$ and $v0$) at the bottom of the vertex stack is the vertices of the last active gate during decompression process. So, we sort the boundary vertices in the reconstructed mesh in counter-clockwise order, set vertex $v1$ to be the first vertex of the boundary, and assign coordinates for them one by one, as shown in Fig. 4.5(R).

Now we can say that we complete the decompression process successfully. The reconstructed mesh is shown in Fig. 4.5(S).

4.5 Quadmesh of higher genus

Non-genus-0 mesh contains handles, which never come up in simple mesh (genus-0 mesh). A handle is an edge shared by two quads which are visited during compression process, where the first quad is a S-type quad (split-type quad, which means one of Q3, Q8, Q9, Q10 or Q11), and the second quad intersects with the first quad at the second edge, the third edge or the fourth edge of the second quad. Here we suppose that the first edge of a quad is the active gate when visiting the quad.

Since we don't know which quad could have a handle binding on it before compression, so we just created an array "*handlesForQuad*" which will be used to store information about handles. The size of the array "*handlesForQuad*" is the same as the number of quads in the mesh. Each element of the array "*handlesForQuad*" is filled with values when processing a S-type quad (which could be the first quad of a handle) during compression. The values are used later when processing a second quad checking if a handle exists between the first quad and the second quad.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When a S-type quad (which could be the first quad of a handle) was processed during compression, we put a mark saying that the gate(s) pushed into stack when processing this quad could possibly create a handle(s).

When a quad (which could be the second quad of a handle) is processed during compression, we check its adjacent quad on its left/right/opposite(above). If one visited adjacent quad is S-type, it means that the edge shared by them is a handle. We push this handle into handle stack.

The handles in the handle stack are ordered according to when the first quad is processed. Handles are stored in a file called "out_handles.txt", which looks like this:

`First line:` *"# of handles"*
`Following lines:` *"index of the second quad" "interaction position (offset)"*

Here "index of the second quad" means the index indicating when the second quad is visited during compression, "interaction position" means the offset between the active gate when visiting the second quad and the edge of the second quad where the second quad interacts with the first quad.

For example, if file "out_handles.txt"contains the following information,

```
2
444 1
555 2
```

we can know that the mesh contains two handles (so the mesh is a genus-1 mesh). For the first handle, the second quad of the handle is the 444th quad visited during compression, and it interacts with the first quad of the handle at the second edge (which is the next edge of the active gate when visiting the second quad). For the second handle, the second quad of the handle is the 555th quad visited during com-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pression, and it interacts with the first quad of the handle at the third edge (which is the opposite edge of the active gate when visiting the second quad).

4.6 Apply encoding schemes for the operation sequence

Currently there two encoding schemes can be used to encode the sequence of operation code. One is developed by Gotman (24), which is shown in table 4.1. the other one is developed by Dr. Mukhopadhyay (25), which is shown in table 4.2. Applying the two encoding schemes for the op-code archives higher compression ratio. Details will be discussed in Chapter 5.

Similar to Touma-Gotsman's algorithm which have been discussed in Chapter 3, we can invites entropy coding, like Huffman to encode the op-code. Table 4.3 shown the Huffman code scheme for sample mesh "Torus".

Interaction Type	Code
Q1	11000
Q2	11001
Q3	11010
Q4	11011
Q5	010
Q6	11100
Q7	11101
Q8	11110
Q9	11111
Q10	011
Q11	100
Q12	101
Q13	00

Table 4.1: Gotsman's encoding scheme for Edgebreaker algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Encoding	Current Quad	Next Quad	Code	Num. of bits
Quad started with Q6-13	Q6	Q1-5	11111	5
		Q6-13	11110	5
	Q7	Q1-5	11101	5
		Q6-13	11100	5
	Q8	Q1-5	11011	5
		Q6-13	11010	5
	Q9	Q1-5	11001	5
		Q6-13	11000	5
	Q10	Q6-13	10111	5
	Q11	Q1-5	10110	5
		Q6-13	10101	5
	Q12	Q6-13	100	3
	Q13	Q6-13	0	1
Quad started with Q1-5	Q1	Q1-5	00	2
		Q6-13	01	2
	Q2	Q1-5	1100	4
		Q6-13	1101	4
	Q3	Q1-5	1010	4
		Q6-13	1011	4
	Q4	Q1-5	1000	4
		Q6-13	1001	4
	Q5	Q6-13	111	3

Table 4.2: Dr. Mukhopadhyay's encoding scheme for Edgebreaker algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Interaction Type	Code
Q1	010
Q4	00
Q6	011
Q7	110
Q12	111
Q13	10

Table 4.3: Huffman coding scheme for sample mesh "Torus"

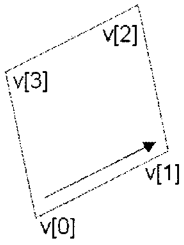
4.7 Time complexity analysis

For the compression process, since each quad is processed exactly once, and the processing time for each quad is constant time, and accessing the next quad is constant time, the overall time complexity of compression process is linear. Same for the decompression process.

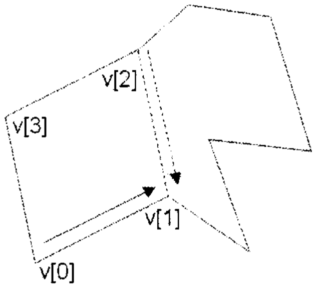
4.8 Summary

In this chapter we discussed how Edgebreaker/Spirale Reversi compression and decompression algorithm for triangle meshes is successfully extended for quadrilateral meshes. Same as Touma-Gotsman's algorithm, the time complexity is linear.

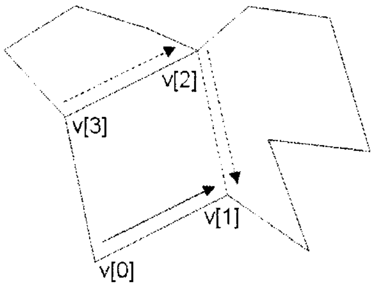
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



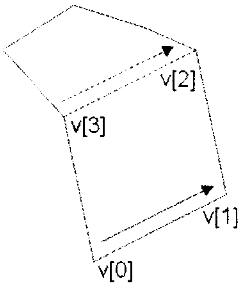
Q1



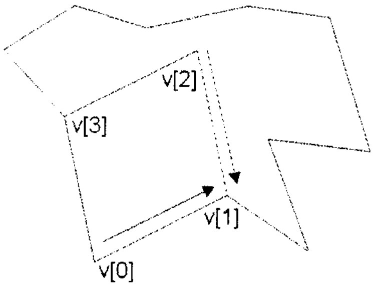
Q2



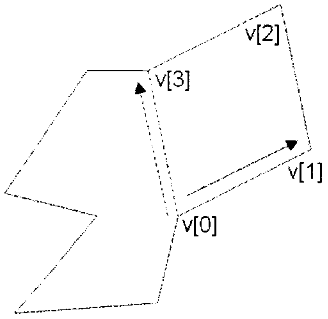
Q3



Q4

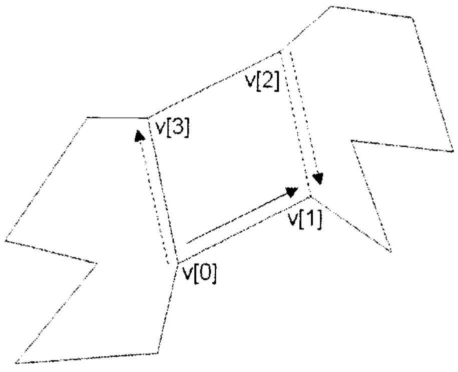


Q5

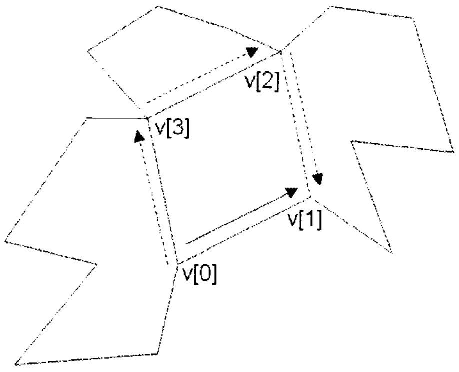


Q6

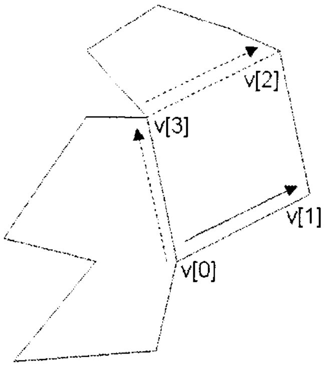
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



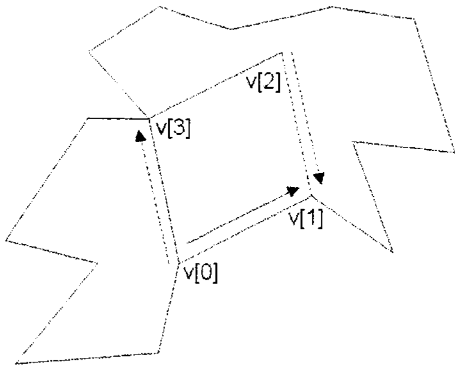
Q7



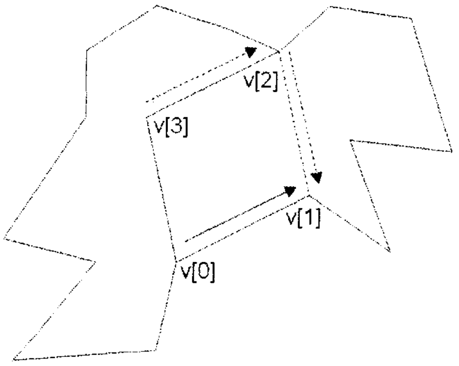
Q8



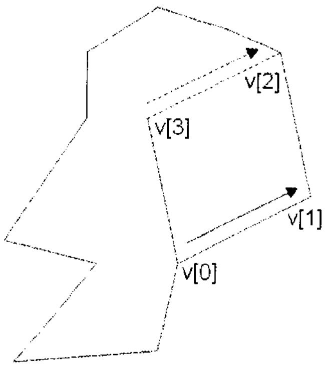
Q9



Q10

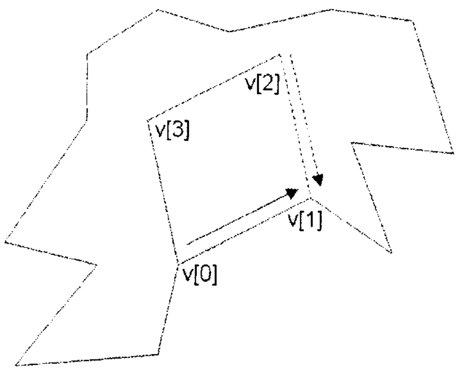


Q11



Q12

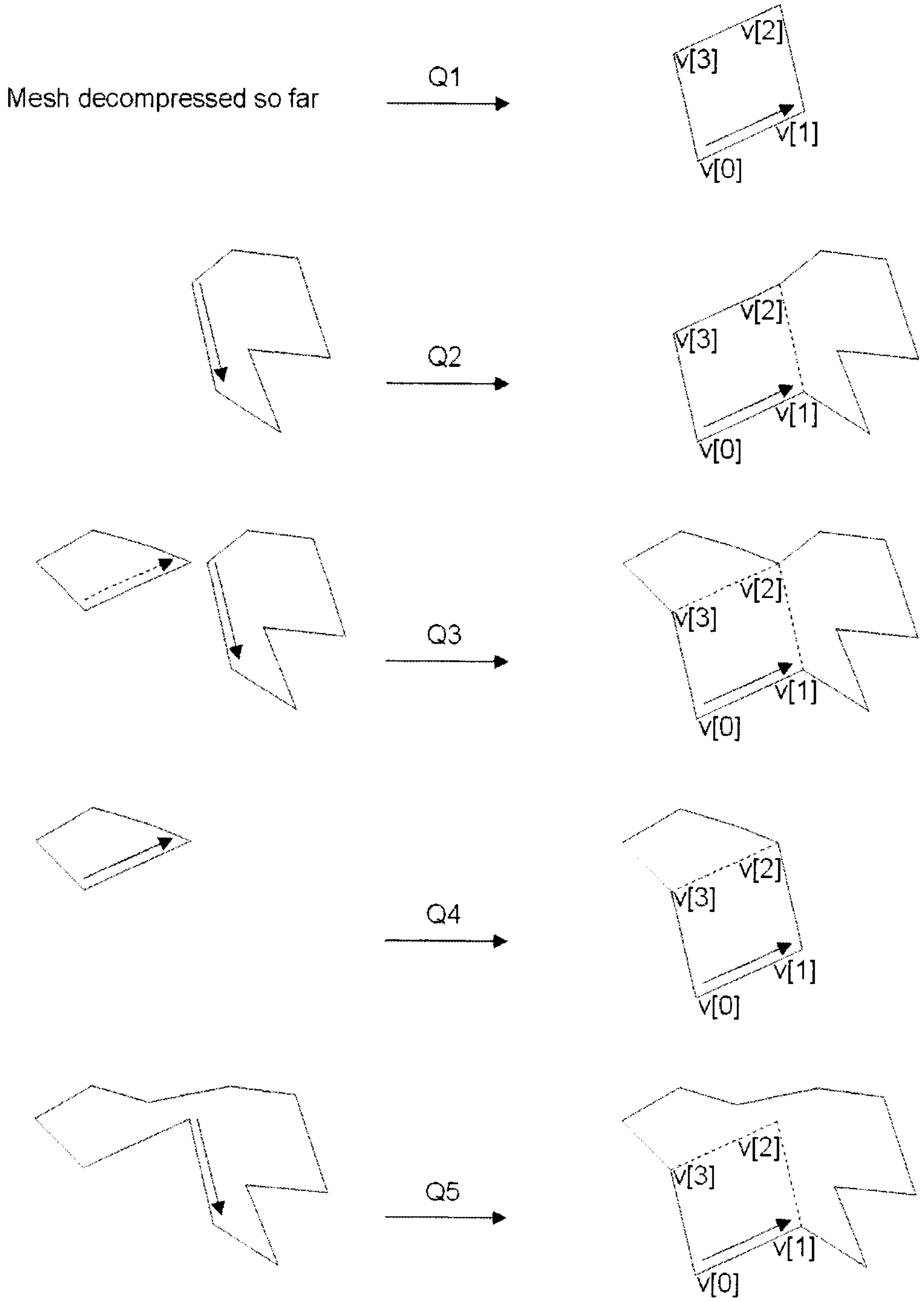
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



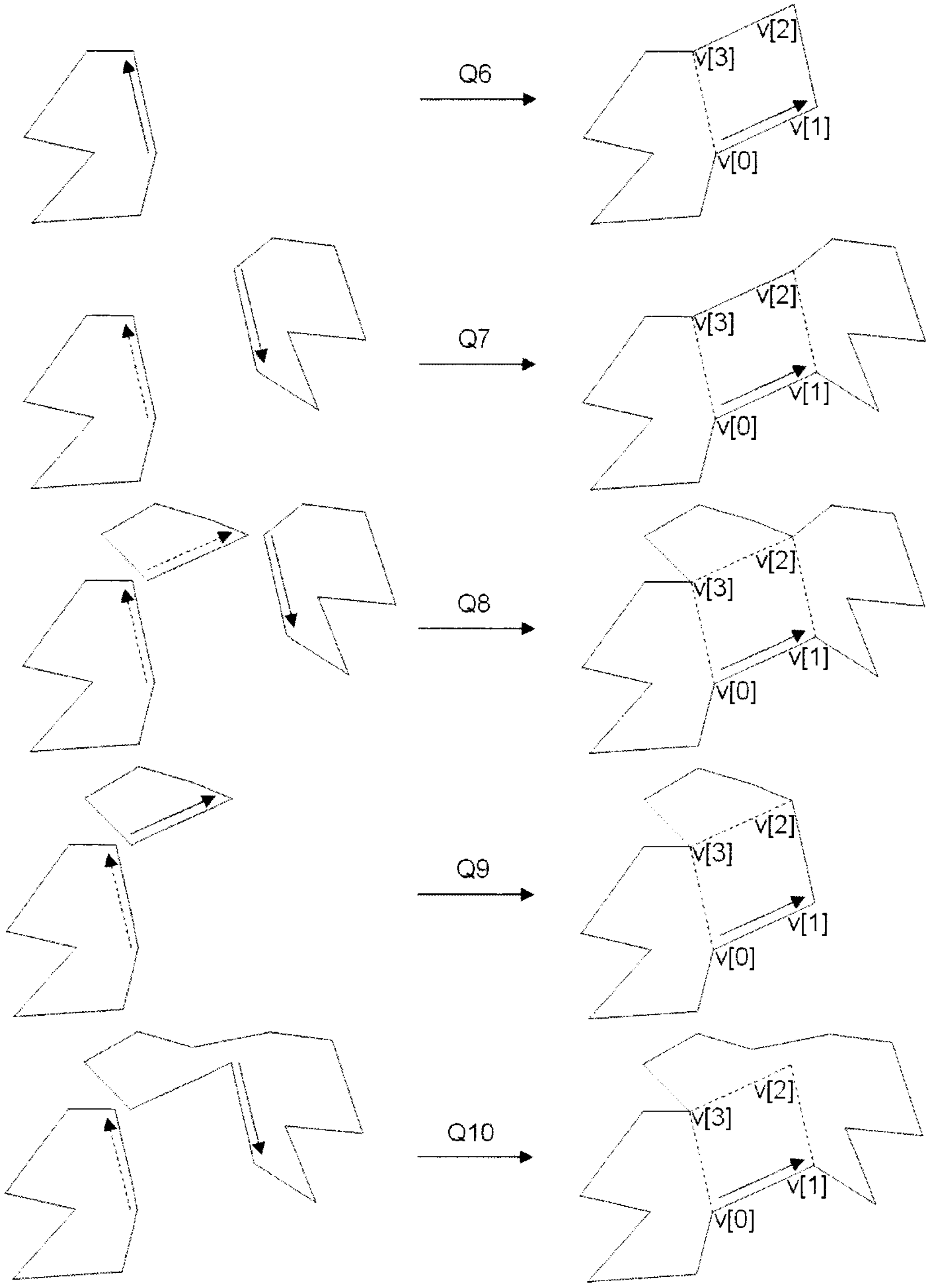
Q13

Figure 4.2: Compression operations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

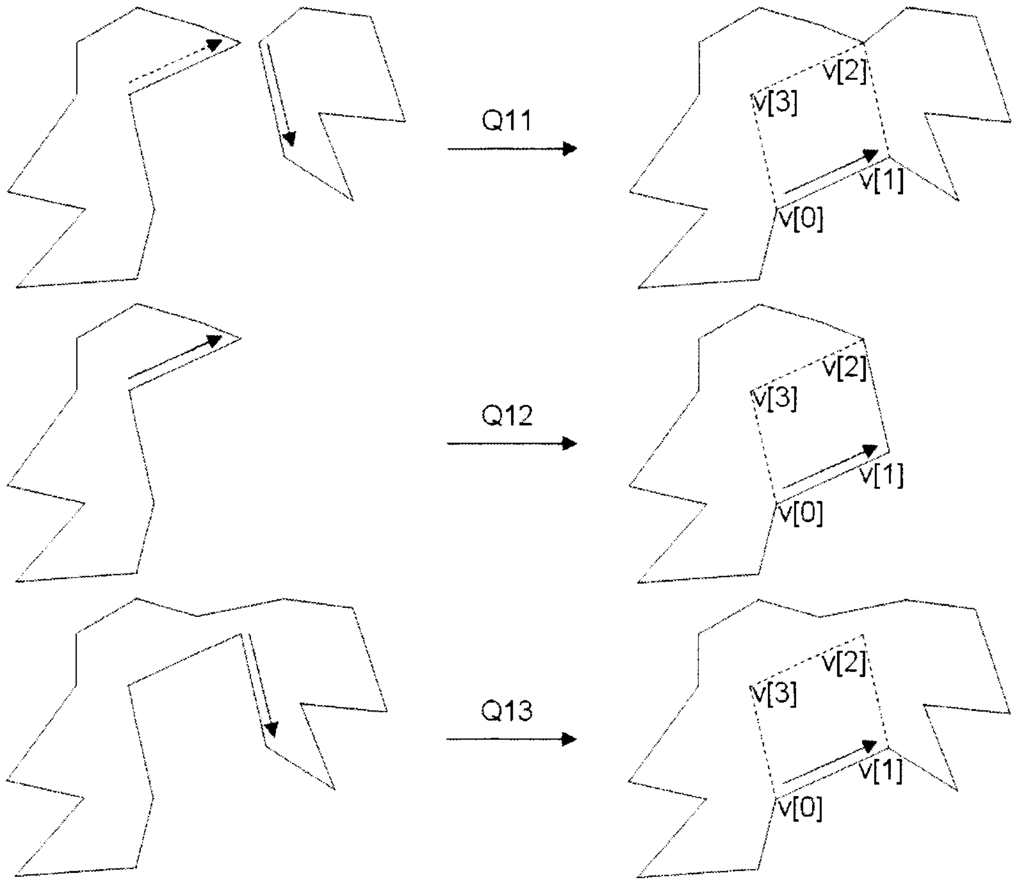


Figure 4.3: Decompression operations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example:
Encode a quadmesh with boundary

Figure 4.4: Compression example (A to S)

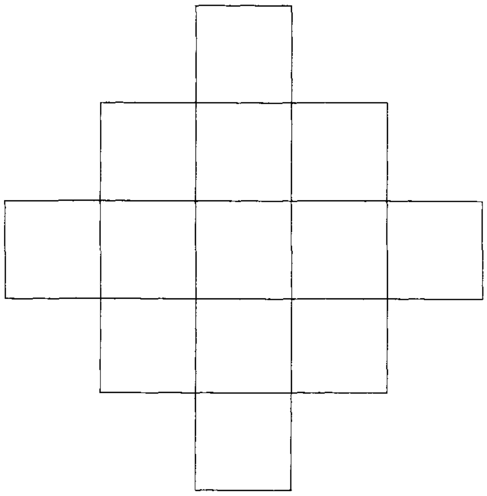


Figure 4.4: Compression example (B)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

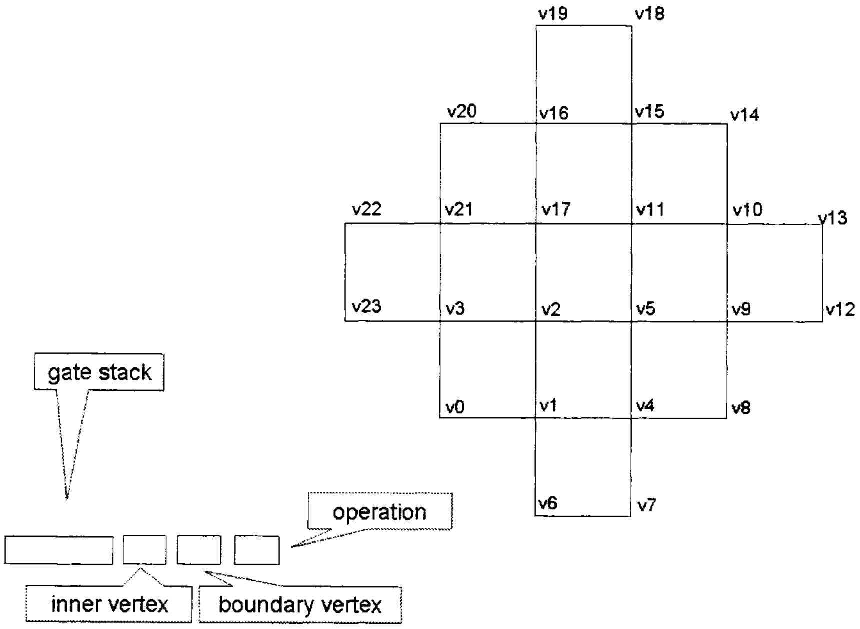


Figure 4.4: Compression example (C)

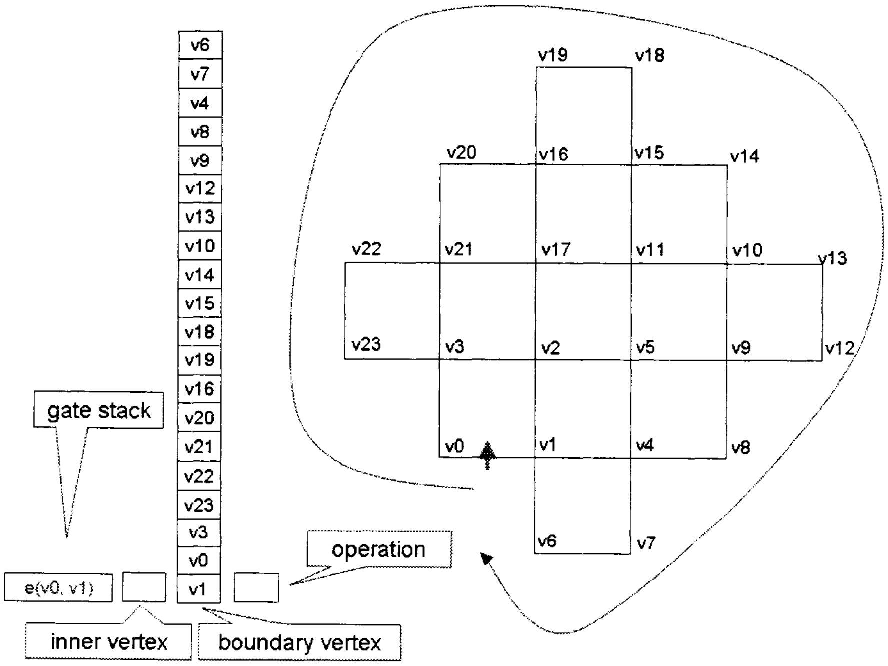


Figure 4.4: Compression example (D)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

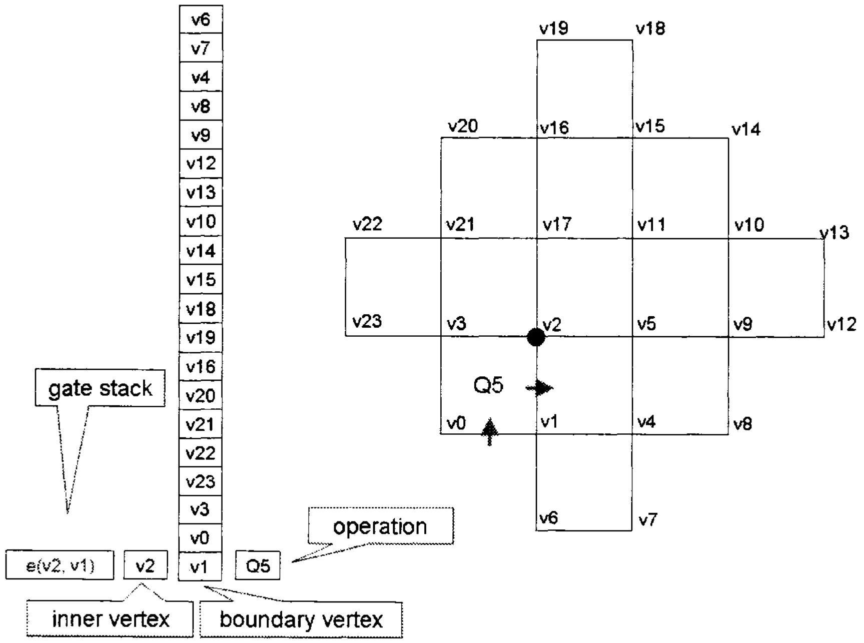


Figure 4.4: Compression example (E)

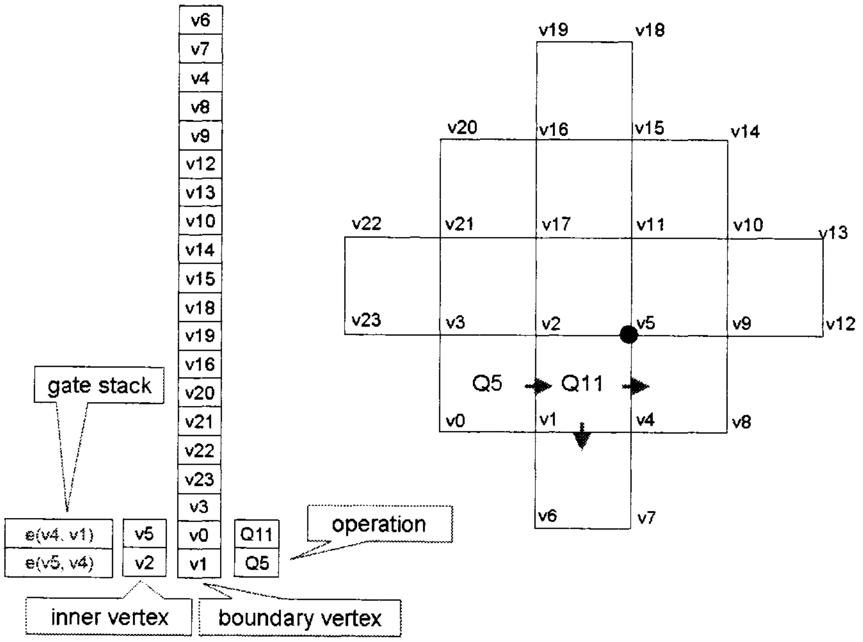


Figure 4.4: Compression example (F)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

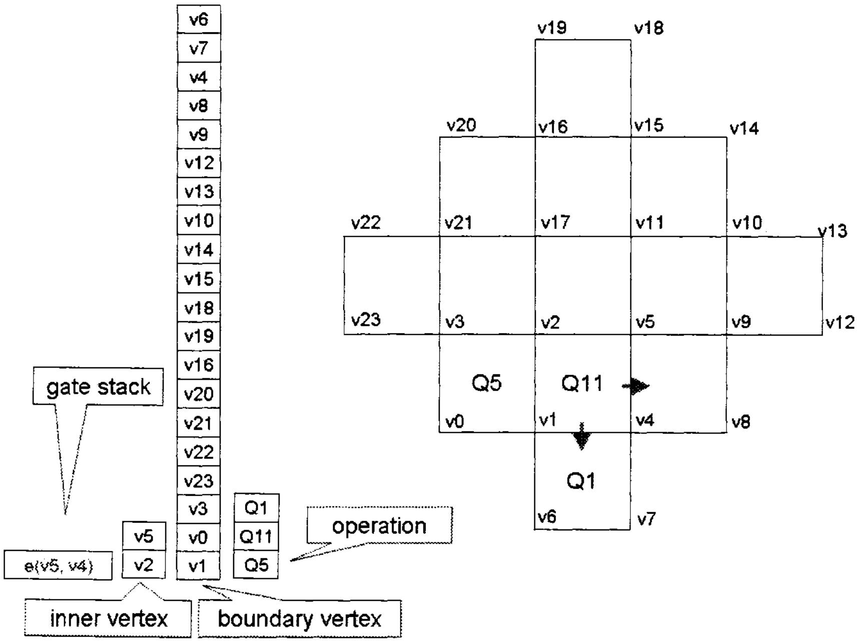


Figure 4.4: Compression example (G)

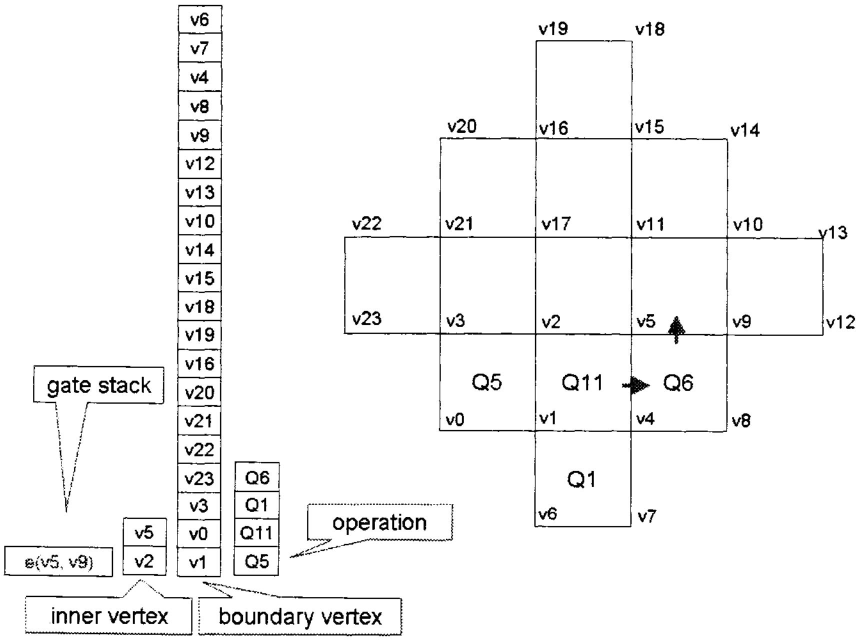


Figure 4.4: Compression example (H)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

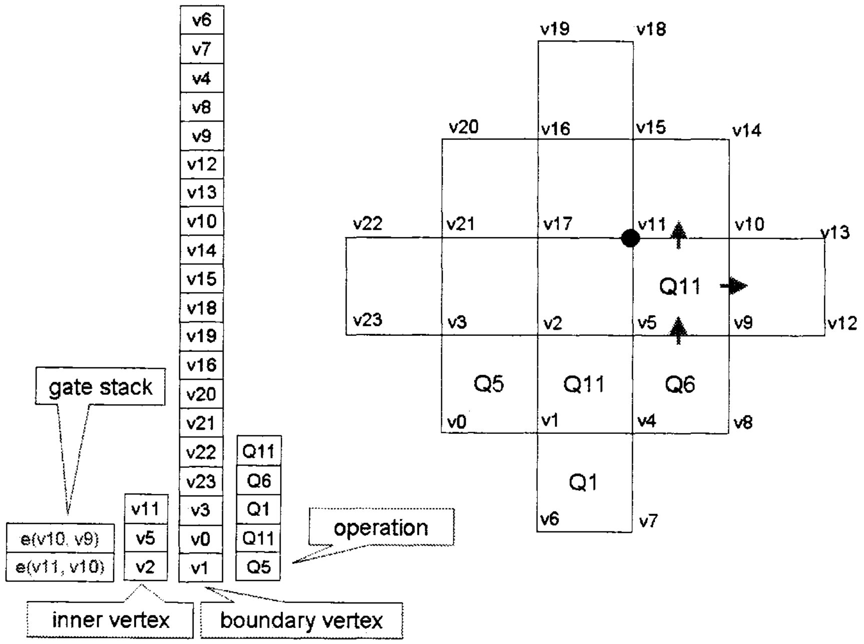


Figure 4.4: Compression example (I)

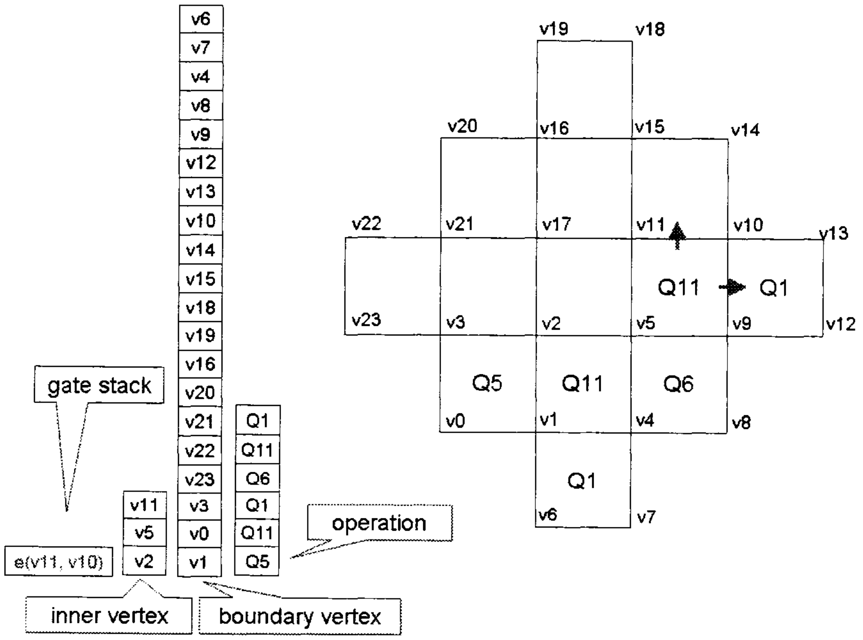


Figure 4.4: Compression example (J)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

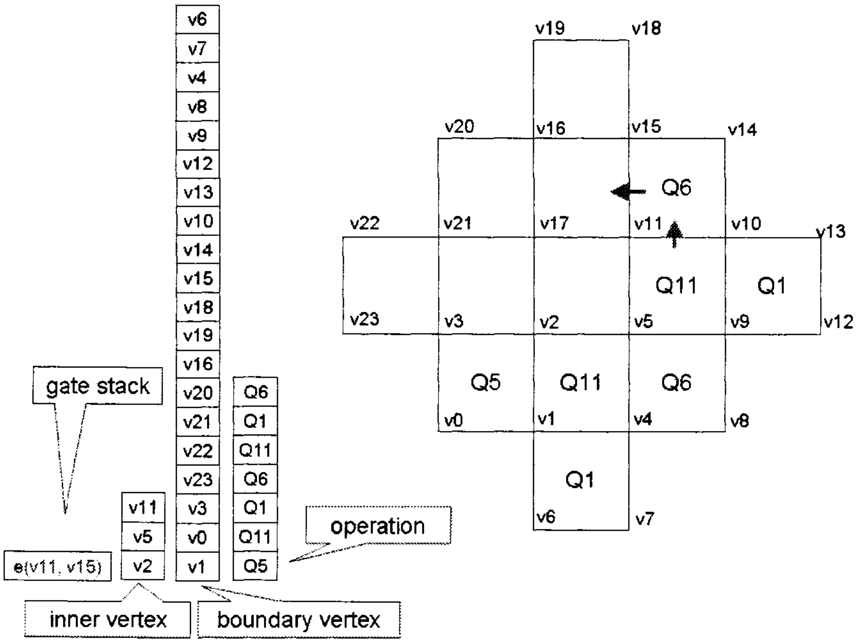


Figure 4.4: Compression example (K)

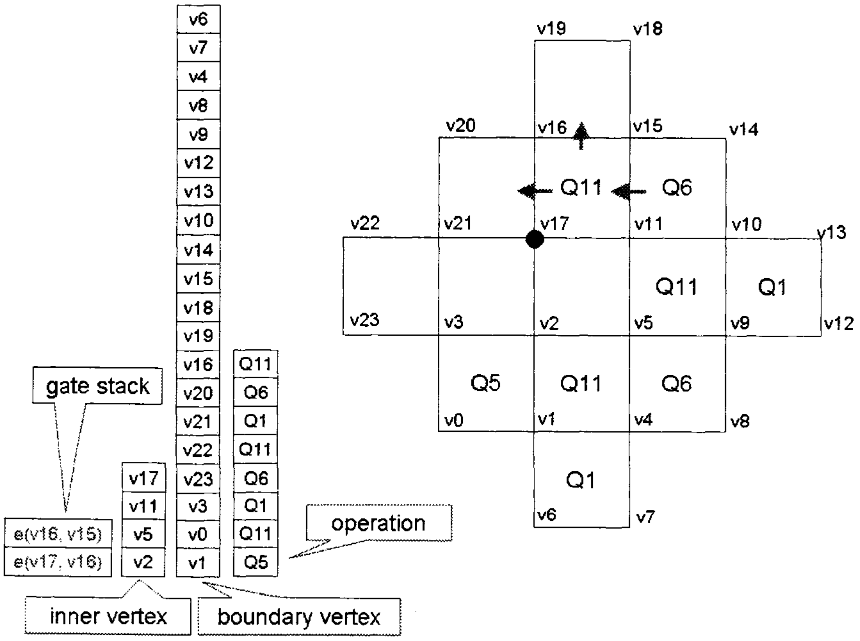


Figure 4.4: Compression example (L)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

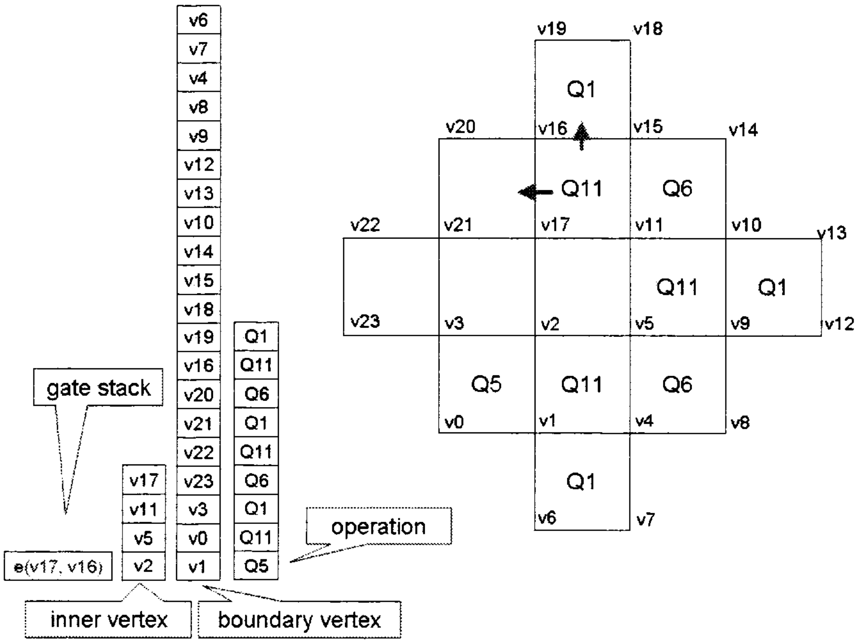


Figure 4.4: Compression example (M)

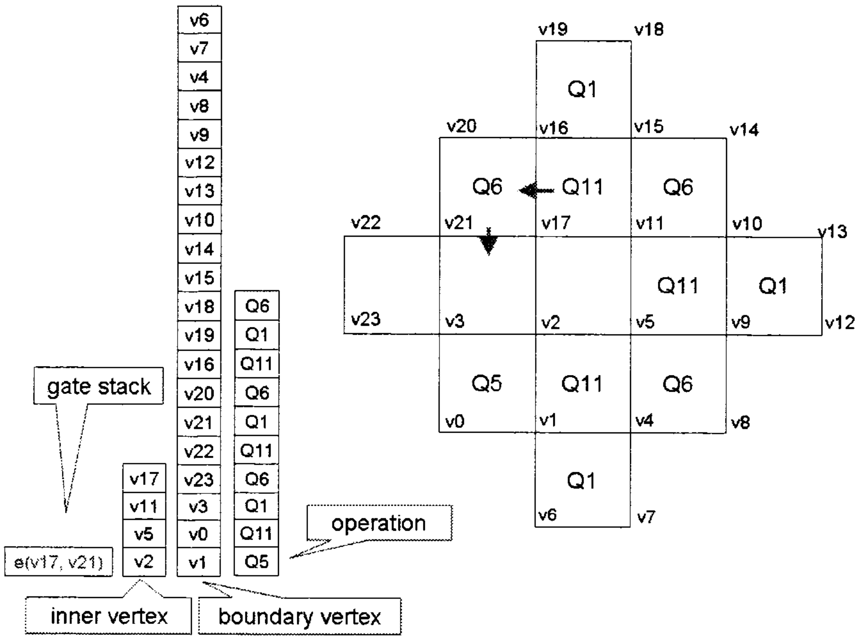


Figure 4.4: Compression example (N)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

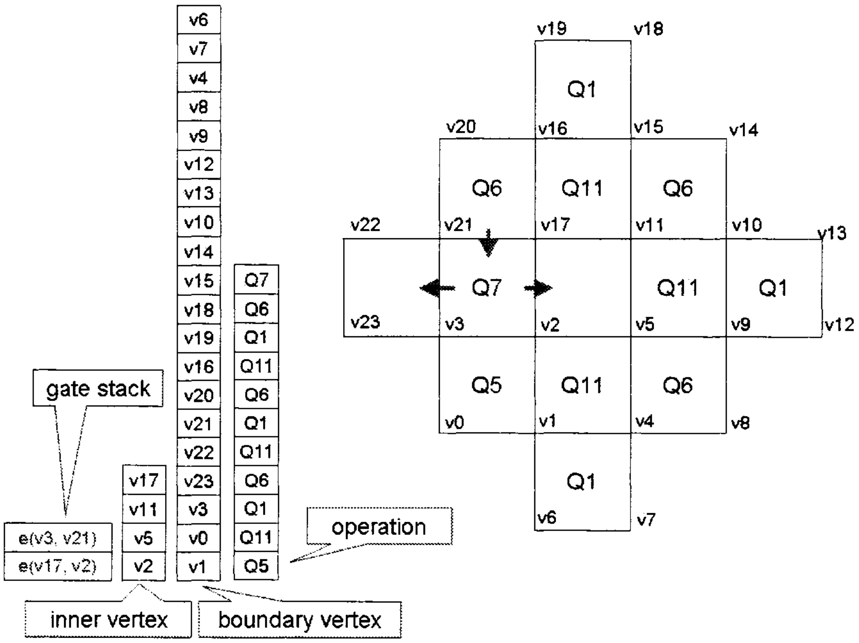


Figure 4.4: Compression example (O)

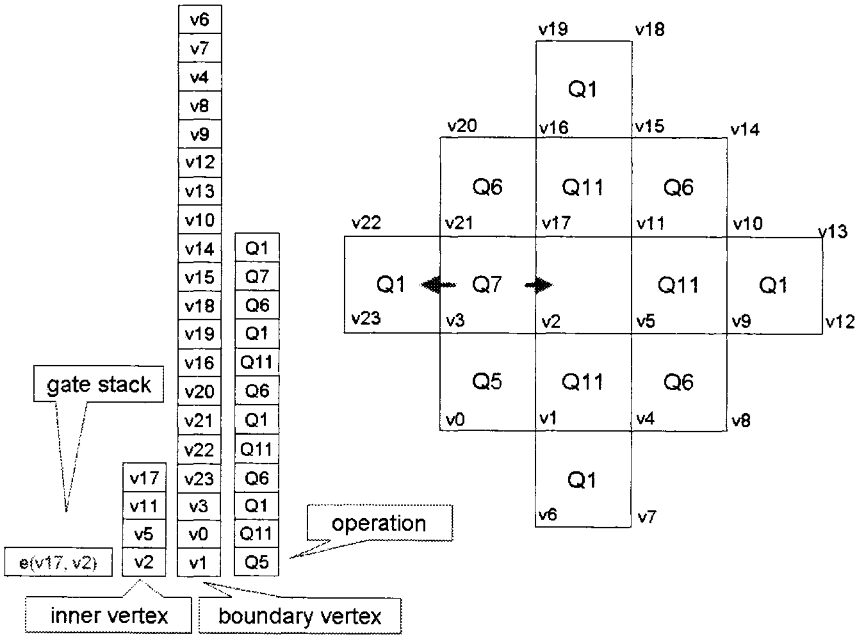


Figure 4.4: Compression example (P)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

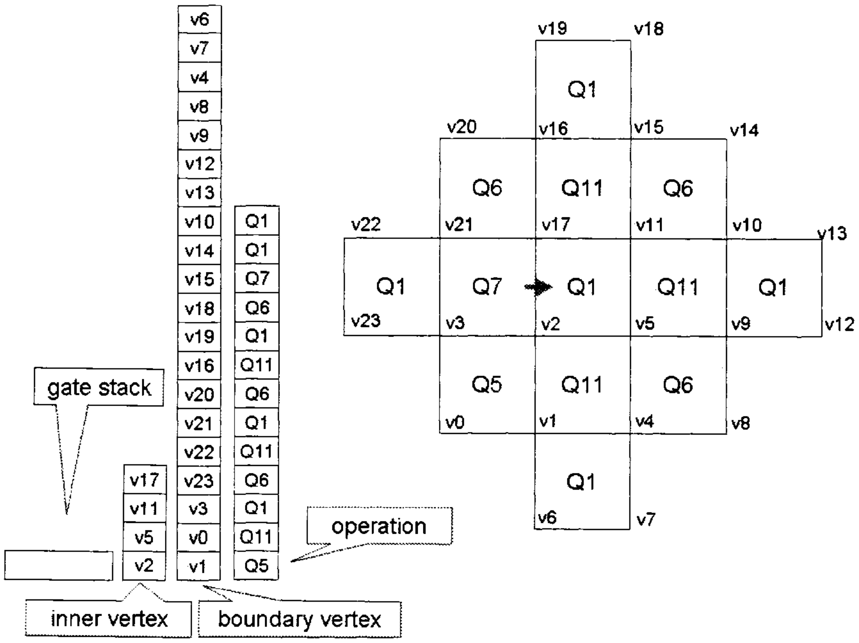
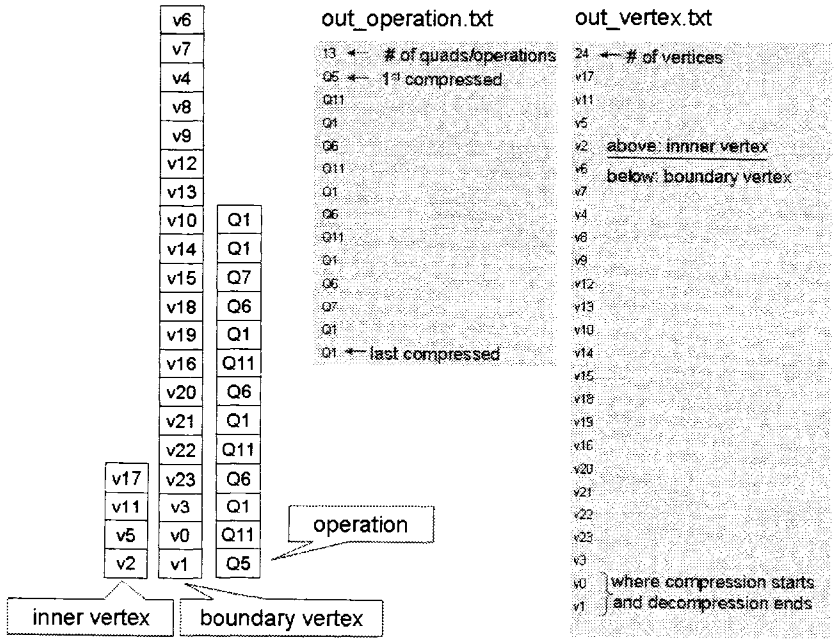


Figure 4.4: Compression example (Q)



out_operation.txt

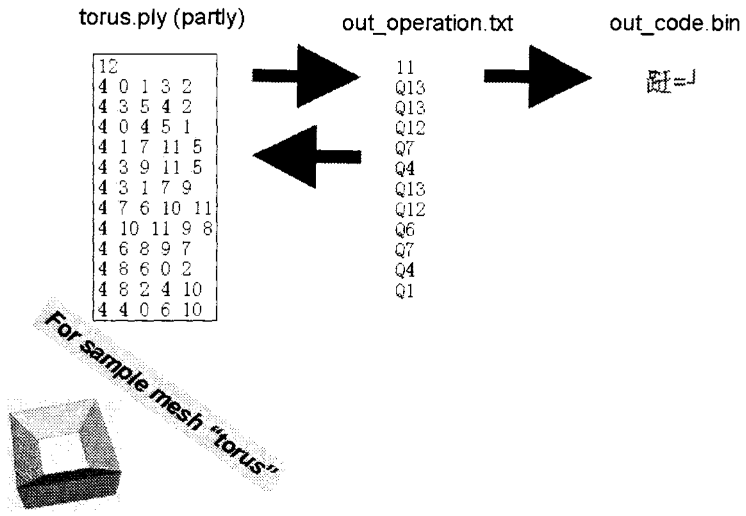
```
13 ←  # of quads/operations
Q5 ←  1st compressed
Q11
Q1
Q6
Q11
Q1
Q6
Q11
Q1
Q6
Q7
Q1
Q1 ← last compressed
```

out_vertex.txt

```
24 ← # of vertices
v17
v11
v5
v2   above: innner vertex
v6   below: boundary vertex
v7
v4
v8
v9
v12
v13
v10
v14
v15
v18
v19
v16
v20
v21
v22
v23
v3
v0  } where compression starts
v1  } and decompression ends
```

Figure 4.4: Compression example (R)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



```
12
4 0 1 3 2
4 3 5 4 2
4 0 4 5 1
4 1 7 11 5
4 3 9 11 5
4 3 1 7 9
4 7 6 10 11
4 10 11 9 8
4 6 8 9 7
4 8 6 0 2
4 8 2 4 10
4 4 0 6 10
```

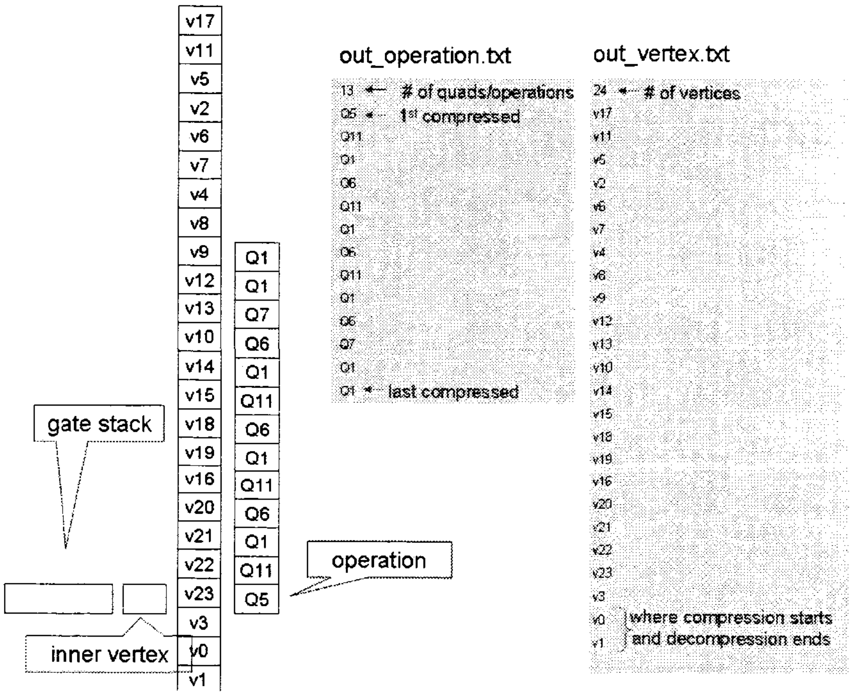
```
11
Q13
Q13
Q12
Q7
Q4
Q13
Q12
Q6
Q7
Q4
Q1
```

Figure 4.4: Compression example (S)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

How to decode using Spirale Reversi algorithm?

Figure 4.5: Decompression example (A to S)



out_operation.txt

```
13 ← # of quads/operations
Q5 ← 1st compressed
Q11
Q1
Q6
Q11
Q1
Q6
Q11
Q1
Q6
Q7
Q1
Q1 ← last compressed
```

out_vertex.txt

```
24 ← # of vertices
v17
v11
v5
v2
v6
v7
v4
v8
v9
v12
v13
v10
v14
v15
v18
v19
v16
v20
v21
v22
v23
v3
v0 } where compression starts
v1 } and decompression ends
```

Figure 4.5: Decompression example (B)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

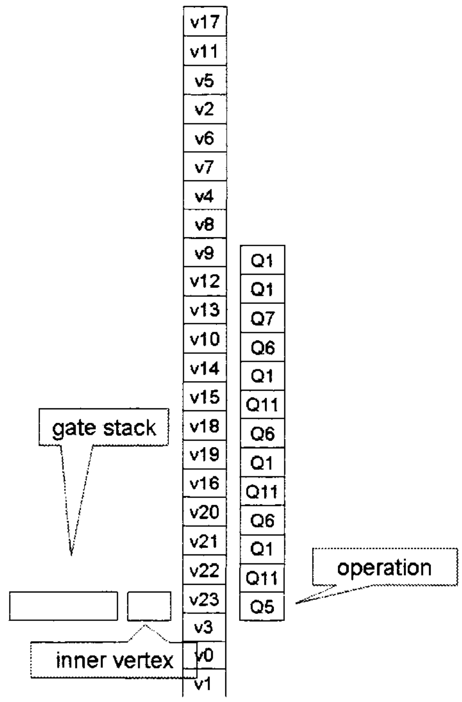


Figure 4.5: Decompression example (C)

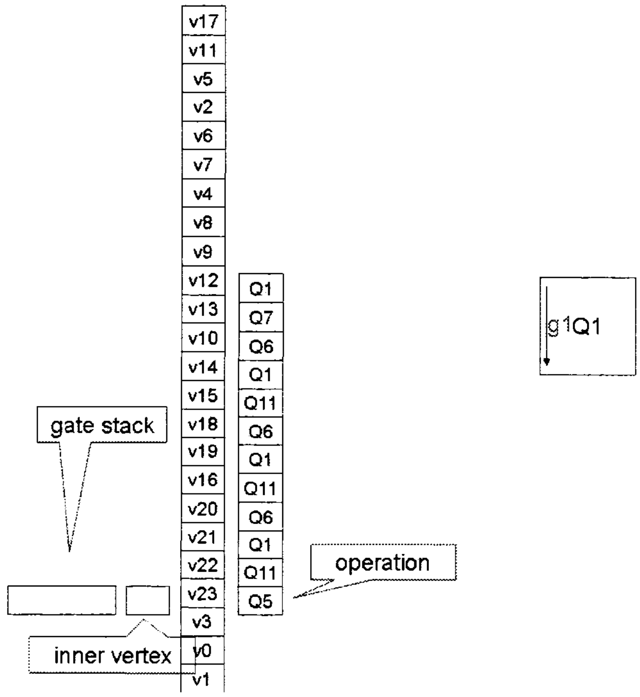


Figure 4.5: Decompression example (D)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

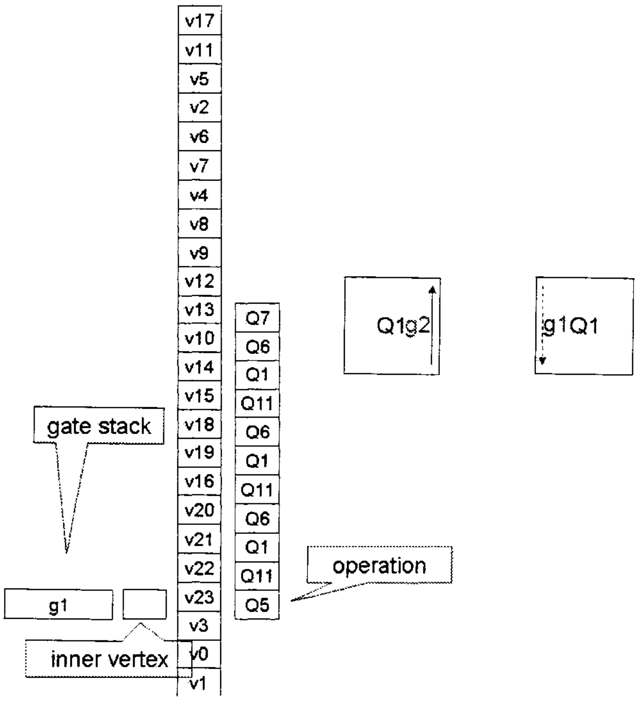


Figure 4.5: Decompression example (E)

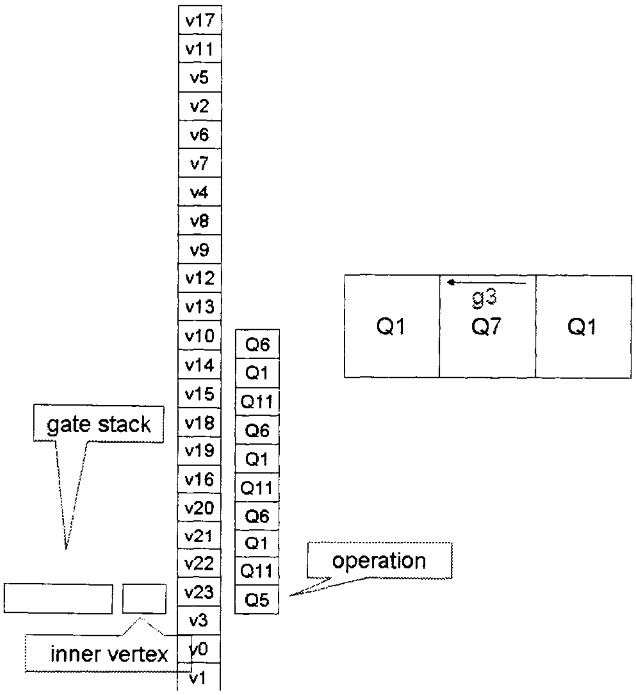


Figure 4.5: Decompression example (F)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

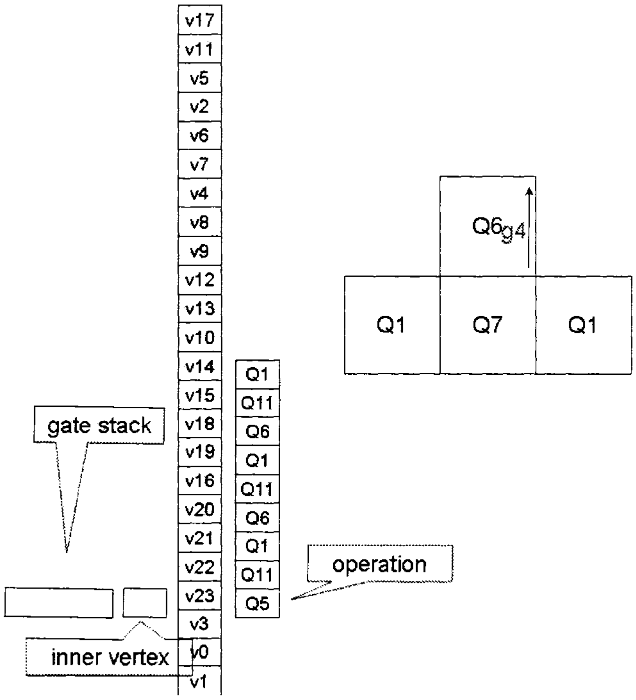


Figure 4.5: Decompression example (G)

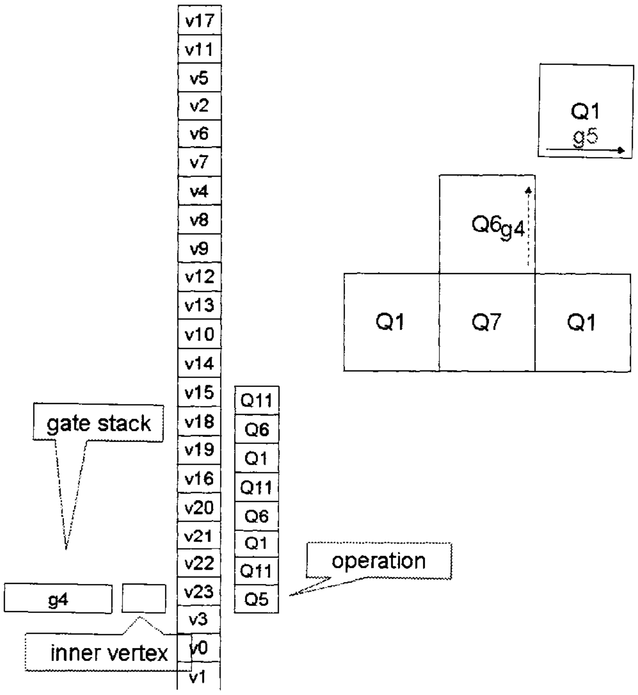


Figure 4.5: Decompression example (H)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

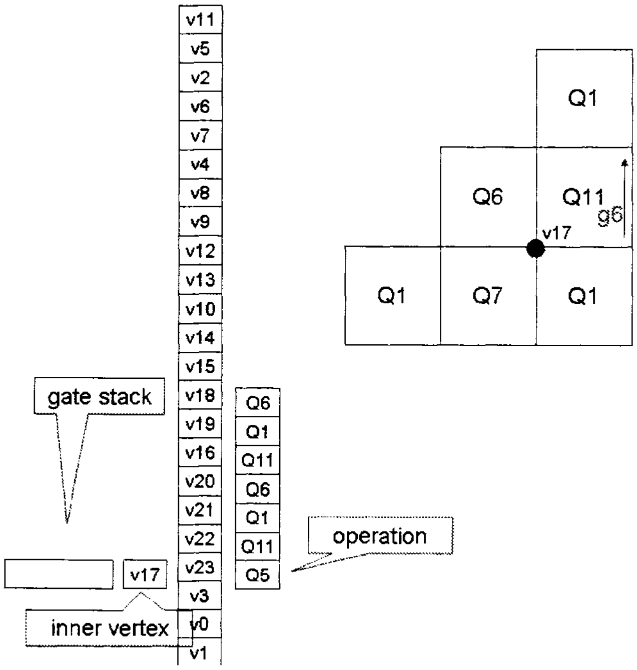


Figure 4.5: Decompression example (I)

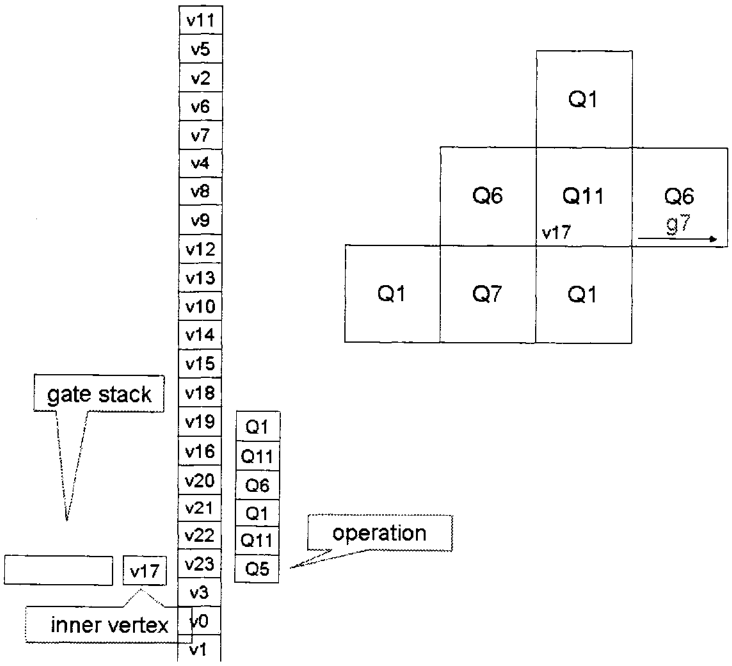


Figure 4.5: Decompression example (J)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

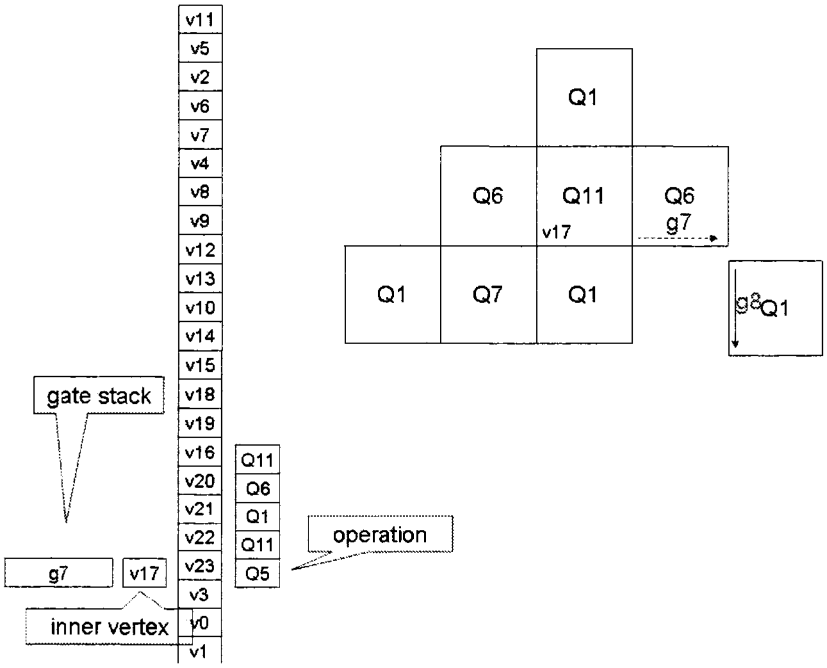


Figure 4.5: Decompression example (K)

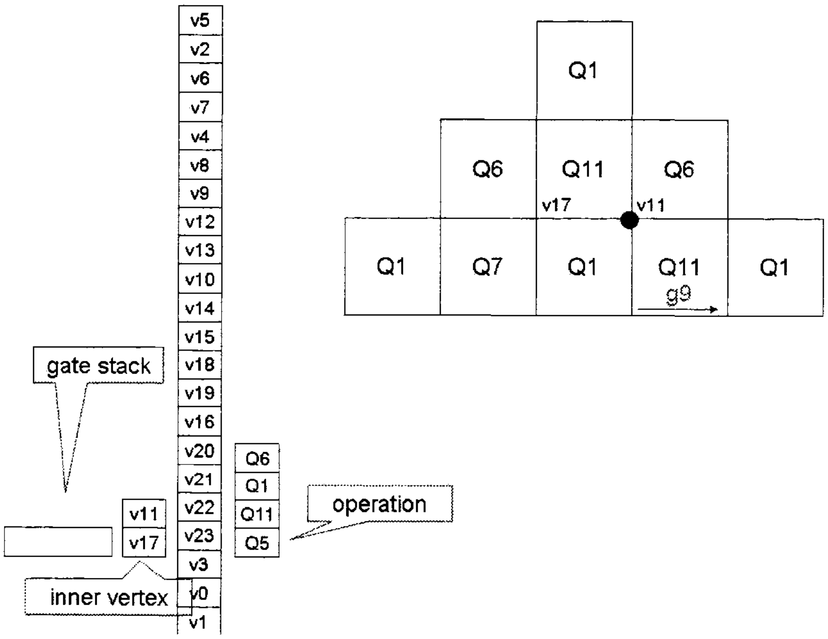


Figure 4.5: Decompression example (L)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

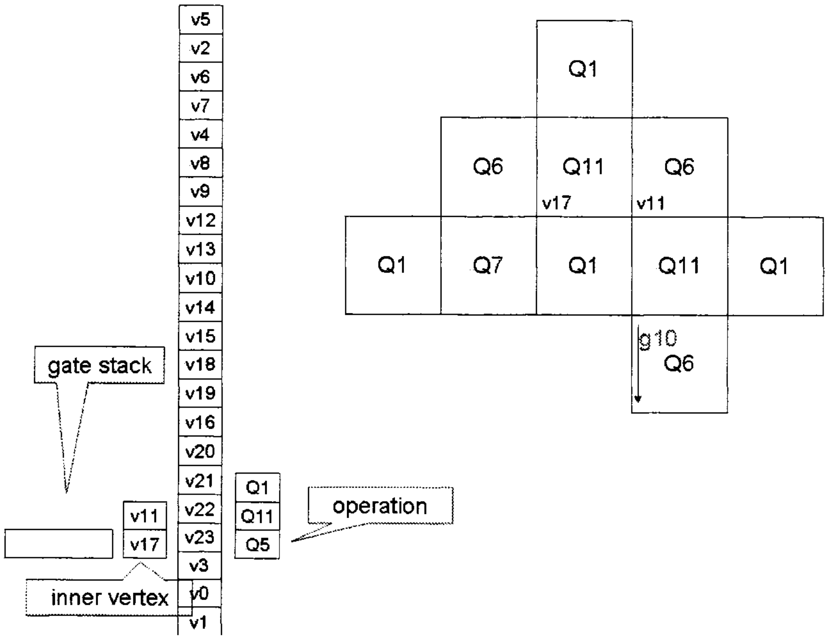


Figure 4.5: Decompression example (M)

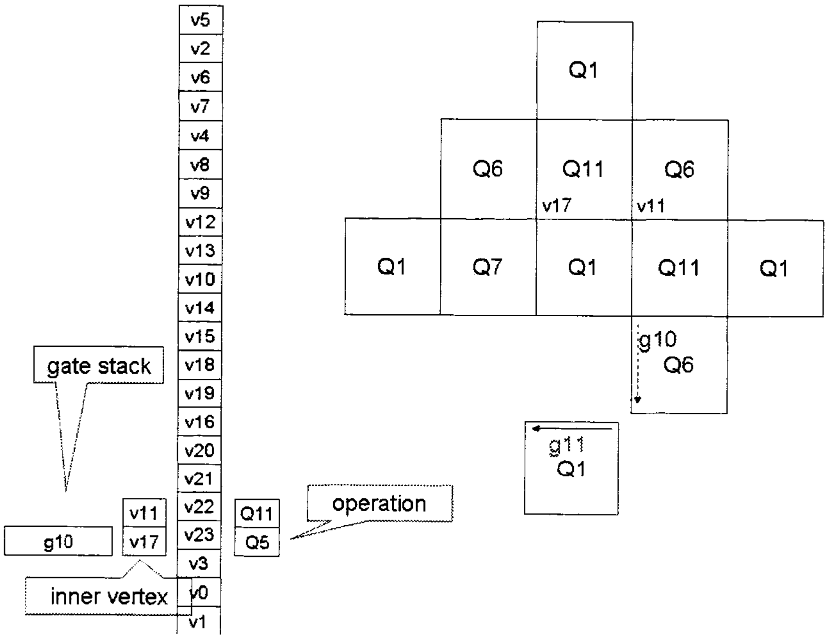


Figure 4.5: Decompression example (N)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

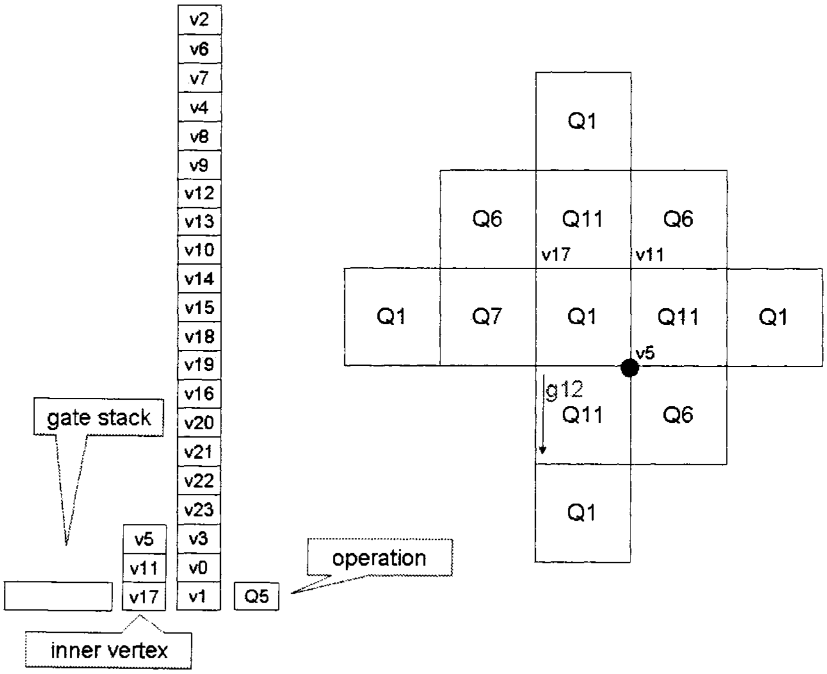


Figure 4.5: Decompression example (O)

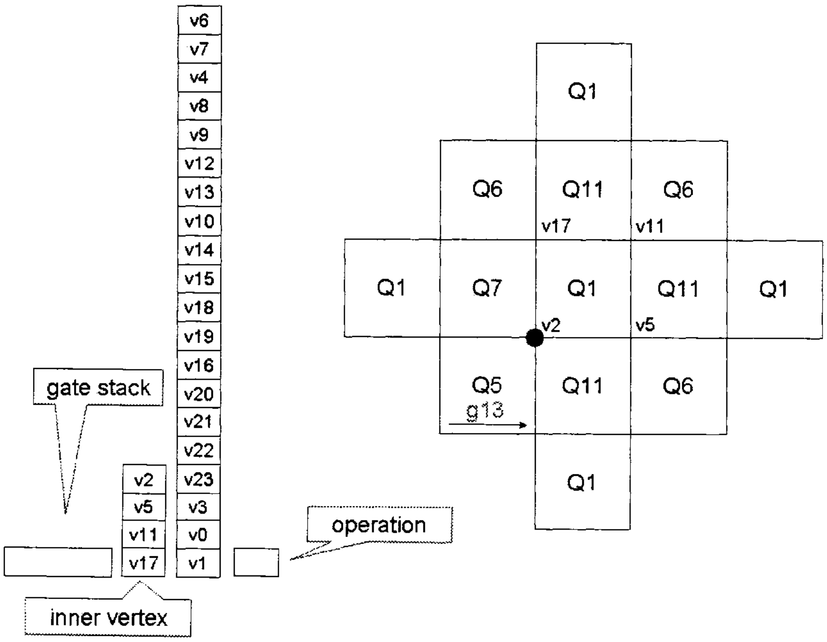


Figure 4.5: Decompression example (P)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

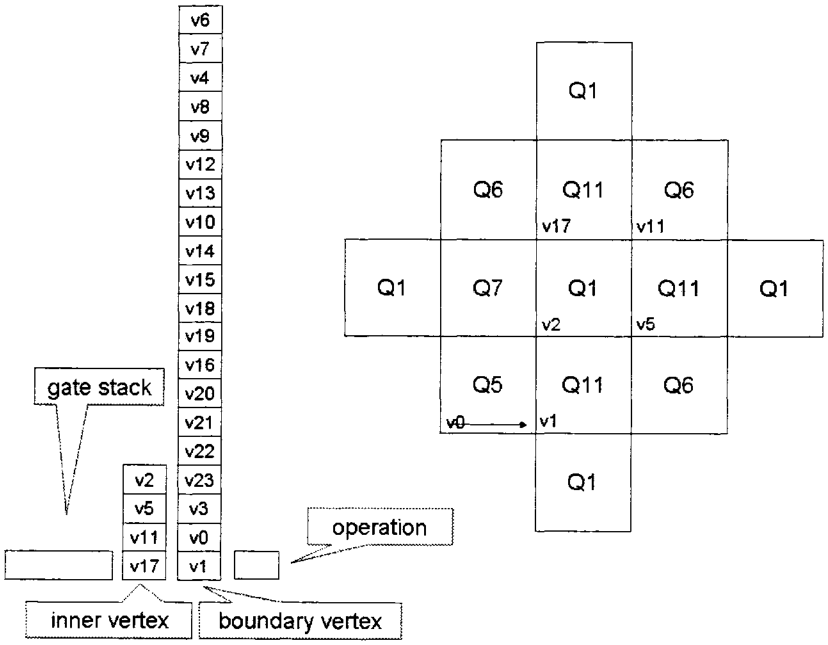


Figure 4.5: Decompression example (Q)

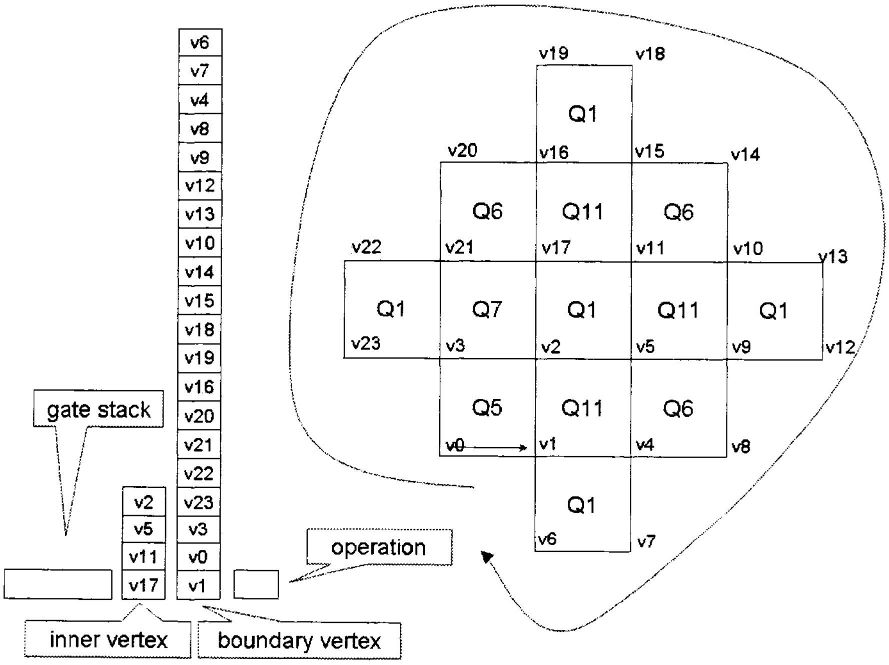


Figure 4.5: Decompression example (R)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

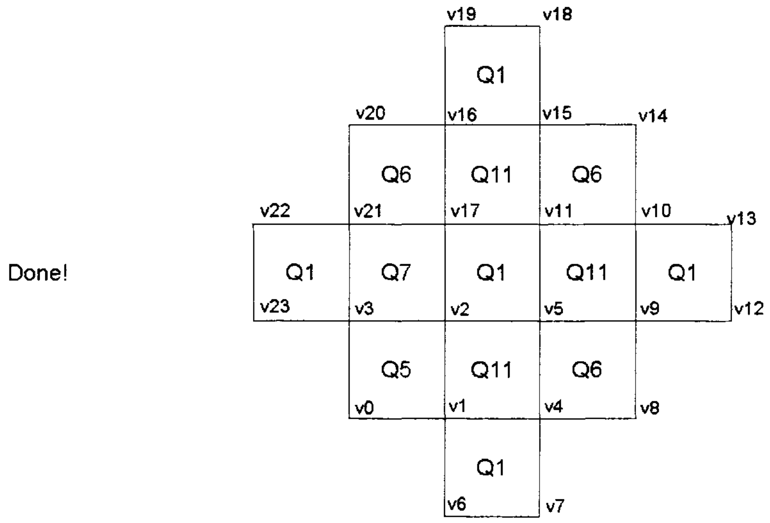


Figure 4.5: Decompression example (S)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Experimental Results

In this chapter, the implementation of compression/decompression of quad meshes is introduced in section 5.1. Then, the experimental results are presented in section 5.2.

5.1 Implementation

Implementation of Touma-Gotsman's algorithm have been discussed in chapter 3, while implementation of Edgebreaker/Spirale Reversi algorithm have been discussed in chapter 4. The source code is developed using Java under JDK 1.5. Since Java is platform-independent, the programs are portable to other operating systems.

Both implementations works for non-genus-0 mesh, but not both of them works for mesh with boundary. Our implementation of Touma-Gotsman's algorithm only works for quadmesh without boundary, while the implementation of Edgebreaker/Spirale Reversi algorithm works also for quadmesh with boundary.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Experimental results

Term "compression ratio" is used to quantify the reduction in data quantity produced by the compression algorithms. Here we declare that compression ratios are expressed as a percentage in the following form

$$Compression\ Ratio = \frac{Compressed\ Size}{Original\ Size}$$

Thus a 100MB file that compresses to 20MB would have a compression ratio of 20:100, or 20

For comparison purpose, we only select quadmeshes without boundary for experiments. We tested the programs on 11 different quadmeshes, pictures of which are shown in table 5.1.

Detailed experimental results are shown in table 5.2, table 5.3, table 5.2 and table 5.2. In all these tables, file size are measured by bytes.

For edgebreaker algorithms, we use three different encoding schemes to encode the sequence of operations: Gotsman's encoding scheme, Dr. Mukhopadhyay's encoding scheme, and Huffman code. In the tables, we use "I", "II" and "III" to represent the three encoding schemes separately.

Comparisons about the compression ratios are shown in Fig. 5.1 and Fig. 5.2.

Fig. 5.1 shows the performance of the two mesh compression algorithms comparing to the compression ratio achieved by using general file compression software GZip. From the picture we can see that using GZip we can only have a compression ratio around 23% for large meshes, while using either of the two algorithms, we can get a compression ratio less than 1% This indicates that for connectivity compression, general file compression tools (like GZip) should not be considered as first choice.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

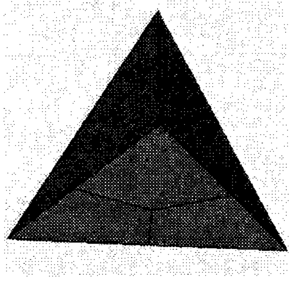
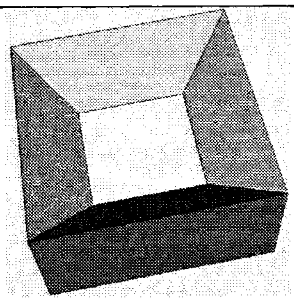
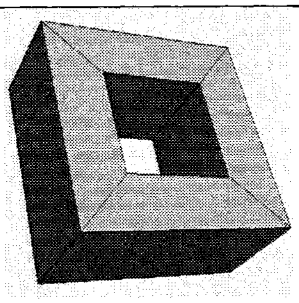
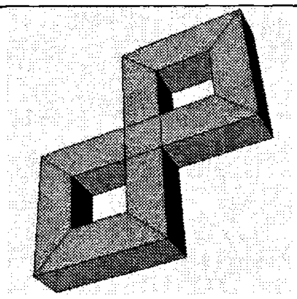
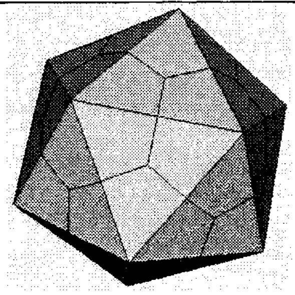
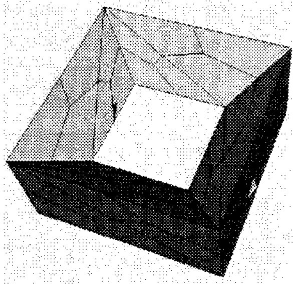
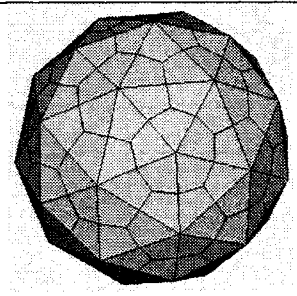
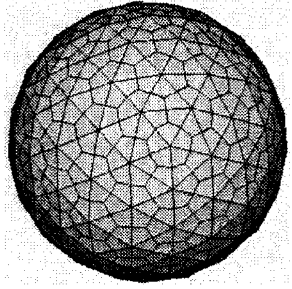
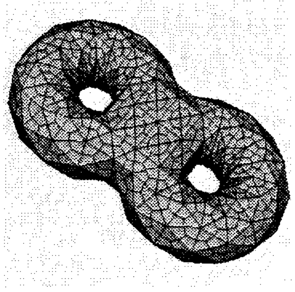
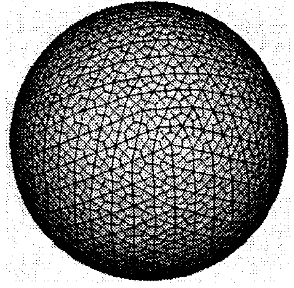
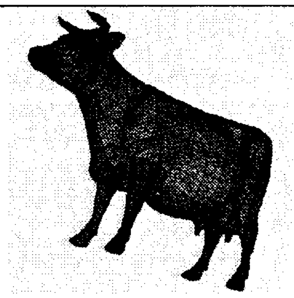
 Tetrahedron	 Torus	 16 Face Torus
 2HandTorus	 Icosahedron	 SplitTorus
 Ball240	 Ball960	 Split2HandleRnd
 Ball3840	 SplitCow	

Table 5.1: Sample quadmeshes for test

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sample Name	# of Vertices	# of Faces	Size of PLY File	Size of Command File	Size of Face File	Size of Connectivity
tetrahedron	14	12	884	58	149	2
torus	12	12	654	75	141	4
16facetorus	16	16	842	91	201	5
2handtorus	32	34	1510	193	467	13
icosahedron	62	60	3387	250	851	12
SplitTorus	72	72	3249	356	1021	27
ball240	242	240	13428	971	4092	51
ball960	962	960	54530	3864	17594	205
Split2HandRnd	1312	1314	64086	5502	25415	326
ball3840	3842	3840	235655	15385	82755	805
SplitCow	17414	17412	895506	70704	419592	4127

Table 5.2: File size for Touma-Gotsman's algorithm

Sample Name	# of Vertices	# of Faces	Size of PLY File	Size of OE File	Size of Op-code File	Size of Face File	Size of Connectivity		
							I	II	III
tetrahedron	14	12	884	713	42	149	5	5	3
torus	12	12	654	446	40	141	6	5	4
16facetorus	16	16	842	618	55	201	7	6	5
2handtorus	32	34	1510	1732	120	467	15	13	10
icosahedron	62	60	3387	3836	225	851	24	23	13
SplitTorus	72	72	3249	3309	264	1021	30	26	21
ball240	242	240	13428	16397	910	4092	96	89	50
ball960	962	960	54530	70419	3664	17594	382	359	187
Split2HandRnd	1312	1314	64086	81986	4974	25415	527	488	282
ball3840	3842	3840	235655	308744	14693	82755	1524	1439	731
SplitCow	17414	17412	895506	1278045	66325	419592	6931	6451	3591

Table 5.3: File size for Edgebreaker algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sample Name	# of Vertices	# of Faces	Size of Connectivity			Compression Ratio of Connectivity		
			I	II	III	I	II	III
tetrahedron	14	149	5	5	3	3.36%	3.36%	2.01%
torus	12	141	6	5	4	4.26%	3.55%	2.84%
16facetorus	16	201	7	6	5	3.48%	2.99%	2.49%
2handtorus	32	467	15	13	10	3.21%	2.78%	2.14%
icosahedron	62	851	24	23	13	2.82%	2.70%	1.53%
SplitTorus	72	1021	30	26	21	2.94%	2.55%	2.06%
ball240	242	4092	96	89	50	2.35%	2.17%	1.22%
ball960	962	17594	382	359	187	2.17%	2.04%	1.06%
Split2HandRnd	1312	25415	527	488	282	2.07%	1.92%	1.11%
ball3840	3842	82755	1524	1439	731	1.84%	1.74%	0.88%
SplitCow	17414	419592	6931	6451	3591	1.65%	1.54%	0.86%

Table 5.4: Compression ratios archived by different encoding schemes for Edgebreaker algorithm

Sample Name	# of Vertices	# of Faces	Size of Face File	Size of Compressed File	Compression Ratio
tetrahedron	14	12	149	104	69.80%
torus	12	12	141	104	73.76%
16facetorus	16	16	201	121	60.20%
2handtorus	32	34	467	219	46.90%
icosahedron	62	60	851	312	36.66%
SplitTorus	72	72	1021	372	36.43%
ball240	242	240	4092	1245	30.43%
ball960	962	960	17594	4927	28.00%
Split2HandRnd	1312	1314	25415	6762	26.61%
ball3840	3842	3840	82755	21260	25.69%
SplitCow	17414	17412	419592	96472	22.99%

Table 5.5: Compression ratios archived by using GZip

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sample Name	Compression Ratio				
	Edgebreaker			Touma-Gotsman	GZip
	I	II	III		
tetrahedron	3.36%	3.36%	2.01%	1.34%	69.80%
torus	4.26%	3.55%	2.84%	2.84%	73.76%
16facetorus	3.48%	2.99%	2.49%	2.49%	60.20%
2handtorus	3.21%	2.78%	2.14%	2.78%	46.90%
icosahedron	2.82%	2.70%	1.53%	1.41%	36.66%
SplitTorus	2.94%	2.55%	2.06%	2.64%	36.43%
ball240	2.35%	2.17%	1.22%	1.25%	30.43%
ball960	2.17%	2.04%	1.06%	1.17%	28.00%
Split2HandRnd	2.07%	1.92%	1.11%	1.28%	26.61%
ball3840	1.84%	1.74%	0.88%	0.97%	25.69%
SplitCow	1.65%	1.54%	0.86%	0.98%	22.99%

Table 5.6: Compression ratios archived by different encoding schemes and GZip

Fig. 5.2 lists the performance of different encoding schemes used for the two algorithms. From this figure, we can find three useful conclusion:

- For Edgebreaker compression algorithm, Dr. Mukhopadhyay's encoding scheme is better than Gotsman's encoding scheme.
- If we apply entropy code like Huffman for both algorithms, Edgebreaker algorithm could get a lower, better compression ratio than Touma-Gotsman's algorithm for large mesh. This is an interesting results.

  When compressing large meshes, Edgebreaker algorithms could generate at most 13 different symbols, while Touma-Gotsman's algorithm could generate much more than 13 different symbols. Thus, when use Huffman code to encode the symbols, Edgebreaker algorithm generally could have lower compression ratio.
- For Edgebreaker algorithm, Huffman code have a lower compression ratio than

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

either of the other two encoding schemes.

Running the standard GZip compression utility on a compressed symbol file does not reduce its size any further. On the contrary, it only increases it. This indicates that our compression algorithms are good, and no additional "general purpose" compression techniques are applicable.

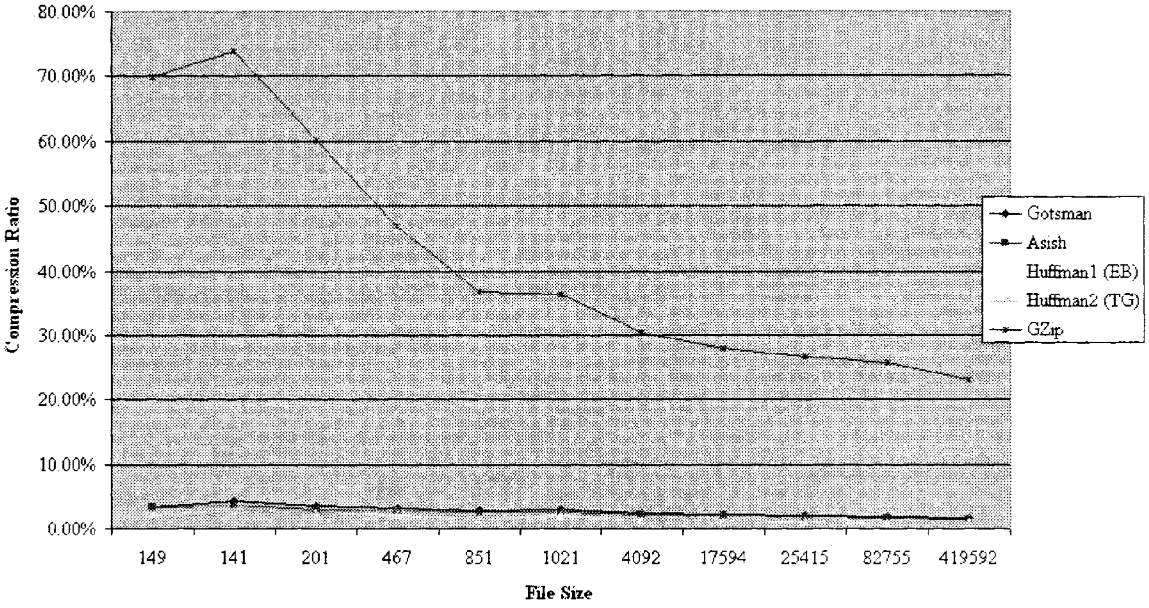


Figure 5.1: Comparison of compression ratios archived by different encoding schemes and GZip

5.3 Summary

This Chapter shows the experimental results for both algorithms. From the experimental results, we we get to useful conclusion for quadmesh compression. First, Edgebreaker archives slightly better compression ratio than Touma-Gotsman's algorithm. Secondly, Dr. Mukhopadhyay's encoding scheme results less bitrate than Gotsman's encoding scheme.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

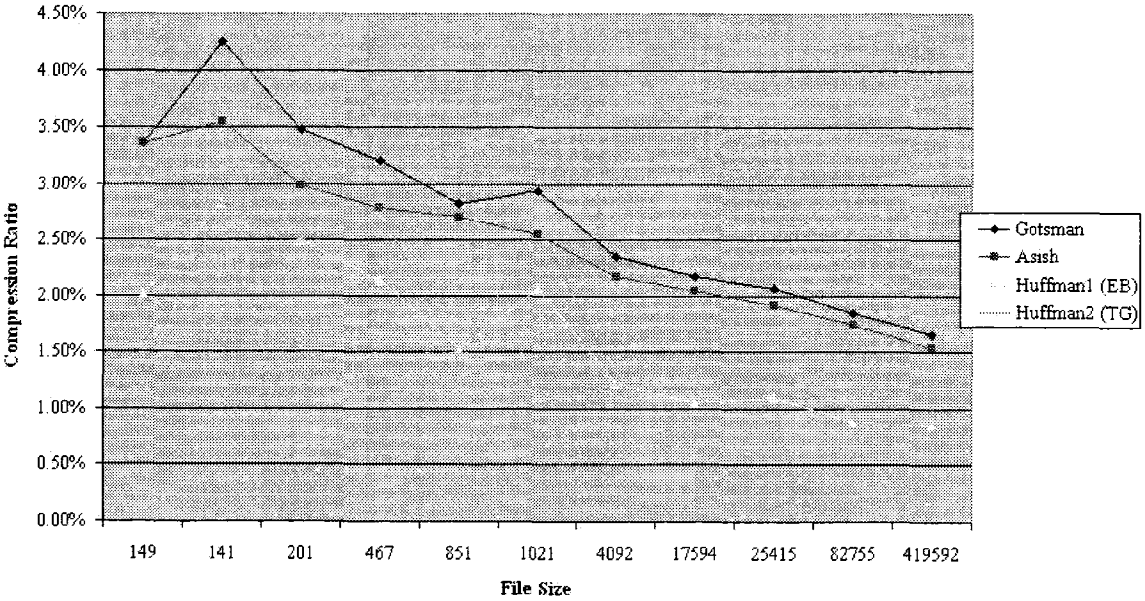


Figure 5.2: Comparison of compression ratios archived by different encoding schemes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Conclusions and Future Work

Although triangle meshes are used most frequency and studied extensively, quadrilateral meshes are used a lot in scientific applications. Traditionally, the problem of connectivity compression of quadrilateral meshes is solved by triangulating the mesh first and then compressing it using triangle compression techniques. This strategy may introduce additional cost. Some researchers have attempted to compress polygon meshes without prior triangulation. In this thesis we presented two simple linear time algorithms for connectivity compression of quadrilateral meshes, which are extended from algorithms for triangle mesh compression.

6.1 Major contributions

There are four major contributions achieved in this thesis, which have been illustrated in chapters 3, 4 and 5. The following is a summary of these contributions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Extend Touma-Gotsman's algorithm for quadmesh compression

Touma and Gotsman provided an efficient algorithm for triangle mesh compression (35). In this thesis, we extended the algorithm for quadmesh compression.

Extend Spirale Reversi algorithm for quadmesh decompression

Spirale Reversi decompression for non-triangle meshes has been mentioned by Kronrod and Gotsman (24), but they never gave a detailed explanation about the implementation. We presents the first detailed description of the Spirale Reversi decompression process for quadmeshes. Jing (21) had done some valuable works in this topic, which gave us some hints for the implementation.

Detailed comparison of the two algorithms for quadmesh compression

The compression algorithms discussed in chapter 3 and chapter 4 create a sequence of symbols, which could be encoded further by applying coding schemes. In this thesis, we encode the sequence using different coding schemes. The experiments confirmed that Dr. Mukhopadhyay's encoding schemes is better than Gotsman's encoding scheme for Edgebreaker algorithm, and Edgebreaker algorithm archives better compression ratio than Touma-Gotsman's algorithm for large mesh compression.

Portable data structures for mesh compression

We defined a set of data structures which includes all kinds of geometric objects for mesh compression. The data structures we defined can also be used in many other implementations.

6.2 Future work

In chapter 2, we said that Alliez and Desbrun (2) proposed a method to further improve the performance of Touma and Gotsman's algorithm. The method could

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

also be used for quadmesh compression.

We agree that it will be possible to achieve compression ratios significantly better than the algorithms discussed in chapter 3 and chapter 4 for mesh connectivity compression. On the other hand, we could use two-pass encoding/decoding methods, in which the connectivity of the quadmesh is first decoded, and then the coordinate decoding is started. The advantage of two-pass methods is, more connectivity information is available at the time of the coordinate decoding.

For triangle meshes containing mainly vertices of degree six, work by Szymczak et al. (31) exploits the reverseness of Spirale Reversi for efficient predictive compression of the labels. This could be extended for quadmesh decompression.

Experimental results show that Dr. Mukhopadhyay's encoding scheme always has a better performance for quadmeshes compressed using Edgebreaker algorithm. The coding schemes might be improved by using more constants, as discussed by Gumhold (17).

When implementing the Edgebreaker/Spirale Reversi algorithm for quadmesh, we used the OE data structure defined by Jing (21), which contains opposite edges information for a quad. Meanwhile, we use a different data structure called OHE when implementing Touma-Gotsman's algorithm for quadmesh. The OHE data structure is easier for understanding comparing to the OE data structure. We believe that both of the algorithms could be implemented using the OHE data structure only, thus make the implementation more understandable.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

References

[1] ISO/IEC 14496-2: Coding of Audio-Visual Objects: Visual. Technical report, 2001.

[2] P. Alliez and M. Desbrun. Valence-Driven Connectivity Encoding for 3 D Meshes. *Computer Graphics Forum*, 20(3):480–489, 2001.

[3] P. Alliez and C. Gotsman. Recent advances in compression of 3D meshes. *Proceedings of the Symposium on Multiresolution in Geometric Modeling*, 3, 2003.

[4] E.M. Arkin, M. Held, J.S.B. Mitchell, and S.S. Skiena. Hamiltonian triangulations for fast rendering. *The Visual Computer*, 12(9):429–444, 1996.

[5] C. Bajaj, V. Pascucci, and G. Zhuang. Compression and Coding of Large CAD Models. Technical report, Technical report, University of Texas, 1998.

[6] C.L. Bajaj, V. Pascucci, and G. Zhuang. Single resolution compression of arbitrary triangular meshes with properties. *Computational Geometry: Theory and Applications*, 14(1-3):167–186, 1999.

[7] R. Carey, G. Bell, and C. Marrin. ISO/IEC 14772-1: 1997 virtual reality modeling language (VRML). Technical report, The VRML Consortium Incorporated, 1997.

[8] M.M. Chow. *Optimized geometry compression for real-time rendering.* PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1997.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[9] M. Deering. Geometry compression. *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20, 1995.

[10] E. Edelsbrunner. *Geometry and Topology for Mesh Generation.* Cambridge, 2001.

[11] F. Evans, S. Skiena, and A. Varshney. Completing sequential triangulations is hard. Technical report, Technical report, Department of Computer Science, State University of New York at Stony Brook, 1996.

[12] F. Evans, S. Skiena, A. Varshney, et al. Optimizing triangle strips for fast rendering. *IEEE Visualization*, 96:319–326, 1996.

[13] C. Gotsman, S. Gumhold, and L. Kobbelt. Simplification and compression of 3d meshes. *European Summer School on Principles of Multiresolution in Geometric Modelling (PRIMUS), Munich*, 2001.

[14] Jonathon L Gross and Jay Yellen. *Graph Theory And Its Applications.* CRC Press, 1998.

[15] A. Guéziec et al. *Converting Sets of Polygons to Manifold Surfaces by Cutting and Stitching.* IBM TJ Watson Research Center, 1998.

[16] S. Gumhold. Improved cut-border machine for triangle mesh compression. *Erlangen Workshop99 on Vision, Modeling and Visualization*, 1999.

[17] S. Gumhold. New bounds on the encoding of planar triangulations. *preprint*, 2000.

[18] S. Gumhold and W. Straßer. Real time compression of triangle mesh connectivity. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 133–140, 1998.

[19] M. Isenburg. Triangle Strip Compression. *Computer Graphics Forum*, 20(2):91–101, 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[20] M. Isenburg and J. Snoeyink. Spirale Reversi: Reverse decoding of the Edgebreaker encoding. *Computational Geometry*, 20(1-2):39–52, 2001.

[21] Q. Jing. Compression and Decompression of Quadrilateral Meshes, 2003.

[22] D.W. Kahn. *Topology: An Introduction to the Point-Set and Algebraic Areas.* Dover Publications, 1995.

[23] D. King and J. Rossignac. Guaranteed 3.67 v bit encoding of planar triangle graphs. *11th Canadian Conference on Computational Geometry*, 149, 1999.

[24] B. Kronrod and C. Gotsman. Efficient Coding of Non-Triangular Mesh Connectivity. *Graphical Models*, 63(4):263–275, 2001.

[25] A. Mukhopadhyay and Q. Jing. Encoding Quadrilateral Meshes. *15th Canadian Conference on Computational Geometry (2003).*

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

[27] J. Rossignac and A. Szymczak. Wrap&Zip decompression of the connectivity of triangle meshes compressed with Edgebreaker. *Computational Geometry*, 14(1-3):119–135, 1999.

[28] M. Schindler. A fast renormalisation for arithmetic coding. In *DCC '98: Proceedings of the Conference on Data Compression*, page 572, Washington, DC, USA, 1998. IEEE Computer Society.

[29] D. Shikhare. State of the Art in Geometry Compression. *National Centre for Software Technology*, 2000.

[30] B. SPECKMANN. Easy triangle strips for TIN terrain models. *International Journal of Geographical Information Science*, 15(4):379–386, 2001.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[31] A. Szymczak, D. King, and J. Rossignac. An Edgebreaker-based efficient compression scheme for regular meshes. *Computational Geometry: Theory and Applications*, 20(1-2):53–68, 2001.

[32] G. Taubin. 3D Geometry Compression and Progressive Transmission. *Eurographics STAR report*, 3, 1999.

[33] G. TAUBIN, W.P. HORN, F. LAZARUS, and J. ROSSIGNAC. Geometry Coding and VRML. *PROCEEDINGS OF THE IEEE*, 86(6), 1998.

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

[35] C. Touma and C. Gotsman. Triangle mesh compression. *PROC GRAPHICS INTERFACE. pp. 26-34. 1998*, 1998.

[36] G. Turan. Succinct representation of graphs. *DISCRETE APPL. MATH.*, 8(3):289–294, 1984.

[37] WT Tutte. A census of planar triangulations. *Canad. J. Math*, 14(1):21–38, 1962.

[38] X. Xiang, M. Held, and J.S.B. Mitchell. Fast and effective stripification of polygonal surface models. *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 71–78, 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA AUCTORIS

Mr. Demin Yin was born in 1977 in Nanjing, Jiangsu, China.

He graduated from Sichuan University in 1999, and later worked for several Internet companies in China. In year 2003, Mr. Yin immigrated to Canada, and lived in Windsor, Ontario.

In fall 2004, Mr. Yin enrolled as master student at School of Computer Science, University of Windsor. He studied there for the next two years under Dr. Asish Mukhopadhyay's supervision.

Personal information

Name: Demin Yin

Email: yin6@uwindsor.ca

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.