A Novel Fully Progressive Lossy-to-Lossless Coder for Arbitrarily-Connected Triangle-Mesh Models of Images and Other Bivariate Functions

by

Jiacheng Guo
B.Sc., University of Electronic Science and Technology of China, 2014

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

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

A Novel Fully Progressive Lossy-to-Lossless Coder for Arbitrarily-Connected Triangle-Mesh Models of Images and Other Bivariate Functions

by

Jiacheng Guo
B.Sc., University of Electronic Science and Technology of China, 2014

Supervisory Committee

Dr. Michael D. Adams, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Pan Agathoklis, Departmental Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Michael D. Adams, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Pan Agathoklis, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

A new progressive lossy-to-lossless coding method for arbitrarily-connected triangle mesh models of bivariate functions is proposed. The algorithm employs a novel representation of a mesh dataset called a bivariate-function description (BFD) tree, and codes the tree in an efficient manner. The proposed coder yields a particularly compact description of the mesh connectivity by only coding the constrained edges that are not locally preferred Delaunay (locally PD).

Experimental results show our method to be vastly superior to previously-proposed coding frameworks for both lossless and progressive coding performance. For lossless coding performance, the proposed method produces the coded bitstreams that are 27.3% and 68.1% smaller than those generated by the Edgebreaker and Wavemesh methods, respectively. The progressive coding performance is measured in terms of the PSNR of function reconstructions generated from the meshes decoded at intermediate stages. The experimental results show that the function approximations obtained with the proposed approach are vastly superior to those yielded with the image tree (IT) method, the scattered data coding (SDC) method, the average-difference image tree (ADIT) method, and the Wavemesh method with an average improvement of 4.70 dB, 10.06 dB, 2.92 dB, and 10.19 dB in PSNR, respectively.

The proposed coding approach can also be combined with a mesh generator to form a highly effective mesh-based image coding system, which is evaluated by comparing to the popular JPEG 2000 codec for images that are nearly piecewise smooth. The images are compressed with the mesh-based image coder and the JPEG 2000 codec at the fixed compression rates and the quality of the resulting reconstructions

are measured in terms of PSNR. The images obtained with our method are shown to have a better quality than those produced by the JPEG 2000 codec, with an average improvement of 3.46 dB.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

This thesis would never have been finished without the help from numerous people. I would like to take this chance to thank:

My supervisor Dr. Michael Adams . Thank you for spending much time teaching me C++ and all the other knowledge related to my research project. You have always been very patient whenever I ask you questions and you always provide me with detailed answers. Besides, thank you for saving me a huge amount of time by helping me with the interface design of the software for my research. I am also grateful for all the instructions you gave me regarding the academic writing, this thesis would never be written without your guidance.

My committee member Pan Agathoklis. Thank you for being my committee member and reviewing my thesis.

My course instructors. I would like to thank all my instructors: Dr. Wu-Sheng Lu, Dr. Alexandra Branzan Albu, Dr. Michael Adams, Dr. Sue Whitesides, and Dr. Mihai Sima. Thank you for offering me those impressive lectures.

Other students in my research group I would like to thank Dan Han, Yue Tang, Xiao Feng, Jun Luo, Yue Fang, Ali, and all other students in my research group. Thank you for helping me during my study and research. I also really appreciate all the group meetings you presented since I learned a lot from them. It is my pleasure to be a teammate with you.

My friends I would like to thank my friends: Jun Luo, Yue Fang, Zhuo-Li Xiao, Kasem, Yukinoshita, and all other friends. Thank you for supporting me when life is tough. The time we spent together will never be forgotten.

My family I would like to thank my parents Wei Guo and Fei Wu. Thank you for your love and supporting my study.

DEDICATION

To my family

Chapter 1

Introduction

1.1 Mesh Modelling and Mesh Coding of Bivariate Functions

Bivariate functions are of great interest to a wide range of scientific applications, such as digital elevation maps in geographic information systems (GIS), images representation in signal processing, and math functions in surface modelling. Nonuniform content-adaptive sampling has proven to highly beneficial for many types of bivariate functions.

One very popular class of representation for bivariate functions that allows for nonuniform sampling is the 2.5-dimensional (2.5-D) triangle mesh. An example of a 2.5-D triangle mesh is shown in Figure 1.1. The original bivariate function is shown in Figure 1.1(a). The mesh is constructed by partitioning the domain of the function to be represented into a set of nonoverlapping triangles, where the vertices of the triangles correspond to the sample points. Then, an approximating function is defined over each triangle to yield a model for the original function over its entire domain, as shown in Figure 1.1(b).

A great many choices are possible for the connectivity of the triangulation used to partition the function domain. At one extreme is the Delaunay triangulation [11], which is (up to degeneracies) uniquely determined from the sample points. At the other extreme, the triangulation connectivity is chosen arbitrarily in a manner dependent on the underlying dataset, leading to what is commonly known as a data-dependent triangulation [16, 15, 24, 25, 34, 20, 21, 17]. For most functions, considerably more accurate representations can be achieved by allowing for arbitrary connec-

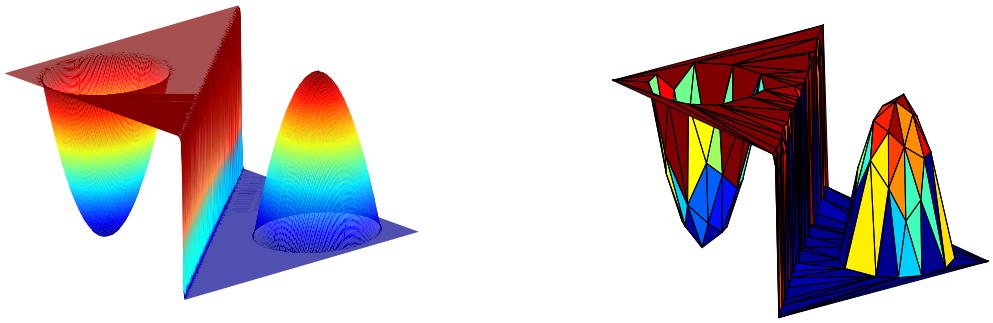
tivity [26]. Consequently, meshes with arbitrary connectivity are of great practical interest. In the discussion above, the "2.5-D" qualifier for triangle mesh refers to the fact that the surface associated with the mesh is a function defined on the plane (or a subset thereof). In contrast, one can also speak of a 3-dimensional (3-D) triangle mesh, which represents a true 2-manifold embedded in 3-D space and is used, for example, in geometric modelling. A 2.5-D mesh is much more constrained in its behavior than its 3-D counterpart, however. To better illustrate the difference between a 2.5-D and a 3-D mesh, examples of a 2.5-D and a 3-D dataset are shown in Figure 1.2(a) and Figure 1.2(b) respectively. The mesh in Figure 1.2(a) can be described as a bivariate function using the equation $z = f(x, y)$, since no vertical line perpendicular to the xy-plane intersects the surface at more than one point. In contrast, the mesh in Figure 1.2(b) can not be described directly as a bivariate function, as there exist some vertical lines perpendicular to the xy-plane (e.g., z-axis) that intersect the surface at multiple points.

As 2.5-D meshes find use in a growing number of applications, techniques for efficiently coding such datasets for storage and communication are becoming increasingly important. Moreover, many applications strongly favor coding methods that offer progressive lossy-to-lossless coding functionality. With such functionality, the decoder need not wait for the entire coded bitstream to be received before decoding. Instead, decoding can commence after having received only a very small fraction of the coded bitstream. Then, as more of the coded bitstream is received, progressively better approximations of the coded dataset are obtained, until finally lossless reproduction is achieved after the entire coded bitstream has been decoded. Motivated by the above, we focus our attention herein on the problem of progressive lossy-to-lossless coding of 2.5-D meshes.

1.2 Historical Perspective

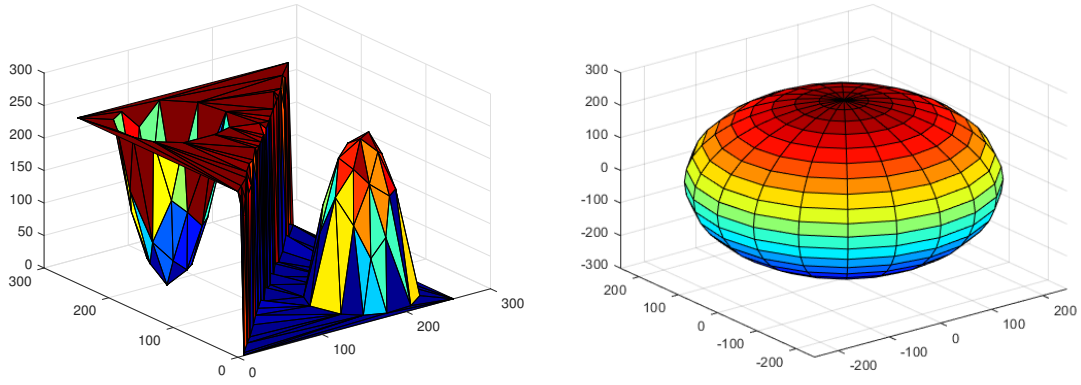
Since a 2.5-D mesh is a special case of 3-D mesh, techniques for coding 3-D triangle meshes could, in principle, be used to code 2.5-D triangle meshes. Over the years, 3-D mesh coding methods have been studied extensively in the literature. Two excellent surveys of such methods can be found in [23] and [22]. Of the various methods proposed to date, two very well-known ones with publically available software implementations are Edgebreaker and Wavemesh.

Wavemesh [32, 31] is a lossy to lossless progressive coder that is based on wavelets



(a) (b)

Figure 1.1: An example of a 2.5-D triangle mesh model. (a) the original bivariate function and (b) a 2.5-D triangle mesh of the function.



(a) (b)

Figure 1.2: Examples of mesh models. (a) a 2.5-D mesh model and (b) a 3-D mesh model.

for irregular 3-D meshes. A mesh is first simplified according to a subdivision scheme. Each face is subdivided into two, three, or four faces, or remains unchanged. After simplification, the method builds a hierarchical relationship between the original mesh and the simplified one. Therefore, the information of the original mesh can be approximated by applying the wavelet decomposition.

Edgebreaker [27, 28] is a single-rate (i.e., non-progressive) coder for compressing 3D triangle meshes. The Edgebreaker method is based on the triangle-traversal approach. At each step, the coder encodes the topological relation between the current triangle and the boundary of the remaining part of the mesh. The decoder performs the same traversal to travel the mesh from one triangle to an adjacent one. An implementation of the Edgebreaker method can be found in [28].

Unfortunately, using a 3-D mesh coder for a 2.5-D dataset is far from ideal. This is due to the fact that 2.5-D meshes are much more constrained in nature than their 3-D counterparts, and a 3-D mesh coder is unable to take advantage of this, leading to inefficient coding.

Compared to the 3-D case, relatively little attention has been given to the problem of coding 2.5-D meshes, with only very limited work on progressive 2.5-D triangle mesh coders having been performed to date. Unfortunately, the most effective progressive coders that have been proposed in the literature cannot handle meshes with arbitrary connectivity. That is, such coders do not code connectivity information at all, and instead presume that the connectivity is known through some other means (e.g., by assuming Delaunay connectivity). Of the few progressive 2.5-D mesh coders in the literature, the **scattered data coding (SDC)** method [12], **image tree (IT)** method [5], and **average-difference image tree** (ADIT) method [8] have proven to be highly effective.

The SDC method applies a technique called adaptive thinning, which is a recursive point removal scheme that works with decremental Delaunay triangulations. The adaptive thinning is used to obtain a scattered set of most significant pixels. Then, the information of those pixels is coded by a hierarchical coding scheme, which works with recursive subdivisions of octree cells.

The IT method is based on a quadtree data structure. The coder partitions the image domain recursively along with an iterative sample value averaging process. The ADIT method employs another tree-based representation of the 2.5-D triangle mesh, called the average-difference image tree, which shares some similarities with the image tree proposed in IT method. The main difference is that the ADIT method uses a

completely different approach to capture the function values of the sample points. Due to this variation, the progressive coding performance of the ADIT method is vastly superior to that of the IT method.

None of the above methods, however, can code meshes with arbitrary connectivity. In fact, the author is not aware of any fully progressive lossy-to-lossless coders for 2.5-D meshes in the current literature that can handle arbitrary connectivity.

1.3 Overview and Contributions of the Thesis

This thesis is concerned with addressing the problem of efficiently coding 2.5-D triangle meshes with arbitrary connectivity for storage and communication. The main contribution of this thesis is the proposal of a new progressive lossy-to-lossless coding scheme that codes 2.5-D triangle mesh models of images and other bivariate functions. A novel representation of a 2.5-D mesh dataset called a bivariate-function description (BFD) tree is developed. The BFD tree captures all of the information required to characterize the mesh, and more importantly, this data structure is particularly well suited for progressive coding. Another contribution is that the connectivity coding cost for meshes is vastly reduced by the proposed approach since the coder only codes a small subset of original edges from the mesh that is sufficient to recover the whole connectivity of the mesh.

Our framework is loosely based on ideas from the ADIT mesh coder described in [8]. Many substantial contributions have been made, however, beyond this earlier work. The most significant weakness of this earlier coder is that it does not code mesh connectivity, as it implicitly assumes the mesh connectivity to be Delaunay. Herein, we have extended this earlier coding scheme in order to code mesh connectivity. Furthermore, numerous other key improvements have been made, leading to a much more effective coder overall. For example, the manner in which information is embedded in the coded bitstream has been changed, leading to much better progressive coding performance. As experimental results will later demonstrate, these improvements allow our new coder proposed herein to significantly outperform the ADIT coder in terms of progressive coding performance.

The remainder of this thesis contains four chapters and one appendix. In what follows, we provide an overview of each of these remaining chapters/appendixes.

Chapter 2 introduces the background information necessary to understand work presented herein. The chapter starts by introducing some basic notation and terminol-

ogy, followed by some geometry concepts such as convex hull and triangulation. Two types of triangulations (namely, preferred Delaunay triangulation and constrained preferred Delaunay triangulation) are then discussed. Next, the definition of a 2.5-D triangle mesh model is formally introduced. This is followed by some background on arithmetic coding. Finally, the average-difference (AD) transform is presented.

Chapter 3 presents our proposed coding method for 2.5-D meshes. First, we introduce a newly proposed representation of the mesh called a BFD tree. The mesh dataset is first represented as a BFD tree, and then the BFD tree is coded. After that, we explain the approach of utilizing a BFD tree to handle progressive coding. Next, the pseudocode of the encoding algorithm is given, followed by the detailed descriptions of the algorithm. After that, two main procedures of the encoding process, namely, the child-configuration-edge-constraints (CCEC) coding and the detailed coefficient refinement (DCR) coding, will be discussed. Finally, the description of the decoding process is presented. Since the encoding process and decoding process have a high degree of symmetry, we focus primarily on describing the aspects of the decoder that cannot be deduced by symmetry.

Chapter 4 evaluates the performance of the proposed coding method by benchmarking it against several other 2.5-D and 3-D mesh coders. Our proposed scheme is shown to achieve a level of coding performance that is vastly superior to 3-D mesh coders for both progressive and lossless coding. For lossless coding performance, the proposed approach is compared with two well-known 3-D coders, namely Wavemesh and Edgebreaker. The experimental results show that the Edgebreaker and Wavemesh schemes produce the coded bitstreams that are 27.03% and 68.19% larger than those generated by the proposed method, on average. In terms of progressive coding performance, our method is shown to have superior performance relative to other state-of-the-art progressive 2.5-D mesh coders, often yielding function reconstructions at intermediate rates during progressive decoding that are better in terms of PSNR by 6.97 dB on average. Lastly, we also demonstrate that our coding method can be combined with a mesh generator to form a highly effective coder for lattice-sampled images. For images that are approximately piecewise smooth, our mesh-based image coder is shown to offer better coding performance than the well-known JPEG 2000 codec [18], both in terms of PSNR and subjective visual quality.

Chapter 5 concludes the thesis with a summary of our key results and some closing remarks. Some recommendations for future work are also suggested.

Appendix A describes the software that implements the proposed 2.5-D triangle

mesh coding framework. The appendix starts with a basic introduction of the software, followed by instructions of how to build and install the software. After that, a detailed description of the command-line interface for the software is given. Finally, some examples of how to use the software are also provided.

Chapter 2

Background

In this chapter, the background information necessary for the reader to understand the work in this thesis is presented. First, some notation and terminology is presented. Next, we introduce several concepts related to triangulations, and formally define the 2.5-D triangle mesh dataset. At last, arithmetic coding and the average difference transform are introduced.

2.1 Notation and Terminology

Before proceeding further, a brief digression is needed to introduce some of the notation and terminology used herein. The cardinality of the set S is denoted $|S|$. The sets of integers and real numbers are denoted as $\mathbb{Z}$ and $\mathbb{R}$, respectively. The following notation is used to denote ranges of integers and intervals on $\mathbb{R}$:

$$[a\,..\,b] = \{x \in \mathbb{Z} : a \leq x \leq b\}, \quad [a\,..\,b) = \{x \in \mathbb{Z} : a \leq x < b\},$$
$$[a, b) = \{x \in \mathbb{R} : a \leq x < b\}, \quad \text{and} \quad [a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}.$$

For $x \in \mathbb{R}$, $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the largest integer no greater than x (i.e., the floor function) and the smallest integer no less than x (i.e., the ceiling function), respectively. As a matter of notation, a line segment with endpoints a and b is denoted $\overline{ab}$ and a triangle with vertices a, b, and c is denoted $\triangle abc$.

Given two (non-parallel) line segments p and q, we can define an arbitrary predicate isPrefDir that tests if the orientation (i.e., direction/slope) of p is preferred over that of q, where isPrefDir(p, q) is 1 if p is preferred over q and 0 otherwise. For the purposes of our work, we define such a predicate using the preferred-directions scheme

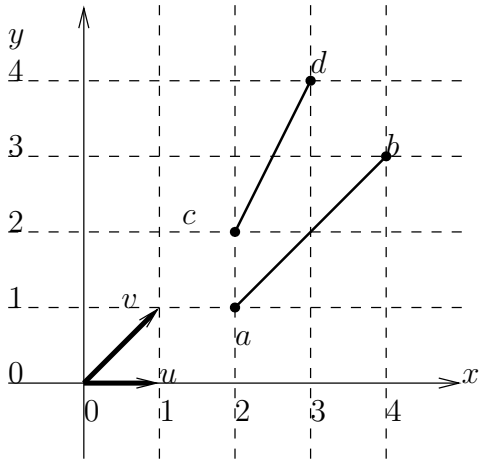


Figure 2.1: An example that illustrates the principle of the preferred-directions scheme.

of [14] as

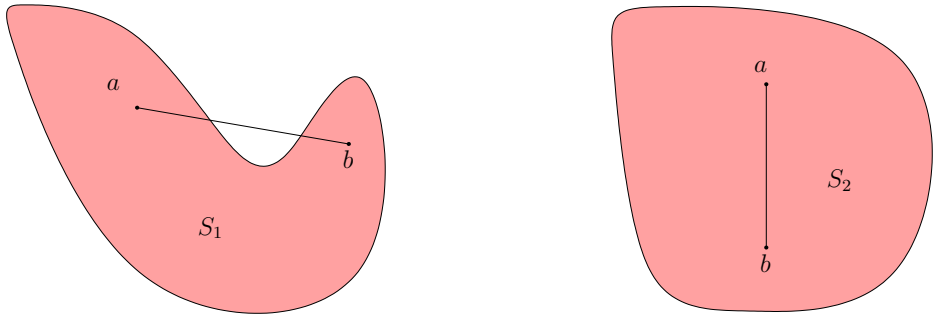
$$\text{isPrefDir}(p,q) = \begin{cases} 1 & \text{if } \theta(p,u) < \theta(q,u); \text{ or if } \theta(p,u) = \theta(q,u) \text{ and } \theta(p,v) < \theta(q,v) \\ 0 & \text{otherwise,} \end{cases}$$

where $u = (1,0)$, $v = (1,1)$, and $\theta(a,b)$ denotes the magnitude of the angle between a and b. In other words, of p and q, we prefer the line segment whose slope is closer to that of u unless both are equally close, in which case v is used in place of u in this comparison to break the tie.

To better illustrate the preferred-direction predicate, an example is shown in Figure 2.1. Two line segments $\overline{ab}$ and $\overline{cd}$ and two unit vectors u and v are plotted in this figure. To decide which line segment of $\overline{ab}$ and $\overline{cd}$ is preferred, we compare the value of $\theta(\overline{ab}, u)$ with the value of $\theta(\overline{cd}, u)$. After calculation, it follows that $\overline{ab}$ is preferred over $\overline{cd}$ (i.e. isPrefDir $\left(\overline{ab}, \overline{cd}\right) = 1$), since $(\theta(\overline{ab}, u) = 45°) < (\theta(\overline{cd}, u) = 60°)$.

2.2 Triangulations

In this section, we focus on the definitions of triangulations. The basic concept of a triangulation will be given first, followed by the definitions of two specific types of triangulation used herein, namely, preferred Delaunay (PD) triangulation and constrained preferred Delaunay (PD) triangulation. In order to introduce the concept of a triangulation, two basic concepts must first be introduced, namely, convex set and



(a) (b)

Figure 2.2: Examples of a (a) nonconvex set and (b) convex set.

convex hull.

Definition 2.1 *(Convex set). A set P of points in $\mathbb{R}^2$ is said to be **convex** if and only if for every pair of points $a, b \in P$, the line segment $\overline{ab}$ is also completely contained in P.*

To better illustrate the concept of a convex set, an example is shown in Figure 2.2. In this illustration, the set S_1 shown in Figure 2.2(a) is not convex, since there exists a line segment $\overline{ab}$ with $a, b \in S_1$ that is not completely contained in S_1. The set S_2 in Figure 2.2(b) is convex, as every line segment $\overline{ab}$ where $a, b \in P$ is always contained in S_2. Given the definition of convex set, we can now present the concept of a convex hull.

Definition 2.2 *(Convex hull). The **convex hull** of a set P of points in $\mathbb{R}^2$, denoted conv(P), is the intersection of all convex sets that contain P.*

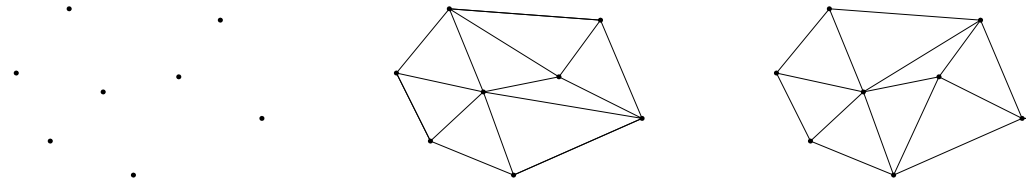
An example to illustrate the notion of a convex hull is shown in Figure 2.3. Figure 2.3(a) shows a set P of points, and the convex hull of P is shown in Figure 2.3(b). The boundary of the convex hull of a set P can also be visualized as a polygon formed by a rubber band that is stretched to enclose all the points of P. With the definition of convex hull established, the concept of a triangulation can be introduced as follows.

Definition 2.3 *(Triangulation) A **triangulation** T of the set P of points in $\mathbb{R}^2$ is a set T of non-degenerate triangles that satisfies the following conditions:*

1. *the union of all triangles in T is the convex hull of P;*
2. *the set of the vertices in all triangles of T is P; and*

(a) (b)

Figure 2.3: Convex hull example. (a) A set P of points, and (b) the convex hull of P.



(a) (b) (c)

Figure 2.4: Examples of triangulations of a set P of points. (a) A set P of points, (b) A triangulation of P, and (c) another triangulation of P.

3. the interiors of any two triangles faces in T do not intersect.

Typically, a set P of points will have (very) many possible triangulations (i.e., many possible connectivities). A set P of points is shown in Figure 2.4(a). With the same point set given, two possible triangulations are generated and illustrated in Figures 2.4(b) and 2.4(c). We can see that the connectivity of the triangulation in Figure 2.4(b) is different from that of the triangulation in Figure 2.4(c).

Next, we would like to introduce two specific types of triangulations. In order to do this, we must first describe the notions of a flippable edge and a circumcircle.

An edge e in a triangulation is said to be **flippable** if e has exactly two incident faces (i.e. is not on the triangulation boundary) and the union of these faces is a strictly convex quadrilateral. The edge e shown in Figure 2.5(a) is flippable while the edge e in Figure 2.5(b) is not flippable.

Definition 2.4 *(Circumcircle of a triangle). The circumcircle of a triangle is defined as the unique circle passing through all three vertices of the triangle.*

To illustrate the definition of a circumcircle, an example is shown in Figure 2.6. A triangle is first given and the circumcircle of the triangle is drawn in a dashed line. With the notions of circumcircle and flippable introduced, we now present the

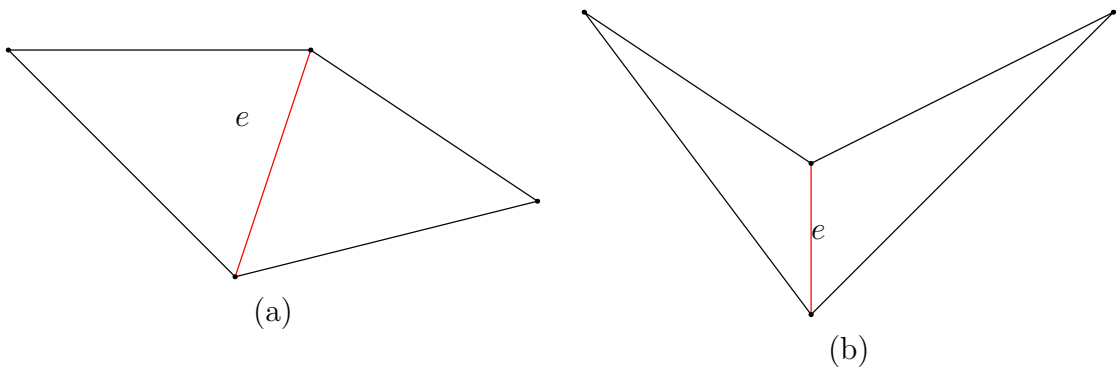


Figure 2.5: Examples of flippable and nonflippable edges. (a) An edge e that is flippable, and (b) an edge e that is not flippable.

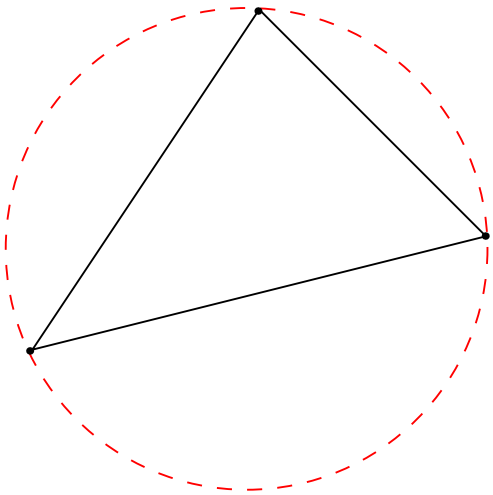


Figure 2.6: An example of circumcircle.

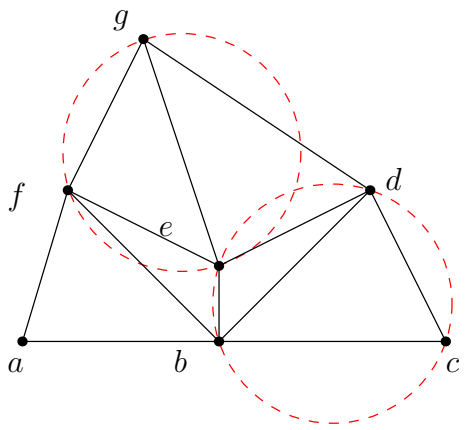


Figure 2.7: An example of a triangulation that contains four types of locally PD.

definitions of locally preferred Delaunay (locally PD) and preferred Delaunay(PD) triangulations.

Definition 2.5 *(Locally preferred Delaunay (locally PD)). An edge $\overline{ac}$ in a triangulation is said to be* ***locally preferred Delaunay (locally PD)*** *if $\overline{ac}$ is not flippable; or $\overline{ac}$ is flippable and it has two incident faces $\triangle abc$ and $\triangle acd$, and either:*

1. *d is outside the circumcircle of $\triangle abc$; or*
2. *d is on the circumcircle of $\triangle abc$ and* isPrefDir $\left(\overline{ac}, \overline{bd}\right) \neq 0$ *(i.e., $\overline{ac}$ is preferred over $\overline{bd}$).*

Definition 2.6 *(Preferred Delaunay (PD) triangulation). The* ***preferred Delaunay (PD) triangulation*** *of a set P of points, denoted* PDT(P)*, is a triangulation for which each of its edges is locally PD.*

To better illustrate the definition of locally PD, an example is given in Figure 2.7. The edge $\overline{af}$ in Figure 2.7 is locally PD since it is on the triangulation boundary thus not flippable. The edge $\overline{be}$ is also locally PD as $\overline{be}$ is not flippable. Moreover, the edge $\overline{ge}$ is locally PD since d is outside the circumcircle of $\triangle gef$. Finally, we consider the edge $\overline{bd}$. As the point c is on the circumcircle of the face $\triangle bed$, the predicate defined in (2.1) is used to determine the value of isPrefDir $\left(\overline{bd}, \overline{ce}\right)$. The edge $\overline{bd}$ is locally PD as isPrefDir $\left(\overline{bd}, \overline{ce}\right) \neq 0$ (i.e., $\overline{bd}$ is preferred over $\overline{ce}$).

Often, a triangulation may be desired that contains certain prescribed edges, known as **constrained edges**. Based on the definition of a PD triangulation, we now present the concept of a **constrained PD triangulation**.

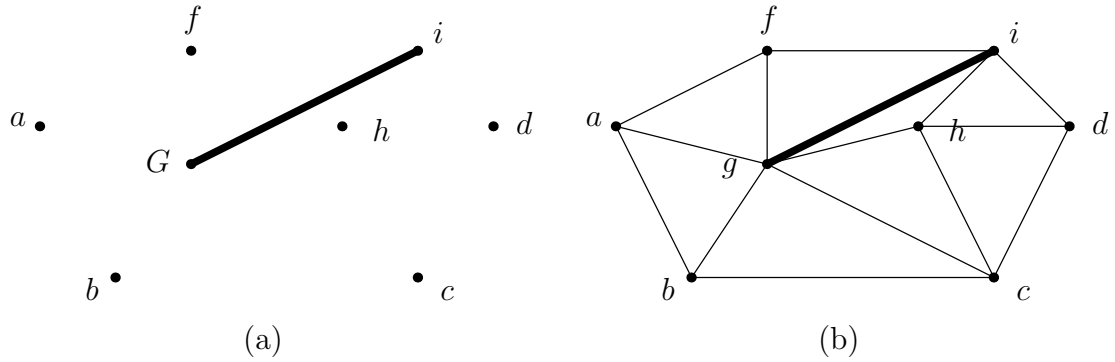


Figure 2.8: Constrained PD triangulation example. (a) A set P of points (where $P = \{a, b, c, d, e, f, g, h, i\}$) and a set E of one segment (where $E = \{\overline{gi}\}$), and (b) the constrained PD triangulation of (P, E), with the circumcircles of triangles in T drawn using dashed lines.

Definition 2.7 *(Constrained PD triangulation). The* ***constrained PD triangulation*** *of a set P of points with the set E of constrained edges, denoted* $\mathrm{CPDT}(P, E)$*, is a triangulation in which each edge is either in E (i.e., constrained) or locally PD.*

To better illustrate the definition of constrained PD triangulation, we consider the example shown in Figure 2.8. Figure 2.8(a) shows a set P of points and a set E of constrained edges, and Figure 2.8(b) demonstrates the corresponding constrained PD triangulation $\mathrm{CPDT}(P, E)$, with the constrained edge drawn with a thick line.

In some sense, a constrained PD triangulation is as close as possible to being a PD triangulation, subject to the constraint that the former must contain certain prescribed (i.e., constrained) edges. For any given P and E, $\mathrm{CPDT}(P, E)$ is always uniquely determined from only P and E [14].

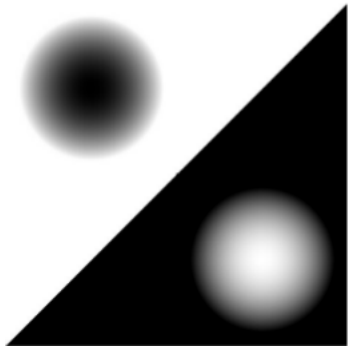
Suppose that we are given a set P of points and an arbitrary triangulation T of P. Let E denote the set of edges in T that are not locally PD. Then, it trivially follows that $\mathrm{CPDT}(P, E)$ is a triangulation with identical connectivity to T. Furthermore, it can be shown [13] that E is the minimal set such that $\mathrm{CPDT}(P, E)$ has the same connectivity as T. In this sense, the connectivity of any triangulation can be completely characterized by a set of edge constraints (through a constrained PD triangulation). As will be seen later, this fact is exploited in our method for triangulation connectivity coding.

2.3 2.5-D Triangle Mesh Models

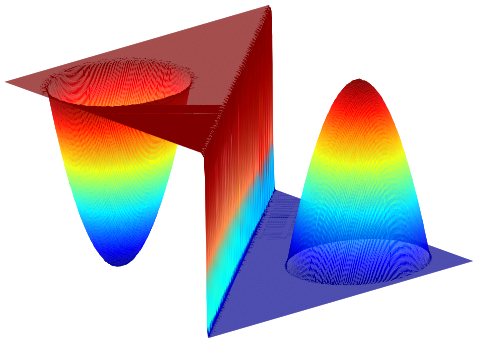
As mentioned earlier, our work addresses the problem of efficiently coding 2.5-D meshes. At this point, we would like to formalize exactly what constitutes such a dataset. In the context of our work, a 2.5-D mesh is a dataset that consists of: 1) a set $P = \{p_i\}$ of sample points with integer coordinates (i.e., $p_i \in \mathbb{Z}^2$); 2) a triangulation T of P, which specifies the connectivity of the sample points; and 3) a set of integer values $Z = \{f_i\}$, where f_i corresponds to the (integer) function value at the sample point p_i. In the case that the function domain is an iso-oriented (i.e., an axis-aligned) rectangle, the extreme convex hull points of P would be the four corners of the function-domain bounding box. It is worth noting, however, that the function-domain need not be an iso-oriented rectangle. It can be any convex polygon.

Given the information for a mesh dataset, a bivariate function can be constructed. In practice, this is normally done by constructing a function over each face of the triangulation T and then combining these functions to obtain a function that is defined over the entire convex hull of P (i.e., the region covered by the triangulation T). Furthermore, the approximating function for each face is most often produced using straightforward linear interpolation. As will be seen later, however, our proposed coding method makes no assumptions about the particular manner in which a function is constructed from the mesh dataset. So, as far as mesh coding itself is concerned, the function-construction process is not important. This said, however, some of our experiments presented later require generating a function from the decoded dataset, in which case a specific choice for the function-construction procedure must be made. In such cases, we simply choose linear interpolation, since (as mentioned above) it is the most common approach.

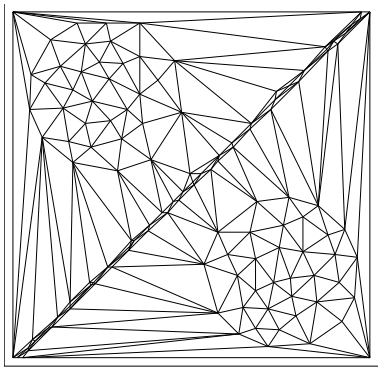
The mesh modelling process, is illustrated in Figure 2.9. Figure 2.9(a) shows the original bivariate function, and Figure 2.9(b) shows the function represented as a surface where brightness corresponds to the height of the surface above the plane. With the bivariate function given, a set of sample points is chosen and used to construct a triangulation, as illustrated in Figure 2.9(c). Next, an approximating function is defined over each triangle face using linear interpolation to yield a model that approximates the original function over its entire domain, as shown in Figure 2.9(d).



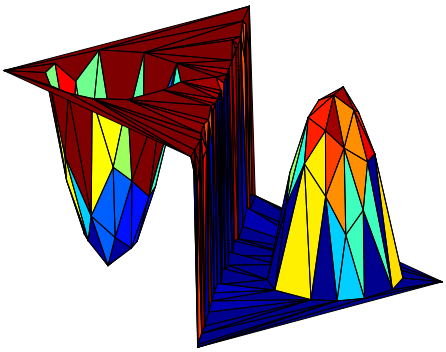
(a)



(b)



(c)



(d)

Figure 2.9: An example of a mesh model of an image. (a) the original bivariate image, (b) the function modelled as surface, (c) a triangulation of the function, and (d) the resulting 2.5-D triangle mesh model.

2.4 Arithmetic Coding

Arithmetic coding is one of the most popular entropy coding schemes used in data compression. The source message is represented as an interval $[0, 1)$ by the arithmetic coding. The interval is narrowed to be shorter as the source message becomes longer, leading to the increase of bits used to represent the interval. Binary arithmetic coding is a special case of the arithmetic coding. The binary arithmetic coding scheme only codes two types of symbols: 0 and 1. In what follows, the main focus is introducing the concept of the binary arithmetic coding.

The encoding procedure is presented as follows. At the beginning of the encoding process, the interval is initialized to $[0, 1)$. At each step of the encoding process, the encoder receives a symbol and the current interval is divided by the encoder into two sub-intervals, each representing a fraction of the current interval proportional to the probability of the received symbol. Next, the current interval is updated to one of the sub-intervals that corresponds to the symbol. Let $[a_1, a_2)$ denote the current interval before encoding the next symbol and let $[b_1, b_2)$ denote the interval that corresponds to the probability distribution of the next symbol. Then the new interval $[c_1, c_2)$ is calculated as given by:

$$\begin{cases} c_1 = a_1 + (a_2 - a_1) \times b_1 \\ c_2 = a_2 - (a_2 - a_1) \times (1 - b_2). \end{cases} \tag{2.1}$$

The encoder keeps updating the current interval based on the received symbols and after all symbols have been encoded, the resulting interval clearly identifies the sequence of symbols that generated it.

The decoding process also starts with the interval $[0, 1)$. The decoder determines the value of each symbol based on the message received from encoder and updates the interval based on value of that symbol. Let $[a_1, a_2)$ denote the current interval and let $[b_1, b_2)$ denote the interval that corresponds to the probability distribution of the decoded symbol. The new interval $[c_1, c_2)$ can also be calculated by (2.1). The decoder also needs to know where the bit stream ends so it can terminate at the appropriate point. The arithmetic coder is said to be **context based** if the probability distribution of symbols is chosen based on the contextual information, rather than always being fixed. Moreover, the arithmetic coding is called **adaptive** if the probability of each symbol is adjusted based on the symbols that have been

Table 2.1: Probability distribution of the symbols {0,1}

Symbol	Probability	Interval
0	0.7	[0.0,0.7)
1	0.3	[0.7,1.0)

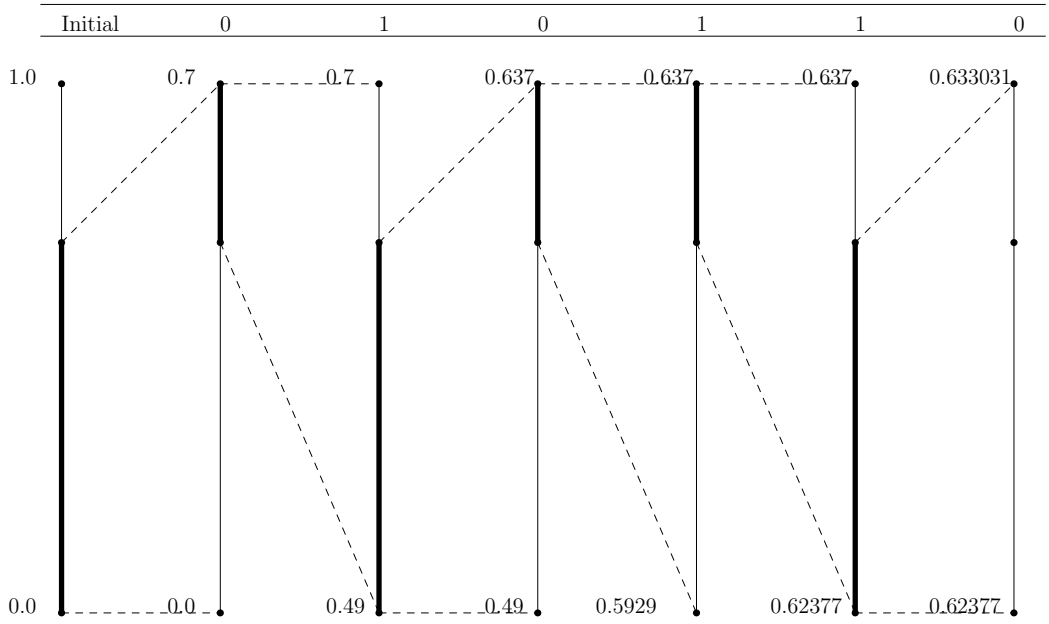


Figure 2.10: Graphic representation of the arithmetic encoding process.

coded.

In what follows, examples are shown to demonstrate the (binary) arithmetic encoding and decoding processes. The source message $\{0, 1, 0, 1, 1, 0\}$ contains six symbols selected from the binary alphabet $\{0, 1\}$. Table 2.1 shows the probability distribution of the symbols.

We first present how encoder works. Figure 2.10 is presented to help illustrate the updates of the interval in the encoding process. The encoder first initializes the interval to $[0, 1)$. The first symbol received by encoder is 0, which corresponds to the interval $[0, 0.7)$ as shown in Table 2.1. By following (2.1), the current interval $[0, 1)$ is updated to a sub-interval $[0, 0.7)$. Similarly, the second symbol 1 narrows the interval from $[0.0, 0.7)$ to $[0.49, 0.70)$ using the same equation. Repeating the same approach, the following symbols $0, 1, 1, 0$ are encoded by the encoder one after another. When all symbols are encoded, the final interval is updated to $[0.62377, 0.633031)$, which is sufficient to recover the source message. As it is not necessary for encoder to code both sides of the interval, only the number 0.63477 (lower bound) is selected and

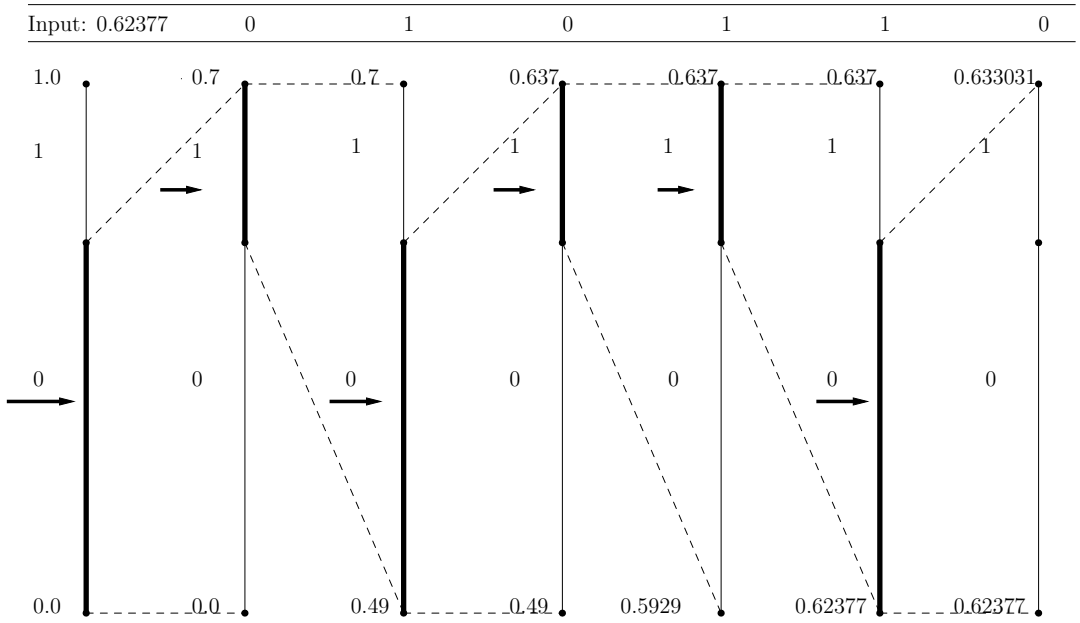


Figure 2.11: Graphic representation of the arithmetic decoding process.

encoded.

Now we consider the decoding process. The updates of the interval is illustrated in Figure 2.11. The decoder also starts with the interval $[0, 1)$. The transmitted number 0.62377 is in the range $[0, 0.7)$, which is same as the interval of the symbol 0 in the initial range. Consequently, the decoder decodes a 0 for the first symbol and the interval is updated to $[0, 0.7)$ based on (2.1). As the number 0.62377 is located at the range $[0.49, 0.7)$, which corresponds to the interval of symbol 1 relative to the current interval $[0, 0.7)$, the second symbol decoded is 1. The symbol 1 narrows the current interval from $[0, 0.7)$ to $[0.49, 0.7)$ by following (2.1). Repeating the same approach, the remaining symbols $0, 1, 1, 0$ are decoded one after another, and the decoding process ends after the sixth symbol is successfully decoded.

A binary arithmetic coder is only capable of coding binary symbols. In some real world applications, certain **binarization** schemes must be applied to convert a non-binary symbol to a sequence of binary symbols for coding.

2.5 Average-difference Transform (ADT)

In anticipation of what comes later, we introduce a simple transformation known as the **average-difference transform (ADT)**. The ADT, denoted ADT, is the

mapping from $\mathbb{Z}^2$ to $\mathbb{Z}^2$ given by

$$\mathrm{ADT}\{(x_0, x_1)\} = (f_{\mathsf{avg}}(x_0, x_1), f_{\mathsf{diff}}(x_0, x_1)), \tag{2.2}$$

where $f_{\mathsf{avg}}(x_0, x_1) = \left\lfloor \frac{1}{2}(x_0 + x_1) \right\rfloor$ and $f_{\mathsf{diff}}(x_0, x_1) = x_1 - x_0$. In other words, the ADT maps a pair of integers to their approximate average and difference. Due to the form of (2.2), if x_0 and x_1 can each be represented with n bits (i.e., an n-bit integer), $f_{\mathsf{avg}}(x_0, x_1)$ and $f_{\mathsf{diff}}(x_0, x_1)$ can be represented with n and $n + 1$ bits, respectively, a fact that we make use of later. The transform computed by (2.2) is invertible, with its inverse given by

$$\mathrm{ADT}^{-1}\{(y_0, y_1)\} = \left(y_0 - \left\lfloor \tfrac{1}{2} y_1 \right\rfloor, y_0 + \left\lceil \tfrac{1}{2} y_1 \right\rceil\right). \tag{2.3}$$

Chapter 3

Proposed Mesh-Coding Method

In developing a coding scheme for 2.5-D meshes with arbitrary connectivity, a strategy for describing the mesh connectivity must be chosen. The most straightforward way to characterize the mesh connectivity would be to view it directly as a graph. In our work, however, a different approach is taken. Instead of viewing the mesh connectivity as a graph, we describe the connectivity as a set of edge constraints for a constrained PD triangulation. For a given mesh to be coded with the set P of sample points, we select the minimal set E of edge constraints such that CPDT(P, E) yields a triangulation with the same connectivity as the given mesh. Then, E is used to convey the mesh connectivity for coding purposes. As mentioned earlier, this set E is easily determined. In particular, we choose E as the set of all edges in the mesh that are not locally PD. For non-Delaunay meshes of practical interest, the fraction of edges that are not locally PD is typically less than 25% and often significantly less for some types of datasets. Furthermore, it has been shown that this fraction cannot exceed 50% for any mesh [13]. Thus, this strategy yields a particularly compact description of the mesh connectivity. Moreover, the compactness of this description increases (i.e., $|E|$ decreases) as the mesh connectivity more closely approaches PD connectivity (where, in the case of PD connectivity, $|E| = 0$). This allows for a very low connectivity coding cost for meshes with Delaunay connectivity.

With the above strategy, the encoder determines the set E from the mesh to be coded and encodes this information in the coded bitstream; the decoder then recovers the correct connectivity by constructing a constrained PD triangulation with the decoded edge constraints. Essentially, this transforms the problem of coding a 2.5-D mesh into one of coding a constrained PD triangulation with a function value (i.e., f_i value) for each vertex.

3.1 Bivariate-Function Description (BFD) Tree

Our coding method, to be introduced shortly, employs a novel representation of a 2.5-D mesh dataset proposed herein called a bivariate-function description (BFD) tree. With our coding approach, a given mesh dataset is first represented as a BFD tree, and then this BFD tree is coded. To begin, we first introduce the BFD-tree representation of a mesh. Then, we proceed to describe how the information in a BFD tree can be efficiently coded.

Let us consider a mesh dataset having: 1) the set $P = \{p_i\}$ of sample points; 2) the set $F = \{f_i\}$ of function values, each with a sample precision ρ of bits/sample; 3) the triangulation T of P; and 4) the set E of edges in T that are not locally PD (i.e., E is the minimal set needed to ensure CPDT(P, E) has the same connectivity as T). Without loss of generality, we assume the sample points $\{p_i\}$ to be contained in a rectangular region of the form $B = [0, W) \times [0, H)$ for some positive integer constants W and H. (This assumption can always be made to hold by adding an appropriate constant bias to the coordinates of the sample points.) As a matter of terminology, the padded bounding box B' (of a mesh dataset) is defined as the rectangular region $B' = [0, 2^D) \times [0, 2^D)$ where D is the smallest positive integer such that the $B \subset B'$ (i.e., B' is the smallest square region with a power-of-two width/height that contains B). A **cell** is a rectangular region of the form $C = [x_0, x_1) \times [y_0, y_1)$, where $x_0, x_1, y_0, y_1 \in \mathbb{Z}$ and the width (i.e., $x_1 - x_0$) and height (i.e., $y_1 - y_0$) of C are strictly positive integers and powers of two. A cell is said to be **occupied** if it contains at least one sample point (i.e., element of P). Two occupied cells C and C' are said to be **constraint connected** if there exists a sample point in C that is connected by an edge constraint to a sample point in C'. The **representative point** of a cell $C = [x_0, x_1) \times [y_0, y_1)$ is defined as the point $(x_{\mathsf{m}}, y_{\mathsf{m}})$, where $x_{\mathsf{m}} = \left\lfloor \frac{1}{2}(x_0 + x_1) \right\rfloor$ and $y_{\mathsf{m}} = \left\lfloor \frac{1}{2}(y_0 + y_1) \right\rfloor$. That is, the representative point of C is its (exact) centroid, except when the width or height of C is 1, in which case this representative point is on the boundary of C.

A BFD tree is binary tree that captures all of the information needed to completely characterize a 2.5-D mesh dataset (namely, P, F, T, and E). Such a tree is associated with a recursive binary partitioning of B' into cells, similar to the partitioning associated with a k-d tree [10]. Each node in the tree has a corresponding cell. As a matter of terminology, two leaf nodes in a BFD tree are said to be **constraint-connected neighbours** (or, equivalently, **constraint connected**) if the cell of one node is constraint connected to the cell of the other node. Each internal node in a

BFD tree can have either one or two children, and each (internal or leaf) node in the tree consists of: 1) a cell, which is always occupied; 2) an approximation coefficient, which is an approximate average of the function values f_i taken over all of the sample points in the node's cell; 3) in the case of an internal node with exactly two children, a detail coefficient, which specifies the difference in the approximation coefficients of the node's two children; and 4) in the case of a leaf node, the node's set of constraint-connected neighbours. (i.e., the set containing each other leaf node in the tree whose cell is constraint connected to the cell of this node). For convenience, in what follows, we denote the approximation coefficient of the root node as a_{root}.

At this point, we need to explain how to determine which nodes are present in a BFD tree and the cells of those nodes. As mentioned earlier, a BFD tree is associated with a recursive binary partitioning of B'; so, perhaps not surprisingly, this decision process is specified recursively. Since a BFD tree must contain at least one node, it always has a root node. The cell of the root node is chosen as B'. Then, we define a recursive process for adding more nodes as follows. Given a node u at level ℓ in the tree with cell $C = [x_0, x_1) \times [y_0, y_1)$, we proceed as follows to determine which children of u are present and what the cell of each child node is. Let u_0 and u_1 denote the first and second child nodes of u, each of which may or may not be present in the tree. The cell C is split into two new cells C_0 and C_1, in a manner that depends on ℓ, as follows: 1) if ℓ is even, $C_0 = [x_0, x_{\mathsf{m}}) \times [y_0, y_1)$ and $C_1 = [x_{\mathsf{m}}, x_1) \times [y_0, y_1)$, where $x_{\mathsf{m}} = \frac{1}{2}(x_0 + x_1)$ (i.e., C is split by a vertical line through its centroid to yield C_0 and C_1); 2) if ℓ is odd, $C_0 = [x_0, x_1) \times [y_0, y_{\mathsf{m}})$ and $C_1 = [x_0, x_1) \times [y_{\mathsf{m}}, y_1)$, where $y_{\mathsf{m}} = \frac{1}{2}(y_0 + y_1)$ (i.e., C is split by a horizontal line through it centroid to yield C_0 and C_1). Once C_0 and C_1 have been determined, the decision of which of the child nodes u_0, u_1 of u are present is made by recalling the invariant that a cell's node must always be occupied. In particular, for $i \in \{0, 1\}$, if node C_i is occupied, then the node u has the child u_i with cell C_i. The preceding rule for adding children (i.e., new leaf nodes) is applied recursively until each leaf node is such that its cell contains only a single point in $\mathbb{Z}^2$ (where this point corresponds to the cell's representative point). This occurs when ℓ equals $L_{\mathsf{max}} = 1 + 2D$. At level L_{max} in the tree, each (leaf) node's cell has a width and height of 1. By construction, the representative point of each leaf node's cell (at level L_{max}) is one of the sample points in the mesh. Thus, there is a one-to-one correspondence between leaf nodes in the tree and sample points (i.e., mesh vertices).

Next, we specify how the approximation and detail coefficients are defined for

nodes in a BFD tree. First, let us consider the case of determining the coefficient information for a leaf node u. Since only a node with two children has a detail coefficient, u (which has no children) does not have a detail coefficient. So, we must only specify how the approximation coefficient of u is determined. Recall that a leaf node always corresponds to a sample point in the mesh. Let p_i denote this sample point and let f_i denote the corresponding function value. We simply define the approximation coefficient a of u as $a = f_i$. Next, let us consider the case of an internal node u. There are two possibilities to consider, depending on the number of children possessed by u (which is either 1 or 2). First, we consider the case that u has exactly one child. In this case, u has no detail coefficient (for a similar reason as in the case of a leaf node above) and the approximation coefficient a of u is given by $a = a_i$, where a_i is the approximation coefficient of the child of u. Next, we consider the case that u has exactly two children. Let u_0 and u_1 denote the child nodes of u with their respective approximation coefficients a_0 and a_1. In this case, the approximation coefficient a and detail coefficient d of u are given by the ADT as $a = f_{\mathsf{avg}}(a_0, a_1)$ and $d = f_{\mathsf{diff}}(a_0, a_1)$ (where the ADT was defined earlier in (2.2)).

Due to the manner in which the ADT is defined, if each function value f_i can be represented as an n-bit integer then: 1) each approximation coefficient (including a_{root}) can be represented as an n-bit integer; and 2) each detail coefficient can be represented as an $(n+1)$-bit signed integer. We exploit this fact later in our proposed coding scheme. It is also important to note that a significant amount of redundancy exists in the approximation and detail coefficients of a BFD tree. For example, a_{root} and the set of all detail coefficients is sufficient to completely characterize all of the approximation coefficients in the tree. Therefore, in order to fully capture the coefficient information for a BFD tree, it is sufficient to code only a_{root} along with the detail coefficients.

An example of a BFD tree for a simple mesh dataset is shown in Figure 3.1. Figure 3.1(a) shows a mesh dataset with 5 sample points and a padded bounding box of $B' = [0, 4) \times [0, 4)$. Each sample point p_i is shown labelled with its corresponding function value f_i. The edges in the triangulation are shown, with each locally PD edge drawn as a thin line and each edge that is not locally PD drawn as a thick line. As can be seen from the figure, only 1 of the 8 triangulation edges is not locally PD. Figure 3.1(b) shows the BFD tree corresponding to the mesh dataset in Figure 3.1(a). Each node in the tree is labelled with its cell, approximation coefficient, and, if one exists, detail coefficient (in that order). Pairs of constraint-connected nodes

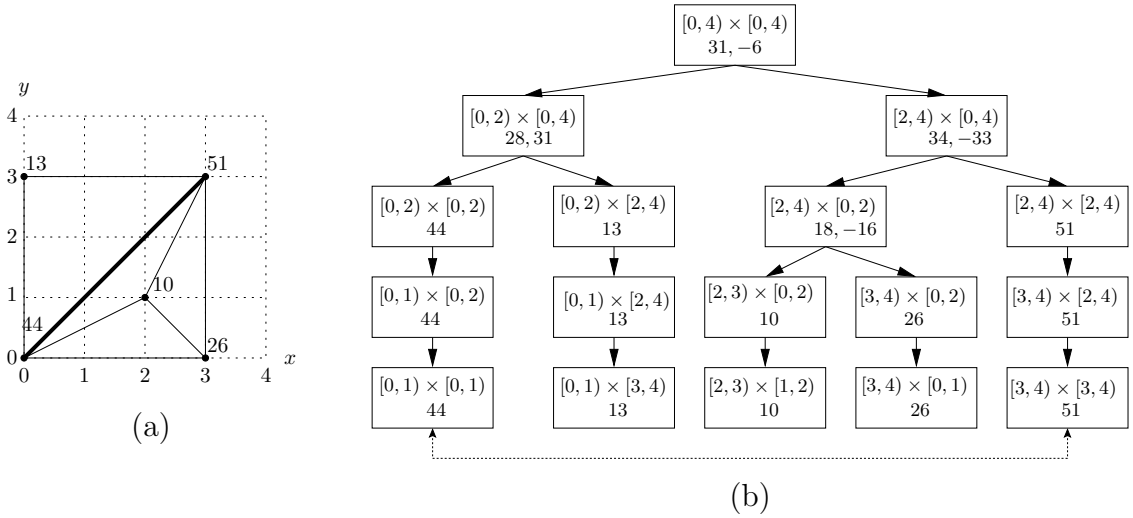


Figure 3.1: BFD tree example. (a) A 2.5-D mesh dataset (i.e., sample points, function values, and sample-point connectivity) and (b) its corresponding BFD tree.

(which must be leaf nodes) are shown connected by a dotted line. The single pair of constraint-connected nodes appearing in Figure 3.1(b) corresponds to the single edge in the mesh dataset that is not locally PD.

3.2 Progressive Coding

As it turns out, a BFD tree is particularly well suited for progressive coding. In order to understand the basic principle behind the progressive coding of such a tree, it is important to recall from earlier that, the leaf nodes of a BFD tree have a one-to-one correspondence with mesh vertices (i.e., sample points). In particular, each leaf node corresponds to a vertex positioned at the representative point of the node's cell. To progressively code a BFD tree, we code the information in the tree starting from the root node and proceeding downwards in the tree. Initially, the root approximation coefficient of the tree is included (in a header) at the start of the coded bitstream. We then code a single node at the root, which corresponds to a degenerate triangulation with a single vertex and no edge constraints. Then, we proceed to successively code how to add new leaf nodes to the tree. As each new leaf node is added, information is coded indicating how to appropriately update the constraint-connected relationships between leaf nodes. Along with the addition of new leaf nodes, the detail coefficients are also coded from the most-significant to least-significant bit position.

At any given point, a partially decoded tree can be used to obtain an approx-

imation of the original mesh dataset as follows. The vertices of the decoded mesh are given by the representative points of the leaf nodes' cells in the partially-decoded tree. The edge constraints to be used for determining the mesh connectivity are given by the constraint-connected relationships of the leaf nodes. In particular, the vertices associated with two leaf nodes are connected by an edge constraint if and only if their corresponding nodes are constraint connected. The function values corresponding to the mesh vertices are determined, through the application of the inverse ADT, using the approximation coefficient of the root node and the values of the detail coefficients decoded so far.

3.3 Encoding Algorithm

Having introduced the BFD tree representation of a mesh, we now present our proposed method for efficiently coding the information in such a tree. Our coder employs context-based adaptive binary arithmetic coding [33]. Since the encoding and decoding processes in our method have a high degree of symmetry, the decoding process can be mostly inferred from the encoding process. For this reason, in the interest of brevity, we focus primarily on describing the encoder herein, only commenting on aspects of the decoder that cannot be deduced by symmetry.

Conceptually, the encoder employs two BFD trees called the reference and current trees. The **reference tree** is an entire BFD tree for the mesh being coded. This tree is constructed at the beginning of the encoding process and is never modified subsequently. It is used only to query values at various stages in the encoding process. The **current tree** represents the part of the reference tree that has been coded so far. As far as the coding process is concerned, the tree of primary interest is the current tree, as it it holds the current coding state. The encoding algorithm employs two queues, each of which holds nodes from the current tree. The first queue, called the **splitting (S) queue**, is a priority queue. It is used only to hold leaf nodes at even levels in the current tree. The second queue, called the **refinement (R) queue**, is a first-in first-out (FIFO) queue. It is used only to hold non-leaf nodes from the current tree.

Given a mesh to be coded, the encoder proceeds as follows. First, it constructs the reference tree (i.e., the BFD tree for the mesh being coded). Then, the encoder writes a small fixed-size header to the coded bitstream containing several key BFD-tree parameters (e.g., W, H, ρ, and a_{root}) and initializes the arithmetic coding engine.

Next, the current tree is initialized to contain only a single (i.e., root) node and this node is inserted on the S queue. The encoding process then alternates between processing nodes on each of the S and R queues, with the switching between queues being controlled based on the number of binary symbols coded with the arithmetic coder. Each node removed from the S queue is processed by the **child-configuration and edge-constraint (CCEC)** coding procedure, which refines mesh vertices (i.e., sample points) and their positions and updates information on edge constraints. The CCEC coding procedure also causes nodes to be placed on the R queue. Each node removed from the R queue is processed by the **detail-coefficient refinement (DCR)** coding procedure, which refines the values of detail coefficients (i.e., function value information). The encoding process continues until both queues are empty, at which point all information in the mesh dataset has been coded.

The above encoding algorithm is described in more detail in pseudocode form in Algorithm 1. In the pseudocode, the reference and current trees are referred to as `refTree` and `curTree`, respectively. In passing, we note that, although the mesh datasets considered herein are such that the coordinates of the sample points p_i and the function values f_i are integers, real values can easily be accommodated by quantizing the original data (to obtain integer quantizer indices), adding the quantization parameters (e.g., quantizer step sizes) to the header of the coded bitstream, and then coding the quantized integer data. In order to complete the description of the encoding algorithm, we still need to specify the CCEC and DCR coding procedures and explain how the S queue priority function sQueuePri is defined. In what follows, we provide these additional details.

S queue priority function (i.e., sQueuePri*).* In Algorithm 1 above, the sQueuePri function is used (in steps 7 and 24) to calculate the priority with which a node should be inserted on the S queue. The priority of a node u is defined as $\text{sQueuePri}(u) = a(c+1)$, where a is the area of the cell of u and c is the number of constraint-connected neighbours of u. The priority function controls the order in which different regions of the mesh are refined during (progressive) coding. Herein, we have chosen sQueuePri to achieve good overall rate-distortion performance. In passing, we note that other choices are possible. For example, although not explored in our work, the priority function could be chosen to prioritize reducing the error in a particular region (or regions) in the mesh in order to provide a basic region-of-interest coding functionality. In such a case, the priority of a node could be made to depend on the position of the node's cell in relation to the region of interest.

Algorithm 1 Encoding algorithm

```
1: procedure encodeMesh
2:     define several constants as follows: sThresh = 100, dThresh = 10, and η = 3
3:     set refTree to the BFD tree of the mesh to be coded (refTree is never
       modified after being initialized in this step)
4:     encode the header information (i.e., W, H, ρ, and a_root).
5:     initialize the arithmetic-coding engine
6:     create current BFD tree curTree with the root node rootNode (where
       curTree represents the part of the BFD tree coded so far)
7:     clear the S and R queues, and insert rootNode on the S queue with priority
       sQueuePri(rootNode)
8:     sBudget := sThresh
9:     dBudget := dThresh
10:    while S and R queues are not both empty do
11:        while sBudget > 0 and S queue is not empty do
12:            set curNode to the node at the front of the S queue and remove this
               node from the queue
13:            b := 0
14:            invoke the CCEC coding procedure for curNode; let sNodeList be a
               list containing the newly-created nodes at an even level in curTree
               (i.e., the new leaf nodes); let rNodeList be a list containing curNode
               and the newly created nodes at an odd level in curTree; increment b
               by the number of binary symbols coded (by the arithmetic coder) in
               this step
15:            for each node in rNodeList do
16:                if node has a detail coefficient then
17:                    invoke the DCR coding procedure η times for node; increment
                       b by the number of binary symbols coded (by the arithmetic
                       coder) in this step
18:                    if node has more DC bits to code then
19:                        insert node on the R queue
20:                    endif
21:                endif
22:            endfor
23:            for each node in sNodeList do
24:                insert node on the S queue with priority sQueuePri(node)
25:            endfor
26:            sBudget := sBudget - b
27:        endwhile
```

```
28:         while rBudget > 0 and R queue is not empty do
29:             set node to the node at front of R queue and remove this node from
                the queue
30:             invoke the DCR coding procedure once for node; set b to the number
                of binary symbols coded (by the arithmetic coder) in this step
31:             if still more bits of DC data to code for node then
32:                 insert node on the R queue
33:             endif
34:             rBudget := rBudget - b
35:         endwhile
36:         sBudget := min{sThresh, sBudget + sThresh}
37:         rBudget := min{rThresh, rBudget + rThresh}
38:     endwhile
39: endprocedure
```

Binarization schemes. In the CCEC and DCR coding procedures (to be discussed shortly), the need sometimes arises to code nonbinary symbols. Since a binary arithmetic coder is employed for coding purposes, any nonbinary symbols must be converted to a sequence of binary symbols through some binarization process in order to be coded. In what follows, we introduce the various types of nonbinary symbols used by our coder and describe the binarization scheme used for each type (i.e., ternary, senary, unsigned integer and signed integer).

In what follows, let c_{half} and c_{third} each denote an arithmetic-coder context with a fixed probability distribution in which the probability of a one is $\frac{1}{2}$ and $\frac{1}{3}$, respectively. The first type of nonbinary symbol used is a ternary symbol with a fixed uniform probability distribution. The ternary symbol $n \in [0\,..\,2]$ is coded as a bit with value $\lfloor n/2 \rfloor$ using context c_{third} followed, if $\lfloor n/2 \rfloor = 0$, by a bit with value $\mathrm{mod}(n, 2)$ coded using context c_{half}. The second type of nonbinary symbol employed is a senary (i.e., 6-ary) symbol with a fixed uniform probability distribution. The senary symbol $n \in [0\,..\,5]$ is coded as a bit with value $\lfloor n/3 \rfloor$ using context c_{half} followed by a ternary symbol (with a fixed uniform probability distribution) with value $\mathrm{mod}(n, 3)$.

The third type of nonbinary symbol employed is an n-bit unsigned integer. For this type of symbol, we employ the UI binarization scheme described in [5]. This binarization scheme has two parameters n and f and is denoted $\mathrm{UI}(n, f)$, where n is the number of bits in the integer to be coded and f is a parameter that controls which symbol values are associated with independent probabilities. The method uses $2^f + n - f$ contexts to code bits of integer value $x \in [0\,..\,2^n)$. The contexts are used in such a way that symbols with values in the range $[0\,..\,2^f)$ can have distinct probabilities, while symbols with the remaining values (if any) are partitioned into ranges of the form $[2^i\,..\,2^{i+1})$ for $i \in [f\,..\,n)$, where values within each range must have the same probabilities.

The last type of nonbinary symbol employed is an n-bit signed integer (i.e., an integer with $n-1$ magnitude bits plus a sign bit). For handling this type of symbol, we define the $\mathrm{SI}(n, f)$ binarization scheme, as a trivial extension of the UI method introduced above. To code an n-bit signed integer x with $\mathrm{SI}(n, f)$ binarization, we code $|x|$ with $\mathrm{UI}(n-1, f)$ binarization except that immediately after the first nonzero bit in $|x|$ is coded, a bit indicating the sign of x is coded using a fixed uniform probability distribution.

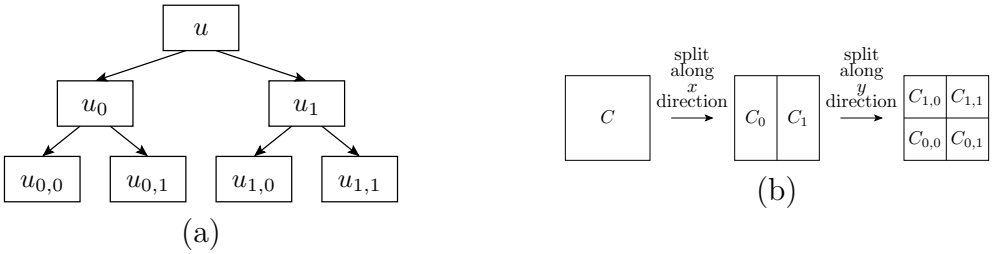


Figure 3.2: Potentially new nodes added by CCEC coding procedure. (a) The subtree rooted at u showing the six positions (relative to u) at which new nodes may potentially be inserted and (b) the cells corresponding to these nodes.

3.3.1 CCEC Coding Procedure

In step 14 of Algorithm 1, the child-configuration and edge-constraint (CCEC) coding procedure is utilized. This procedure is always invoked for a leaf node u at an even level in the current BFD tree (i.e., `curTree`). The CCEC coding procedure codes information that specifies how new nodes should be inserted in the tree (as descendants of u) and how edge-constraint information should be updated.

For a given node u (which is a leaf at an even level in the tree), the CCEC coding procedure adds any children and grandchildren of u (i.e., any nodes with a depth of 1 or 2 relative to u). In other words, this procedure potentially adds nodes at each of the six positions in the tree relative to u shown in Figure 3.2(a), where the nodes u, u_0, u_1, $u_{0,0}$, $u_{0,1}$, $u_{1,0}$, and $u_{1,1}$ are associated with the respective cells C, C_0, C_1, $C_{0,0}$, $C_{0,1}$, $C_{1,0}$, and $C_{1,1}$ shown in Figure 3.2(b). A new node is only *potentially* added at each of the six positions shown in the figure since, as we recall from earlier, a BFD tree only contains nodes with occupied cells. Consequently, only the nodes with occupied cells are added. As a matter of terminology, the particular arrangement of new nodes to be added to the tree is referred to as the **child configuration**. To specify the child configuration, it is sufficient to specify which of $\{C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}\}$ are occupied. Since the cell of a node is contained in the cell of its parent, knowing which of $C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}$ are occupied also implies which of $\{C_0, C_1\}$ are occupied.

Once the child configuration has been determined, the CCEC coding procedure proceeds to add new nodes to the tree. This is accomplished by first adding the (one or two) children of u, and then, for each child added, adding its children. When adding the children for a node v, one of two possibilities can occur: 1) v has exactly one child; or 2) v has exactly two children. As a matter of terminology, the process of adding exactly two children to a node is called a **vertex split**, while the process

of adding exactly one child to a node is called a **vertex drag**. Figure 3.3 illustrates the notion of vertex splits and drags. Suppose that we are given a node v to which its (one or two) children are to be added. The scenario in Figure 3.3(a), where two children are added to v, corresponds to a vertex split, while each of the scenarios in Figures 3.3(b) and (c), where only one child is added to v, correspond to vertex drags. The process of adding all of the appropriate new nodes for u can be viewed as a sequence of vertex split and vertex drag operations. In passing, we note that, during a single invocation of the CCEC coding procedure, at most three vertex splits can occur, which corresponds to the case when each of $\{C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}\}$ is occupied.

Now, we must consider what information needs to be coded in order to indicate changes to the edge-constraint set that results from vertex split and vertex drag operations. For this, we need to understand how these operations transform the current mesh, which is associated with the leaf nodes in the current tree (i.e., `curTree`). Recall that each leaf node in the BFD tree corresponds to a vertex in the mesh that is positioned at the representative point (i.e., approximate centroid) of the node's cell. Since a vertex split replaces the leaf node v with two new leaf nodes, this operation can be viewed as splitting the vertex associated with node v into two new vertices (i.e., the vertices associated with the two child nodes of v). Similarly, since a vertex drag replaces the leaf node v with a single new leaf node, this operation can be viewed as moving the vertex associated with the node v to the vertex associated with the single child node of v. Because vertex splits and vertex drags change some vertices in the mesh, we must consider what information (if any) must be coded to convey potential changes in edge constraints for the mesh. For convenience in what follows, for a node u, cell(u) and vertex(u) denote the cell of u and vertex of u, respectively.

First, we consider the case of a vertex drag. Let v be the node to which the single child v_i has been added. Since v has only the single child v_i, cell(v) $\setminus$ cell(v_i) is an unoccupied cell (i.e., does not contain any sample points) and therefore cannot contain any endpoint for an edge constraint. Consequently, each edge constraint with an endpoint in cell(v) must be such that its endpoint is specifically in cell(v_i). In other words, vertex(v_i) must have the same incident edge constraints as vertex(v). Thus, no information needs to be coded to specify how to update edge constraints in the case of a vertex drag. In effect, a vertex drag simply moves the vertex vertex(v) to the new position vertex(v_i), pulling any incident edge constraints along with it.

Next, we consider the case of a vertex split. Let v be the node to which the children v_0 and v_1 have been added, and let N be the set of constraint-connected

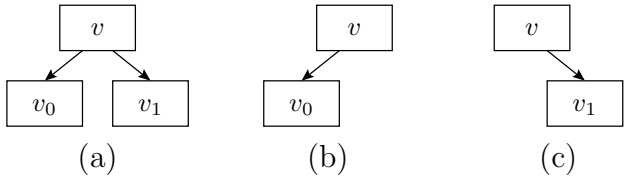


Figure 3.3: Vertex split and drag operations. (a) Vertex split operation and (b) and (c) vertex drag operations.

neighbours of v (prior to the adding of v_0 and v_1). In the case of a vertex split, since cell(v_0) and cell(v_1) are both occupied, they each contain sample points that could potentially serve as endpoints for edge constraints. Consequently, each node $u \in N$ could potentially be constraint connected to (only) v_0 or (only) v_1 or both. Since we cannot know which is the case without additional information, this information must be coded. Furthermore, we have one extra complication that does not arise in the vertex drag case. Since a vertex split adds two new vertices vertex(v_0) and vertex(v_1) (instead of one), the possibility exists that these two new vertices may be connected by an edge constraint. Since it cannot be deduced whether or not this is the case without additional information, this information must also be coded. Thus, in the case of a vertex split, the following information must be coded in order to allow the edge constraint information to be updated correctly: 1) for each $u \in N$, if u is constraint connected to v_0 or v_1 or both; and 2) if v_0 is constraint connected to v_1. In the encoder, the information to be coded is determined by examining the reference tree (i.e., `refTree`).

With all of the above in mind, the CCEC coding procedure codes child configuration information followed by edge-constraint update information for each vertex split encountered while adding new nodes. In order to complete our description of this procedure, we simply need to explain the manner in which the child configuration and edge-constraint update information is coded, which we do next.

Child configuration coding. To convey the child configuration for the node u, we code a count n of how many of $\{C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}\}$ are occupied followed by an indication of specifically which n cells are occupied. This information is determined by the encoder by examining the reference tree (i.e., `refTree`). For a given node u, the child configuration is coded as follows. Let n denote the number of occupied child cells $\{C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}\}$ associated with u. The value $n - 1 \in [0\,..\,3]$ is coded using UI$(2, 2)$ binarization, conditioned on $\ell/2$ and $\min\{6, m\}$, where ℓ is the level in the tree at which u resides and m is number of constraint-connected neighbours

of u. (Note that $\ell/2 \in \mathbb{Z}$ since ℓ is always even, as noted above.) Which n of the child cells $\{C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}\}$ are occupied is coded as follows. If $n \in \{1, 3\}$, four configurations are possible, and the particular configuration is coded using two bits, each with a fixed uniform distribution. If $n = 2$, six configurations are possible, and the particular configuration is coded as a senary symbol with a fixed uniform distribution. If $n = 4$, only one configuration is possible (with all four child cells being occupied) and consequently no information need be coded.

Edge-constraint coding. Now, we consider the scheme used to code the information required to update the edge constraints after a vertex split. Let v denote the node to which the two children v_0 and v_1 have been added. Let N denote the set of constraint-connected neighbours of v. Let $\theta(u)$ denote a count of how many of $\{v_0, v_1\}$ are constraint connected to u. Let $\gamma(u)$ denote the respective values 0, 1, or 2, if in the direction of the cell split, u lies strictly between v_0 and v_1, lies strictly outside v_0 and v_1, or neither. For each $u \in N$, we perform the following. A binary symbol is coded indicating if $\theta(u) = 2$, conditioned on $\gamma(u) \in \{0, 1, 2\}$. If $\theta(u) = 1$, u is known to be constraint connected to exactly one of v_0 or v_1, and we must code which one. To do this, we proceed as follows: 1) If vertex(u) is not equidistant to vertex(v_0) and vertex(v_1), we predict u to be constraint connected to v_0 if $\|\text{vertex}(v_0) - \text{vertex}(u)\| < \|\text{vertex}(v_1) - \text{vertex}(u)\|$ and v_1 otherwise, and then code a single binary symbol indicating if this prediction is correct, conditioned on the level of v in the tree. 2) If vertex(u) is equidistant to vertex(v_0) and vertex(v_1), one bit with a fixed uniform distribution is coded indicating if u is constraint connected to v_0. If $\theta(u) = 2$, u is already known to be constraint connected to each of v_0 and v_1 and no information need be coded to indicate this. Lastly, a single binary symbol is coded, indicating if v_0 and v_1 are constraint connected, conditioned on the level of v in the tree.

3.3.2 DCR Coding Procedure

In steps 17 and 30 of Algorithm 1, the detail-coefficient refinement (DCR) coding procedure is invoked. As mentioned earlier, this procedure is responsible for coding detail-coefficient data (i.e., sign and magnitude bits for detail coefficients). In what follows, we describe this procedure in detail.

The DCR coding procedure is invoked for a particular node in the current tree (i.e., `curTree`) (which can be at any level in the tree), where the node must have a detail coefficient. Let u denote this node and let d denote its detail coefficient. The

DCR coding procedure proceeds as follows. If this is the first time that the coding procedure is being invoked for u (i.e., no detail-coefficient bits have been previously coded for u), initialize the $\mathrm{SI}(\rho+1, \min\{\rho+1, 4\})$ binarization process for coding d, where ρ is as defined earlier. If no bits in $|d|$ remain to be coded, the procedure is complete; otherwise, processing continues. Next, the most significant bit of $|d|$ that has not yet been coded (using an earlier invocation of the DCR coding procedure) is coded. This may also, as a side effect, code an additional bit for the sign of d, since SI binarization automatically codes the sign bit of an integer immediately after coding its first nonzero magnitude bit. If the detail coefficient d has n magnitude bits, the DCR coding procedure must be invoked n times for u in order to fully code d.

3.4 Decoding

Although the encoding and decoding algorithms of our method are mostly symmetric, some small asymmetries exist. Consequently, a few aspects of the decoding algorithm cannot be inferred from a description of the encoding process. In what follows, we comment on these particular aspects of the decoder.

At intermediate stages of progressive decoding, it is possible to obtain a decoded mesh that is not topologically valid. That is, it is possible to obtain a set of decoded edge constraints with nontrivial intersections (i.e., intersections excluding those that simply share a common endpoint). In such cases, a postprocessing step must be applied to the mesh in order to restore its validity. This postprocessing is performed as follows. To avoid nontrivially intersecting edge constraints, the constrained PD triangulation in the decoder is constructed by inserting each decoded edge constraint in the triangulation, one at a time, in order of increasing edge length. If an edge constraint to be inserted would nontrivially intersect an edge constraint already in the triangulation, the insertion of the constraint is skipped. Since this process avoids introducing nontrivial intersections, this process will always result in a valid mesh.

At intermediate stages of progressive decoding, some of the bits of some detail coefficients may be unknown. (The root approximation coefficient is always known since it is included in the header at the start of the coded bitstream.) Due to the manner in which the bits of the detail coefficients are coded, when one or more bits of a detail coefficient are unknown, one of two situations must be the case: 1) at least one nonzero magnitude bit (as well as the sign bit) is known; or 2) no nonzero magnitude bits are known. In the second case, the detail coefficient should be decoded

as zero. In the first case, the detail coefficient is known to lie in the range $[d_0 \,..\, d_1]$ (where d_0 and d_1 are determined by the bits that are known), and the decoded value should be chosen as the approximate midpoint of this range. After any unknown bits in detail coefficients have been chosen as explained above, the inverse ADT is applied in a straightforward manner to obtain the reconstructed function values.

In many applications, the function domain associated with the mesh is an iso-oriented (i.e., axis-aligned) rectangle. This is often the case for images. In such circumstances, the progressive coding performance can be improved by ensuring that the extreme convex hull points of the function domain (i.e., the four corners of the function-domain bounding box) are always present in the reconstructed mesh. During the earlier stages of progressive coding, these points will often be missing. This can potentially lead to high distortion in the function reconstruction near the border of the function domain. A simple postprocessing step can be applied to mitigate this problem, however. For each of the four corner points, if a point is found to be missing during (intermediate stages of progressive) decoding, the missing point is added with its function value chosen as the function value of the closest sample point in the progressively decoded dataset. This postprocessing step typically leads to lower distortion in the function reconstruction near the function-domain boundary. In applications where the function domain is known to always be an iso-oriented rectangle, this postprocessing step can simply be applied automatically.

Chapter 4

Evaluation of the Proposed Method

In this chapter, we evaluate the proposed method by comparing it with several mesh coders for both lossless and progressive coding. The competitors include 3-D coders (namely, Wavemesh [31] and Edgebreaker [28]) and 2.5-D coders (namely, ADIT [8], IT [5], and SDC [12]). Moreover, the proposed mesh coding method can also be combined with a mesh generator to form a highly effective mesh-based image coder, which is benchmarked against the popular JPEG 2000 codec for compressing images that are nearly piecewise smooth.

4.1 Test Data

Before stepping into the experimental results, a brief discussion is needed to introduce the test data used herein. A large set of meshes are used in our experiments, which include 40 PD (preferred Delaunay) meshes and 58 non-PD (non-preferred Delaunay) meshes. Those triangle meshes are generated from a variety of bivariate functions, which include images taken mostly from the standard test sets [30, 19] and elevation maps [29]. The meshes are generated with a variety of schemes such as those described in [6] and [7]. For the purposes of presentation, individual results are only shown for the 15 test meshes (10 non-PD and 5 PD) in Table 4.1. This table shows the name of each mesh as well as the number of vertices, edges, and faces in the mesh, the fraction of non-locally-PD edges, the function-domain bounding-box sizes, the number of bits per function value, and the function type (i.e, image versus elevation map). We can

Table 4.1: Test meshes

Name	V†	E†	F†	Fraction of Non-LPD† Edges (%)	Unpadded Bounding Box	Bits Per Sample	Type
kodim15@0.02	7864	23583	15720	20.5	768×512	8	image
lena@0.03	7864	15578	23441	21.0	512×512	8	image
ct@0.01	2621	7840	5220	19.9	512×512	12	image
cr@0.005	17858	53419	35562	19.8	1744×2048	10	image
muttart@0.0025	3637	10858	7222	21.7	1912×761	8	image
question2@0.02	38400	115193	76794	19.8	1200×1600	8	image
checkerboard@0.01	2621	7753	5133	19.1	512×512	8	image
n27@0.0025	3606	10711	7106	13.9	1201×1201	13	EM†
n35@0.0025	3606	10715	7110	13.8	1201×1201	12	EM
n49@0.01	14424	43132	28709	13.0	1201×1201	12	EM
animal@0.005	7397	22146	14750	0	1238×1195	8	image
bull@0.02	15728	47073	31346	0	1024×768	8	image
peppers@0.04	10485	31297	20813	0	512×512	8	image
wheel@0.04	3451	10346	6896	0	512×512	8	image
n45@0.01	14424	43036	28613	0	1201×1201	10	EM

†non-LPD stands for non-locally-PD
†EM stands for elevation map
†V, E, and F stand for number of vertices, number of edges, and number of faces, respectively

see from the Table 4.1 that those 15 meshes are chosen to be quite diverse in terms of the characteristics listed in the table. The first 10 meshes are not PD while the last five meshes are PD.

4.2 Lossless Coding Performance

To begin, we compare our coder with others in terms of lossless bit rate. Each of the 98 meshes (i.e. 58 non-PD and 40 PD) is losslessly compressed using the proposed mesh coder and 3-D methods and the final bit rate is measured. Since the ADIT, SDC, and IT methods can only code PD meshes, only 40 PD meshes are used for comparison with those 2.5-D methods. The summaries of the statistical results obtained for non-PD, PD, and all meshes are shown in Tables 4.2(a), 4.2(b), and 4.2(c), respectively. A representative subset of the lossless coding results for 10 non-PD meshes and 5 PD meshes are also listed in Table 4.3(a) and Table 4.3(b), respectively.

First, we analyze how the proposed method compares to the 3-D coders (namely, Wavemesh and Edgebreaker) in terms of lossless bit rate. Examining the overall re-

Table 4.2: Summary of lossless coding results for the various methods under consideration for the cases of (a) 50 non-PD meshes, (b) 48 PD meshes, and (c) 98 all meshes.

(a)

Mesh	Mean Bit rate (bits/vertex)		
	Proposed	Edgebreaker	Wavemesh
non-PD	**19.19**	22.36	29.72

(b)

Mesh	Mean Bit rate (bits/vertex)					
	Proposed	SDC	IT	ADIT	Edgebreaker	Wavemesh
PD	15.29	15.31	15.32	**15.23**	21.43	27.02

(c)

Mesh	Mean Bit rate (bits/vertex)		
	Proposed	Edgebreaker	Wavemesh
All meshes	**18.08**	21.94	28.79

sults shown in Tables 4.2(a) and 4.2(c), we can see that the proposed method is vastly superior to the Edgebreaker and Wavemesh methods. More specifically, the Edgebreaker and Wavemesh schemes produce coded bitstreams that are 3.84 bits/vertex and 10.71 bits/vertex larger than those generated by the proposed method, on average. It is not surprising that the Edgebreaker method is superior to the Wavemesh method since the former is a single rate coder while latter is a progressive coder. Moreover, although not listed in the table, the experimental results also show that the proposed method outperforms the Edgebreaker and Wavemesh schemes in every test case, by margins of 2.04 to 12.05 bits/vertex and 7.09 to 25.46 bits/vertex, respectively. The individual results for 10 non-PD meshes and 5 PD meshes shown in Tables 4.3(a) and 4.3(b) are consistent with the overall statistical results. Based on the above statistical results, we conclude that the proposed method outperforms 3-D coders Wavemesh and Edgebreaker in terms of lossless coding performance.

Having considered 3-D coders, we now compare the proposed coder with the SDC, IT, and ADIT methods. Since these competitors can only handle PD meshes, only PD meshes are used for evaluation purposes in this case. Examining the overall results obtained for PD meshes shown in Table 4.2(b), we can see that the 3-D coders Edgebreaker and Wavemesh are significantly worse than all of the 2.5-D coders. The proposed coder is comparable to the SDC method, IT method, and ADIT method,

Table 4.3: Subset of lossless coding results for the various methods under consideration for the cases of (a) non-PD meshes and (b) PD meshes

(a)

Mesh	Bit rate (bits/vertex)		
	Proposed	Edgebreaker	Wavemesh
n49@0.01	**21.30**	25.90	34.39
kodim15@0.02	**18.32**	21.67	27.71
lena@0.03	**17.92**	20.80	25.65
ct@0.01	**22.90**	25.52	36.36
cr@0.005	**21.16**	24.29	29.02
n27@0.0025	**24.41**	28.70	43.00
n35@0.0025	**23.18**	26.64	35.52
question2@0.02	**15.42**	18.35	23.00
muttart@0.0025	**22.78**	31.24	43.74
checkerboard@0.01	**17.10**	20.18	23.79

(b)

Mesh	Bit rate (bits/vertex)					
	Proposed	SDC	IT	ADIT	Edgebreaker	Wavemesh
animal@0.005	14.28	**14.25**	14.39	14.34	20.00	24.15
bull@0.02	**12.79**	12.83	12.89	12.84	18.90	19.86
peppers@0.04	13.15	13.18	13.19	**13.11**	18.94	20.85
wheel@0.04	12.10	**11.92**	12.16	12.06	19.25	25.67
n45@0.01	**15.47**	15.75	15.54	15.54	21.30	27.34

differing by a margin of less than 0.1 bits/vertex on average. In spite of the fact that the proposed coder is at significant disadvantage due to handling a more general type of dataset (i.e, meshes with arbitrary connectivity) than the competitors, our method still outperforms the IT method, with the average lossless bit rate being 0.03 bits/vertex less. The individual results for 5 PD meshes are shown in Table 4.3(b). Examining the Table 4.3(b), we can see that the lossless coding performances of all the 2.5-D coders are very close, which is consistent with the overall results that we obtained.

4.3 Progressive Coding Performance

As mentioned earlier, each mesh in the test set was generated from a bivariate function sampled on a rectangular grid. For evaluation of progressive coding performance, each of the 98 meshes is first encoded losslessly using the proposed method and the Wavemesh method. The Edgebreaker method is not used for this comparison as it is a single rate coder and does not support progressive coding. Since the ADIT, IT, and SDC methods can only code PD meshes, only 40 PD meshes are used for comparison with these 2.5-D coders. After a mesh is losslessly encoded, the compressed bitstream is decoded at many intermediate rates. The decoded mesh at each stage is rasterized to produce a sampled function with same dimensions as the original function, and the PSNR value relative to the original function is computed. The reason we compare the rasterized functions instead of comparing the meshes directly is that the difference between meshes is difficult to measure and the function reconstruction error is also of great practical interest. The PSNR value is measured against the original function instead of the losslessly reconstructed function to avoid arbitrarily large PSNR value when the bit stream is nearly fully decoded, which makes graphs difficult to interpret.

A representative subset of the experimental results is given in Table 4.4. The results for non-PD meshes are shown in Table 4.4(a) while the results for PD meshes are shown in Table 4.4(b). The results in Table 4.4(a) only contain those obtained with the proposed method and the Wavemesh method since other methods can not code meshes with arbitrary connectivity. Examining the individual results in these two tables, we can see that the proposed method always produces the reconstructed function of best quality at every intermediate rate. More specifically, the proposed method outperforms the 3-D Wavemesh method by margins of 7.08 to 13.28 dB. For PD meshes, in spite of the fact that the SDC, IT, and ADIT methods all have the

advantage of handling a more constrained type of dataset (PD mesh), the proposed method still outperforms the SDC, IT, and ADIT methods by margins of 5.55 to 11.90 dB, 0.81 to 8.58 dB, and 0 to 4.29 dB, respectively. In order to show the difference in terms of the visual quality, examples of reconstructed functions for the non-PD and PD cases are shown in Figures 4.1 and 4.2, respectively. Figure 4.1(a) shows the function obtained with the proposed method, while Figure 4.1(b) demonstrates the function generated by the Wavemesh method. The lossy bit rate is chosen as 14874 bytes. It is clear that the visual quality of the function in Figure 4.1(a) is vastly superior to that of the function in Figure 4.1(b). Figure 4.2 shows results for a PD mesh. The functions obtained with the proposed method, SDC method, IT method, ADIT method, and Wavemesh method are shown in Figures 4.2(a), (b), (c), (d), and (e), respectively. We can see that the function obtained with the proposed method is of best quality. The ADIT method is the second best, followed by the IT, Wavemesh, and SDC methods.

Although not provided in this thesis, the progressive coding results for all the 98 meshes were obtained. The overall results show that the proposed method is vastly superior to all the other competitors in every test case, which is consistent with the individual results discussed above. More specifically, the proposed method outperforms the Wavemesh method, SDC method, IT method, and ADIT method by average margins of 11.18 dB, 10.56 dB, 4.88 dB, and 3.15 dB, respectively.

4.4 Comparison with Traditional Image Coder

4.4.1 Mesh-based Image Coding System

The proposed mesh-coding method can be combined with a mesh generation method to achieve a highly effective mesh-based image coding system. To better illustrate this coding system, a diagram is shown in Figure 4.3. As illustrated in Figure 4.3, the image coding system consists of two parts, an image encoder and image decoder. The image encoder is formed by combining the proposed mesh-encoding method with a mesh generator, while the image decoder includes the proposed mesh-decoding method and a mesh rasterizater. At the beginning, the original image is given as the input. The image is first processed by the mesh generator to produce a 2.5-D triangle mesh model. Next, this mesh model is encoded by the proposed encoding method and a compressed image is produced. To decode the compressed image, the proposed

Table 4.4: Comparison of progressive coding results for the various methods under consideration. (a) Non-Delaunay (b) Delaunay.

(a)

Mesh	Rate(bytes)	PSNR (dB)	
		Proposed	Wavemesh
kodim15@0.02	2200	**20.53**	8.92
	4300	**21.85**	14.38
	9400	**25.10**	19.31
	15100	**31.72**	23.36
	21000	**32.03**	26.39
lena@0.03	2800	**22.00**	14.92
	5200	**23.70**	18.41
	9400	**26.66**	20.95
	14800	**33.66**	24.95
	20400	**34.21**	28.10
ct@0.01	600	**23.48**	17.52
	2100	**27.10**	22.38
	4000	**27.82**	25.03
	5300	**41.10**	27.82
	7500	**43.16**	32.33

(b)

Mesh	Rate (bytes)	PSNR (dB)				
		Proposed	SDC	IT	ADIT	Wavemesh
animal@0.05	1000	**25.70**	15.75	23.31	23.44	12.46
	4000	**30.01**	19.71	28.42	27.92	19.14
	7000	**35.02**	22.70	31.10	30.61	24.65
	10000	**41.43**	29.53	33.13	36.74	27.00
	13000	**43.56**	38.01	40.49	43.56	33.80
bull@0.02	1100	**25.23**	13.34	23.52	23.67	15.55
	5600	**29.66**	18.83	27.57	28.23	24.23
	10100	**31.50**	22.38	30.19	30.64	27.03
	15600	**36.50**	25.94	32.52	34.02	30.69
	20600	**42.08**	30.99	36.28	40.25	34.88
peppers@0.04	1000	**21.02**	11.62	19.69	19.68	10.58
	4900	**24.46**	13.02	22.58	23.02	17.51
	8500	**26.77**	17.77	24.98	25.29	19.18
	12400	**32.93**	22.66	26.37	30.23	21.75
	16300	**34.35**	28.11	29.70	34.25	26.02

(a) (b)

Figure 4.1: Progressive coding example for non-PD case. Reconstructed images obtained after decoding 14874 bytes of mesh for lena image using the (a) proposed (33.70 dB) and (b) Wavemesh (24.95 dB) methods.

(a) (b) (c)

(d) (e)

Figure 4.2: Progressive coding example for PD case. Reconstructed images obtained after decoding 9100 bytes of the mesh for the animal image using the (a) proposed (38.47 dB), (b) SDC (26.39 dB), (c) IT (32.05 dB), (d) ADIT (34.10 dB), and (e) Wavemesh (27.00 dB) methods.

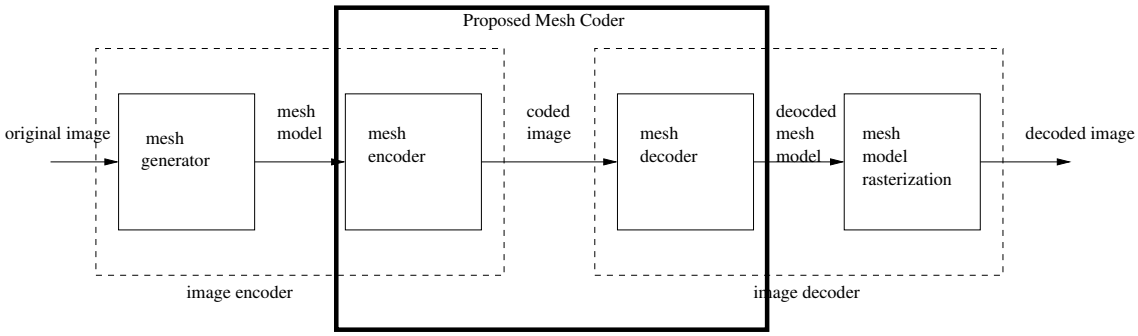


Figure 4.3: The image coding system consisting of the proposed coding method and a mesh generator.

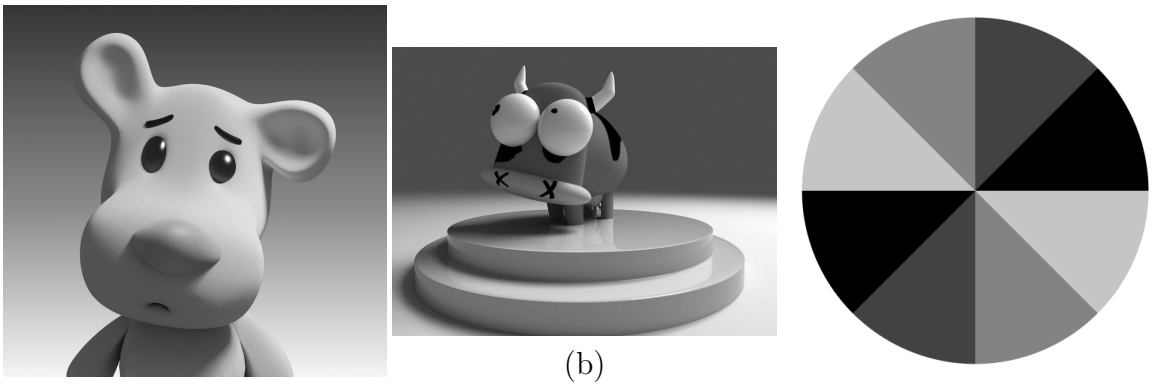
decoding method is first used to recover the 2.5-D triangle mesh model, which is then rasterized to produce an image reconstruction. For the purposes of evaluation, two mesh generators are chosen and used in the experiments, namely the methods described in [7] and [17].

4.4.2 Comparison with JPEG 2000 Codec

The image coding system is evaluated by comparing to the well-known JPEG 2000 codec for images that are nearly piecewise smooth. First, the given image is encoded by the JPEG 2000 codec and the mesh-based image encoder as illustrated in Figure 4.3, respectively. We fix the compression rate (the ratio of the size of the original image to the size of the coded image) so the size of the coded image obtained with proposed method is same as that of the coded image produced by the JPEG-codec. After that, the coded images are decoded by the mesh-based image decoder and the JPEG 2000 codec, which results in two reconstructed images. The quality of the decoded images is measured in terms of PSNR.

The subset of images used for comparison are shown in Figure 4.4. The information of those three images is given in Table 4.5. For each image, we compress it with the mesh-based image coding system and the JPEG 2000 codec at a wide range of compression rates, and the rate-distortion performance is measured and shown in Table 4.6. Although only three images are presented in this section, 15 images from [30, 19] were tested in total. The mesh-based image coding system outperforms the JPEG 2000 codec consistently for images that are nearly piecewise smooth so only a representative subset of results is given.

Examining the results in Table 4.6, we can see that for all those three test cases,



(a) (b) (c)

Figure 4.4: The test images used for comparing the proposed method with JPEG 2000 codec. (a) animal, (b) bull, and (c) wheel.

Table 4.5: Test images

Image	Size	Bits/Sample	Description
bull	1024×768	8	cartoon animal
animal	1238×1195	8	cartoon animal
wheel	512×512	8	computer generated image

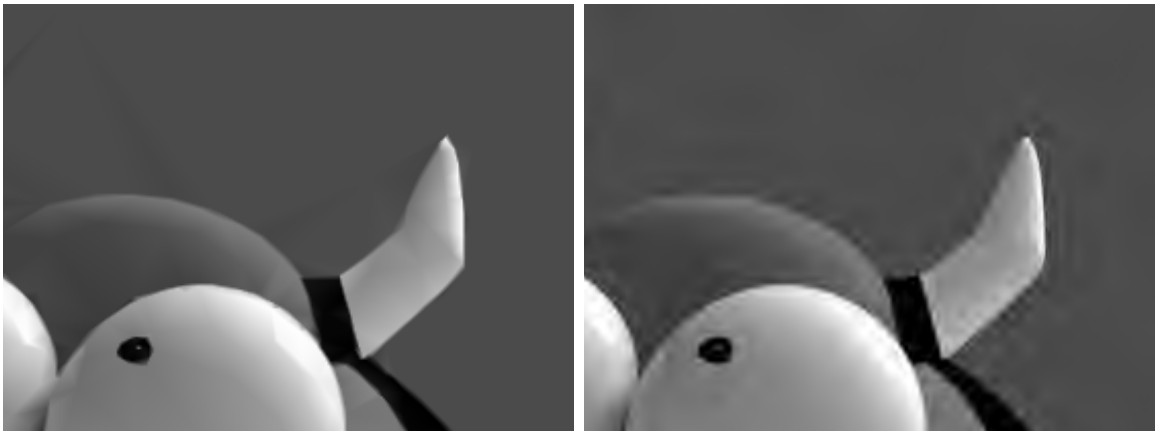
the mesh-based image coding system yields image reconstructions of better quality than those obtained with JPEG 2000 codec for most compression rates. The only exception is that the JPEG 2000 codec beats the the proposed method by 1.04 dB when compressing the image bull at compression ratio 40:1. Moreover, for images bull and animal, the mesh-based image coding system outperforms the JPEG 2000 codec by a margin of 1.93 dB on average. For image wheel, the JPEG 2000 codec is vastly beaten by the proposed method by margins of 11.33 to 17.98 dB. Moreover, some examples of reconstructed images are also given to demonstrate the differences in terms of their visual quality. The examples are shown in Figures 4.5, 4.6, and 4.7. Some of those images are magnified to show differences more clearly. It is clear that the images generated by the proposed coder in Figures 4.5(a), 4.6(a), and 4.7(a) are reasonably good and very close to the original images, while those obtained with JPEG 2000 codec (Figures 4.5(b), 4.6(b), and 4.7(b)) are blurry. This is due to the fact that JPEG 2000 codec is a wavelet-based coder, which usually results in ringing artifacts that blur the edges. Our image coder is a mesh-based coder, so it can preserve edges by avoiding the ringing artifacts.

Table 4.6: Comparison of lossy coding results

Image	Comp. Ratio	PSNR (dB)	
		Mesh	JPEG 2000
bull	40:1	44.63	**45.67**
	100:1	**42.51**	41.20
	200:1	**39.41**	38.00
	250:1	**39.08**	37.22
animal	114:1	**43.22**	42.83
	241:1	**41.84**	40.28
	435:1	**40.47**	37.63
	526:1	**39.84**	35.68
wheel	71:1	∞	39.42
	121:1	**52.20**	34.22
	200:1	**46.98**	30.13
	250:1	**39.63**	28.30

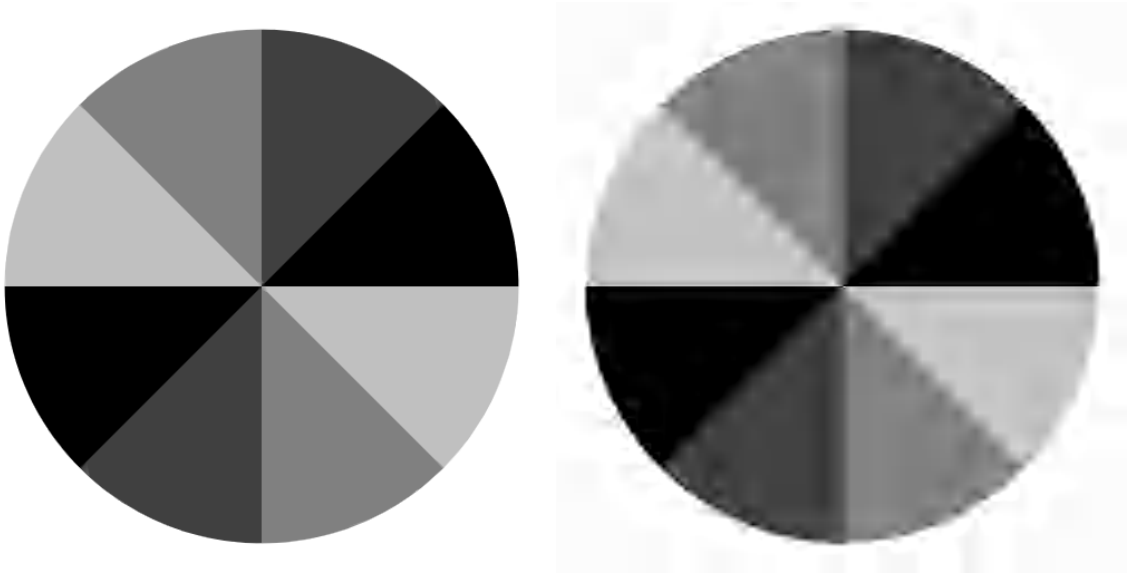
(a) (b)

Figure 4.5: Part of the reconstructed images obtained for the animal image: compression with the (a) proposed (39.84 dB) and (b) JPEG 2000 (35.68 dB) methods at compression ratio 526:1.



(a) (b)

Figure 4.6: Part of the reconstructed images obtained for the bull image: compression with the (a) proposed (39.08 dB) and (b) JPEG 2000 (37.22 dB) methods at compression ratio 250:1.



(a) (b)

Figure 4.7: Part of the reconstructed images obtained for the wheel image: compression with the (a) proposed (39.63 dB) and (b) JPEG 2000 (28.30 dB) methods at compression ratio 250:1.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

In this thesis, we proposed a new progressive lossy-to-lossless coding method for 2.5-D triangle meshes with arbitrary connectivity. A novel representation for 2.5-D mesh datasets called the BFD tree was proposed. A BFD tree captures all of the information needed to completely characterize a 2.5-D mesh model. The proposed coding method first represents the given mesh dataset as a BFD tree, and then codes the BFD tree in a top-down manner. For connectivity coding, instead of viewing the mesh connectivity as a graph, our approach describes the connectivity as a set of edge constraints for a constrained PD triangulation. The coding method only encodes the edge constraints rather than the whole connectivity, which yields a particularly compact representation.

For the evaluation of our proposed mesh-coding method, we compared it with several other 2.5-D and 3-D mesh coders. Experimental results showed that the coding performance of the proposed approach is vastly superior to that of 3-D mesh coders for lossless coding. For example, the proposed method outperforms the Wavemesh method and Edgebreaker method needing 27.3% and 68.1% less bits on average for coding mesh datasets losslessly. For progressive coding, the proposed method is shown to beat both 3-D mesh coders and 2.5-D mesh coders by a margin of up to 13.28 dB.

A highly effective mesh-based image coder for lattice-sampled images was achieved by combining the proposed method with a mesh generator. The combined coder was compared with the well-known JPEG 2000 codec for images that are approximately piecewise smooth. The images were compressed with the proposed mesh-based image

coder and the JPEG 2000 codec respectively at the same compression rates. Then the compressed files were decoded and the resulting reconstructions were measured by their PSNR. The experimental results showed that our mesh-based image coder outperforms the JPEG 2000 codec by an average margin of 3.46 dB for images that are nearly piecewise smooth.

5.2 Future Research

Although the proposed mesh coder offers excellent performance for both progressive and lossless coding, some additional work could still be done in this area. In what follows, some potential avenues for future work are given.

As presented in Section 3.3.1, in the case of a vertex split, a node v is split into two new nodes v_0 and v_1. For each constraint-connected neighbor u of the node v, certain information must be coded in order to indicate to which of v_0 and v_1 the node u is constraint-connected after a vertex split if the node u is not constraint-connected to both of v_0 and v_1. The coder predicts u to be constraint connected to the child node whose representative vertex is closer to the representative vertex of u. The efficiency of the coder is increased due to this prediction. Sometimes, however, vertex(u) is equidistant to vertex(v_1) and vertex(v_2). For this boundary case, the coder currently codes a single bit with a fixed uniform distribution to indicate if u is constraint connected to v_0. Coding symbols in this manner is often inefficient compared to adaptive coding since the distribution of symbols is likely to be more skewed in the latter. If an efficient prediction scheme is employed for handling the case where vertex(u) is equidistant to vertex(v_1) and vertex(v_2), the coding efficiency could be further increased.

Another area that is worth further researching is the choice of the priority function parameter sQueuePri. This function parameter controls the order in which different regions of the mesh are refined. As mentioned in Section 3.3, in our work, the parameter sQueuePri is chosen to achieve good overall progressive coding performance. Other choices of priority functions, however, could also be made for different purposes. For example, the priority function could be chosen to prioritize coding a particular region in the mesh to provide a basic region-of-interest coding functionality. In such a case, the priority of a node could be determined based on the position of that node in relation to the region of interest.

Appendix A

Software User Manual

A.1 Introduction

As part of this work, a software implementation of the mesh-coding method proposed herein was developed. The software was written in C++ and consists of around 10000 lines of code, which includes some fairly complex data structures and algorithms. The libraries utilized in this software include the Computational Geometry Algorithm Library (CGAL) [3], the Boost Library [1], the Signal Processing Library (SPL) [9], and the SPL Extensions Library (SPLEL).

Basically, our software consists of two executable programs:

1. `mesh_encode`, which reads a mesh from standard input in OFF format [4], encodes the mesh, and writes the coded bitstream to standard output.

2. `mesh_decode`, which reads the compressed bitstream from standard input, decodes the mesh, and writes the reconstructed mesh to standard output in OFF format.

The remainder of this appendix provides detailed information about how to build, install, and use the software.

A.2 Build and Install the Software

Since our program utilizes some features of `C++17`, the compiler should support `C++17`. GCC 8.1 or higher version is recommended to be used as the compiler. Also, all the

libraries mentioned above need to be installed before building the software. The versions of those libraries that are known to work are:

- Boost 1.59.0
- CGAL 3.8.2
- SPL 2.0.4
- SPLEL 2.0.5

The CMake [2] tool with version 3.2.2 or later should be installed prior to building our software. In what follows, `$SOURCE_DIR` denotes the top-level directory of the software distribution (i.e., the directory containing the CMakeLists.txt), `$BUILD_DIR` denote denotes a directory to be used for building the software, and `$INSTALL_DIR` denotes the directory under which the software should be installed. To build and install our software, perform the following steps in order:

1. Change the current working directory to `$SOURCE_DIR`
2. Create native build files by running the command:
   `cmake -H. -B$BUILD_DIR -DCMAKE_INSTALL_PREFIX=$INSTALL_DIR`
3. To install the executables, libraries, include files, and other auxiliary data, use the command:
   `cmake --build $BUILD_DIR --clean-first --target install`

Once the software is installed, the native build files can be removed by using the command:
`rm -rf $BUILD_DIR`.

A.3 Detailed Program Descriptions

As mentioned before, the software consists of two executable programs: `mesh_encode` and `mesh_decode`. In what follows, we provide detailed information of how to use them.

A.3.1 `mesh_encode`

SYNOPSIS

`mesh_encode [OPTIONS]`

DESCRIPTION

This program reads a mesh from standard input in `OFF` format, encodes the mesh and writes the compressed bitstream to standard output.

OPTIONS

The following options are supported:

`-i`	`$initialDC`	Sets the number of times that the detail coefficient coding procedure is invoked initially to `$initialDC`. The default value is 3.
`-d`	`$r_threshold`	Sets the threshold value for the `R_queue` to `$r_threshold`. The default value is 10.
`-g`	`$s_threshold`	Sets the threshold value for the `S_queue` to `$s_threshold`. The default value is 100.
`-z`	`$z_quantization`	Sets the quantization step size for the z coordinate (function value) to `$z_quantization`. The default value is 1.
`-x`	`$x_quantization`	Sets the quantization step size for the x coordinate to `$x_quantization`. The default value is 1.
`-y`	`$y_quantization`	Sets the quantization step size for the y coordinate to `$y_quantization`. The default value is 1.
`-r`	`$max_rate`	Sets the maximum number of bytes that will be encoded to `$max_rate`. The default value is ∞.

The program exits with status 0 if the software finishes without any errors, and

1 otherwise.

A.3.2 `mesh_decode`

SYNOPSIS

`mesh_decode [OPTIONS]`

DESCRIPTION

This program reads a coded mesh from standard input, decodes the bitstream, and writes the decoded mesh to standard output in `OFF` format. The decoding process can be terminated at any intermediate stage to allow progressive decoding.

OPTIONS

The following options are supported:

`-s`	`$edge_sort_policy`	Sets the order in which the edge constraints will be sorted as `$edge_sort_policy`. There are three choices for `$edge_sort_policy`: `ascending`, `descending`, and `random`. The default value is `ascending`
`-r`	`$max_rate`	Sets the maximum number of bytes that will be decoded to `$max_rate`. The default value is ∞.

The program exits with status 0 if the software finishes without any errors, and 1 otherwise.

A.4 Examples of Software Usage

Some examples are provided in order to illustrate how to use the software with different options.

Example A

Suppose that we want to encode the mesh in a file named `input_mesh.off` and write the coded data to a file named `output.coded` with the following requirements:

- the value of threshold for `R_queue` is set to be 8;
- the value of threshold for `S_queue` is set to be 90;
- the number of times that DC coding procedure is invoked initially is set to be 4; and
- default options are used for other parameters.

The above can be accomplished with the following command:

```
mesh_encode -g 90 -d 8 -i 4 <input_mesh.off >output.coded
```

Example B

Suppose that we want to encode the mesh in a file named `input_mesh.off` and write the coded data to a file named `output.coded` with the following requirements:

- the value of the quantization steps for z coordinate is set to be 2;
- the value of threshold for `S_queue` is set to be 110;
- the number of times that DC coding procedure is invoked initially is set to be 2;
- the number of bytes that will be encoded is set to be 8000; and
- default options are used for other parameters.

The above can be accomplished with the following command:

```
mesh_encode -g 110 -f 2 -i 2 -r 8000 <input_mesh.off >output.coded
```

Example C

Suppose that we have a compressed file called `mesh.coded`, which was produced by the `mesh_encode` program. We would like to decode the full bitstream and write the decoded mesh to the file `reconstructed_mesh.off`. The above can be accomplished

with the following command:

```
mesh_decode <mesh.coded >reconstructed_mesh.off
```

Example D

Suppose that we have a compressed file called `mesh.coded`, which was produced by the `mesh_encode` program. The size of `mesh.coded` is 10000 bytes. We would like to decode the first 5000 bytes from the input bitstream and use descending order for inserting all the edge constraints, and write the decoded mesh to file `reconstructed_mesh.off`. The above can be accomplished with the following command:

```
mesh_decode -r 5000 -s descending <mesh.coded >reconstructed_mesh.off
```

Bibliography

[1] Boost C++ library, April 2018. `http://www.boost.org`.

[2] CMake tool, May 2018. `https://cmake.org`.

[3] Computational geometry algorithms library, April 2018. `https://www.cgal.org/`.

[4] OFF, object file format, May 2018. `http://segeval.cs.princeton.edu/public/off_format`.

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

[6] M. D. Adams. A flexible content-adaptive mesh-generation strategy for image representation. *IEEE Transactions on Image Processing*, 20(9):2414–2427, September 2011.

[7] M. D. Adams. A highly-effective incremental/decremental Delaunay mesh-generation strategy for image representation. *Signal Processing*, 93(4):749–764, April 2013.

[8] M. D. Adams. An improved progressive lossy-to-lossless coding method for arbitrarily-sampled image data. In *Proc. of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 79–83, Victoria, BC, Canada, August 2013.

[9] M. D. Adams. Signal processing library, May 2018. `http://www.ece.uvic.ca/~frodo/SPL/`.

[10] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[11] B. Delaunay. Sur la sphere vide. *Bulletin of the Academy of Sciences of the USSR, Classe des Sciences Mathematiques et Naturelle*, 7(6):793–800, 1934.

[12] L. Demaret and A. Iske. Scattered data coding in digital image compression. In *Curve and Surface Fitting: Saint-Malo 2002*, pages 107–117, Brentwood, TN, USA, 2003. Nashboro Press.

[13] O. Devillers, R. Estkowski, P.-M. Gandoin, F. Hurtado, P. Ramos, and V. Sacristan. Minimal set of constraints for 2D constrained Delaunay reconstruction. *International Journal of Computational Geometry*, 13(5):391–398, 2003.

[14] C. Dyken and M. S. Floater. Preferred directions for resolving the non-uniqueness of Delaunay triangulations. *Computational Geometry—Theory and Applications*, 34:96–101, 2006.

[15] N. Dyn. Data-dependent triangulations for scattered data interpolation and finite element approximation. *Applied Numerical Mathematics*, 12:89–105, 1993.

[16] N. Dyn, D. Levin, and S. Rippa. Data dependent triangulations for piecewise linear interpolation. *IMA Journal of Numerical Analysis*, 10:137–154, 1990.

[17] Y. Fang. An improved lawson local-optimization procedure and its application. M.A.Sc. thesis, Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada, 2018.

[18] *ISO/IEC 15444-1: Information technology—JPEG 2000 image coding system—Part 1: Core coding system*, 2000.

[19] Kodak lossless true color image suite. `http://r0k.us/graphics/kodak`, 2011.

[20] P. Li and M. D. Adams. A tuned mesh-generation strategy for image representation based on data-dependent triangulation. *IEEE Transactions on Image Processing*, 22(5):2004–2018, May 2013.

[21] X. Ma and M. D. Adams. An improved error-diffusion approach for generating mesh models of images. *Signal Processing*, 117:17–32, December 2015.

[22] A. Maglo, G. Lavoue, F. Dupont, and C. Hudelot. 3D mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys*, 47(3):44:1–44:41, February 2015.

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

[24] E. Quak and L. L. Schumaker. Least squares fitting by linear splines on data dependent triangulations. In P. J. Laurent, A. Le Mehaute, and L. L. Schumaker, editors, *Curves and Surfaces*, pages 387–390. Academic Press, Boston, MA, USA, 1991.

[25] S. Rippa. Adaptive approximation by piecewise linear polynomials on triangulations of subsets of scattered data. *SIAM Journal on Scientific and Statistical Computing*, 13(5):1123–1141, 1992.

[26] S. Rippa. Long and thin triangles can be good for linear interpolation. *SIAM Journal on Numerical Analysis*, 29(1):257–270, 1992.

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

[28] Y. Tang. Edgebreaker GitHub home page, April 2018. `https://github.com/uvic-aurora/edgebreaker`.

[29] United States Geological Survey. Shuttle radar topography mission void-filled dataset. `http://lta.cr.usgs.gov/SRTMVF`, 2017.

[30] University of Southern California. USC-SIPI image database. `http://sipi.usc.edu/database`, 2011.

[31] S. Valette. Wavemesh GitHub home page, April 2018. `https://github.com/valette/Wavemesh`.

[32] S. Valette and R. Prost. Wavelet-based progressive compression scheme for triangle meshes: wavemesh. *IEEE Trans. on Visualization and Computer Graphics*, 10(2):123–129, March 2004.

[33] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[34] X. Yu, B. S. Morse, and T. W. Sederberg. Image reconstruction using data-dependent triangulation. *IEEE Computer Graphics and Applications*, 21(3):62–68, May 2001.