A New Progressive Lossy-to-Lossless Coding Method for 2.5-D Triangle Meshes with Arbitrary Connectivity

by

Dan Han
B.Sc., University of Posts and Telecommunications, 2012

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 New Progressive Lossy-to-Lossless Coding Method for 2.5-D Triangle Meshes with Arbitrary Connectivity

by

Dan Han
B.Sc., University of Posts and Telecommunications, 2012

Supervisory Committee

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

Dr. Wu-Sheng Lu, Departmental Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

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

Dr. Wu-Sheng Lu, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

A new progressive lossy-to-lossless coding framework for 2.5-dimensional (2.5-D) triangle meshes with arbitrary connectivity is proposed by combining ideas from the previously proposed average-difference image-tree (ADIT) method and the Peng-Kuo (PK) method with several modifications. The proposed method represents the 2.5-D triangle mesh with a binary tree data structure, and codes the tree by a top-down traversal. The proposed framework contains several parameters. Many variations are tried in order to find a good choice for each parameter considering both the lossless and progressive coding performance. Based on extensive experimentation, we recommend a particular set of best choices to be used for these parameters, leading to the mesh-coding method proposed herein.

The lossless and progressive coding performance of the proposed method are evaluated by comparing with other methods, namely, the general-purpose compression algorithm Gzip, the 3-D mesh-coding method Edgebreaker, and the modified scattered data coding (MSDC) method for the 2.5-D meshes with Delaunay connectivity. The experimental results show that the proposed method outperforms Gzip with the lossless coding bit rate of the proposed method being 5 to 6 times lower than that of Gzip. Moreover, Gzip cannot achieve progressive coding functionality. The proposed method also outperforms the Edgebreaker method by using 8.1% less bits on average in terms of the lossless coding if the mesh connectivity does not deviate too far from a preferred-direction Delaunay triangulation, with the edge-flipping distance no larger than 37.38%. Here the distance 37.38% means, 37.38% of edges need to be flipped before transforming the triangulation of the original mesh to be preferred-direction Delaunay. In addition, the Edgebreaker method cannot perform progressive coding. For progressive performance, we compare the proposed method with the MSDC method by testing on the meshes with Delaunay connectivity. Since the direct comparison between different

meshes is tricky to perform, instead, we generate image approximations from the meshes and then compare the mean squared errors of the image approximations in terms of peak-signal-to-noise ratio (PSNR) metric. Therefore, the experiments measure the progressive performance using PSNR values of image reconstructions during the progressive decoding procedure. The experimental results show that the proposed method can yield image approximations of considerable higher quality in terms of PSNR than those obtained with the MSDC method. For example, in order to obtain similar-quality image approximations (i.e., with the PSNR being 75% of the maximum PSNR obtained for lossless reconstruction), the bit rate used by the proposed method is 55% to 86% of that used by the MSDC method.

During the course of the work described herein, the author discovered that the PK method cannot, in practice, handle meshes with large-valence vertices. The proposed framework provides a divide-and-conquer approach by introducing a parameter to avoid the combinatorial blowup in the PK method when handling large-valence vertices. With the partitioning scheme, the proposed method improves the previous PK method to be more practically useful. Besides the problem of large-valence vertices, the author also discovered another problem of the PK method. When the PK method updates the faces of the 3-D dataset, in certain circumstances, some extra faces can be generated in the lossless reconstructed mesh that do not exist in the original. In our work, the face information is not of concern in the 2.5-D dataset. If we consider the basic linear interpolation on the mesh, however, the proposed framework provides a method to generate the faces without having the extra-face problem.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

This thesis would never have been written without the help and support from numerous people. I would very like to take this opportunity to express my appreciation to certain individuals in particular.

First and foremost, I would like to thank my supervisor, Dr. Michael Adams. Thank you for spending so much time and effort on teaching me C++ from the basic knowledge to the higher-level complex functionalities. With all your efforts, I have developed my interests in programming, which will surely help me further into my career. Thank you for your time and patience on guiding me through the thesis writing procedure. From this procedure, I have learned a lot about academic writing. I am truly grateful for all the guidance and support you gave me throughout my graduate study.

Next, I would like to express my gratitude to my supervisory committee member, Dr. Wu-Sheng Lu. Thank you for being on my supervisory committee and spending time on reviewing my thesis. Thanks for delivering the courses of Digital Signal Processing and Engineer Design Optimization, from which I learned a lot. I also would like to thank all the other course instructors during my graduate studies, Dr. Alexandra Branzan Albu, Dr. Sue Whitesides, and Dr. Peter Driessen. Thank you for offering all the wonderful lectures, and spending so much time and effort on the preparation of the courses.

Moreover, I would like to thank my friends who have accompanied me during my life in Victoria. My best friend, Xia Meng, thank you for being there sharing the good and bad times together, and thank you for always backing me up like a family member. Thanks Yue Tang for providing her earlier experimental results for comparison in this thesis. I also would like to express my gratitude to Xiao Feng, Yue Fang, Xiao Ma, Ali Mostafavian, and Badr El Marzouki. Thank you for your inspirations, and I am grateful for being in the same research group with you. All my other friends, thanks for supporting me and sharing all the joys with me. My life here is not complete without any of you.

Furthermore, I would like to thank our fantastic faculty staffs, Moneca Bracken (retired), Janice Closson (retired), Dan Mai, Amy Rowe, Ashleigh Burns, Kevin Jones and Erik Laxdal. Thank you for providing a helpful and comfortable study environment.

Last but not least, I would like to thank my dearest family. I would like to thank my parents Nianxue Han and Guisong Kong. Thank you for being so understanding, supportive and encouraging. I am so lucky to have your unconditional endless love. My elder brother, Guolin Han, thanks for always being there and taking care of me, which makes me feel fearless.

DEDICATION

To my family.

Chapter 1

Introduction

1.1 Mesh Modeling and Mesh Coding

Bivariate functions are used extensively in a variety of scientific applications, for example, digital elevation maps in geographic information systems (GIS), images in signal processing, and range images in computer vision. One representation of bivariate functions is offered by 2.5-dimensional (2.5-D) triangle meshes. An example of a 2.5-D triangle mesh is shown in Figure 1.1. In this example, the sample points of the dataset are triangulated and the domain is partitioned into nonoverlapping triangle faces. The function value at each sample point is represented by the height of the surface above the x-y plane. The difference between a 2.5-D and a 3-D dataset is the 2.5-D has more restrictions on the data. To better illustrate this difference, a 3-D dataset with a shape of sphere is shown in Figure 1.2. We can see each point (x, y) in the 2.5-D mesh shown in Figure 1.1 only has one possible function value, but in the 3-D mesh shown in Figure 1.2, one (x, y) can have two function values. Unless explicitly mentioned as 3-D mesh, "mesh" stands for 2.5-D dataset in the context of this thesis.

In the mesh shown in Figure 1.1, the points are distributed evenly and uniformly sampled on a truncated lattice. In real-world applications, however, the information contained in the 2.5-D dataset is generally nonstationary, so uniform sampling is usually not optimal. Another sampling method called content-adaptive sampling is more practically useful. With the content-adaptive sampling, the density of the sample points usually increases in the areas of more intense variation in function values. To better illustrate this nonuniform sampling, an example of a 2.5-D triangle mesh model of an image is illustrated in Figure 1.3. The original image is shown in Figure 1.3(a) and a set of sample points with a triangulation is shown in Figure 1.3(b). From Figure 1.3(b), we can see that the density of the sample points is increased in areas with more detailed information and decreased in others. This nonuniform

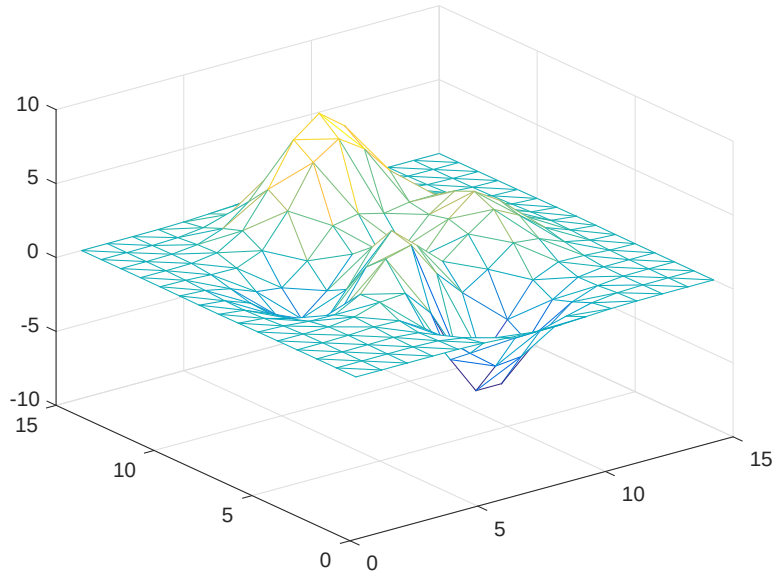


Figure 1.1: An example of a 2.5-D triangle mesh model.

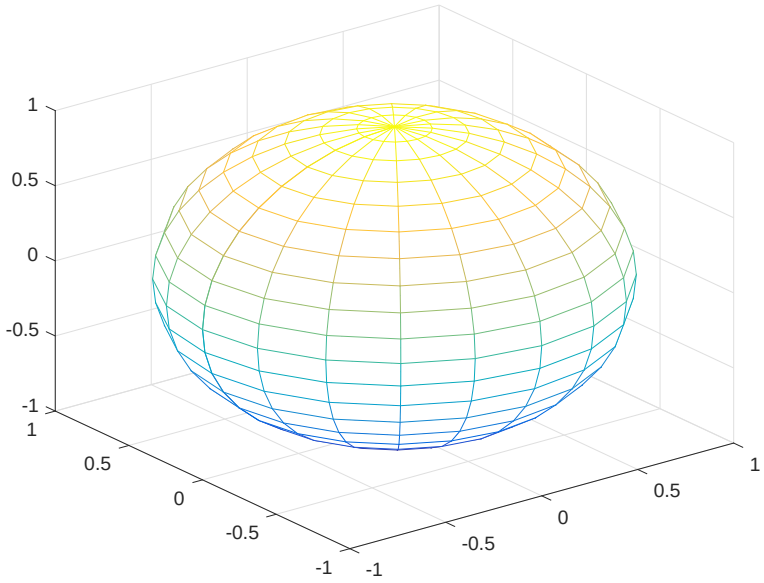
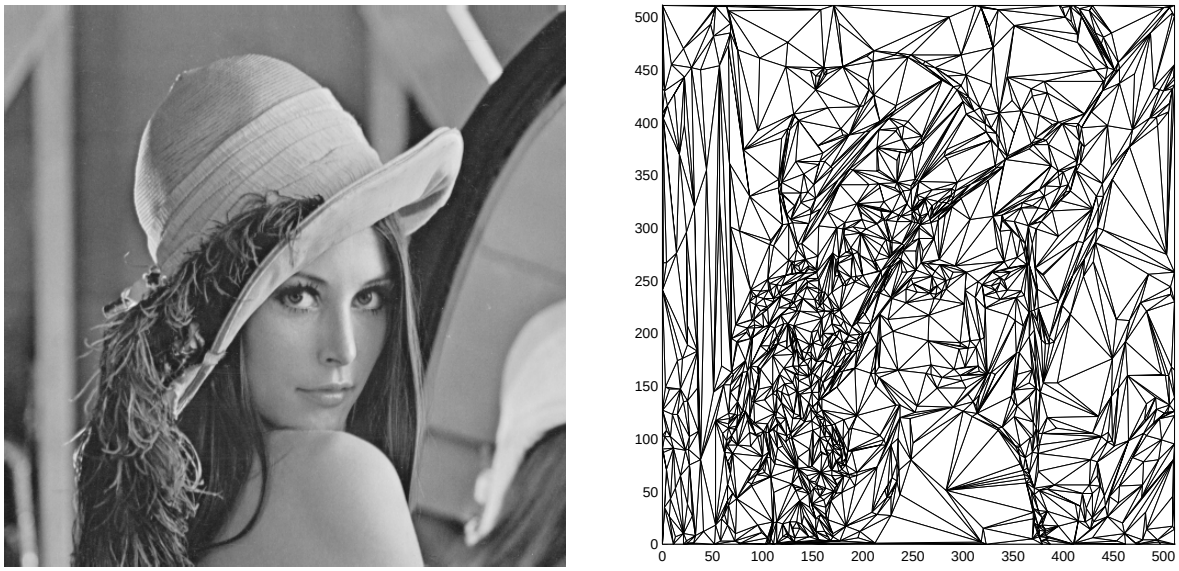


Figure 1.2: An example of a 3-D mesh model.



(a)

(b)

Figure 1.3: (a) An image, and (b) a set of nonuniformly sampled points with a triangulation on these points partitions the image domain into nonoverlapping triangles.

sampling is not only efficient by using less sample points to convey more information, but can also capture the geometric structure inherent in the images, such as edges. The nonuniform sampling has proven to be very useful in some applications such as feature detection [13], filtering [21], image/video compression [9, 10, 33, 23], and computer vision [35].

From a mathematical viewpoint, a 2.5-D triangle mesh can be described and analyzed as a triangulation over a subset of the plane, and a bivariate function defined on this subset (i.e., $z = f(x, y)$). So, the information contained in the 2.5-D mesh can be divided into the following three categories: 1) the locations (x, y) of the sample points in the subset, 2) the connectivity of a triangulation of the sample points, and 3) the function value at each sample point.

Because mesh models can be very large, we are often interested in compressing such models. Although many mesh coding methods have been proposed, they can all be classified as either lossy or lossless. If the decompressed mesh is identical to the original one, we call the coding method lossless; otherwise, the method is lossy. Lossy methods often permanently discard certain information to reduce the size of the information for storage, handling and transmitting. In many cases, however, the original and the recovered data being identical is very important, like coding executable programs, source codes, and medical images.

Because the desire for transmitting complex meshes over networks with limited bandwidth and many applications requiring real-time interaction, progressive coding has become very popular. Progressive coding methods can decode full bitstreams, or partial bitstreams if the decoding procedure is terminated in an intermediate stage. Non-progressive methods decode all of the information as a whole and cannot meaningfully decode partial bitstreams. In this thesis, our interest is to propose a coding framework that can provide progressive lossy-to-lossless coding functionality for 2.5-D triangle meshes.

1.2 Historical Perspective

Because of the growing interest in graphic 3-D data, much effort has been devoted to 3-D triangle meshes. Earlier research has proposed numerous methods for coding 3-D meshes based on the triangle strip [14], the spanning tree [38], the layered decomposition [11], the triangle conquest [22, 34], connectivity-driven compression [32, 37, 12, 26] and geometry-driven compression [20, 24, 31, 25], to name a few. An excellent survey of 3-D mesh-coding methods can be found in [30].

One of the well-known methods is **Edgebreaker** [34]. In the Edgebreaker method, the connectivity of the mesh is coded by a traversal of triangles. The Edgebreaker method is not capable of progressive coding. In this thesis, a popular progressive 3-D mesh-coding

method [31] proposed by Peng and Kuo (**PK method**) is of interest. The PK method is based on an octree decomposition. In the PK method, an octree data structure is constructed by recursive partitioning the bounding box of a mesh. Then, the mesh coder performs a top-down traversal of the octree and codes the local changes associated with each cell partitioning, including how many nonempty subcells are generated and how the mesh is updated during the partitioning.

Although 3-D mesh-coding methods could be used directly to handle 2.5-D meshes, this would be inefficient, because 2.5-D datasets have more restrictions than 3-D ones. Compression of 2.5-D meshes has been explored much less, and even less effort has been devoted to progressive coding. Some previously proposed coding methods include the scattered data coding (SDC) scheme [15] and image tree (IT) scheme [8].

The SDC scheme, proposed by Demaret and Iske in [15], views the combination of 2-D sample points and corresponding sample values as points in 3-D space, and then utilizes an octree data structure to partition the space for coding. This method has some important limitations. It can only handle 2.5-D meshes with domains that are square with integer-power-of-two dimensions and only handles the locations and function values of the sample points, assuming the underlying triangulation to have Delaunay connectivity. Since Delaunay connectivity is assumed, the method cannot handle meshes with arbitrary connectivity. Moreover, the SDC method does not provide progressive coding functionality. Later, Adams [9] proposed a modified SDC (**MSDC**) method that removes the preceding limitations, allowing progressively coding meshes with arbitrary rectangular domains. Unfortunately, like the SDC method, the MSDC scheme also cannot handle meshes with arbitrary connectivity.

Another effective method, the IT scheme, was originally proposed by Adams in [8]. This method also assumes the mesh to be coded has Delaunay connectivity. Therefore, it cannot handle meshes with arbitrary connectivity. This method uses an image tree to represent the mesh model, where this tree structure is generated based on a recursive quadtree partitioning of the image domain along with an iterative averaging process for the sample data. By efficiently coding the information in the image tree using a top-down traversal, the method can provide the progressive lossy-to-lossless functionality. Another method called average-difference image-tree (ADIT) based on the IT scheme was proposed by Adams in [10]. It uses another similar tree-based representation. The IT and ADIT methods provide a much better progressive and lossless coding performance compared to the MSDC scheme. Unfortunately, like the IT method, the ADIT scheme cannot handle meshes of arbitrary connectivity.

The meshes with arbitrary connectivity are of practical interest, since many applications have such meshes. In prior work, however, not so much effort has been devoted to effec-

tive coding methods for meshes with arbitrary connectivity. Due to the above statements, proposing a progressive lossy-to-lossless coding method for handling 2.5-D triangle meshes with arbitrary connectivity is our main focus and interest.

1.3 Overview and Contributions of the Thesis

In this thesis, we propose and develop a new mesh-coding framework for progressive lossy-to-lossless code the 2.5-D triangle meshes with arbitrary connectivity. This framework is based on ideas from the ADIT and PK methods. The framework has several associated parameters. After extensive experimentation, we recommend a set of choices for these parameters to yield our proposed method. As we will show later, the proposed method outperforms the general-purpose compression algorithm Gzip, and outperforms the Edgebreaker mesh-coding method for the meshes with connectivity close to Delaunay. Moreover, the proposed method also is superior to the Gzip and the Edgebreaker methods because the latter two cannot achieve the progressive coding. To evaluate the progressive performance of the proposed method, we compare it with the MSDC method. The progressive performance is evaluated by the image approximations generated from the reconstructed meshes, using peak-signal-to-noise ratio (PSNR) values to indicate the quality of image approximations. As we will show later, the proposed method outperforms the MSDC method by generating image approximations of substantially higher quality at lower bit rates in terms of both PSNR values and subjective image quality.

Besides the proposed framework, the thesis also makes a contribution by identifying the problems in the PK method. One problem is that the PK method becomes computationally intractable for meshes with large-valence vertices, due to combinatorial blowup. The other problem is that in certain circumstances, the face updating rules of this method will cause extra faces added to the lossless reconstructed mesh that do not exist in the original. The first problem of the above problems is addressed by a divide-and-conquer approach.

The remainder of the thesis consists of four chapters and one appendix. An overview of each of these chapters is described in what follows.

Chapter 2 presents background information necessary for understanding our work. First, we introduce some basic notation and terminology. Then several fundamentals from geometry are introduced including convex hulls, triangulations, and Delaunay triangulations. Next, a key operation that can be performed on a triangulation, called vertex split, is presented. With the preceding background, 2.5-D triangle meshes are formally defined. This is followed by some background about arithmetic coding. Finally, the average-difference transform is presented.

Chapter 3 introduces the proposed framework with several parameters, and how the proposed method is developed. To begin, we introduce a binary tree data structure used in the framework and how to use this tree to represent the 2.5-D triangle meshes. Then, we explain how to utilize this binary tree to achieve progressive coding. With the preceding background, the proposed mesh-coding framework is presented in detail for the encoding procedure. After that, we study how different choices of the free parameters in the framework influence the coding performance, leading to a set of recommended choices. Then the proposed method is finalized with these recommended choices. Since the proposed method combines ideas from two previous methods, we emphasize the differences between our method and the other two in the end.

Chapter 4 evaluates the performance of the proposed method by comparing with other methods. For lossless coding performance, the proposed method is compared with Gzip and the Edgebreaker method. For progressive coding performance, the proposed method is compared with the MSDC method. Based on the experiment results, the proposed method outperforms Gzip with the lossless bit rate of the proposed method being 5 to 6 times lower than that of Gzip. The proposed method outperforms the Edgebreaker method by using 8.1% less bits on average for the lossless coding if the mesh connectivity does not deviate too far from the preferred-directions Delaunay triangulation. The proposed method also is superior to the Gzip and Edgebreaker methods because the latter two cannot provide progressive coding functionality. Next, the proposed method is compared with the MSDC method for handling the meshes with Delaunay connectivity. The bit rate used by the proposed method is 55% to 86% of that used by the MSDC method, to achieve similar-quality image approximations with the PSNR value being 75% of the maximum PSNR obtained for lossless reconstruction. Furthermore, an extended application of the proposed method in the area of image processing is presented.

Chapter 5 gives the conclusions of the work presented herein. Moreover, some recommendations for the further research are stated in this chapter.

Appendix A describes the software used to implement the proposed framework and collect all of the experimental results. The software was fairly complex to develop, but it was designed to have a user-friendly interface. Some examples are also presented in this appendix to show how to use this software.

Chapter 2

Preliminaries

2.1 Introduction

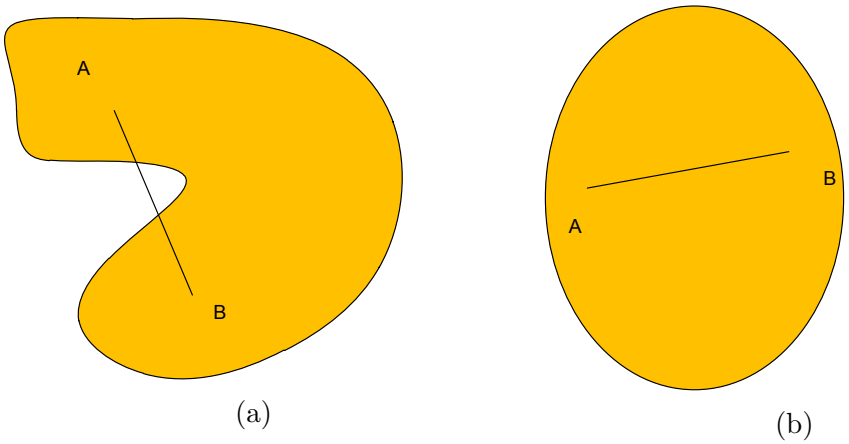
In this chapter, some background information is presented in order to facilitate the understanding of the work presented in this thesis. To begin, we present the notation and terminology used herein. Then, several concepts from geometry are introduced, followed by a description of the 2.5-D triangle meshes. Lastly, arithmetic coding and the average-difference transform are presented.

2.2 Notation and Terminology

Before proceeding further, some basic notation and terminology used throughout the thesis are introduced. The set of real numbers and integers are denoted as $\mathbb{R}$ and $\mathbb{Z}$, respectively. For $a, b \in \mathbb{R}$, we use the following notation for denoting subsets of $\mathbb{R}$: $(a, b) = \{x \in \mathbb{R} : a < x < b\}$, $[a, b) = \{x \in \mathbb{R} : a \leq x < b\}$, $(a, b] = \{x \in \mathbb{R} : a < x \leq b\}$ and $[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$. Note that, $[a, a)$ and $(a, a]$ are empty, and $[a, a]$ only contains one element a. The cardinality of a set S is denoted $|S|$. For $x \in \mathbb{R}$, we use $\lfloor x \rfloor$ to denote the largest integer smaller than x and $\lceil x \rceil$ to denote the smallest integer larger than x. For $m, n \in \mathbb{Z}$, $\mathrm{mod}\,(m, n) = m - n\lfloor m/n \rfloor$ (i.e., the remainder after dividing m by n).

The point (x_1, y_1) is said to be less than (x_2, y_2) in lexicographic order if: $(a)\, x_1 < x_2$; or $(b)\, x_1 = x_2$ and $y_1 < y_2$. The length of the line segment $e = \overline{(x_1, y_1)\,(x_2, y_2)}$ is denoted $\|e\|$ and defined as

$$\|e\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$



(a) (b)

Figure 2.1: Examples of (a) nonconvex and (b) convex sets.

2.3 Geometry

In this section, we present some important concepts in geometry such as a triangulation and Delaunay triangulation. To begin, we introduce the concept of a convex set.

Definition 1. *(Convex set). A set P of points in $\mathbb{R}^2$ is a **convex** 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 notion of a convex set, two different sets are shown in Figure 2.1. The set shown in Figure 2.1(a) is nonconvex, since the line segment $\overline{AB}$ is not completely contained in the set. In the set shown in Figure 2.1(b), we can see that for every pair of two points A and B, the segment connecting these two points must be contained in the set as well. So, the set in Figure 2.1(b) is convex. Having introduced the concept of a convex set, now we can present the notion of a convex hull.

Definition 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 is shown in Figure 2.2 to better illustrate this definition. A set P of points is given in Figure 2.2(a), and the convex hull of P is depicted in Figure 2.2(b). The boundary of the convex hull of P can also be visualized in terms of a rubber band stretched to surround all of the points in P, as illustrated in Figure 2.3. Based on the definition of the convex hull, we can now introduce the concept of a triangulation.

Definition 3. *(Triangulation). A **triangulation** of a finite set P of points in $\mathbb{R}^2$ is a set T of (non-degenerate) triangles such that:*

1. *the set of all the vertices of triangles in T is P;*

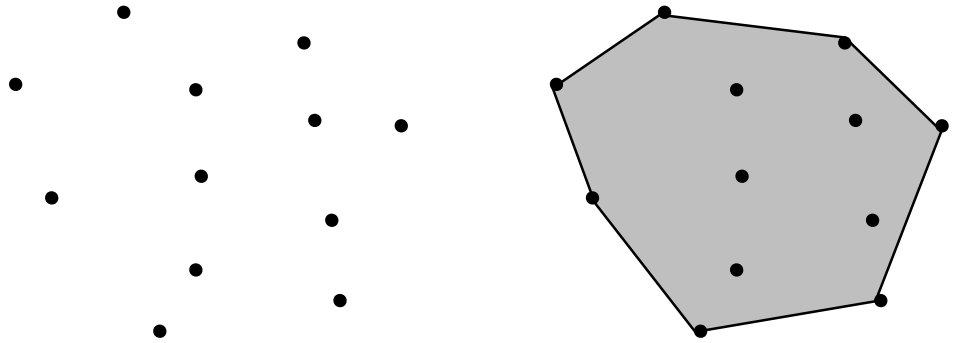


Figure 2.2: An example of a convex hull of a set of points. (a) A set P of points, and (b) the convex hull of the set P.

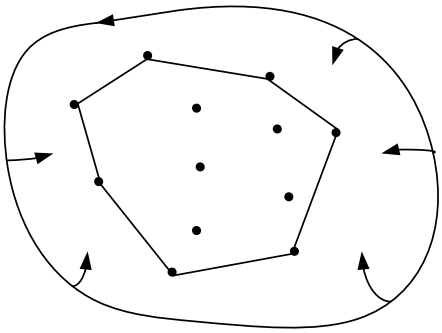


Figure 2.3: The rubber-band visualization of the convex-hull boundary.

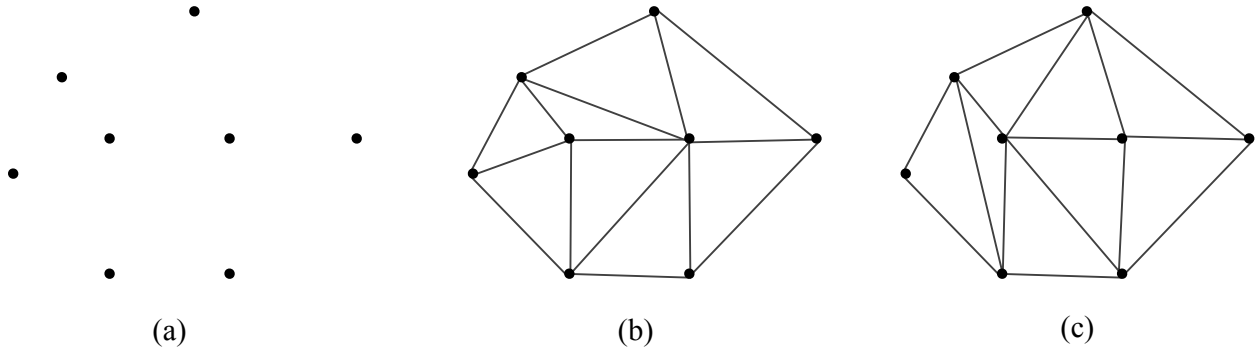


Figure 2.4: Triangulation examples. (a) A set P of points, (b) a triangulation of P, and (c) another triangulation of P.

2. *the interiors of any two triangles in T are disjoint;*

3. *the union of all triangles in T is the convex hull of P; and*

4. *every edge of a triangle in T only contains two points from P.*

For a specific set P of points, the triangulation of P is not necessarily unique. Given the set P as illustrated in Figure 2.4(a), two possible triangulations of P include those illustrated in Figures 2.4(b) and (c). We can see that the edges of the triangulation in Figure 2.4(b) are different from those in Figure 2.4(c).

One basic operation on a triangulation is an **edge flip**. In a triangulation, an edge e is called flippable if e has two incident faces and the union of these two faces is a strictly convex quadrilateral Q. To better understand this operation, an example of edge-flipping is shown in Figure 2.5. The edge e in the triangulation shown in Figure 2.5(a) is flipped to produce another edge e' in the newly generated triangulation shown in Figure 2.5(b). For the same set P of points, one triangulation T can always be transformed into another triangulation T' with a finite sequence of edge flips. In the earlier example in Figure 2.4, the triangulation in Figure 2.4(b) can be transformed into the one in Figure 2.4(c) by three edge flips.

One important type of triangulation is a Delaunay triangulation, which has a number of useful properties. Before introducing Delaunay triangulations, we first introduce the concept of a circumcircle. In geometry, the circumcircle of a triangle t is the unique circle passing through all three vertices of t. An example of a circumcircle of a triangle is shown in Figure 2.6, with the circumcircle drawn with a dashed line. With the notion of a circumcircle at hand, the definition of a Delaunay triangulation is as follows.

Definition 4. *(Delaunay Triangulation). A triangulation T of a set P of points in $\mathbb{R}^2$ is said to be Delaunay if no point in P is strictly inside the circumcircle of any triangle in T.*

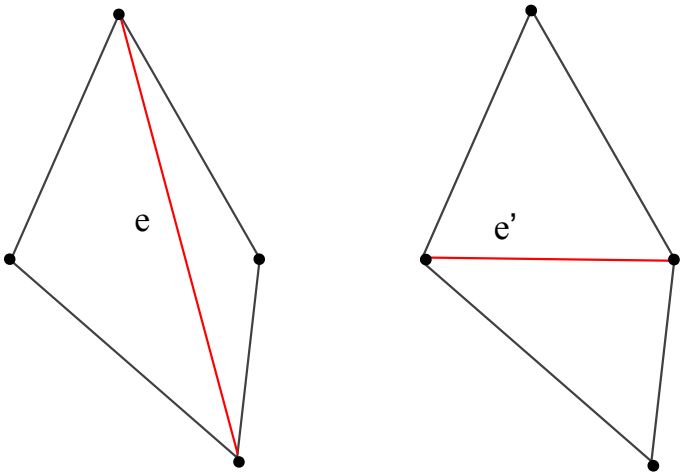


Figure 2.5: An example of an edge-flipping operation, flipping (a) edge e in one triangulation to (b) edge e' in another.

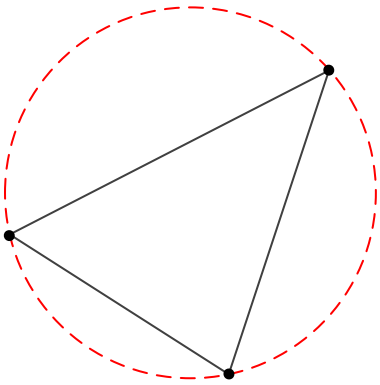


Figure 2.6: An example of a triangle and its circumcircle drawn with a dashed line.

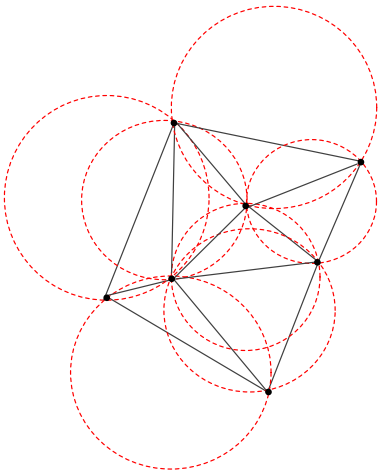


Figure 2.7: An example of a Delaunay triangulation, with the circumcircles of triangles drawn with dashed lines.

An example is shown in Figure 2.7 to better illustrate the concept of a Delaunay triangulation. In this figure, the circumcircle of each triangle is drawn with a dashed line. As is evident from this figure, each circumcircle contains no vertex of the triangulation strictly in its interior. Delaunay triangulations avoid long thin triangles to whatever extent is possible by maximizing the minimum interior angle of all triangles in the triangulation.

For a set P of points, the Delaunay triangulation of P is not guaranteed to be unique. To be specific, the Delaunay triangulation of P is only guaranteed to be unique if no four points in P are co-circular. To better illustrate the non-uniqueness issue, two different Delaunay triangulations of the same set of points are shown in Figure 2.8. A set P of points is shown in Figure 2.8(a), and two Delaunay triangulations of this set are shown in Figures 2.8(b) and (c). In practical situations, the case of having four co-circular points in the set is quite common. So, several techniques have been proposed in order to uniquely choose one Delaunay triangulation amongst all of the possibilities. These techniques include the symbolic perturbation [29, 16, 18, 28] and preferred directions methods [17]. In the preferred-directions scheme, certain rules are established to choose a preferred diagonal, based on its direction, to triangulate the quadrilateral Q generated by the four co-circular points. The unique Delaunay triangulation generated using this scheme is known as the preferred-directions Delaunay triangulation (**PDDT**).

In some applications, it is necessary for certain prescribed edges to be present in the triangulation. Such edges are said to be **constrained**. A triangulation with constrained edges is called a **constrained triangulation**. An essential concept related to constrained triangulations is the planar straight line graph (PSLG), which is defined as follows.

Definition 5. *(Planar straight line graph (PSLG)). A planar straight line graph* (P, E) *is*

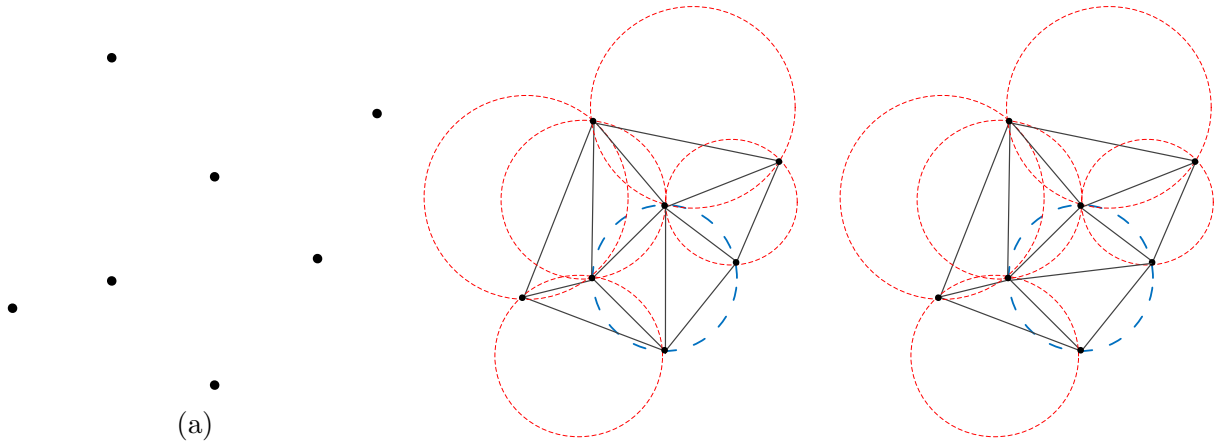


Figure 2.8: Two different Delaunay triangulations of the same set of points. (a) A set P of points, (b) a Delaunay triangulation of P, and (c) another Delaunay triangulation of P. The circumcircles of triangles in (b) and (c) are drawn with dashed lines.

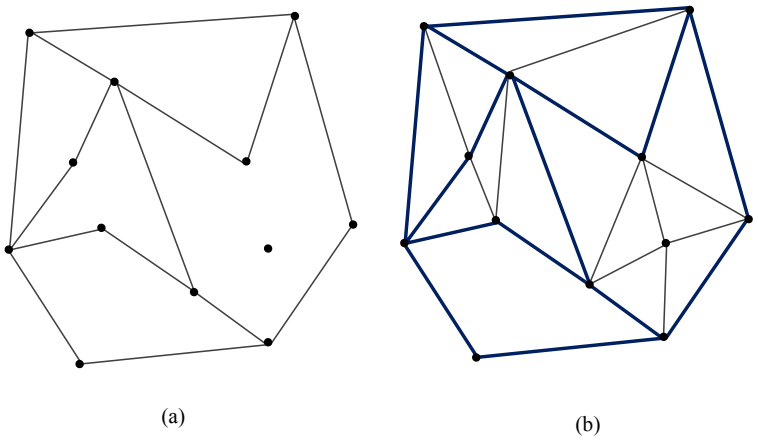


Figure 2.9: An example of (a) a PSLG and (b) a constrained triangulation of the PSLG.

a collection of a set P of points in $\mathbb{R}^2$ and a set E of line segments such that:

1. *the endpoints of each segment of E must be in P; and*

2. *any two segments of E must be disjoint or intersect at most at a common endpoint.*

To better illustrate the preceding definition, an example of a PSLG consisting of a set P of twelve points and a set E of 15 segments is shown in Figure 2.9(a). One possible constrained triangulation of this PSLG is shown in Figure 2.9(b). The constrained triangulation can be viewed as a triangulation T of P with segments in E as edges in T.

Keeping the definitions of Delaunay triangulation and constrained triangulation in mind, we can introduce the notion of a constrained Delaunay triangulation. A constrained Delaunay triangulation combines the constrained and Delaunay features, and this type of triangulation is useful in many applications. To help understand the concept of constrained Delaunay triangulation, the notion of **visibility** must first be introduced.

Definition 6. *(Visibility). Two points A and B are visible to each other in the PSLG (P, E), if and only if segment $\overline{AB}$ does not intersect the interior of any constrained edges in E.*

To better illustrate the notion of visibility, an example is given in Figure 2.10. In the PSLG in Figure 2.10(a), the point A is not visible to D, since the line segment connecting A and D will intersect the interior of the constrained edge $\overline{CM}$. Similarly, the point A is not visible to G. Any two of the other points are visible to each other. Having introduced the concept of visibility, we can now give the formal definition of a constrained Delaunay triangulation.

Definition 7. *(Constrained Delaunay triangulation). Given a PSLG (P,E), a triangulation T of P is said to be constrained Delaunay if each triangle t in T is such that: 1) the interior of t does not intersect any constrained edges in E; and 2) no vertex inside the circumcircle of t is visible from the interior of t.*

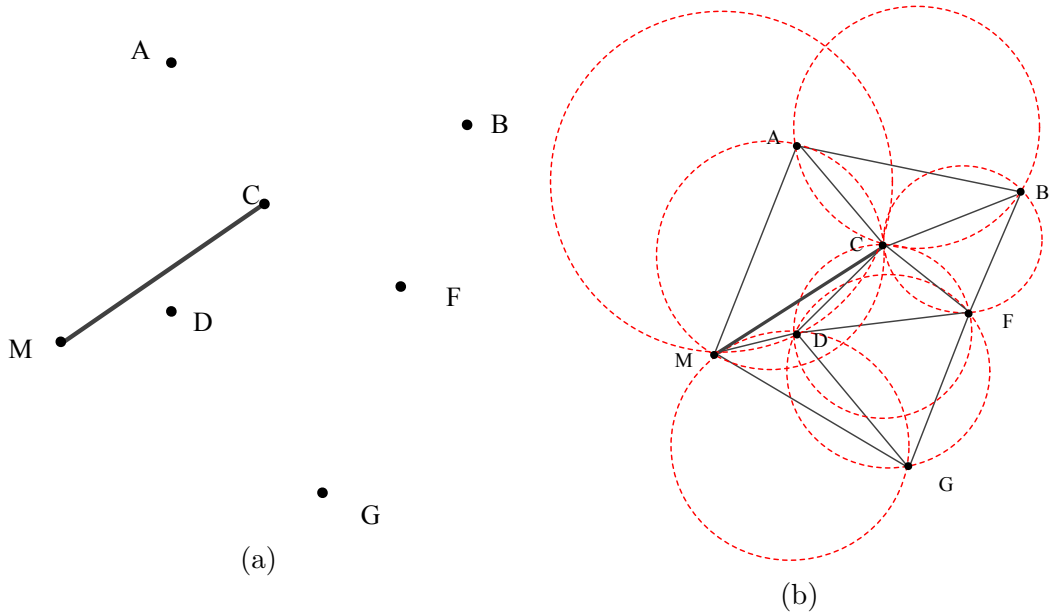


Figure 2.10: Constrained Delaunay triangulation example. (a) A PSLG (P, E) containing a set P of points (where $P = \{A, B, C, D, F, G, M\}$)and a set E of one segment (where $E = \{\overline{CM}\}$), and (b) the constrained Delaunay triangulation T of (P, E), with the circumcircles of triangles in T drawn using dashed lines.

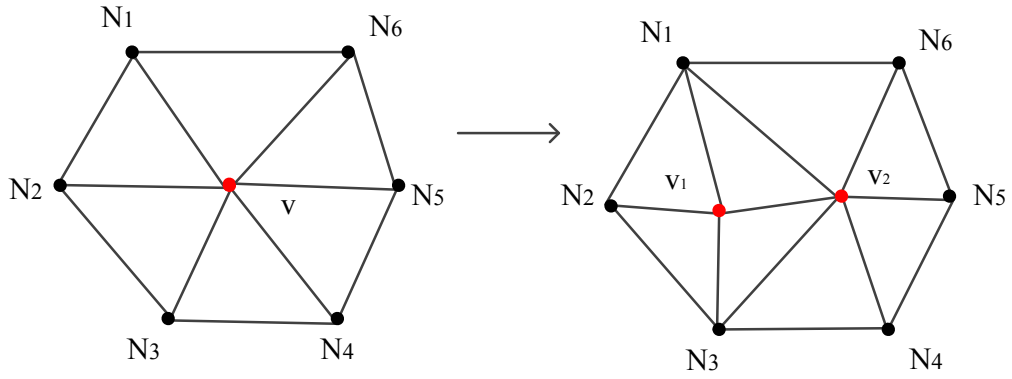


Figure 2.11: An example of vertex split. Vertex v is split into two new vertices v_1 and v_2.

To better illustrate the notion of a constrained Delaunay triangulation, an example is given in Figure 2.10. The PSLG shown in Figure 2.10(a) has the constrained Delaunay triangulation shown in Figure 2.10(b) . In this constrained Delaunay triangulation, the point A is inside the circumcircle of the triangle $\triangle CDM$, but A is not visible to the interior of the triangle, because any segment that connects A and one interior point inside $\triangle CDM$ would intersect the constraint $\overline{CM}$. The point D is inside the circumcircle of the triangle $\triangle ACM$, but similarly, D is not visible to the interior of the triangle either. So the triangulation is constrained Delaunay.

2.4 Vertex Split

Having introduced triangulations, we now discuss an operation that can be performed on triangulations, called a vertex split. In a vertex split, a vertex v in a triangulation with neighbors $\{N_i\}$, is split into two new vertices v_1 and v_2. Thus, a vertex split increases the number of vertices in a triangulation by one. The connectivity changes associated with a vertex split are characterized by:

- whether each N_i is connected to v_1, or v_2, or both v_1 and v_2; and
- whether the new vertices v_1 and v_2 are connected to each other.

An example of vertex split is illustrated in Figure 2.11. In this example, v is the original vertex and has neighbors $\{N_1, N_2, \ldots, N_6\}$, and v_1, v_2 are the new vertices generated from this vertex split. After splitting, the updated connectivity is as follows:

1. N_2 is connected to v_1;
2. N_4, N_5, and N_6 are connected to v_2;

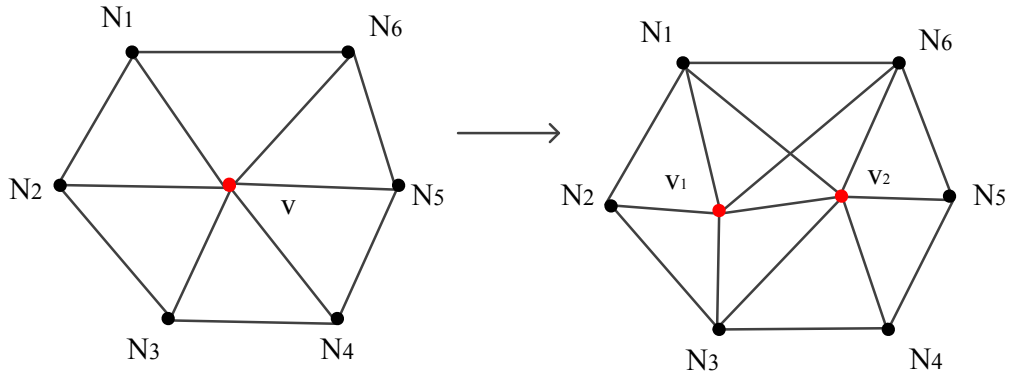


Figure 2.12: An example of vertex split leading to invalid triangulation. (a) Before vertex split and (b) after vertex split.

3. N_1 and N_3 are connected to both v_1 and v_2; and

4. the new vertices v_1 and v_2 are also connected to each other.

Note that, the inverse of a vertex split is called vertex merge, which combines two vertices into one. All neighbors of the previous two vertices are connected to the new vertex after the merge.

As mentioned above, vertex splits can be used to add new vertices to a triangulation. In some situations, however, a vertex split can result in an invalid triangulation. To better illustrate this problem, another example of vertex split is shown in Figure 2.12. The vertex v in Figure 2.12(a) is split into v_1 and v_2 in Figure 2.12(b) with N_1 and N_6 being connected to both two new vertices. This results in the edges $\overline{N_1 v_2}$ and $\overline{N_6 v_1}$ intersecting, which is not valid for a triangulation. Therefore, in this example, the vertex split has led to a triangulation with invalid connectivity.

2.5 2.5-D Triangle Mesh Models

At this point, we now formally introduce 2.5-D triangle meshes. Consider an integer-valued function ϕ defined on $D = [0, W-1] \times [0, H-1]$ and sampled on the integer lattice $S = \{0, 1, 2, \ldots, W-1\} \times \{0, 1, 2, \ldots, H-1\}$ (i.e., a rectangular grid of width W and height H). In the context of this thesis, a 2.5-D triangle mesh is characterized by:

1. a set $P = \{p_i\}$ of sample points, where $P \subset S$ (i.e., geometry information);

2. a triangulation of P (i.e., connectivity information); and

3. a set $Z = \{z_i\}_{i=0}^{|P|-1}$ of function values where $z_i = \phi(p_i)$ (i.e., function value information).

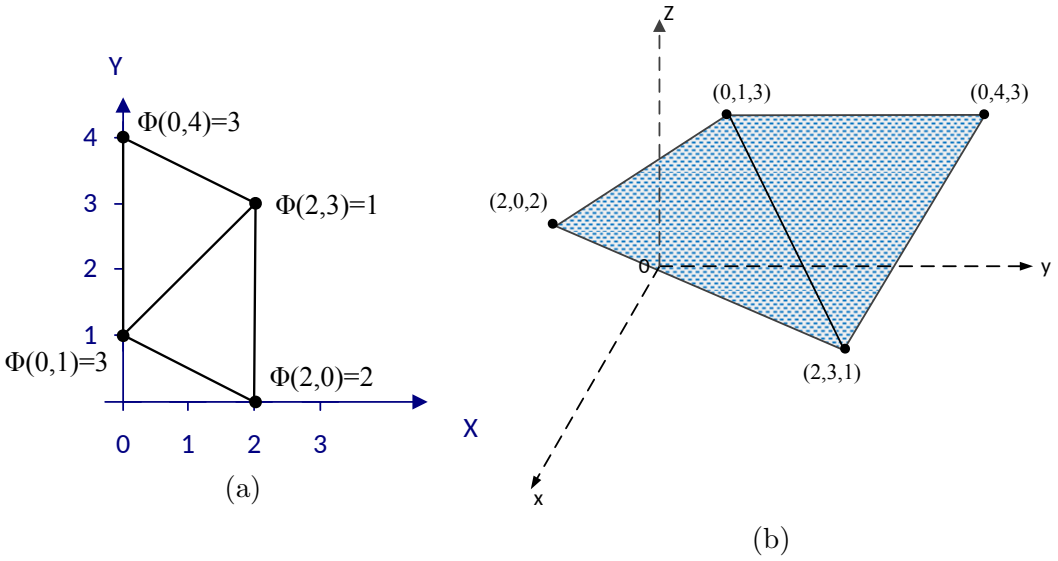


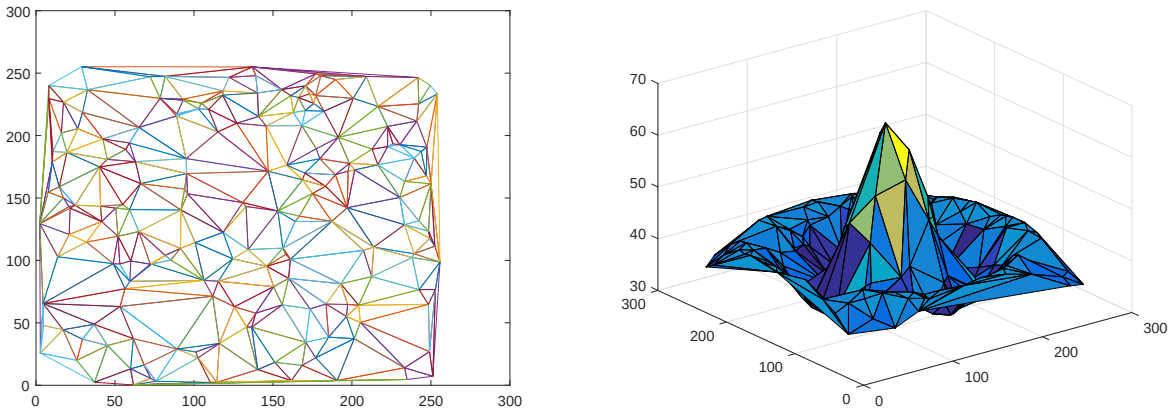
Figure 2.13: An example of a 2.5-D triangle mesh with four sample points. (a) The triangulation and its associated sampled function values, and (b) the surface obtained from piecewise-linear interpolation. Here z-axis represents the function value of $\tilde{\phi}$.

To generate a function $\tilde{\phi}$ defined on the entire domain D (and not just at lattice points in S), the function values Z are used in conjunction with linear interpolation. All of the preceding information needs to be coded in mesh-coding applications.

An example illustrating a 2.5-D triangle mesh is shown in Figure 2.13. In Figure 2.13(a), the set P contains the four points $\{(0,1),(2,0),(0,4),(2,3)\}$ and is triangulated to form two triangles. By applying linear interpolation over each face of the triangulation, a surface $\tilde{\phi}$ is obtained, as shown in Figure 2.13(b).

Although 2.5-D meshes can be used to represent many types of data, this thesis is most interested in using the meshes to model images. We can measure the difference between two meshes by measuring the differences of the corresponding vertices, which is tricky to do. Another way to measure the difference of meshes is to measure the difference between the image reconstructions produced by the meshes. The function values in the original image are integers. So, the image approximation function $\hat{\phi}$ that used to approximate the original function value also needs to be integer-valued, which can be calculated by rounding the non-integer function values of $\tilde{\phi}$ to the nearest integers: $\hat{\phi} = \text{round}\left(\tilde{\phi}\right)$. The approximation function can be generated from the reconstructed mesh using standard rasterization techniques [19].

A 2.5-D triangle mesh model of an image is shown in Figure 2.14. In this example, the original triangulation of the image domain is shown in Figure 2.14(a), with the vertices of the triangulation being the sample points. The mesh model with the piecewise-linear interpolation function $(\tilde{\phi})$ is shown in Figure 2.14(b), from which we can generate an image



(a) (b)

Figure 2.14: An example of mesh model of image. (a) The original triangulation of the image domain and (b) 2.5-D triangle mesh model with the associate piecewise-linear function.

approximation $\hat{\phi}$. To measure the quality of the reconstructed images, we use the **mean squared error (MSE)** to denote the difference between the original image (ϕ) and the approximation ($\hat{\phi}$), using the formula given by

$$\mathsf{MSE} = |P|^{-1} \sum_{p \in P} \left(\phi\,(p) - \hat{\phi}\,(p) \right)^2, \tag{2.1}$$

where P is the set of sample points. In the intermediate stage, the smaller MSE value means the better quality of the reconstructed image.

Generally, another quantity called **peak signal-to-noise ratio (PSNR)** is used instead of MSE for convenience:

$$\mathsf{PSNR} = 20\log_{10} \frac{(2^{\rho} - 1)}{\sqrt{\mathsf{MSE}}}, \tag{2.2}$$

where ρ is the number of bits per function value used by the image ϕ. A larger PSNR value corresponds to a lower MSE.

2.6 Arithmetic Coding

Next, we provide some background that relates to data compression. Of particular interest is a technique known as binary arithmetic coding. Binary arithmetic coding maps a message consisting of binary symbols to a real number n in the interval $[0, 1)$. For the arithmetic encoding, the interval is initially chosen at $[0, 1)$. The current interval is partitioned into two subintervals based on the probabilities of the symbols. Then the current interval is recursively updated to one of the subintervals based on the encoded symbol. After all symbols are encoded, any number in the final interval is sufficient to decode original message, which is sent to the decoder side. The decoder also has the initial interval as $[0, 1)$. The partitioning on the current interval is the same as the encoder. Depending on the value of n from the encoder, the interval is updated to one of the subintervals. Meanwhile, the decoder outputs the corresponding symbol for this subinterval. The decoder terminates by receiving a terminate symbol or after a certain number of symbols are decoded. The probability distribution of symbols is called a **context**. The coder is called **context-based** if the context used in the arithmetic coding procedure is selected based on the contextual information, instead of always being the same one. Moreover, if the probability values in the context can be updated based on the coded symbol, the arithmetic coder is called **adaptive**. A binary symbol is said to be encoded in **bypass mode** if the two possible symbols 0 and 1 have the same fixed probability of 0.5.

To better illustrate how the arithmetic coding works, we consider an example of coding

Table 2.1: A probability distribution and starting intervals for symbols $\{0, 1\}$

Symbol	Probability	Starting Interval
0	0.4	[0, 0.4)
1	0.6	[0.4, 1.0)

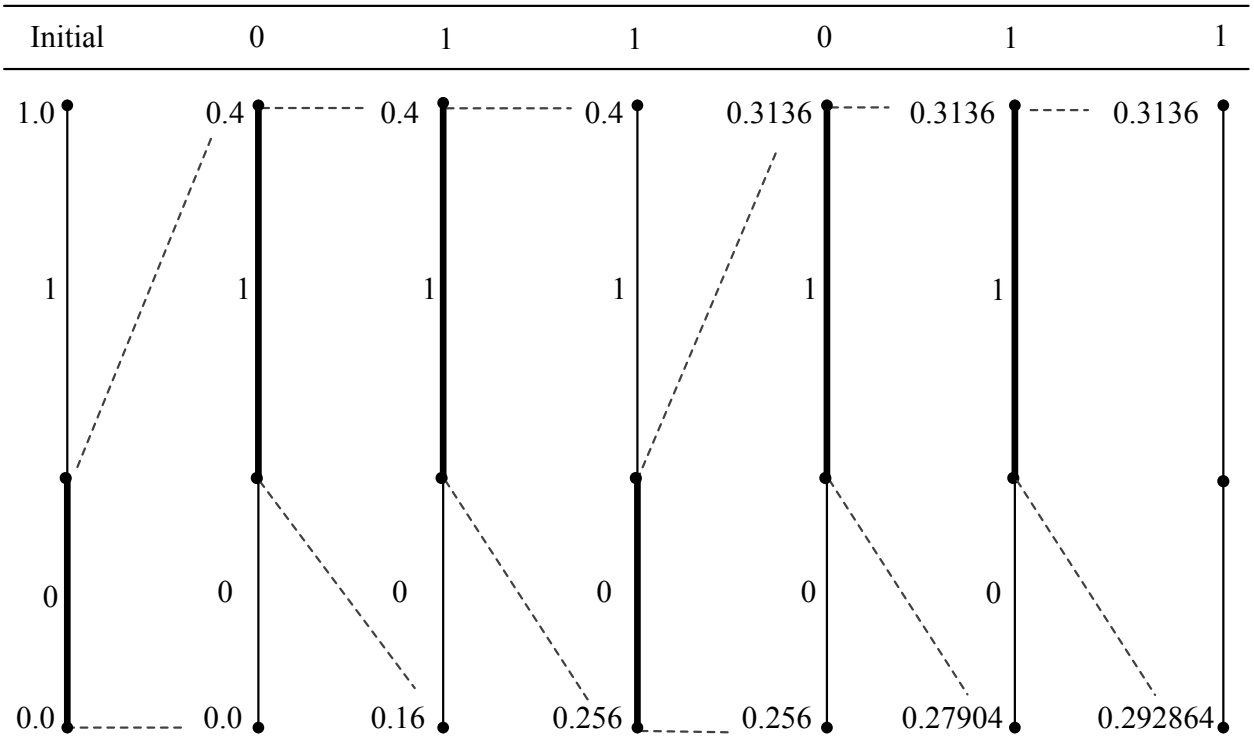


Figure 2.15: Graphic representation of the arithmetic encoding procedure for a particular message $\{0, 1, 1, 0, 1, 1\}$.

of the message $\{0, 1, 1, 0, 1, 1\}$ taken from the binary alphabet $\{0, 1\}$. The probability of each of the symbols is shown in Table 2.1. In order to avoid unnecessary complicating the explanation of arithmetic coding, we only consider how arithmetic coding works with infinite-precision arithmetic, which is sufficient for our needs in the thesis.

To begin, we present the encoding procedure, which is illustrated in Figure 2.15. The encoder starts with the interval $[0, 1)$, and then divides the interval into subintervals based on the probabilities of the symbols. The range will be adjusted based on the symbol encoded. First, after the symbol 0 is encoded, the current interval is set to the first subinterval $[0, 0.4)$, which corresponds to the symbol 0. Then the new current interval is divided into two subintervals $[0, 0.16)$ and $[0.16, 0.4)$ for the symbols 0 and 1, respectively. Since the second symbol is 1, after coding 1, the current interval is set to $[0.16, 0.4)$. The next symbol is 1, and the interval is narrowed further to $[0.256, 0.4)$. Similarly, the encoding procedure continues for the following symbols 0, 1 and 1. After the last symbol is successfully encoded, the final

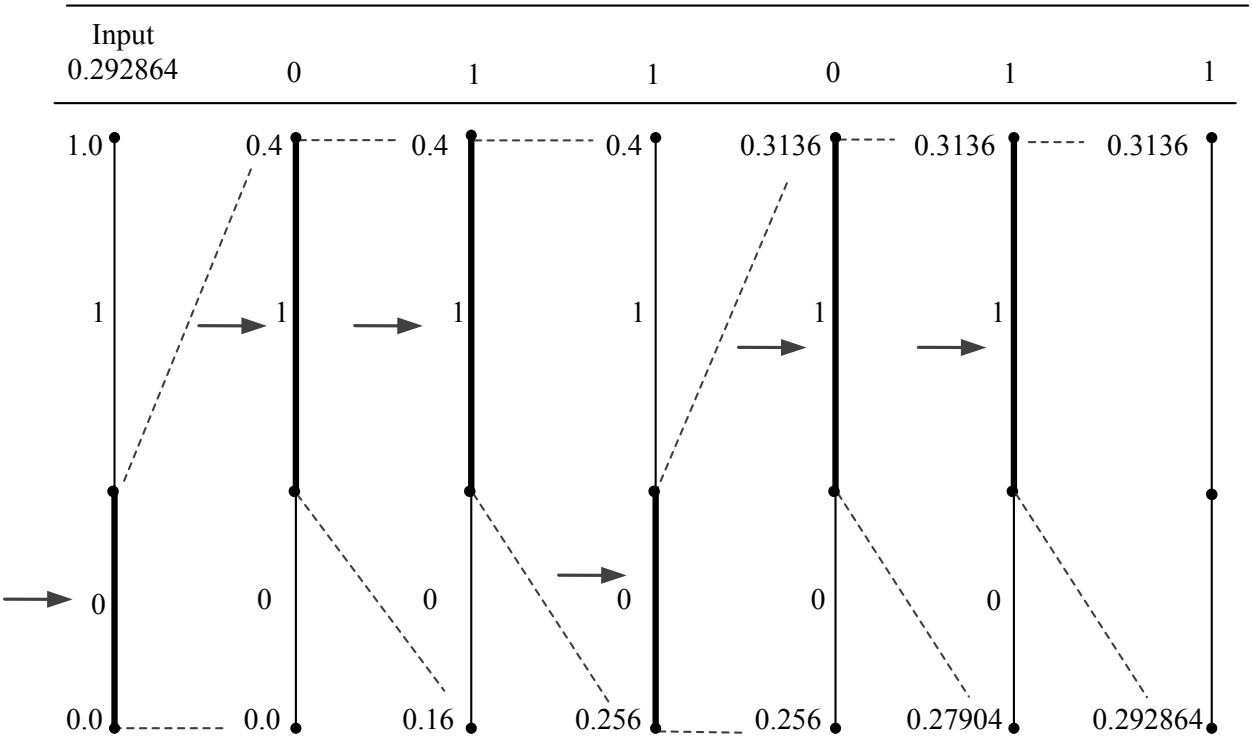


Figure 2.16: Graphic representation of the arithmetic decoding procedure with the input as 0.292864.

interval to represent the message becomes $[0.292864, 0.3136)$. The decoder only needs one number to indicate the range, so we choose the lower bound (i.e., 0.292864) and transmit this value to the decoder.

Now, we consider decoding. Besides the lower bound value, the decoder also needs to know that six symbols are encoded. A graphic representation of the decoding procedure is illustrated in Figure 2.16. With the input result as 0.292864, the decoding procedure is started with the interval $[0, 1)$. The number 0.292864 is contained in the subinterval $[0, 0.4)$. So we know that the first symbol decoded is 0. After decoding 0, the current interval is set to $[0, 0.4)$. In the current interval, two subintervals $[0, 0.16)$ and $[0.16, 0.4)$ are generated based on the symbol probabilities. The number 0.292864 is contained in the second subinterval. So, the second decoded symbol should be 1. The new range is narrowed to $[0.16, 0.4)$. Repeating a similar procedure, we can retrieve later symbols as 1, 0, 1, and 1. After the sixth symbol is successfully recovered, the decoding procedure is terminated.

In some applications, the need may arise to code symbols from a nonbinary alphabet. A binary arithmetic coder, however, can only handle binary alphabet. Therefore, if a binary arithmetic coder is to be used to code symbols from a nonbinary alphabet, such symbols must first be translated into a sequence of binary symbols. Such a process is known as

binarization.

An example of a binarization process is given in what follows. Suppose that each symbol we need to code has the value being one of the four possibilities $\{0, 1, 2, 3\}$. Then, we can represent the four-value symbols using two binary bits $\{00, 01, 10, 11\}$ in order to cover the four possibilities. For example, the sequence of the four-value symbols $\{1, 2, 3\}$ can be represented as $\{01, 10, 11\}$. So instead of coding the previous $\{1, 2, 3\}$, we encode the sequence $\{0, 1, 1, 0, 1, 1\}$ using a binary coder. The encoding and decoding procedures are exactly the same as the preceding binary example. Other complex nonbinary symbols may use less straightforward binarization schemes. The coder used in the proposed coding method is context-adaptive binary arithmetic coder.

2.7 Average-Difference Transform

As we will see later, our work makes use of a transformation known as the average-difference (AD) transform. The AD transform is a two-point transform used in the ADIT method (introduced earlier). The transform maps two integers x_0 and x_1 into two integers y_0 and y_1, as given by

$$y_0 = \left\lfloor \frac{1}{2}(x_0 + x_1) \right\rfloor \quad \text{and} \quad y_1 = x_1 - x_0.$$

Observe that, y_0 is the approximate average of x_0 and x_1, and y_1 is the difference between x_0 and x_1. If x_0 and x_1 are n-bit integers, y_0 and y_1 can be represented using n and $(n + 1)$ bits, respectively. The corresponding inverse transform is given by

$$x_0 = y_0 - \left\lfloor \frac{1}{2} y_1 \right\rfloor \quad \text{and} \quad x_1 = y_1 + x_0.$$

Chapter 3

Proposed Mesh-Coding Method and Its Development

3.1 Introduction

In this chapter, a new progressive lossy-to-lossless coding framework is proposed for 2.5-D triangle meshes with arbitrary connectivity. This framework borrows ideas from the ADIT [10] and PK [31] methods, and uses a tree-based structure to represent the 2.5-D dataset and codes the information in the tree with a top-down traversal.

To begin, we present the tree data structure and describe how to use the tree to store all the information of the mesh. Next, we explain how to use the tree to achieve the progressive coding functionality. With the introduction of the preceding knowledge, we describe the coding framework with several free parameters. After that, the selection of the parameters is discussed in conjunction with experimental results. Then, the proposed method is finalized with a particular recommended set of choices for these parameters. At last, the differences between the proposed method and the other two methods (i.e., the ADIT and PK methods), upon which our work is based, are emphasized.

3.2 Cell Bi-Partitioning Tree-Based Representation of 2.5-D Triangle Meshes

The data structure used in the proposed framework, called a cell bi-partitioning tree (**cbp-tree**), is a binary tree based on spatial partitioning. The cbp-tree is utilized to capture all the information of the mesh, including the geometry, connectivity, and the function values. To begin, we describe how to generate the tree with the information of geometry and function

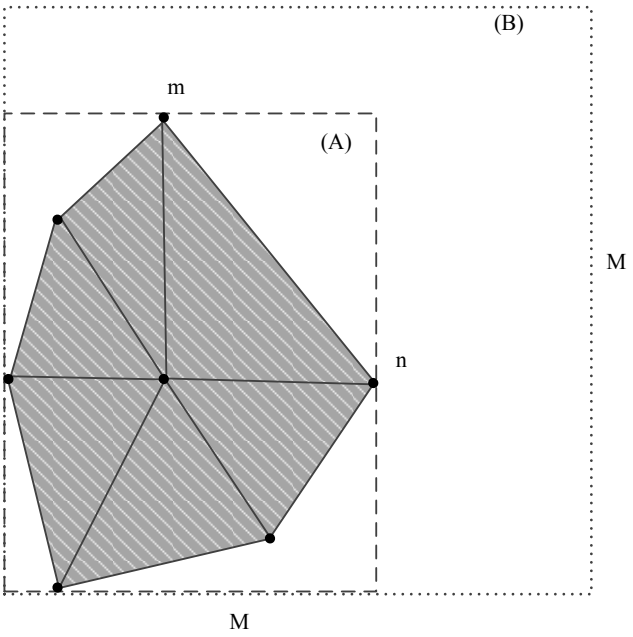


Figure 3.1: An example of root cells for different schemes. The gray area is conv (P), where P is the set of sample points. Dashed lines (A) and (B) represent the root cells under unpadded and padded schemes, respectively.

values. Then, we explain how the connectivity information is stored in the tree.

The cbp-tree is generated by recursively bi-partitioning an initial cell, which contains all the sample points of the original mesh. The initial cell for partitioning is called the **root cell**. Two schemes are provided for selecting the root cell:

1. Unpadded scheme: The root cell is chosen as the smallest isorectangle containing conv (P) (i.e., the convex hull of the set P of sample points).

2. Padded scheme: The root cell is chosen as the smallest isorectangle containing conv (P) that also has dimensions that are equal and integer powers of two.

To better illustrate the difference between unpadded and padded schemes, an example of root cells under different schemes is illustrated in Figure 3.1. In this figure, the gray area is conv (P), the dashed line (A) represents the unpadded root cell with the size $m \times n$ and line (B) represents the padded root cell with the size $M \times M$ and contains (A). Here M is the smallest integer power-of-two that is no smaller than m or n. A free parameter of the framework is used to select different schemes, and we will see how to choose it later.

With the root cell, we can generate the cbp-tree by splitting the root cell through the approximate midpoints along the x and y-axes recursively. Note that, a cell is said to be **empty** if it contains no sample points, and is called **degenerate** if it has zero area. Empty cells are not split further. In particular, a given nonempty cell $C = [x_1, x_2) \times [y_1, y_2)$ is first

split along the x-axis through the approximate midpoint x_m to yield two child cells C_1 and C_2 given by

$$C_1 = [x_1, x_m) \times [y_1, y_2) \quad \text{and} \quad C_2 = [x_m, x_2) \times [y_1, y_2)\,,$$

where $x_m = \lfloor (1/2)\,(x_1 + x_2) \rfloor$. If C_1 and C_2 are not empty, they are split further along the y-axis through the approximate midpoint y_m, where $y_m = \lfloor (1/2)\,(y_1 + y_2) \rfloor$. So together four new cells C_{11}, C_{12}, C_{21}, and C_{22} are generated and given by

$$C_{11} = [x_1, x_m) \times [y_1, y_m)\,, \quad C_{12} = [x_1, x_m) \times [y_m, y_2)\,,$$

$$C_{21} = [x_m, x_2) \times [y_1, y_m)\,, \quad \text{and} \quad C_{22} = [x_m, x_2) \times [y_m, y_2)\,.$$

We provide an example to illustrate the cell bi-partitioning process in Figure 3.2(a). Suppose that the root cell in this example has the size 4×4. First, the root cell is split along the x-axis to yield two nonempty child cells. Then the two child cells are split along the y-axis. Note that, the upper-right cell in the subfigure (III) of Figure 3.2(a) is empty and not split further. The resulting nonempty cells are split along the x-axis again to yield new child cells. The partitioning procedure stops when each nonempty cell contains only one sample point and has an area of one as shown in the subfigure (V) of Figure 3.2(a). Each cell with an area larger than one can generate at most two or one nonempty child cells during the bi-partitioning. The latter case only happens when the root cell does not have a equal power-of-two dimensions.

The above recursive cell partitioning procedure generates a cbp-tree. To be specific, each node in the cbp-tree is associated with a nonempty cell. The cbp-tree is generated from a single root node, which corresponds to the root cell. Each time when a cell C is split to yield two nonempty child cells, two child nodes are added to the corresponding node Q. If C only has one nonempty child cell, Q has one child node added as well. To better illustrate the tree construction, an example of a cbp-tree is shown in Figure 3.2(b) and is related to the recursive cell partitionings in Figure 3.2(a). As can be seen, different numbers $\{(1), (2), \ldots, (10)\}$ are labeled in Figure 3.2(b), which are related to the corresponding cell partitionings labeled with the same numbers in Figure 3.2(a). For example, for the bi-partitioning labeled with "(3)", the cell is split to yield only one nonempty child cell. So the corresponding tree node has one child node added. From Figure 3.2(b), we can see the tree is fully generated when the recursive partitioning procedure is finished and each leaf node in the tree has a one-on-one correspondence with an original sample point.

After the tree is fully generated, besides the nonempty cell, each node is also associated with the following information:

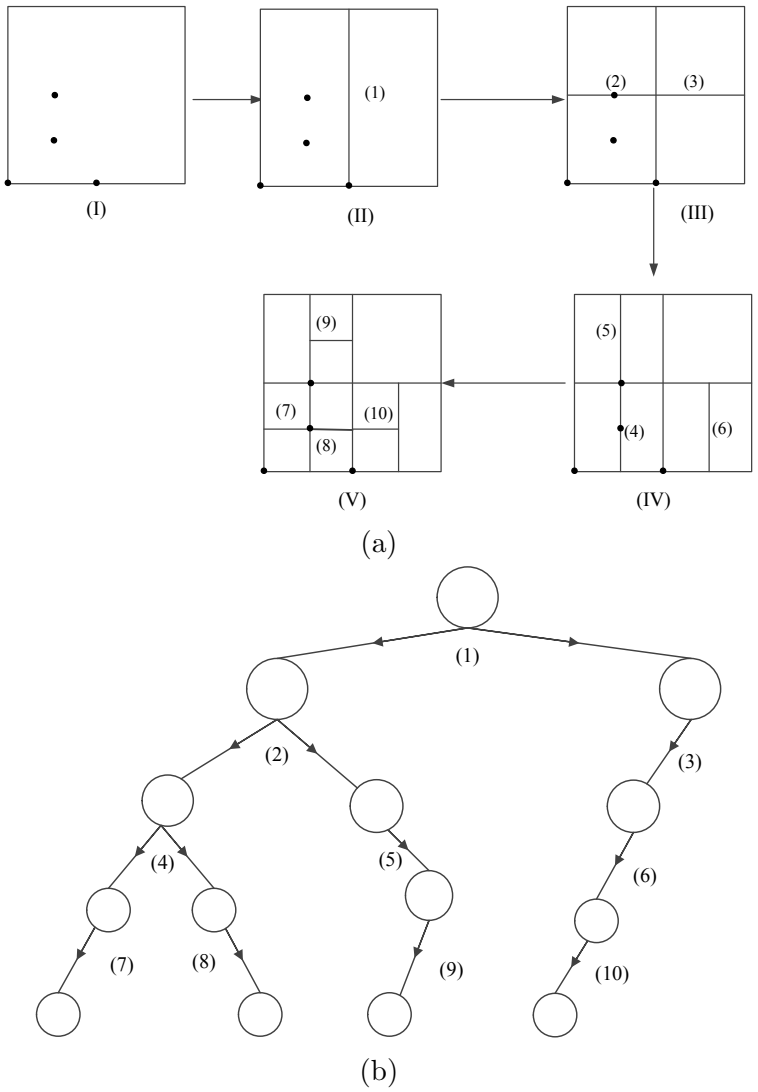


Figure 3.2: An example of (a) a recursive cell partitioning procedure, and (b) the corresponding cbp-tree structure. The labels $\{(1), (2), \ldots, (10)\}$ have one-on-one correspondence in (a) and (b).

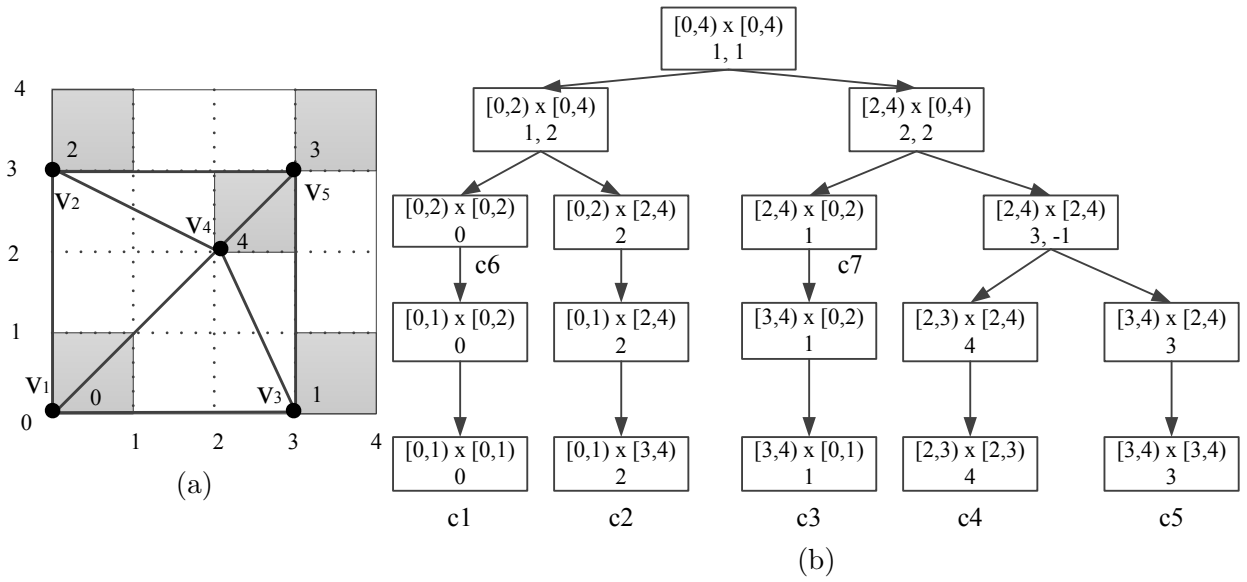


Figure 3.3: An example of a cbp-tree. (a) A mesh with five sample points, and (b) its corresponding cbp-tree representation showing only geometry and function value information.

1. an approximation coefficient;
2. zero or one detail coefficient; and
3. a representative vertex.

For a leaf node, the approximation coefficient is the function value of the single sample point contained in the node's cell, and the representative vertex is chosen as this sample point. For a nonleaf node, the approximation coefficient is the approximate average function value of all samples contained in the cell and the representative vertex is located at the approximate centroid of the cell. A node only has a detail coefficient if it has two children, calculated as the difference between the approximation coefficients of the two child nodes. An example of a cbp-tree is shown in Figure 3.3(b), with the corresponding dataset shown in Figure 3.3(a). The mesh in Figure 3.3(a) consists of five sample points, a triangulation, and a set of function values for these sample points. Each leaf node in Figure 3.3(b) corresponds to a unit cell (i.e., the shaded area in Figure 3.3(a)) containing exactly one integer lattice point which is a sample point. Furthermore, each node is labeled with its associated cell, approximation coefficient, and detail coefficient if any.

In order to capture the geometry information and the function values of the mesh in the cbp-tree without redundancy, we need to specify the following:

1. the width and height of the root cell;
2. the configuration of each node (i.e., how many and which child cells are nonempty);

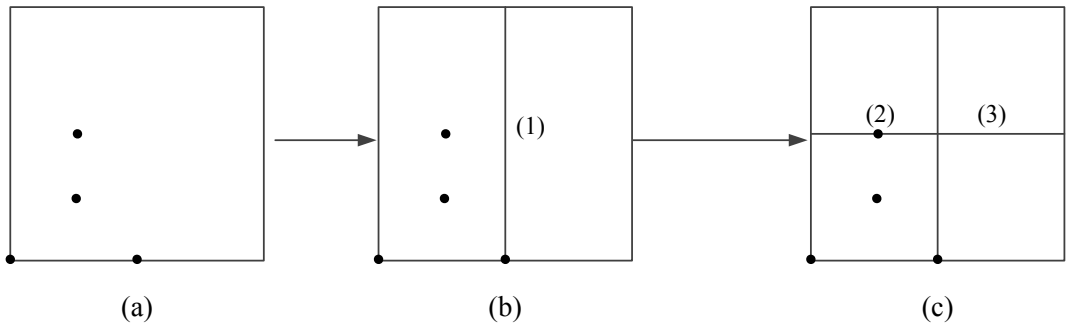


Figure 3.4: An example of a QCP containing three CCPs. (a) Original cell to split. (b) The first CCP along x-axis, and (c) the other two CCPs along y-axis.

3. the approximation coefficient a_r of the root node; and

4. the detail coefficient (**DC**) if any, of each node.

Note that, the approximation coefficients for the non-root nodes are not required, because they can be determined from the inverse AD-transform as described in Section 2.7 (on page 24) using a_r and the appropriate DCs.

Along the cell bi-partitionings, the cbp-tree is generated with the geometry and function values of the mesh contained in the tree nodes. Now, we consider the connectivity information associated with the mesh. We utilize the property that the leaf nodes in the fully generated cbp-tree have a one-on-one correspondence with the actual sample points in the mesh. Since the sample points are connected in a certain way in the original mesh, we can store the same connectivity information to the leaf nodes. For the node cells in the cbp-tree, two cells c_i and c_j are said to be neighbors if at least one edge exists in the original mesh with one endpoint in c_i and the other endpoint in c_j. Returning to the example from Figure 3.3, we can see v_1 is connected to v_2, v_3, and v_4 in Figure 3.3(a). Therefore, in Figure 3.3(b), the cell c_1 is a neighbor to c_2, c_3, and c_4 based on the mesh connectivity. The cells $c_6 = [0, 2) \times [0, 2)$ and $c_7 = [2, 4) \times [0, 2)$ are neighbors to each other, since one edge in the original mesh in Figure 3.3(a) has two endpoints $(0, 0)$ and $(3, 0)$ in cell c_6 and c_7, respectively.

Note that, the previous cbp-tree cell bi-partitioning operations, denoted as CCP, can be viewed in another perspective if we collapse two-level operations into one. The new perspective is a quadtree cell bi-partitioning, denoted as QCP. In particular, a QCP operation starts with bi-partitioning a cell along the x-axis, and then all resulting nonempty child cells are bi-partitioned along the y-axis. Therefore, a QCP on a nonleaf cell contains at most three CCPs. An example of a QCP is illustrated in Figure 3.4 to better understand this operation. In this example, the original cell in Figure 3.4(a) is first split along the x-axis and generates two nonempty child cells in Figure 3.4(b). Then, each of the two cells in Figure 3.4(b) are

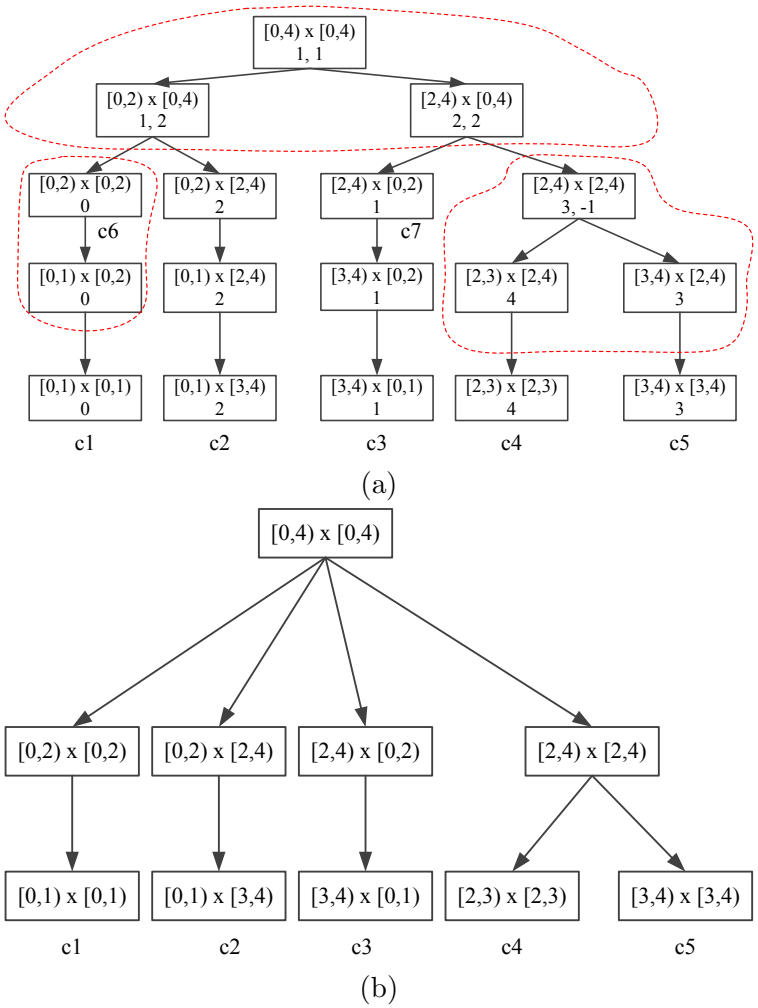


Figure 3.5: (a) Redraw the previous cbp-tree in Figure 3.3(b) with the dashed lines grouping the CCPs into QCPs, and (b) the new representation with the QCP operations.

split further along the y-axis in Figure 3.4(c). As can be seen from Figure 3.4, the QCP contains three CCPs, which are labeled with (1), (2), and (3). As stated before, a CCP on a nonleaf cell can generate at most two or one nonempty child cells. Therefore, a QCP on a nonleaf cell can have at most either four or two subcells, and the latter case only happens when the root cell does not have a square power-of-two dimension. Here the new cells generated from a QCP are called "subcells" in order to be distinguished from "child cells" generated from a CCP. Recall that the previous cbp-tree in Figure 3.3(b) is generated with the recursive CCP operations. We redraw this tree in Figure 3.5(a) using dashed lines to group CCPs into related QCPs. The new representation with QCP operations of the previous cbp-tree is illustrated in Figure 3.5(b). We call this new representation a quadtree, since each operation (QCP) on a tree node can yield at most four child nodes. Essentially, a QCP operation only performs on the nodes on the even levels of the original cbp-tree.

The original cbp-tree with L levels can be replaced by a quadtree with only $\lceil L/2 \rceil$ levels. Here, we say the nodes on the even level l_e of the cbp-tree are of $l_e/2$ level for the quadtree. As we can see in Figure 3.5(b), the quadtree has only three levels, instead of five. This QCP-viewpoint is important in our coding procedure, as we will see later.

3.3 Progressive Coding Mechanism

In this section, we explain how the proposed framework achieves progressive coding functionality using the cbp-tree. First we introduce how the information contained in the tree is encoded. Then, the general idea behind progressive decoding is given.

As we introduced in Section 3.2, the tree is generated along with the recursive cell-partitioning procedure. The cell partitionings are performed on the tree nodes in a specific order, which we will describe in a later section. After the tree is fully generated with all the mesh information contained in the tree nodes, the encoding procedure is started. The encoder performs a top-down traversal of the cbp-tree in previous cell-partitioning order. For each cell partitioning, the encoder encodes the changes of corresponding tree node. The changes on geometry information are about how many and which child cells are nonempty. Each time when a cell is partitioned into two nonempty child cells, the corresponding tree node has two child nodes. Recall that each node has a representative vertex. Thus, we can say the representative vertex of the original node is split into the two representative vertices of the child nodes. The changes on connectivity information can be characterized using the vertex split as we described in Section 2.4 (on page 17). In addition, a detail coefficient is generated during this cell partitioning (as described on page 29). Therefore, If the node has two child nodes, the encoder also codes the changes on connectivity information caused by the corresponding vertex split and the changes on function values indicated by the detail coefficient of the node.

Next, we explain the general idea of the progressive decoding. Since the encoder codes the information in the cbp-tree by a top-down traversal, the decoder can start with one root node. Based on the geometry information from the encoder, the decoder can perform cell partitionings and generate the cbp-tree progressively. The decoder also receives the connectivity information from the encoder. So, the neighbor relations of the current leaf nodes are also stored in the nodes. When the tree is at a certain pruned version, the current leaf nodes can be represented using the representative vertices located at the centroids of the nodes' cells. With these vertices, an intermediate mesh is generated with the edges in the mesh representing the neighbor relations of all the current leaf nodes. The function values of the non-root nodes can be recovered using the approximation coefficient of the root node

and the appropriate detail coefficients recovered from the encoder. As more information is received, the tree reconstructed in the decoder side will have more leaf nodes, and the reconstructed mesh will become closer to the original one. In this way, progressive decoding functionality is achieved.

3.4 Proposed Mesh-Coding Framework

With the background information presented in the preceding sections, we now can introduce our proposed coding framework for 2.5-D triangle meshes which has several free parameters. The remaining part of this section is mainly focused on the details of the encoding procedure. The decoder is mostly symmetric with the encoder, and the few asymmetries will be considered later.

3.4.1 Overview of the Encoding Procedure

As we stated before, the information contained in the mesh includes a set of sample points, a triangulation, and a set of function values (at the sample points). In the proposed framework, all mesh information is encoded in two different procedures. The first is called cell-partitioning (**CP**) coding and is used to code the information of the local geometry and connectivity changes during several cell bi-partitionings. The second is called detail-coefficient (**DC**) coding and is used to code the DCs of the tree nodes.

Besides the cbp-tree, the encoder also utilizes two queues: 1) the CI queue and 2) the RDC queue. To encode the nodes stored on the CI queue, the CP coding procedure is invoked once and the DC coding procedure is invoked several times for coding the initial DC information. A parameter `usePriorityScheme` is used to set the CI queue as prioritized or non-prioritized, which will be described later. The RDC queue is first-in first-out (FIFO), and the nodes on the RDC queue are encoded for the remaining DC information.

The overall encoding procedure is given in Algorithm 1. To begin, a short header is output, containing the width W, height H, and the lower-left corner p of the unpadded root cell, the approximation coefficient a_r of the root node, and several parameters, which will be introduced later. The quantities W, H, and p can be used to determine the root cell. The context-adaptive binary arithmetic-coding engine is initialized. The root cell is selected according to the parameter `usePaddedRootCell`. In particular, the `usePaddedRootCell` is either set to 1 or 0 for the padded or unpadded scheme. Next, the root node is placed on the CI queue. Then the encoder codes the geometry, connectivity, and DC information by invoking the CP and DC coding procedures for nodes from the two queues. The encoder switch-

Algorithm 1 Encoding procedure.

```
1: ciBudget := thresholdCI, rdcBudget := thresholdRDC
2: encode header information (i.e., W, H, p, a_r, usePaddedRootCell, initialDC, remainDC,
   and valenceMax)
3: initialize CI queue and RDC queue, and initialize the arithmetic-coding engine
4: insert root node into CI queue
5: while CI queue not empty or RDC queue not empty do
6:   while ciBudget > 0 and CI queue not empty do
7:     set currentNode as the first node in CI queue, and remove the node from the queue
8:     invoke CP coding procedure for currentNode and set b1 to the number of encoded
       bits, new nodes {n_i} generated during CP coding procedure
9:     invoke initialDC (where initialDC ≥ 0) times of DC coding procedure for each n_i
       having a DC and set b2 to the number of encoded bits in this step
10:    ciBudget := ciBudget - b1 - b2
11:    insert new nodes n_i that have not been processed for CP coding to CI queue
12:    insert n_i with more bits to code for the DC on RDC queue
13:  end while
14:  while rdcBudget > 0 and RDC queue not empty do
15:    set currentNode as the first node in RDC queue and remove the node from the queue
16:    invoke the DC coding procedure remainDC (where remainDC ≥ 1) times for
       currentNode, set b to the number of encoded bits
17:    if more bits of DC need to code for currentNode then
18:      insert currentNode on RDC queue.
19:    end if
20:    rdcBudget := rdcBudget - b
21:  end while
22:  ciBudget := min(thresholdCI, ciBudget + thresholdCI)
23:  rdcBudget := min(thresholdRDC, rdcBudget + thresholdRDC)
24: end while
```

es between these two queues controlled by two thresholds `thresholdCI` and `thresholdRDC`. When the queue currently being processed becomes empty or the number of coded bytes has exceeded the current threshold, the processing switches to the other queue. During the CP coding procedure, the geometry and connectivity information of a node is coded. During the CP coding procedure, a parameter `valenceMax` is used to control the coding behavior, which we will explain later. As shown in Algorithm 1, for each new node n_i generated from the previous CP coding procedure with a detail coefficient, the DC coding procedure is invoked `initialDC` times (where `initialDC` ≥ 0). Then the new nodes that have not undergone the CP coding are placed on the CI queue, and the nodes with more bits of the DCs to be coded are moved to the RDC queue. After a node is handled from the RDC queue by invoking the DC coding procedure `remainDC` times (where `remainDC` ≥ 1), the node is only returned

to the queue if more bits of the DC remain to be coded.

In summary, the framework has the parameters `usePaddedRootCell`, `thresholdCI`, `thresholdRDC`, `usePriorityScheme`, `initialDC`, `remainDC`, and `valenceMax`. How these parameters should best be chosen will be discussed later.

3.4.2 Binarization Schemes

Before describing the details about the CP and DC coding procedures, a brief digression is in order concerning the binarization schemes used in our work. In the proposed framework, we utilize a context-adaptive binary arithmetic coder. If the symbol to be coded is not binary, a binarization scheme must be utilized to transform the symbol into a sequence of binary symbols. Four types of nonbinary symbols need to be handled during the coding process, and are introduced in what follows.

The first type of nonbinary symbol is a ternary symbol (i.e., an element of $\{0, 1, 2\}$) with a fixed uniform distribution. To handle this type of symbol, we define a context c_{third} with a fixed probability value for one equal to 1/3. The ternary value $n \in \{0, 1, 2\}$ can be binarized as follows:

1. Code $b_1 = \lfloor n/2 \rfloor$, using the context c_{third}.
2. If $b_1 = 0$, code another symbol $b_2 = \mathrm{mod}\,(n, 2)$ in bypass mode.

The second type of nonbinary symbol is a hexary symbol (i.e., an element of $\{0, 1, 2, 3, 4, 5\}$) with a fixed uniform distribution. The hexary value $n \in \{0, 1, \ldots, 5\}$ can be binarized as follows:

1. Code $b_1 = \lfloor n/3 \rfloor$ in bypass mode.
2. Code a ternary symbol $b_2 = \mathrm{mod}\,(n, 3)$ using the preceding binarization scheme.

The third type of nonbinary symbol is an n-bit unsigned integer. For this type of symbol, we use the UI-binarization scheme proposed in the IT method [8] with a single parameter $f \in [1, n]$. The binarization scheme works as follows. First, the n-bit unsigned integer v is represented as $v = \sum_{i=0}^{n-1} b_i 2^i$. Then, for each bit position k from $n-1$ down to 0, we perform the following steps:

1. Code the kth bit b_k using context c, where

$$c = \begin{cases} 2^{f-1} - 1 + \sum_{i \in [k+1, f)} (2b_i - 1)2^{i-1}, & k \in [0, f-1), \\ 2^f - f + k - 1, & k \in [f, n-1]. \end{cases}$$

2. If $b_k = 1$ and $k \geq f$, the remaining bits $\{b_i\}_{i\in[0,k)}$ will be coded in bypass mode and the loop will be terminated earlier.

Using this scheme, each symbol in the range $\left[0, 2^f\right)$ is assigned a distinct probability and the remaining symbols (if any), are partitioned into the ranges $[2^i, 2^{i+1})$ for $i \in [f, n)$, where the values within each range are equiprobable. If $f = n$, each symbol is coded using a distinct probability.

The last type of nonbinary symbol is an $(n+1)$-bit signed integer. For this type, We utilize the SI-binarization scheme used in the ADIT method [10], which is similar to UI-binarization with an extra sign bit coded in bypass mode.

3.4.3 Cell-Partitioning Coding Procedure

Now, we can describe the remaining details of the coding framework. To begin, we describe the CP coding procedure, which handles the coding of the changes in geometry and connectivity information. In general, for each cell bi-partitioning, the geometry changes consist of how many nonempty child cells are generated and which of them are nonempty. The connectivity changes, caused by vertex splits as described in Section 2.4 (on page 17), consist of how the original neighbor vertices and the new vertices are connected. In what follows, we describe the encoding of each of the geometry and connectivity changes.

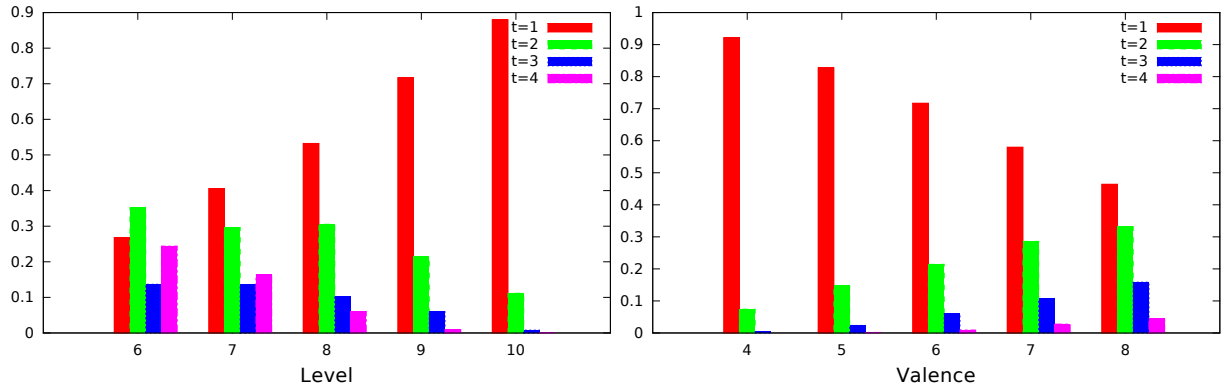
Geometry changes. When a cell is split into two child cells, the corresponding cbp-tree node can have at most two child nodes. In order to provide richer geometry information for coding, we consider the geometry information changes in a QCP operation. As stated in Section 3.2 (on page 31), after a QCP operation, each nonleaf node in the tree can have at most four or two children, and the latter case only happens when the root cell does not have a square power-of-two dimension. Let M denote the maximum number of nonempty subcells, so that $M \in \{2, 4\}$. During each QCP operation, the information related to geometry changes is given by

1. how many nonempty subcells generated from this QCP operation; and

2. which of the subcells are nonempty.

Suppose that after one QCP operation, T subcells are nonempty. Recall that T cannot be zero since empty cell will not be partitioned.

First the quantity value of T is encoded. Generally, the cell on a higher level of the tree (i.e., closer to the root cell) or with more neighbors is more possible to have a larger T after a QCP operation. Note that, the cell on the highest level of the tree is the root cell, and has the smallest level index (i.e., $l = 0$). The number of neighbor cells is called the **valence**



(a) (b)

Figure 3.6: Distributions of T under different valences and levels. (a) Same valence (i.e., 6) with different levels. (b) Same level (i.e., 9) with different valences.

of the cell. The distributions of T under different quadtree levels and valences are shown in Figure 3.6. We count the numbers of nonempty subcells for all cells being partitioned with the valences as 6 but with different level indices, and for all cells with the level indices as 9 but with different valences, then plot the histograms in Figures 3.6(a) and (b), respectively, to show the distributions of T under different situations. From Figure 3.6(a) we can see that with the same valence, the distributions of T vary for different levels. The chance of T to be 4 is much smaller for nodes on the lower levels (i.e., with larger level indices) of the tree. In Figure 3.6(b), the cells with larger valences tend to have larger possibilities to have more nonempty subcells. Based on the observation, T is arithmetic coded conditioned on the cell's valence and the quadtree-level, leading to a decrease of coding bit rate by 61.9% compared with encoding T in bypass mode as a $(\log_2 M)$-bit integer, where $M = 2$ or $M = 4$.

Next we consider the encoding of a configuration for which T of M subcells are nonempty. If $M = 2$, T has two possible values, namely, $T \in \{1, 2\}$. If $T = 1$, only two configurations are possible and can be represented using one single bit and coded in bypass mode. If $T = 2$, only one configuration is possible and no information needs be coded. If $M = 4$, $T \in \{1, 2, 3, 4\}$. We need to consider the following three cases.

1. $T = 1$ or $T = 3$. In this case, four configurations are possible and can be represented using two bits. Each of the two bits is coded in bypass mode.

2. $T = 2$. In this case, six configurations are possible. We can use a hexary symbol to represent the six possibilities and utilize the binarization scheme described in Section 3.4.2 (on page 35).

3. $T = 4$. In this case, one configuration is possible and no information needs be coded.

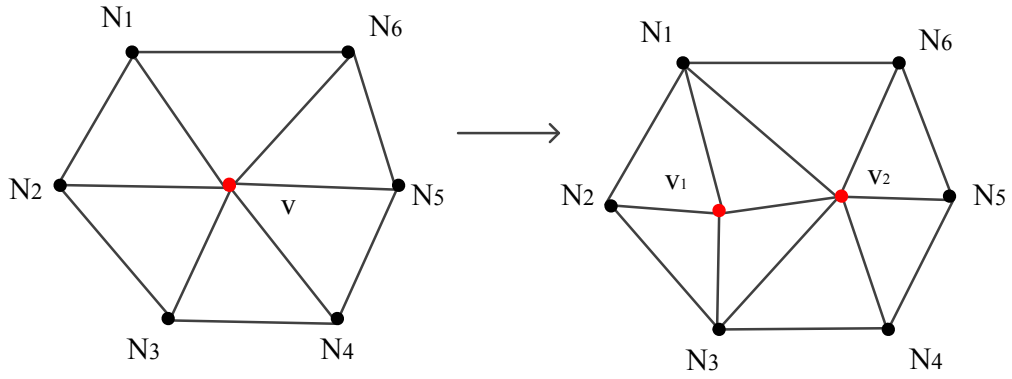


Figure 3.7: An example of vertex split. Vertex v is split into two new vertices v_1 and v_2.

Connectivity changes. Having encoded the geometry changes, now we can encode the connectivity changes. As we described before, local connectivity changes are associated with vertex splits. Before we proceed further, two concepts need to be introduced first. After a vertex split, if an original neighbor connects to both of the new vertices, we call it a **pivot**; otherwise, we call it a **nonpivot**. In the example of vertex split illustrated in Figure 3.7, N_1 and N_3 are pivots. The other neighbors N_2, N_4, N_5, and N_6 are nonpivots. Let us denote the vertex to be split as v, and the two new vertices as v_1 and v_2. Assume that the vertex v has M neighbors before splitting, denoted as $\{N_1, N_2, \ldots, N_M\}$. To fully cover the connectivity changes during the vertex split, we need to code the following information:

- the number and the configuration of the pivots (i.e., how many vertices among M and which of them are pivots);
- for each nonpivot n, to which new vertex v_1 or v_2 does n connect; and
- whether v_1 and v_2 are neighbors or not in the refined mesh.

Suppose that P of the M vertices are pivots, so that $P \in \{0, 1, 2, \ldots, M\}$. Since the value of P is related to the value of M, P is arithmetic coded conditioned on M. The distributions of P under different values of M are illustrated in Figure 3.8. Figure 3.8(a) shows the distribution of P when the number of neighbor vertices is six (i.e., $M = 6$). We can see that P is concentrated at locations around $P = 2$. Figures 3.8(b), (c), and (d) show the distributions of P when $M = 7$, $M = 8$, and $M = 9$, respectively. The possible values of P differ with different M values, but all distributions of P are very skewed, which means the arithmetic coding of P should be very efficient. The conclusion is justified by experimental results, showing the bit rate decreased by 74.3% on average with adaptive arithmetic coding compared with encoding P in bypass mode as a $\lceil \log(M + 1) \rceil$-bit integer.

After encoding the number of pivots P, next we encode the configuration of which P among M neighbors are pivots. The total number of different configurations is $\binom{M}{P}$ and

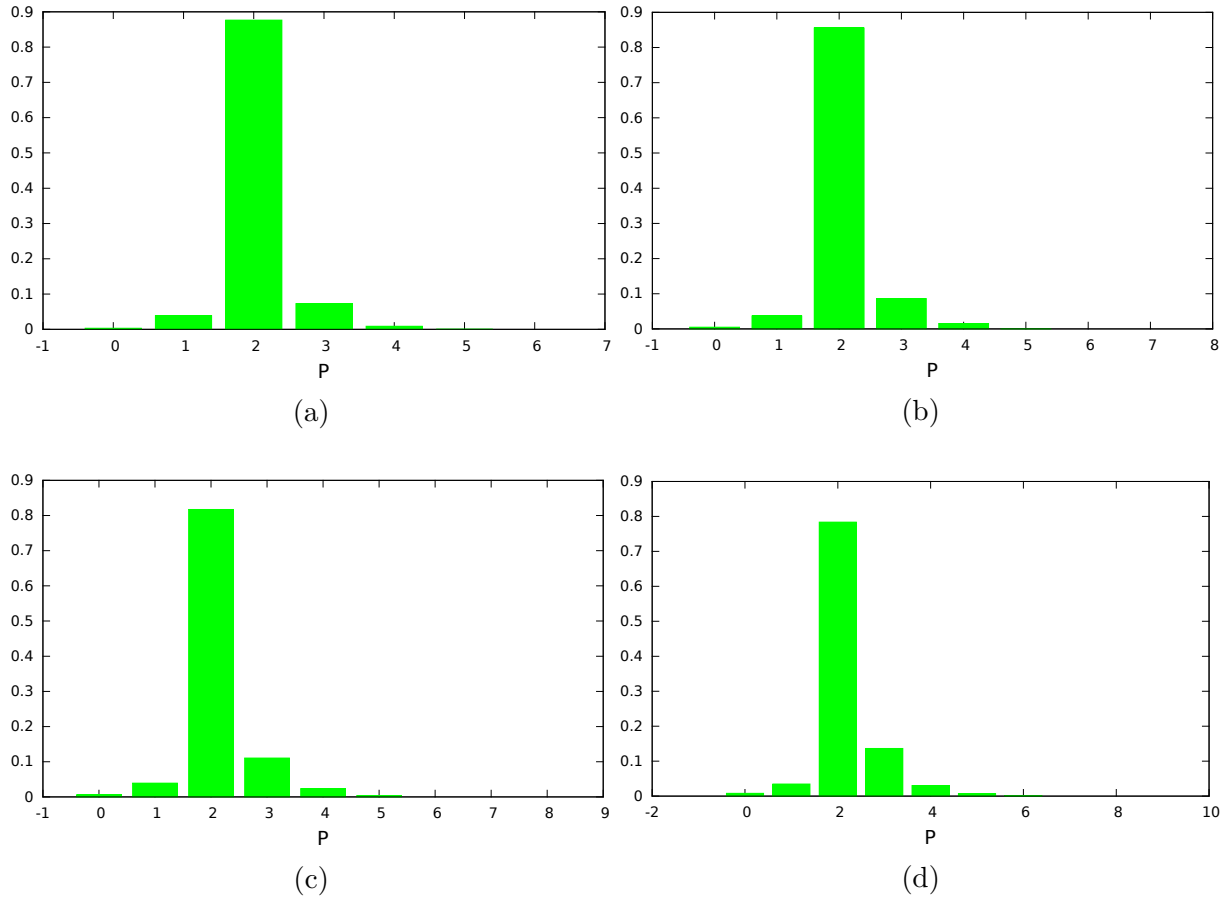
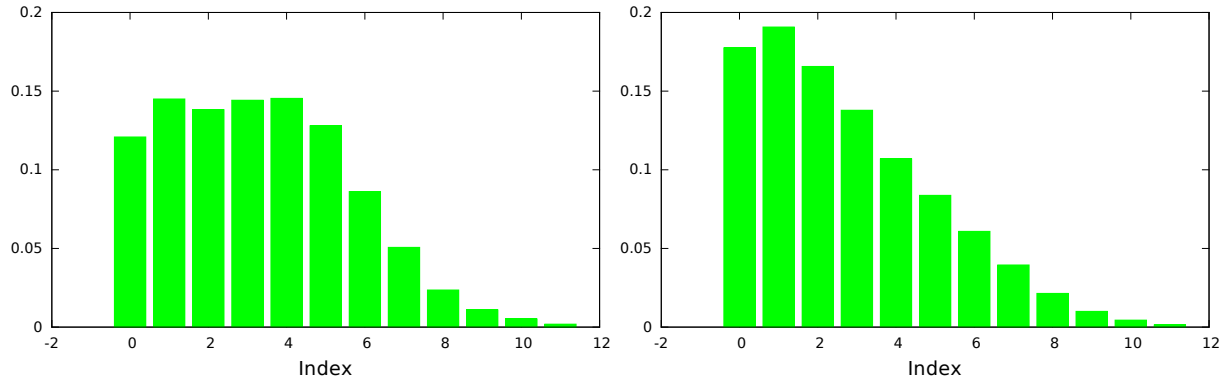


Figure 3.8: Distributions of P when (a) $M = 6$, (b) $M = 7$, (c) $M = 8$, and (d) $M = 9$.



(a) (b)

Figure 3.9: Distributions of indices of pivot-vertex tuple (a) before and (b) after priority calculation.

each one contains P neighbor vertices. This configuration containing P elements is called a P-tuple. The desired configuration containing all P pivots is called a pivot-vertex tuple. Each P-tuple is assigned an index i, so $i \in \left\{0, 1, 2, \ldots, \binom{M}{P} - 1\right\}$.

Before encoding the index of the pivot-vertex tuple, we first estimate the priority of each P-tuple to be the actual pivot-vertex tuple, generate a frequency table, sort all possible P-tuples based on their priorities in descending order, and then encode the index of the actual pivot-vertex tuple in this sorted frequency table. To calculate the priority of a certain P-tuple, we first estimate the possibility of each vertex to be a pivot. Then, the priority of the tuple is calculated by summing the probabilities of the neighbor vertices it contains. For each neighbor N_i, the possibility to be a pivot is determined by the formula

$$p_i = r_i = \frac{\sigma_i}{2s} = \frac{\sqrt{s\,(s-a)\,(s-b)\,(s-c)}}{2s},$$

where a, b, and c are the lengths of the three edges in $\triangle N_i v_1 v_2$, σ_i is the area of $\triangle N_i v_1 v_2$ and $s = (a+b+c)/2$. This possibility also represents the regularity of the triangle. Therefore, if the triangle is more regular, the neighbor N_i will have a larger chance to be pivot.

To demonstrate the effectiveness of the priority scheme, we count the numbers of different indices encoded for the pivot-vertex tuple during the entire encoding procedure before and after applying the priority scheme, and plot the histograms in Figures 3.9(a) and (b), respectively. Note that, in Figure 3.9, we only plot the histograms for the indices to be at most 12, since this value can cover the vast majority of cases. The indices in Figure 3.9(b) are more concentrated in lower-index locations than in Figure 3.9(a). After utilizing the preceding priority scheme, the distribution of indices becomes more skewed, which means the arithmetic coding should be more efficient. Based on the observation, the index of the

pivot-vertex tuple is arithmetic coded conditioned on P and M. This priority scheme leads to an average decrease in the bit rate by 53.10% compared with coding the index as a $\lceil \log \binom{M}{P} \rceil$-bit integer in bypass mode. Also, the bit rate using the priority scheme leads an average decrease in the bit rate by 2.0% compared to not using the priority scheme.

One potential problem occurs in the preceding coding procedure when the valence of the vertex is large, however. If the valence is large, to store the results of all $\binom{M}{P}$ possible configurations requires tremendous amount of memory and the calculations also consume tremendous time. To avoid this computationally intractable situation due to combinatorial blowup and be more practically useful, the proposed framework uses a divide-and-conquer approach by introducing a free parameter `valenceMax` to set a threshold value for M. If the current v to be split has more than `valenceMax` neighbors, we will first divide the neighbor list into several small lists, each with at most `valenceMax` elements. For example, if `valenceMax` = 10 and the size of original neighbors is 15, two neighbor sublists are generated as $\{N_1, N_2, \ldots, N_{10}\}$ and $\{N_{11}, N_{12}, \ldots, N_{15}\}$. Note that, in order to ensure that the encoder and decoder split the original neighbor list in the same way, the neighbor vertices are sorted first based on the lexicographic order before splitting. Then, the number and the configuration of pivots need to be coded for each sublist. The choice of this free parameter `valenceMax` will be described in a later section.

Next, we need to encode the information of nonpivots. Recall from the earlier definition, the nonpivots are the neighbors only connect to one of the v_1 or v_2. Before encoding the connectivity information of nonpivots, we need to partition them into several segments. The partitioning rules are given by

1. each nonpivot connects to more than two other vertices in N_i forms a segment by itself; and

2. other nonpivots are partitioned into maximum-connected segments.

To better explain the above rules, two examples of nonpivots partitioning are illustrated in Figure 3.10. In Figure 3.10(a), N_4 and N_8 are pivots. Since the nonpivots N_1 and N_7 are both connected to three other neighbors, they form segments separately by themselves. The remaining nonpivots are partitioned into the two segments $\{N_2, N_3\}$ and $\{N_5, N_6\}$. Overall, the nonpivots are partitioned into the four segments $\{N_1\}$, $\{N_2, N_3\}$, $\{N_5, N_6\}$, and $\{N_7\}$. Similarly, in Figure 3.10(b), the nonpivots are partitioned into the two segments $\{N_1, N_2, N_3\}$ and $\{N_5, N_6, N_7\}$. In the above figures, different labels are assigned to different segments.

For each segment, one bit is coded to indicate whether the vertices in this segment are connected to the same new vertex v_1 or v_2. If not, the vertices are treated as separate segments. For example, in the fourth segment $\{N_5, N_6\}$ as shown in Figure 3.10(a), N_5 is

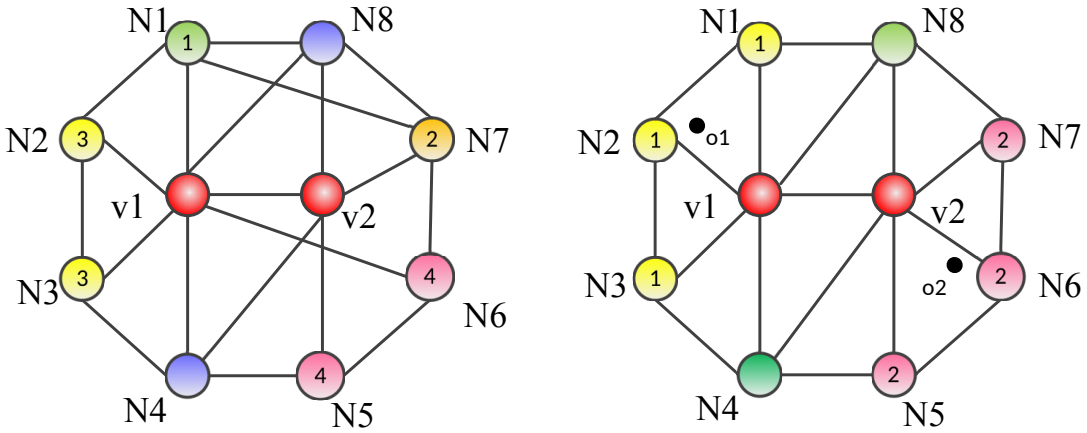


Figure 3.10: Two examples of nonpivots partitioning. In (a), four segments are generated for the nonpivots. In (b), two segments are generated for nonpivots with the centroids of the segments denoted as o_1 and o_2.

connected to v_2, and N_6 is connected to v_1. In a manifold mesh or "almost manifold" mesh, however, almost every segment of nonpivots is connected to the same one of v_1 and v_2. So, this bit can be arithmetic coded efficiently. The total cost for coding this bit is only about 0.03 bits per vertex on average.

Next, we need to specify to which of v_1 or v_2 does each segment connect. For each segment, its centroid o_i is calculated by averaging the locations of the nonpivots contained in the segment. Then, we calculate the distances d_1 and d_2 between the centroid of the segment o_i and the new vertices v_1 and v_2, respectively, and predict the segment connected to one of the v_1 and v_2 with the smaller distance. In the example shown in the Figure 3.10(b), o_1 and o_2 are the centroids of the two segments of nonpivots. Since o_1 is closer to v_1 than v_2, we predict that the segment with label "1" is connected to v_1. Similarly, the other segment with label "2" is predicted as connected to v_2. One bit is arithmetic coded to indicate whether the prediction is correct or not. The total cost for coding this bit is about 2.38 bits per vertex on average, reduced by 44% relative to coding in bypass mode.

Besides encoding the connectivity changes of the original neighbor vertices, we also need to encode the connectivity of the two new vertices. One bit is coded to indicate whether the two new vertices v_1 and v_2 are connected to each other after the vertex split. This bit can also be arithmetic coded efficiently, with the total cost around 0.43 bits per vertex on average, reduced by 57% relative to coding in bypass mode.

Summary of cell-partitioning coding procedure. Suppose that after one QCP operation, the cell of a nonleaf node is partitioned into T nonempty subcells. Then this QCP operation includes $\max(T-1,0)$ cell bi-partitionings that split a cell into two nonempty child cells, corresponding to $\max(T-1,0)$ vertex splits. To better understand this relation between the QCP operation and cell bi-partitionings, four examples of QCP operations are illustrated in Figure 3.11. In Figure 3.11(a), after a QCP operation, the actual number of nonempty subcells is $T=1$ and the maximum number of nonempty subcells is $M=4$. We can see the first bi-partitioning along the x-axis generates one nonempty child cell, which is bi-partitioned further along the y-axis and also generates one nonempty child. Therefore, no cell bi-partitioning generates two nonempty child cells. In Figure 3.11(b), $T=2$ and $M=4$. The first bi-partitioning along the x-axis generates two nonempty child cells, each of which is bi-partitioned further along the y-axis and only generates one nonempty child cell. Therefore, only one bi-partitioning generates two nonempty child cells. Similarly, we can see in Figure 3.11(c), $T=3$ and $M=4$, and two bi-partitionings generate two nonempty child cells. In Figure 3.11(d), since the width of the cell to be partitioned is one, after bi-partitioning along the x-axis, a degenerate child cell with zero area is generated. Then the nondegenerate one is bi-partitioned along the y-axis and generates one nonempty child cell.

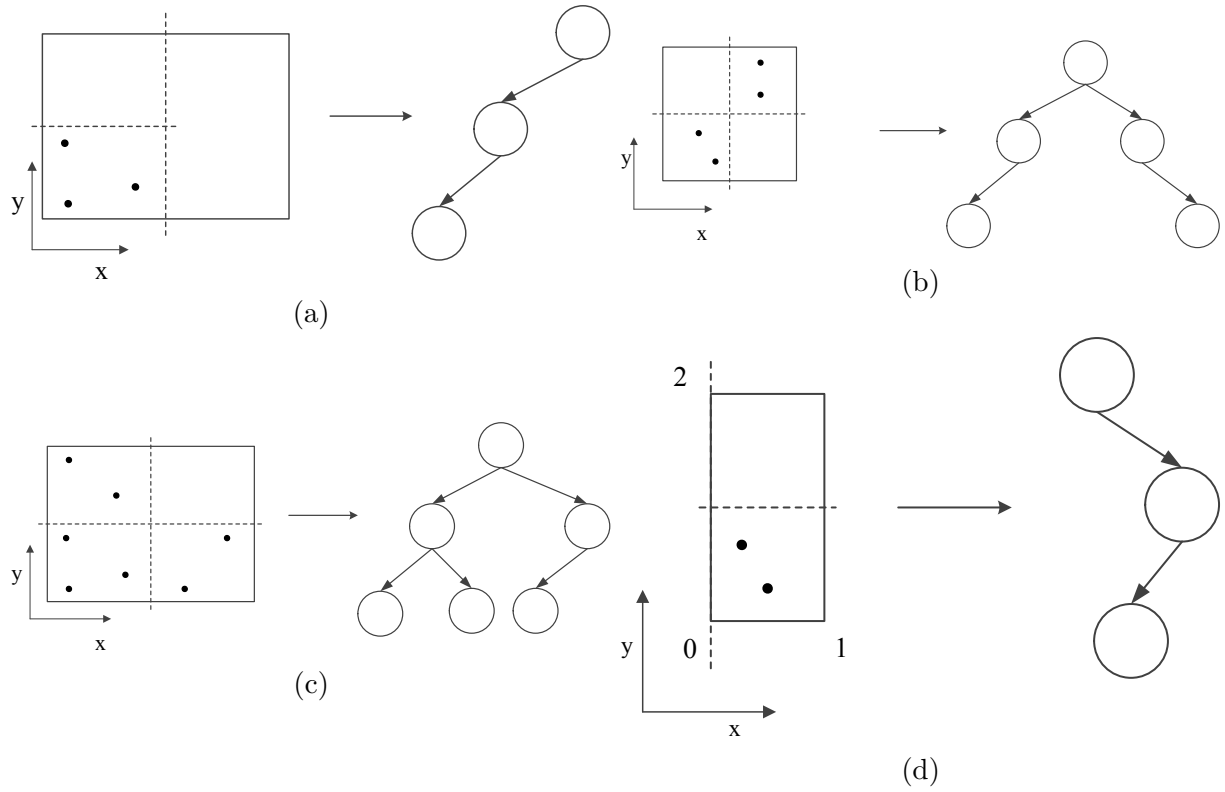


Figure 3.11: Four examples of QCP. After a QCP, (a) the actual number of nonempty cells is $T = 1$ and the maximum number of nonempty cells is $M = 4$. Similarly, (b) $T = 2$ and $M = 4$ and (c) $T = 3$, $M = 4$. (d) Since the cell bi-partitioning along x-axis generates a degenerate cell, so the maximum number of nonempty cells $M = 2$, and the actual number is $T = 1$.

Therefore, in the case of Figure 3.11(d), $M = 2$, $T = 1$, and no cell bi-partitioning generates two nonempty child cells. To summarize, the information coded during the CP coding procedure includes the geometry changes (the number T and configuration of nonempty subcells) and the connectivity changes caused by $\max(T - 1, 0)$ vertex splits.

3.4.4 Detail Coefficient Coding Procedure

To handle the function values, the proposed framework uses a similar idea from the ADIT method. Instead of coding the function values directly, we only code the approximation coefficient of the root node, and all of the DCs of the nodes in the tree. Then using the inverse AD-transform, we can calculate the approximation coefficients of the other nodes.

Suppose that each function value of the mesh can be represented using a ρ-bit unsigned integer. Then, the DCs can be represented as $(\rho + 1)$-bit signed integers, with ρ magnitude bits plus one sign bit. Each DC is coded using the SI-binarization scheme. The DCs are coded starting from the most-significant magnitude bit. The sign bit is coded in bypass mode immediately after the first nonzero magnitude bit. Every time when the DC coding procedure is invoked, one extra magnitude bit is coded. Two free parameters related to DC coding procedure, namely `initialDC` and `remainDC`, control how many times the DC coding procedure is invoked at each of two places in Algorithm 1 in Section 3.4.1 (on page 32). Details about how to choose these two free parameters will be discussed in a later section.

3.4.5 Decoding

The decoding procedure is almost a mirror of the encoding procedure. Therefore, decoding is not described in detail here. The decoder starts with a single root node, whose representative vertex is set to the centroid of the root cell. After the geometry changes information is decoded and the value T (i.e., the number of nonempty cells) is obtained, the connectivity changes information related to $\max\{T - 1, 0\}$ vertex splits to refine the mesh is decoded. The decoding procedure continues and processes information from the bitstream, until the end of the bitstream is reached. After the decoding procedure is terminated, the reconstructed mesh consists of the vertices associated with the current leaf nodes in the tree, and the edges representing the neighbor relations between these nodes. As we have described in Section 2.4 (on page 17), vertex splits can potentially lead to a triangulation with invalid connectivity. Therefore, a reconstructed mesh produced at intermediate stages of decoding is not guaranteed to have valid triangulation connectivity. To enforce valid connectivity, we represent the connectivity by a constrained Delaunay triangulation with all of the edges as constraints. From the implementation viewpoint, the edges are set as constraints and

Table 3.1: Category A: Delaunay triangulation meshes (twelve meshes). Edge-flipping distances are always zero for these cases.

	Mesh Information			Statistics about Valence					
Nickname	#Vertices	#Edges	#Faces	Max.	Min.	Median	Mean	Width×Height	ρ
B1	3932	11756	7825	15	2	6	5.97965	1024×768	8
B2	7864	23532	15669	13	2	6	5.98474	1024×768	8
B3	15728	47042	31315	13	2	6	5.98194	1024×768	8
B4	31457	94112	62656	12	2	6	5.98353	1024×768	8
L1	1310	3880	2571	11	2	6	5.92366	512×512	8
L2	2621	7788	5168	11	2	6	5.94277	512×512	8
L3	5242	15612	10371	11	2	6	5.95651	512×512	8
L4	10485	31283	20799	11	2	6	5.96719	512×512	8
P1	1310	3905	2596	12	3	6	5.96183	512×512	8
P2	2621	7814	5194	12	3	6	5.96261	512×512	8
P3	5242	15641	10400	13	3	6	5.96757	512×512	8
P4	10485	31322	20838	11	3	6	5.97463	512×512	8

inserted to the triangulation in a particular order. If a constrained edge intersects with a previous inserted edge, the previous constraint will be removed. The insertion order is determined by the parameter `edgeInsertion`, which can be one of the following four cases:

1. edge-iteration order as determined by the data structure employed in the implementation;
2. in order of descending edge length;
3. in order of ascending edge length; and
4. pseudo-random order.

We will see later about how to choose this parameter.

3.5 Test Data

Before proceeding further, a brief digression is in order concerning the test data used herein. The author's main application focus is image processing. Therefore, we employ a set of 64 2.5-D meshes generated from images using several mesh-generation schemes, including [27]. The test meshes are grouped into three categories A, B, and C based on their underlying connectivity, and listed in Tables 3.1, 3.2, and 3.3, respectively. In these three tables, meshes are denoted by nicknames for convenience. The original filenames for the meshes are also listed in Tables 3.4, 3.5, and 3.6 for reference.

Category A meshes as listed in Table 3.1, have Delaunay connectivity. Category B and C meshes, as listed in Tables 3.2 and 3.3, have arbitrary connectivity with good and bad

Table 3.2: Category B: Non-Delaunay triangulation meshes with good quality (44 meshes)

	Mesh Information			Statistics about Valence						
Nickname	#Vertices	#Edges	#Faces	Max.	Min.	Median	Mean	Width×Height	ρ	Distance †
A1	7397	22140	14744	40	3	6	5.98621	1238×1195	8	40.41%
A2	14794	44311	29518	27	3	6	5.9904	1238×1195	8	39.55%
A3	29588	88657	59070	49	3	6	5.99277	1238×1195	8	36.23%
A4	44382	132985	88604	31	3	6	5.99274	1238×1195	8	35.10%
B5	3932	11757	7826	21	3	6	5.98016	1024×768	8	49.86%
B6	7864	23533	15670	19	3	6	5.98499	1024×768	8	45.93%
B7	15728	47063	31336	22	2	6	5.98461	1024×768	8	41.49%
B8	23592	70585	46994	20	2	6	5.98381	1024×768	8	38.73%
CH1	655	1950	1296	17	3	5	5.9542	512×512	8	36.41%
CH2	1310	3859	2550	22	3	5	5.8916	512×512	8	46.26%
CH3	2621	7753	5133	41	3	5	5.91606	512×512	8	57.53%
CR1	17858	53419	35562	19	3	6	5.98264	1744×2048	10	34.29%
CR2	35717	106893	71177	22	3	6	5.98555	1744×2048	10	33.73%
CR3	71434	213899	142466	41	3	6	5.98872	1744×2048	10	33.81%
CR4	107151	320873	213723	54	2	6	5.98917	1744×2048	10	33.80%
CT1	1310	3914	2605	22	2	6	5.97557	512×512	12	38.68%
CT2	2621	7840	5220	25	2	6	5.98245	512×512	12	39.32%
CT3	5242	15703	10462	29	2	6	5.99122	512×512	12	40.01%
CT4	7864	23559	15696	27	2	6	5.99161	512×512	12	39.30%
K1	1966	5889	3924	18	3	6	5.99084	768×512	8	39.68%
K2	3932	11787	7856	30	3	6	5.99542	768×512	8	39.68%
K3	7864	23583	15720	46	3	6	5.99771	768×512	8	37.38%
K4	11796	35377	23582	47	3	6	5.99813	768×512	8	36.60%
L5	657	1948	1292	14	3	6	5.92998	513×513	8	42.76%
L6	1315	3893	2579	14	2	6	5.92091	513×513	8	41.74%
L7	2631	7812	5182	17	3	6	5.93843	513×513	8	41.27%
L8	5263	15659	10397	15	2	6	5.9506	513×513	8	39.95%
L13	1310	3882	2573	18	3	6	5.92672	512×512	8	40.86%
L14	2621	7797	5177	17	3	6	5.94964	512×512	8	41.46%
L15	5242	15611	10370	17	3	6	5.95612	512×512	8	40.68%
L16	7864	23441	15578	16	2	6	5.9616	512×512	8	39.31%
M1	3637	10858	7222	18	3	6	5.97086	1912×761	8	52.04%
M2	7275	21745	14471	19	3	6	5.97801	1912×761	8	49.92%
M3	14550	43491	28942	19	3	6	5.97814	1912×761	8	47.69%
M4	29100	86988	57889	21	3	6	5.97856	1912×761	8	45.38%
P5	327	965	639	14	3	5	5.90214	512×512	8	33.23%
P6	655	1947	1293	16	3	6	5.94504	512×512	8	31.86%
P7	1310	3898	2589	15	3	6	5.95115	512×512	8	30.30%
P8	2621	7799	5179	17	3	6	5.95116	512×512	8	30.75%
P9	5242	15628	10387	19	3	6	5.96261	512×512	8	30.80%
Q1	7864	23465	15602	19	3	6	5.9677	512×512	8	56.12%
Q2	9600	28793	19194	50	3	6	5.99854	1200×1600	8	52.86%
Q3	19200	57593	38394	93	3	6	5.99927	1200×1600	8	47.42%
Q4	38400	115193	76794	141	3	6	5.99964	1200×1600	8	41.53%

† Distance: edge-flipping distance between the mesh triangulation and the PDDT.

Table 3.3: Category C: Non-Delaunay triangulation meshes with poor quality (eight meshes)

	Mesh Information			Statistics about Valence						
Mesh Nickname	#Vertices	#Edges	#Faces	Max.	Min.	Median	Mean	Width×Height	ρ	Distance †
L9	1310	3919	2610	395	3	5	5.98321	512×512	8	145.65%
L10	2621	7843	5223	459	3	5	5.98474	512×512	8	148.83%
L11	5242	15690	10449	522	3	5	5.98626	512×512	8	149.66%
L12	7864	23539	15676	575	3	5	5.98652	512×512	8	150.51%
M5	3637	10883	7247	266	3	4	5.9846	1912×761	8	138.00%
M6	7275	21786	14512	392	3	4	5.98928	1912×761	8	146.06%
M7	14550	43580	29031	638	3	4	5.99038	1912×761	8	155.22%
M8	29100	87183	58084	887	3	5	5.99196	1912×761	8	163.26%

† Distance: edge-flipping distance between the mesh triangulation and the PDDT.

Table 3.4: The original filenames and nicknames of the meshes in category A

Original Name	Nickname
`bull@1-GPRFS-ED-G4@default@0.005@model`	B1
`bull@1-GPRFS-ED-G4@default@0.01@model`	B2
`bull@1-GPRFS-ED-G4@default@0.02@model`	B3
`bull@1-GPRFS-ED-G4@default@0.04@model`	B4
`lena@1-ABOVE1-40@default@0.005@model`	L1
`lena@1-ABOVE1-40@default@0.01@model`	L2
`lena@1-ABOVE1-40@default@0.02@model`	L3
`lena@1-ABOVE1-40@default@0.04@model`	L4
`peppers@1-MGH@default@0.005@model`	P1
`peppers@1-MGH@default@0.01@model`	P2
`peppers@1-MGH@default@0.02@model`	P3
`peppers@1-MGH@default@0.04@model`	P4

Table 3.5: The original filenames and nicknames of the meshes in category B

Original Name	Nickname
animal@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	A1
animal@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	A2
animal@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	A3
animal@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh	A4
bull@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	B5
bull@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	B6
bull@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	B7
bull@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh	B8
checkerboard_antialiased@sqrErr:h25:jndfe:pef:fe@41@0.0025@mesh	CH1
checkerboard_antialiased@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	CH2
checkerboard_antialiased@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	CH3
cr@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	CR1
cr@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	CR2
cr@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	CR3
cr@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh	CR4
ct@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	CT1
ct@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	CT2
ct@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	CT3
ct@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh	CT4
kodim15@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	K1
kodim15@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	K2
kodim15@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	K3
kodim15@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh	K4
lena_513x513@sqrErr:h25:jndfe:pef:fe@41@0.0025@mesh	L5
lena_513x513@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	L6
lena_513x513@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	L7
lena_513x513@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	L8
lena@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	L13
lena@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	L14
lena@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	L15
lena@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh	L16
muttart@sqrErr:h25:jndfe:pef:fe@41@0.0025@mesh	M1
muttart@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	M2
muttart@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	M3
muttart@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	M4
peppers@gh_hybrid@41@0.0025@mesh	P5
peppers@gh_hybrid@41@0.005@mesh	P6
peppers@gh_hybrid@41@0.01@mesh	P7
peppers@gh_hybrid@41@0.02@mesh	P8
peppers@gh_hybrid@41@0.03@mesh	P9
question2_grayscale@sqrErr:h25:jndfe:pef:fe@41@0.0025@mesh	Q1
question2_grayscale@sqrErr:h25:jndfe:pef:fe@41@0.005@mesh	Q2
question2_grayscale@sqrErr:h25:jndfe:pef:fe@41@0.01@mesh	Q3
question2_grayscale@sqrErr:h25:jndfe:pef:fe@41@0.02@mesh	Q4

Table 3.6: The original filenames and nicknames of the meshes in category C

Original Name	Nickname
lena@pae:pae:fe:none@41@0.005@mesh	L9
lena@pae:pae:fe:none@41@0.01@mesh	L10
lena@pae:pae:fe:none@41@0.02@mesh	L11
lena@pae:pae:fe:none@41@0.03@mesh	L12
muttart@sqrErr:pae:fe:none@41@0.0025@mesh	M5
muttart@sqrErr:pae:fe:none@41@0.005@mesh	M6
muttart@sqrErr:pae:fe:none@41@0.01@mesh	M7
muttart@sqrErr:pae:fe:none@41@0.02@mesh	M8

Table 3.7: Images used to generate the test datasets

Image Name	Image Description
`lena`	woman [6]
`bull`	cartoon animal
`peppers`	collection of peppers
`ct`	CT scan of head, from JPEG-2000 test set [7]
`muttart`	architecture
`question`	question mark
`checkerboard`	computer generated image
`kodim15`	digital image, from Kodak test set [3]

qualities, respectively. In each table, several characteristics of each mesh are listed, including the number of vertices, edges, and faces of the mesh, the maximum, minimum, average, and median valences of the vertices in the mesh, the width and height of the unpadded root cell, and the edge-flipping distance between the mesh triangulation and the PDDT (i.e., the percentage of edges need to be flipped before transforming the original triangulation into the PDDT). We note that the good-quality meshes usually have small edge-flipping distances , and the poor-quality meshes usually have very large distances. Note that, the poor-quality meshes are not of practical interest and are only included to allow us to see how robust our proposed framework is to handling datasets with bizarre statistical properties.

The 64 meshes in Tables 3.1, 3.2, and 3.3 were chosen to be quite diverse, with different numbers of bits per function value, unpadded root cell sizes, valence statistics, and sampling densities. The meshes are generated from images in standard test sets such as [6, 7, 3], which contain varied image types as shown in Table 3.7. From the original mesh filenames, one can infer from which image each mesh was generated. For example, the mesh with filename `lena@sqrErr:h25:jndfe:pef:fe@41@0.03@mesh` was generated from the image `lena`, which is an image of a woman.

3.6 Development of Proposed Method and Selection of Parameters

As introduced earlier, the proposed coding framework for 2.5-D triangle meshes has several parameters, namely:

1. the parameter `usePaddedRootCell`, which determines whether the encoder starts with unpadded or padded mode;

2. the parameters `thresholdCI` and `thresholdRDC`, which control the switching between queues during the coding procedure;

3. the parameter `usePriorityScheme`, which determines whether the CI queue is to be prioritized or not;

4. the parameter `valenceMax`, which sets a threshold value for the valence of the vertex to split during the CP coding procedure;

5. the parameters `initialDC` and `remainDC`, which determine how many times the DC coding procedure is invoked when a node is coded from the CI and RDC queue, respectively; and

6. the parameter `edgeInsertion`, which determines the order used to insert the edge constraints when generating the constrained Delaunay triangulation in the decoder.

In what follows, we study how various choices of these parameters affect lossless and progressive coding performance. Based on these experimental results, we ultimately recommend a particular set of choices to be used for these parameters, leading to our mesh-coding method proposed herein. In the sections that follow, several experiments are performed in which our framework is used to code meshes. During some of these experiments, we vary one parameter while keeping the others fixed at default values. For the purpose of these experiments, these defaults are as follows: 1) `usePaddedRootCell` = 1 (i.e., padded scheme), 2) `thresholdCI` = 512, `thresholdRDC` = 128, 3) `usePriorityScheme` = 1 (i.e., prioritized), 4) `valenceMax` = 12, 5) `initialDC` = 3, `remainDC` = 1, and 6) `edgeInsertion` = 1 (i.e., in order of descending edge length).

3.6.1 Choice of Root Cell Selection Strategy

To begin, we consider how different root cell selection strategies affect the progressive and lossless coding performance. The parameter `usePaddedRootCell` is provided to choose different schemes. If `usePaddedRootCell` is 1, the padded root cell is used; otherwise, the unpadded root cell is employed. In order to find the best choice for `usePaddedRootCell`, we consider meshes where the unpadded root cells do not have dimensions that are equal and powers of two. Since different schemes will not make a difference if the meshes already have the unpadded and padded root cells being the same. Among the 64 datasets listed in Tables 3.1, 3.2, and 3.3 in Section 3.5, 36 meshes fall into this category. In what follows, we only change the parameter `usePaddedRootCell`, and set the other parameters in the framework as default values as described at the beginning of Section 3.6.

First, we consider the influence of `usePaddedRootCell` on progressive performance (i.e., decoding in a lossy manner to various intermediate rates). Using each of the padded and

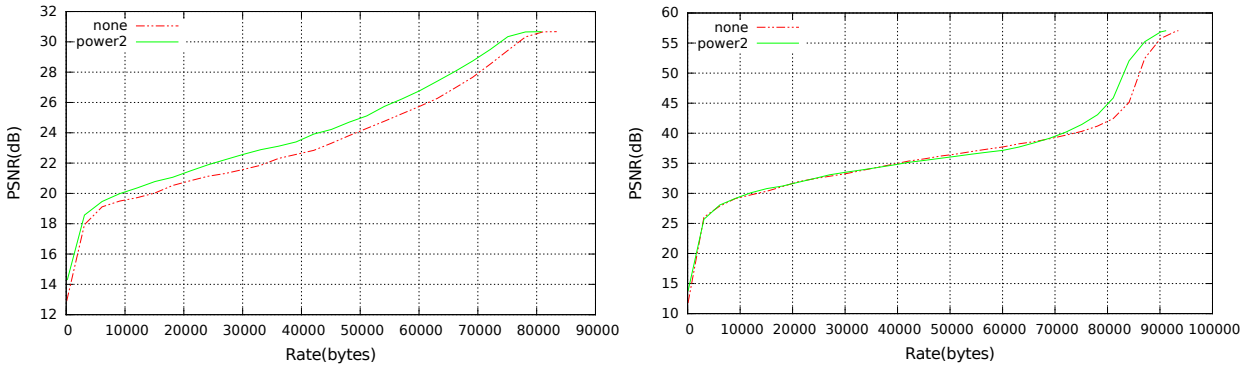


Figure 3.12: Progressive performance using different schemes for (a) mesh M4 and (b) mesh Q4. Label "none" represents unpadded scheme and "power2" represents padded scheme.

unpadded schemes, each mesh is encoded once losslessly, and then decoded at many intermediate rates. In each case, the decoded dataset was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. We give a representative subset of these results involving two datasets, namely, M4 and Q4. These results are given in Figure 3.12. Each of the two graphs in Figure 3.12 shows the PSNR of the reconstructed image plotted against rate. On each graph, the far left corresponds to no information having been decoded, while the far right corresponds to the coded bitstream having been fully decoded. Since the PSNR is measured relative to the original lattice-sampled image and the original mesh only approximates the original image, the lossless reconstruction does not achieve zero error. Thus, the PSNR obtained for the lossless reconstruction is not infinity. In Figure 3.12, the label "none" represents the unpadded scheme and "power2" represents the padded scheme. Next, we will examine the results in Figure 3.12 more closely.

From Figure 3.12(a), it is clear that the image reconstructions generated with a padded root cell are consistently better (often around one dB in terms of PSNR) than the one obtained with the unpadded cell. From Figure 3.12(b), we can see the difference between the padded and unpadded cases is not significant. We illustrate the original datasets M4 and Q4, in Figures 3.13 and 3.14, respectively, to explain the above observation. As can be seen from Figures 3.13 and 3.14, the quality of these two meshes are very good, showing clearly the inherent geometric structure in the original images. The difference is that, in the mesh Q4 as shown in Figure 3.14, not so many sample points are near the function-domain boundary. Since the most direct influence of the padded scheme is on the boundary-area points, if not many samples are in that area, the benefit of using the padding scheme is less significant, as we can see in Figure 3.12(b). To study whether the padded scheme is robust to

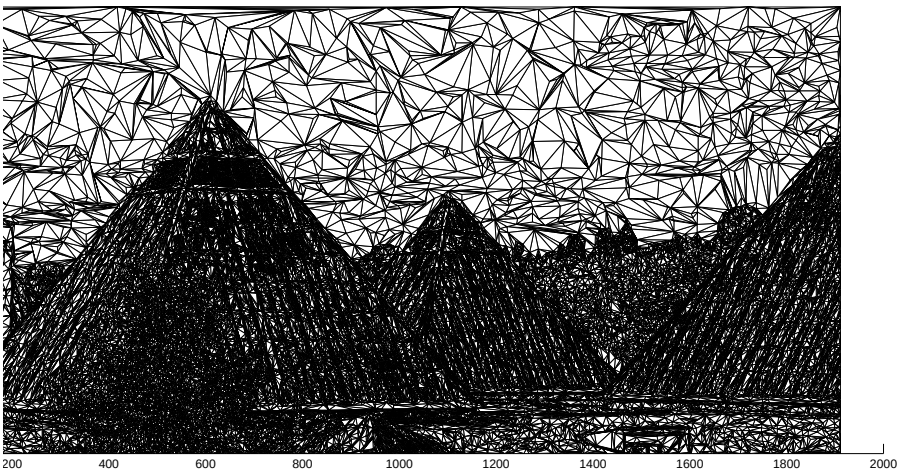


Figure 3.13: The original dataset with good quality, mesh M4

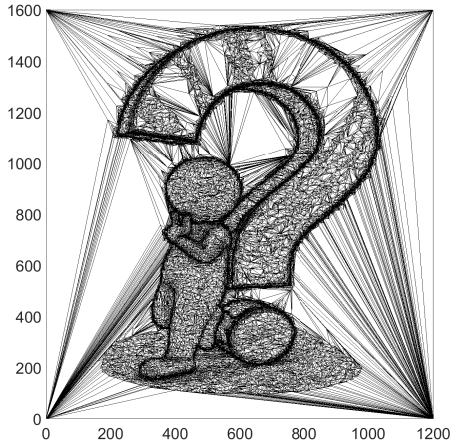


Figure 3.14: The original dataset with good quality, mesh Q4

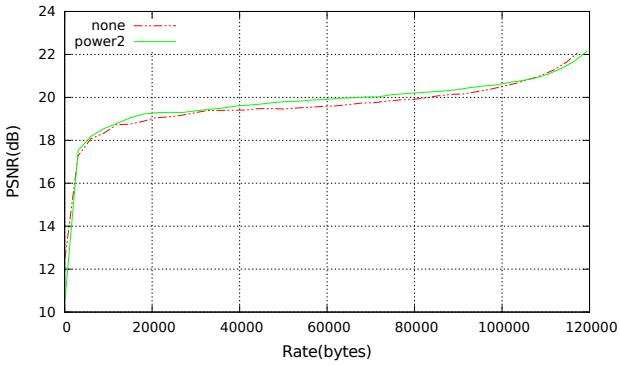


Figure 3.15: Progressive coding performance for M8 with different schemes. Label "none" represents unpadded scheme and "power2" represents padded scheme.

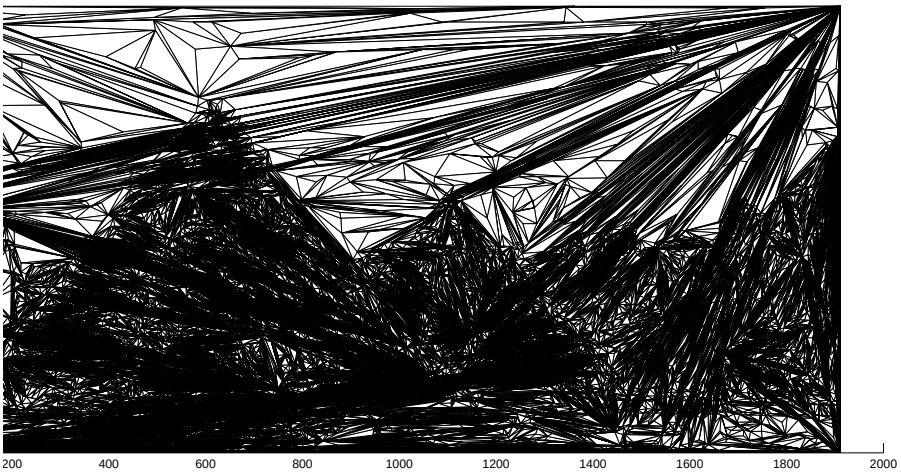


Figure 3.16: The original dataset with poor quality, mesh M8

the poor-quality meshes, we present a representative result involving a poor-quality dataset M8 in Figure 3.15. The original dataset M8, as illustrated in Figure 3.16, is generated from the same image as M4, but with a different mesh-generation method. From Figure 3.16, we can clearly see the connectivity is so poor that we can barely see the inherent geometric structure in the image. Besides the bad connectivity, we also observe that M8 has less points in the boundary area compared to M4 in Figure 3.13. As can be seen from Figure 3.15, the results with the padded scheme is only slightly better than with the unpadded scheme, with a less significant improvement relative to the results in Figure 3.12(a), as expected based on the previous analysis.

To summarize, we conclude that, in terms of progressive coding performance, using a padded root cell works better than an unpadded one, and the improvement becomes less

Table 3.8: Comparison of the lossless coding performance with different root cell selection strategies

	Rate (bits/vertex)					
	Unpadded (`usePaddedRootCell` = 0)			Padded (`usePaddedRootCell` = 1)		
Individual Mesh	Total	GD†	Connectivity	Total	GD†	Connectivity
A1	22.45	14.50	7.92	**22.30**	**14.35**	**7.91**
A4	17.95	10.80	7.14	**17.69**	**10.67**	**7.01**
B5	**24.75**	15.49	**9.20**	24.83	**15.31**	9.45
B8	**19.22**	11.72	**7.48**	19.46	**11.70**	7.74
CR1	23.93	16.89	7.02	**23.73**	**16.83**	**6.88**
CR4	20.11	13.35	6.75	**19.86**	**13.30**	**6.56**
L5	**27.70**	**18.37**	8.93	27.78	18.49	**8.90**
L8	**21.72**	**13.95**	7.72	21.79	14.04	**7.69**
M2	26.90	16.39	10.48	**25.78**	**16.36**	**9.38**
M3	24.90	15.14	9.74	**24.00**	**15.09**	**8.89**
M5	**33.78**	16.85	**16.86**	34.06	**16.64**	17.35
M8	**32.33**	13.11	**19.22**	32.82	**12.76**	20.05
Q3	21.61	12.61	8.99	**21.17**	**12.45**	**8.71**
Q4	19.50	11.02	8.47	**18.98**	**10.87**	**8.10**

† GD: geometry and DC information.

significant if the tested mesh has less samples near the function-domain boundary area. The conclusion is also the same for the poor-quality meshes.

Next, we consider the lossless coding performance. For the same 36 meshes as used in the previous experiments, we losslessly code each of them with padded and unpadded root cells, and measure the final bit rates. A representative subset of the results is given in Table 3.8. In this table, we separate the total information into two parts. One part includes the geometry and DC information, denoted as GD, and the other part includes the connectivity information. We also record and list the bit rates used to encode these two parts separately. The best results for each test case are typeset with bold font.

Examining the results in Table 3.8, we can see that for most cases, the GD bit rates obtained with the unpadded root cell are larger than those with the padded one, except for L5 and L8. For meshes L5 and L8, the unpadded root cell has the size 513×513 and the padded root cell has the size 1024×1024. The padded root cell is much larger than the unpadded one (i.e., the fraction of the unpadded size over padded is only 25.1%). With the padded root cell, the information that needs to be coded is much more than with the unpadded scheme. Therefore, the lossless bit rates obtained with the padded root cell are larger. Among the results for all test datasets, 96% of the cases have the GD coding results obtained with the padded root cell better than those with the unpadded cell. Although the

connectivity bit rates are less predictable than the GD information, still more cases (74%) have better results with the padded scheme than with the unpadded scheme. In terms of total lossless coding results, 90% of the cases have better results with the padded scheme than with the unpadded scheme. From the above results, we conclude that the lossless coding performance with different root-cell selection schemes are influenced by two factors together. Generally, on the one hand, if the root starts with the unpadded cell, the encoder needs to handle the degenerate cases. With the extra bits on degenerate cases, the overall lossless coding performance is degraded for the unpadded scheme. On the other hand, however, if the padded size is much larger than the original unpadded size, the lossless bit rates obtained with the padded root cell tend to be higher than with the unpadded cell, since much more information needs to be coded under the padded scheme.

To summarize, the padded scheme can provide a better progressive performance, and a better lossless coding performance in a majority of cases without degrading too much in other situations. Therefore, we recommend the coding starts with the padded root cell. The experiment described above was also repeated with different choices of the fixed parameters in the framework, and similar results were obtained.

3.6.2 Choice of Prioritized or Non-Prioritized CI Queue

As described in Section 3.4.1 (on page 33), the nodes on the CI queue are handled for the CP information (i.e., the geometry and connectivity information involved in the CP coding procedure) and the initial DC information. Therefore, the order of the nodes on the CI queue determines the splitting order of the node cells. If the more important nodes on the CI queue are split first, a better mesh quality will be obtained with a lower bit rate. In this regard, we observe that:

1. a cell with higher valence tends to contain more sample points;
2. a cell with larger size has more impact on the mesh quality after splitting; and,
3. a cell with larger distance from its neighbor cells implies more influence on the refinement of the mesh after splitting.

Based on the preceding observations, the importance value I of a node can be determined as

$$I = vsl, \tag{IV}$$

where v is the valence of the cell to be partitioned, s is the size of the cell, and l is the average distance between the target cell's centroid and its neighbor cells' centroids. The parameter

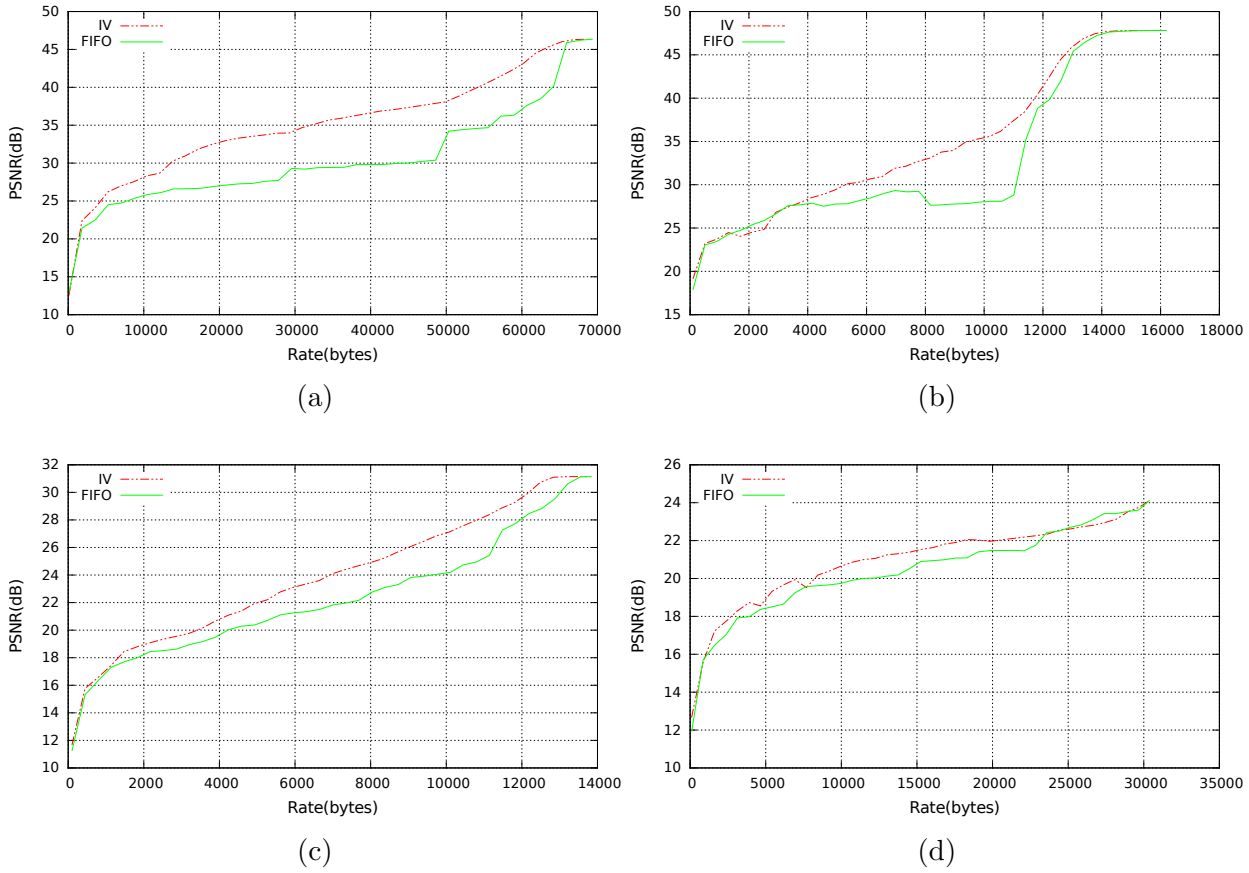
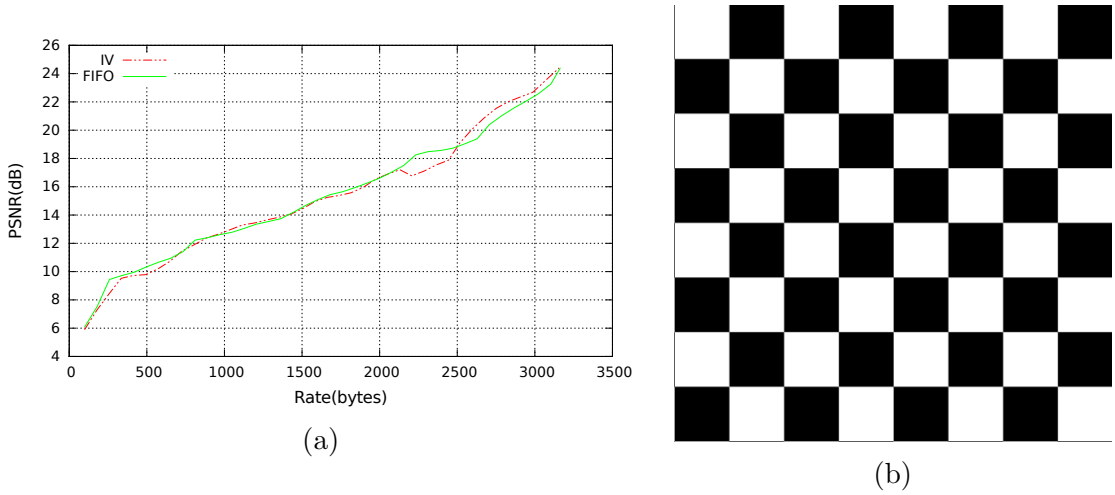


Figure 3.17: Progressive coding performance for meshes (a) A3, (b) CT3, (c) P8, and (d) L12 using different values of `usePriorityScheme`. Labels "IV" and "FIFO" represent the results obtained with `usePriorityScheme` set as 1 and 0, respectively.

`usePriorityScheme` is used to determine the order in which elements are removed from the CI queue. In the case that `usePriorityScheme` is 1, the scheme in (IV) is used; otherwise, FIFO order is used.

First, we study the influence of different values of `usePriorityScheme` on progressive coding performance by testing with the 64 meshes as listed in the Tables 3.1, 3.2, and 3.3 in Section 3.5 (on page 46). Using each of the FIFO and IV orders with `usePriorityScheme` set to 0 and 1 respectively, each mesh is encoded losslessly, and then decoded at many intermediate rates. In each case, the decoded dataset was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. The other parameters in the framework are set to the default values mentioned earlier at the beginning of Section 3.6 (on page 52). We choose a representative subset of these results involving several datasets, which is shown in Figure 3.17. Each of the four graphs in the figure shows the PSNR of the reconstructed image plotted against rate. On each graph, the far left corresponds to no information having been decoded, while the far right corre-



(a)

(b)

Figure 3.18: (a) Progressive performance for the mesh CH2 using different values of `usePriorityScheme` (1 labeled with "IV", and 0 labeled with "FIFO"). (b) The original image `checkerboard` used to generate CH2.

sponds to the coded bitstream having been fully decoded. The maximum PSNR obtained corresponds to lossless reconstruction of the mesh. In each graph, the results obtained with `usePriorityScheme` set to 1 and 0, are labeled with "IV" and "FIFO", respectively.

Now, we examine the results in Figure 3.17 more closely. The results in Figure 3.17 are obtained for the four test datasets A3, CT3, P8, and L12. The first three meshes have good quality and the last one has poor quality. In Figures 3.17(a), (b), and (c), it is clear that the PSNR results obtained with the IV order are better (often by several dB) for the image reconstructions than those obtained with the FIFO order. Since L12 is a mesh with poor quality, the result in Figure 3.17(d) suggests that the IV order also works better for poor mesh quality. The reason is, with the IV order, more important nodes are coded first, so more information for a given bit rate can be decoded. Another representative subset of the results, namely, for the `checkerboard` image, is given in Figure 3.18. The `checkerboard` image is illustrated in Figure 3.18(b). From this figure, we can see that the black and white blocks are distributed evenly in the image domain. Unlike an image like `woman` or `animal`, the `checkerboard` image has less varying content and the mesh information is more evenly distributed in the tree nodes. In Figure 3.18(a), the results with different values of `usePriorityScheme` are very close to each other. Therefore, if the mesh information is less varying, the benefit of using the IV order disappears since the information stored in the tree nodes tends to be equally important. From the above analysis, we conclude that in terms of progressive performance, the results with `usePriorityScheme` set to 1 are better relative to those with the parameter set to 0. If the information of the datasets is distributed more evenly, the improvement is less significant.

Table 3.9: Comparison of the lossless coding performance with different values of `usePriorityScheme`. (a) Individual results for seven datasets, and (b) overall average results for meshes in different categories.

(a)

	Rate(bit/vertex)	
Individual Mesh	1 (IV order)	0 (FIFO order)
A1	**22.27**	22.42
B6	**22.73**	22.90
CH3	**20.34**	20.42
CT4	**23.37**	23.49
L8	**21.75**	21.85
M2	**25.76**	25.86
P4	**17.45**	17.50

(b)

	Rate (bit/vertex)	
Category	1 (IV order)	0 (FIFO order)
A	**19.03**	19.16
B	**22.69**	22.78
C	**32.26**	32.30

Next, the influence of different values of `usePriorityScheme` on lossless coding performance is studied. For the same 64 meshes as listed in Section 3.5, we losslessly encode each mesh with each of the IV and FIFO orders, and then measure the final bit rates. The other fixed parameters in the framework are still set to default values (as shown on page 52). A representative subset of these results for several individual datasets is given in Table 3.9(a), and the average results for all meshes based on categories are given in Table 3.9(b). The different categories of meshes were listed earlier in Tables 3.1, 3.2, and 3.3 (on page 47). In Tables 3.9(a) and (b), the best result in each case is typeset with bold font. As can be seen in these two tables, the results obtained with the IV order are better than those obtained with the FIFO order. From the results of all test datasets, we obtain the statistics that the lossless bit rates with the IV order are decreased compared to the results with the FIFO order, by 0.75%, 0.43%, and 0.14% for meshes in categories A, B, and C, respectively. With more important nodes being handled first to provide more information, the estimation of the later information is more accurate. As a result, the coding efficiency can be improved.

Besides the lossless bit rates, we also consider the time cost with the different values of `usePriorityScheme`. For the purpose of making time measurements, very modest hardware was employed, namely, a 13-year-old computer with a 3.16 GHz Intel Core2 Duo CPU and 4.0 GB of RAM. To get the precise timing results, we run each test case for the encoder

Table 3.10: Comparison of the lossless coding performance with different values of `usePriorityScheme`. (a) Individual results for seven datasets, and (b) overall average results for meshes in different categories.

(a)

	Time (s)†	
Individual Mesh	1 (IV order)	0 (FIFO order)
A1	0.84	**0.64**
B6	0.87	**0.66**
CH3	0.20	**0.17**
CT4	0.77	**0.60**
L8	0.51	**0.40**
M2	0.93	**0.67**
P4	0.98	**0.70**

(b)

	Time (s)†	
Category	1 (IV order)	0 (FIFO order)
A	0.77	**0.55**
B	1.80	**1.24**
C	1.91	**1.50**

† Time: Encoding time in seconds, chosen as the median value among the 9 times running for each case.

nine times, measuring the coding time in each case, and then compute the median of these values. A representative subset of these results for several individual datasets is listed in Table 3.10(a), and the average results for all meshes based on categories are summarized in Table 3.10(b). As can be seen from Tables 3.10, the time cost with the IV order are higher relative to with the FIFO order. We consider the average time cost. The encoding procedures with the IV order consume 24.27%, 23.99%, and 16.79% more time than with the FIFO order for meshes of categories A, B, and C, respectively. Therefore, the extra cost of using the IV order is spending more time on calculating and updating the importance values for the nodes on the queue.

Summarizing the results of this section, the progressive results obtained with the IV order are better than those obtained with the FIFO order by using 3.69 to 9.83 bits/vertex less (depending on the particular dataset being coded) to achieve a similar quality of reconstructed images with PSNR values around 75% of the maximum PSNR obtained for the lossless reconstructions. Furthermore, if a mesh has a better quality and contains more varying information, the improvement IV order makes over FIFO order is more significant. For lossless coding performance, the coding bit rate obtained with the IV order is decreased by approximately 0.46% relative to with the FIFO order, with 23.14% more time cost. Because

Table 3.11: Three thresholding schemes of different values for the parameters `thresholdCI` and `thresholdRDC`

Scheme	`thresholdCI`	`thresholdRDC`
1	512	128
2	512	256
3	512	512

of the better progressive and lossless coding performance, we are willing to take the extra time-cost and recommend the parameter `usePriorityScheme` be set as 1 (i.e., using the IV order to select the next node on the CI queue). The above experiments were repeated with different choices of the fixed parameters in the framework (on page 52), and we obtained a similar conclusion.

3.6.3 Choices of Threshold Values of Two Queues

As explained earlier, the framework parameters `thresholdCI` and `thresholdRDC` are provided to control the switching between the CI and RDC queues during the coding procedure. In what follows, we study the influence of different values of `thresholdCI` and `thresholdRDC` on progressive coding performance by considering three different thresholding schemes as given in Table 3.11. In Table 3.11, the values of `thresholdCI` and `thresholdRDC` have different values in different thresholding schemes. The two values have the largest margin (i.e., 512 and 128) in scheme 1, and smallest margin (i.e., 512 and 512) in scheme 3.

The test datasets are still the 64 meshes (as listed previously in Tables 3.1, 3.2, and 3.3 in Section 3.5 on page 47). Using each of the thresholding schemes in Table 3.11, each mesh was losslessly encoded, and then decoded at many intermediate rates. In each case, the decoded mesh was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. The other parameters in the framework are set to the default values described earlier at the beginning of Section 3.6 (on page 52). A representative subset of these results for four datasets, namely, A4, B8, K4, and L10, is listed in Figure 3.19. Each of the four graphs in the figure shows the PSNR of the reconstructed image plotted against rate, where the far left corresponds to no information having been decoded and the far right corresponds to the coded bitstream having been fully decoded.

Now, we analyze the results introduced above in detail. In Figures 3.19(a), (b), and (c), it is clear that the image reconstructions generated with scheme 1 are consistently better (often less than or around one dB in terms of PSNR) than with the other two schemes. The reason is, the nodes on the first CI queue are coded for the geometry, connectivity, and initial

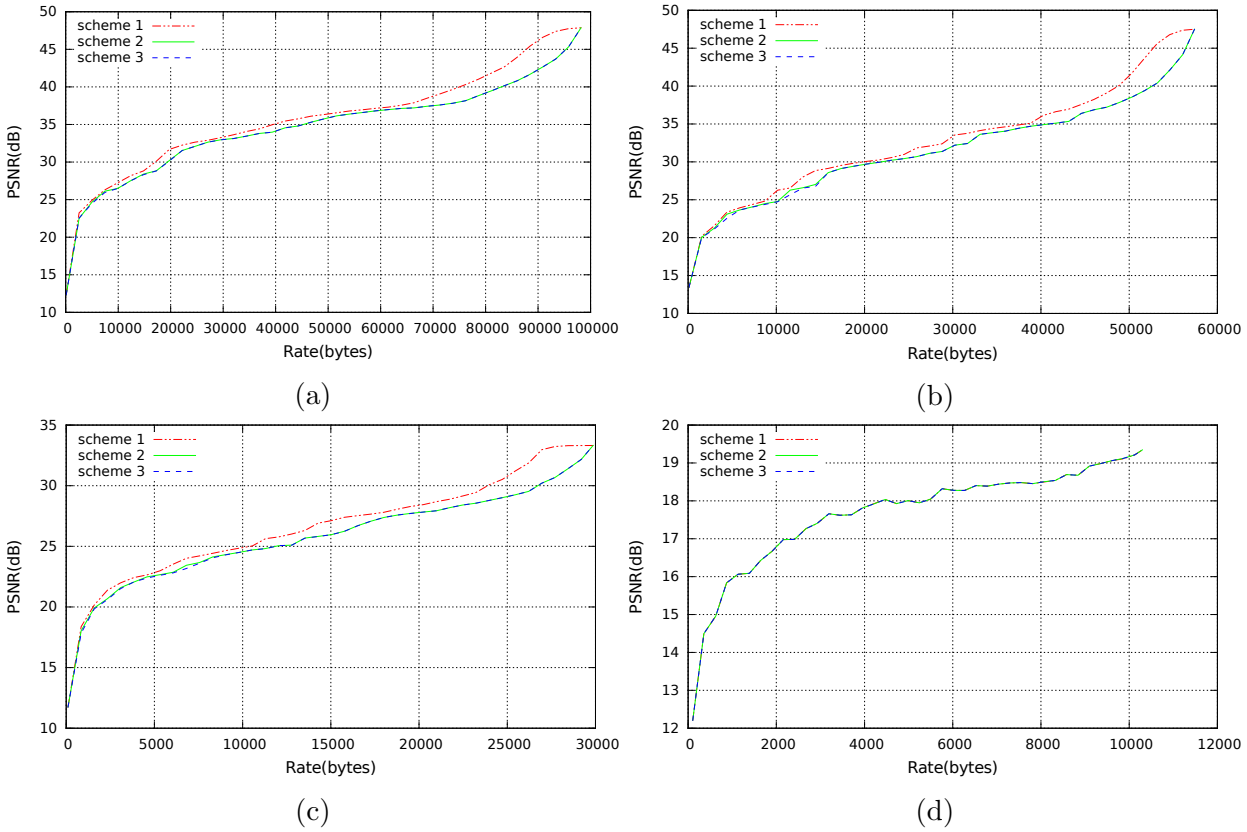


Figure 3.19: Progressive performance for meshes (a) A4, (b) B8, (c) K4, and (d) L10 using different thresholding schemes as shown in Table 3.11.

DC information and the nodes on the second RDC queue are coded for the remaining DC information. The important information to recover the structure of the mesh is provided by the geometry and connectivity information. So, we need to spend more bits on coding this important information before having unnecessarily-detailed information for recovering function values. The mesh L10 has poor quality and the results in Figure 3.19(d) obtained with different schemes are close to each other, which at least suggests that scheme 1 does not degrade the performance for the poor-quality meshes.

In terms of progressive coding performance, we conclude that the most effective choices for `thresholdCI` and `thresholdRDC` are 512 and 128, respectively. These choices do not degrade performance for poor-quality meshes. We repeated the above experiments with the different choices of the fixed parameters in the framework (on page 52), and obtained the similar results.

3.6.4 Choice of the Threshold Value for Valence

As explained earlier, in order to avoid combinatorial-explosion problems as described in Section 3.4.3 (on page 41), we use the parameter `valenceMax`. In what follows, we consider the influence of different values of `valenceMax` on progressive and lossless coding performance.

First we study the influence of `valenceMax` on progressive coding performance. In this experiment, we use the 64 meshes listed in Tables 3.1, 3.2, and 3.3 in Section 3.5 (on page 47). We consider four values for `valenceMax`, namely, 8, 10, 12, and 14. With each of the values of `valenceMax` under consideration, each mesh is losslessly encoded, and then decoded at many intermediate rates. In each case, the decoded dataset was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. The other parameters in the framework are set to default values (on page 52). A representative subset of these results involving two datasets is illustrated in Figure 3.20. Each of the two graphs in the figure shows the PSNR of the reconstructed image plotted against rate, where the far left corresponds to no information having been decoded and the far right corresponds to the coded bitstream having been fully decoded. The maximum PSNR in the graphs corresponds to the lossless reconstruction of the mesh.

Now, we analyze the results from above in detail. The results of progressive performance with different `valenceMax` are very close for the two datasets L16 and A4, as shown in Figures 3.20(a) and (b), respectively. In order to examine the results more closely, we provide specific numerical values in Table 3.12 for the mesh L16. In this table, the PSNR values of the reconstructed images obtained at different intermediate rates are given. At each decoding rate, the best result obtained with different `valenceMax` is typeset with bold

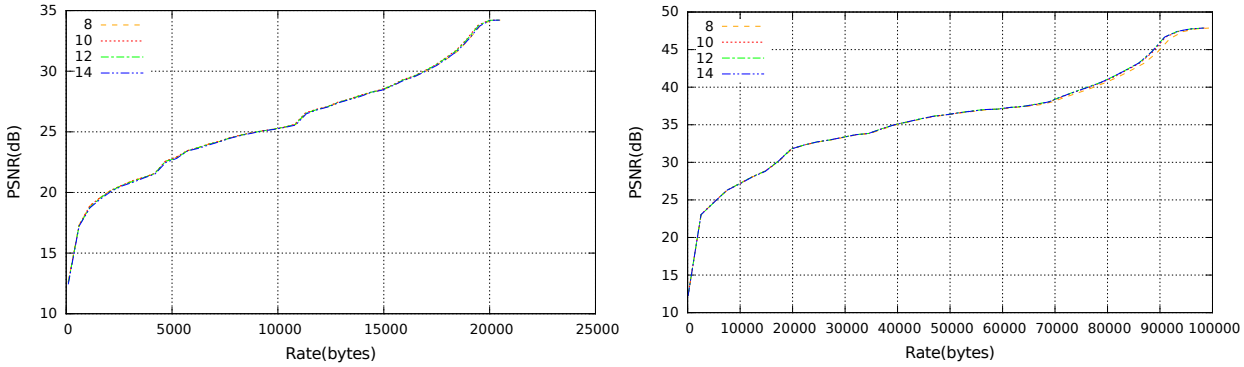


Figure 3.20: Progressive performance for meshes (a) L16 and (b) A4 using different `valenceMax` as 8, 10, 12, and 14.

Table 3.12: Reconstruction quality at various (lossy) decoding rates for the mesh L16

	PSNR (dB)			
Decoded Bytes	`valenceMax` = 8	`valenceMax` = 10	`valenceMax` = 12	`valenceMax` = 14
100	13.49	13.43	**13.70**	13.65
1100	**18.11**	18.05	17.91	17.87
5100	**21.66**	21.66	21.62	21.60
10100	**24.31**	24.31	24.29	24.27
15100	28.04	**28.09**	28.01	27.91
20100	34.20	**34.20**	34.20	34.20

Table 3.13: Lossless bit rates for meshes (a) L16 and (b) A4 using different `valenceMax` as 8, 10, 12, and 14

(a)

`valenceMax`	Bit Rate (bits/vertex)
8	20.83
10	**20.73**
12	20.75
14	20.81

(b)

`valenceMax`	Bit Rate (bits/vertex)
8	17.89
10	17.74
12	**17.69**
14	17.70

font. From the data in the table, we can clearly see the PSNR values at each rate are very close with no value of `valenceMax` consistently performing better than the others. Thus, we conclude that the choice of `valenceMax` does not have a significant impact on the progressive coding performance.

Next, we study the influence of `valenceMax` on lossless coding performance. Again, we use the 64 meshes as listed in Section 3.5 (on page 47), and consider the same four values for `valenceMax` (i.e., 8, 10, 12, and 14). We losslessly code each of the meshes with the values of `valenceMax` under consideration, and measure the final bit rates. The other parameters in the framework are still set to default values (as on page 52). A representative subset of these results is given in Table 3.13. The best result for each test case is typeset with bold font. Recall that the parameter is a threshold related to valences of vertices being split. This fact motivates us to examine the statistics of the valences for all the vertices being split during the coding procedure.

To order to explain the results in Table 3.13, we collect some statistics for the vertex valences encountered during the coding procedure by recording the valence of each vertex being split. The resulting statistics are given in Tables 3.14 and 3.15, for the two datasets L16 and A4, respectively. First, we consider the results for L16. Among the total 7863 vertex splits, 2010, 432, and 60 vertices have the valences exceed 8, 10, and 12, respectively. For mesh A4, we obtain the statistics that among all 44381 vertex splits, 2418 and 418 cases with valences exceed 10 and 12. Recall that the number of pivots P is coded conditioned on M, and the index of pivot-vertex tuple is coded conditioned on M and P (as described on page 38). If M is set to a sufficiently large number, context dilution will occur and the

Table 3.14: The numbers of vertices being split with specific valences (i.e., 0, 1, ..., 19) during the coding procedure for mesh L16

Valence	0	1	2	3	4	5	6	7	8	9
Counts	1	1	6	57	281	823	1498	1687	1499	1030
Valence	10	11	12	13	14	15	16	17	18	19
Counts	548	269	103	37	14	5	3	1	0	0

Table 3.15: The numbers of vertices being split with specific valences (i.e., 0, 1, 2, ...) during the coding procedure for mesh A4

Valence	0	1	2	3	4	5	6	7	8	9
Counts	1	1	5	230	1343	4352	8176	9973	8953	5882
Valence	10	11	12	13	14	15	16	17	18	$\geq$19
Counts	3047	1386	614	234	102	41	22	8	4	7

Table 3.16: (a) The numbers of vertices being split with specific valences (i.e., 0, 1, 2, ...) during the coding procedure for mesh L12, and (b) lossless bit rate for L12 using different `valenceMax`

(a)

Valence	0	1	2	3	4	5	6	7	8	9
Counts	1	1	2	55	196	415	646	762	799	757
Valence	10	11	12	13	14	15	16	17	18	19
Counts	725	622	572	472	403	340	263	223	168	98
Valence	20	21	22	23	24	25	26	27	28	$\geq$29
Counts	96	57	55	36	27	19	6	8	7	32

(b)

`valenceMax`	Bit Rate (bits/vertex)
8	**30.84**
10	30.86
12	30.92
14	31.08

efficiency of arithmetic coding will be degraded. For mesh L16, only 432 vertices being split have more than 10 neighbors, and for mesh A4, only 418 vertices being split have more than 12 neighbors. So, for L16, if `valenceMax` is set to a value larger than 10, the overall lossless coding performance is degraded as shown in Table 3.13(a). For A4, the overall lossless coding performance is degraded with `valenceMax` set to a larger value than 12, as shown in Table 3.13(b).

Based on the above analysis, we conclude that the influence of `valenceMax` on lossless coding performance is related to the statistics of the valences of the vertices being splits during coding. If more vertices being split have larger valences, the lossless results obtained with a larger value of `valenceMax` tend to be better than those obtained with smaller values. If less vertices being split have larger valences, the lossless results with a smaller value of `valenceMax` tend to be better.

We also consider the case of poor-quality meshes. For poor-quality meshes, the valences during the vertex splits are distributed more evenly on a larger range. To better illustrate the situation for poor-quality meshes, a representative dataset L12 is considered in what follows. Table 3.16(a) shows the statistics for the vertex valences encountered during the coding procedure, from where we can see that the valence distribution is wider compared with the preceding two examples. The lossless coding bit rates for L12 with different `valenceMax` are listed in Table 3.16(b). As can be seen from Table 3.16(b), the lossless coding performance is less predictable, but the results obtained with `valenceMax` chosen as 12 is not too bad.

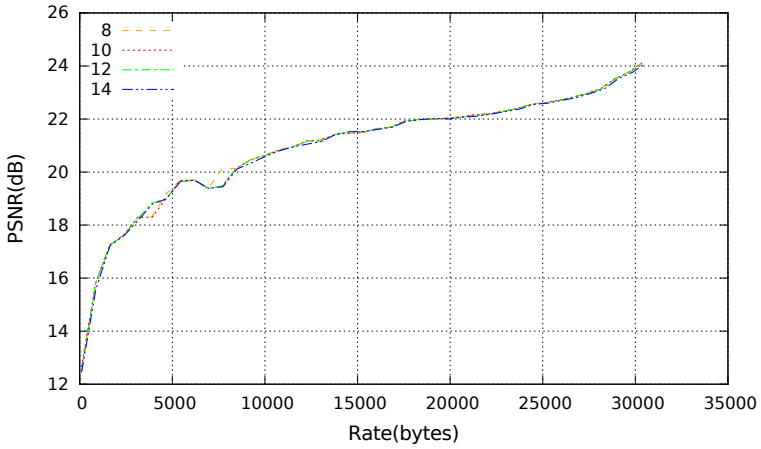


Figure 3.21: Progressive performance for mesh L12 using different `valenceMax` as 8, 10, 12, and 14.

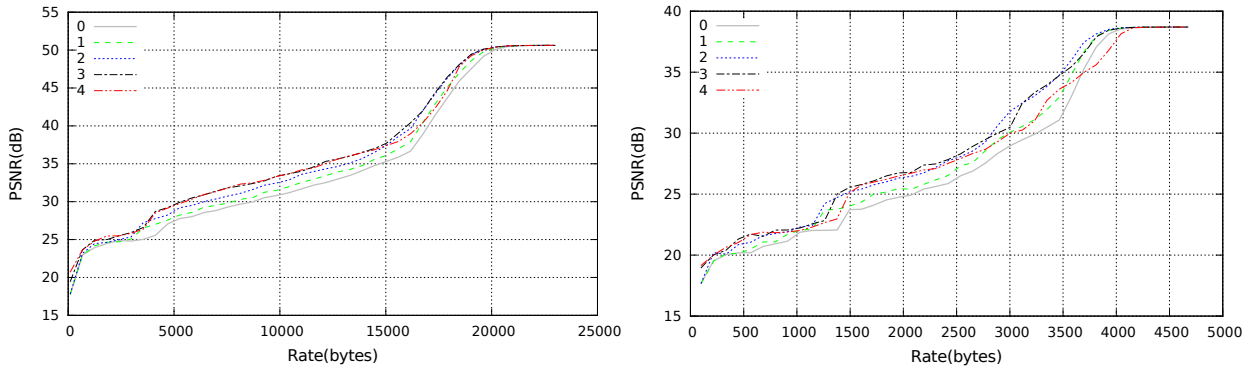
The results of progressive coding performance with different `valenceMax` are still very close to each other, as shown in Figure 3.21.

Based on the above observations, we recommend 12 as the best choice of `valenceMax`, since it can yield better lossless coding performance for most cases, without degrading the performance too much in other cases. We also repeated the previous experiments with the different choices for the fixed parameters (on page 52), and obtained the similar results.

3.6.5 Choices of Invoking of the DC Coding Procedure

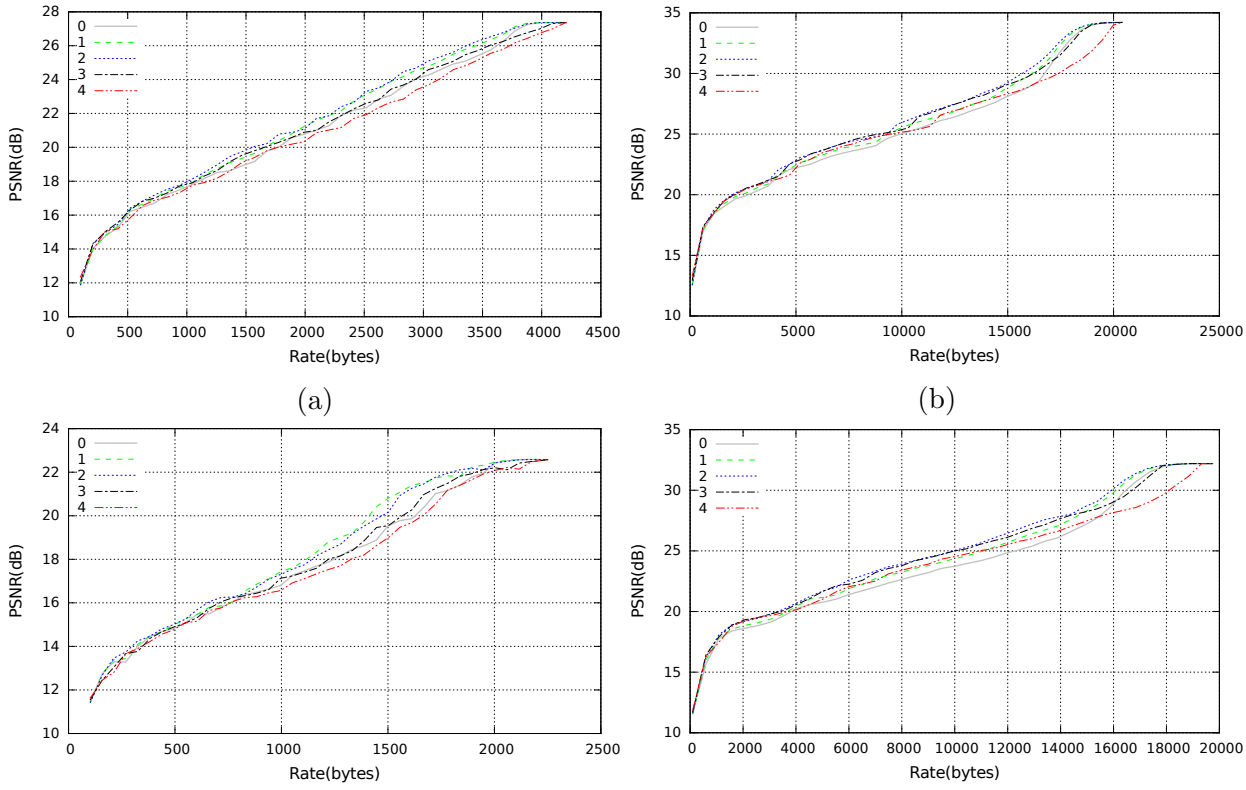
Next, we consider the influence of the framework parameters `initialDC` and `remainDC` on progressive coding performance. We start with the first parameter `initialDC` involved in the CI queue, and then study the `remainDC` parameter involved in the RDC queue.

Parameter `initialDC`. First, we study how different choices of the `initialDC` parameter affect the progressive performance. The datasets used in this part are the 64 meshes as listed in previous Tables 3.1, 3.2, and 3.3 in Section 3.5 (on page 47). We consider the values 0, 1, 2, 3, and 4 for `initialDC`. Using each of the values for `initialDC` under consideration, each of the meshes is losslessly encoded, and then decoded at many intermediate rates. In each case, the decoded mesh was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. During the experiments, the other parameters in the framework, including `remainDC`, are set to the default values as listed earlier (on page 52). A representative subset of these results is illustrated in Figures 3.22 and 3.23. Each of these graphs shows the PSNR of the reconstructed image plotted against rate. On each graph, the far left corresponds to no information having been decoded, while the far right corresponds to the coded bitstream



(a) (b)

Figure 3.22: Progressive performance obtained with `initialDC` set as 0, 1, 2, 3, and 4 for meshes (a) CT4 (sampling density is 0.03) and (b) CT1 (sampling density is 0.005).



(a) (b)

(c) (d)

Figure 3.23: Progressive performance for meshes (a) L13 (sampling density is 0.005), (b) L16 (sampling density is 0.03), (c) P4 (sampling density is 0.0025), and (d) P9 (sampling density is 0.03) using different values of `initialDC` as 0, 1, 2, 3, and 4.

having been fully decoded.

Now, we analyze the results introduced above in detail. The results in Figure 3.22 are obtained for the meshes CT4 and CT1, which are generated from the `ct` image with the sampling densities 0.03 and 0.005, respectively. The number of bits used to represent the function values for the `ct` image is 12 (i.e., $\rho = 12$). Examining the results in Figure 3.22(a), we can see that the results with `initialDC` set to 3 and 4 are close to each other during most stages of the progressive decoding and are better than the results with `initialDC` set to the other values. In Figure 3.22(b), the results with `initialDC` set to 2 and 3 are very close and are better than the result with `initialDC` set to 4. Therefore, choosing `initialDC` as 3 is a good tradeoff. Another four sets of results are shown in Figure 3.23, obtained for the four datasets L13, L16, P4, and P9. The meshes, L13 and L16, are generated from the `lena` image with the sampling densities 0.005 and 0.03, respectively. The meshes, P4 and P9, are generated from the `peppers` image with the sampling densities 0.0025 and 0.03, respectively. Both of the `lena` and `peppers` images use 8 bits to represent the function values (i.e., $\rho = 8$). Examining the results in Figure 3.23, we can make the following observations. The results in Figures 3.23(a) and (c) are obtained for the meshes with lower-sampling densities, and the results obtained with `initialDC` set to 2 and 1 are better than those obtained with the other values. The results in Figures 3.23(b) and (d) are obtained for the meshes with higher-sampling densities, and the results obtained with `initialDC` set to 2 and 3 are better than those obtained with the other values. So, for the the above four test cases, choosing `initialDC` as 2 is a good tradeoff. Compared with the previous two results in Figure 3.22, the four results in Figure 3.23 lead to a lower recommended value (i.e., 2), which is because the first two datasets have a higher ρ than the latter four datasets.

Based on the above results, we observe that the influence of `initialDC` on progressive performance is related to ρ and the sampling density of the mesh. To summarize, if the mesh has a lower sampling density, it is better to choose a smaller value for `initialDC`, so more bits spent on the CI queue are concentrated on coding the geometry and connectivity information in order to recover a relatively good approximation of the mesh first. Otherwise, if the mesh has a higher sampling density, the number of bits spent on the DCs can be increased in order to recover more accurate function values, so the quality of reconstructed mesh can be improved. If the mesh has a higher value of ρ to represent the function values, spending more bits on coding DCs also helps to recover higher mesh quality. Therefore, we recommend the value of `initialDC` be set to $\lfloor \frac{\rho}{4} \rfloor$, as this is a good choice for most cases. We repeated the preceding experiments with different values of the other parameters in the framework (on page 52), and the results obtained showed that the previous conclusion still holds.

Parameter `remainDC`. Next, we consider the influence of the choice of the `remainDC`

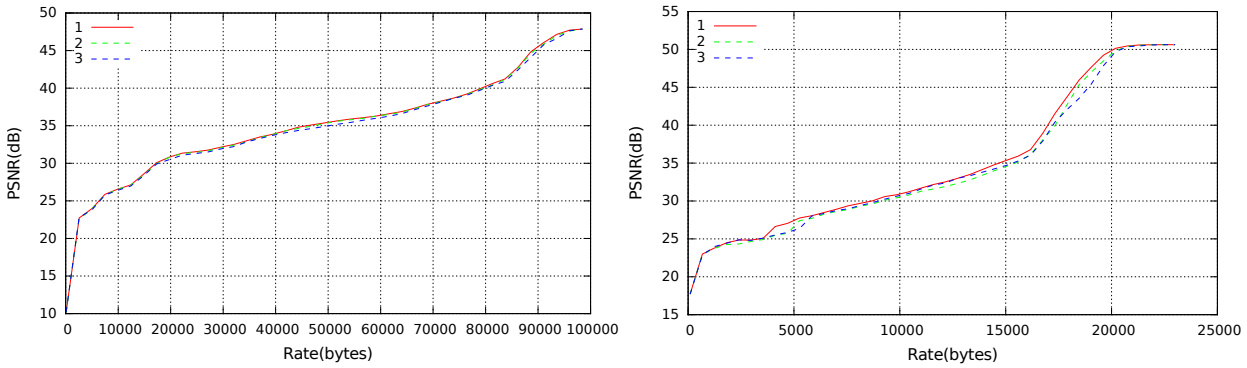


Figure 3.24: Progressive performance for meshes (a) A4 and (b) CT4 using different values of `remainDC` as 1, 2, and 3.

parameter on progressive coding performance. Again, the datasets used in this part are the same 64 meshes listed in Tables 3.1, 3.2, and 3.3 in Section 3.5 (on page 47). We consider the values 1, 2, and 3 for `remainDC`. Using each of the values of `remainDC` under consideration, each of the meshes is losslessly encoded, and then decoded at many intermediate rates. In each case, the decoded dataset was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. The other parameters in the framework, including `initialDC`, remain as the default values listed earlier (on page 52). A representative subset of the results obtained is illustrated in Figure 3.24. Each of the graphs in the figure shows the PSNR of the reconstructed image plotted against rate, where the far left corresponds to no information having been decoded and the far right corresponds to the coded bitstream having been fully decoded. The maximum PSNR obtained corresponds to the lossless reconstruction of the mesh.

Now, we analyze the above results in detail. The results in Figures 3.24(a) and (b) are obtained for the two datasets A4 and CT4, respectively. As can be seen from these graphs, the results obtained with `remainDC` set as 1 are slightly better than those obtained with other values. We also consider the case of poor-quality meshes. To better illustrate the situation for poor-quality meshes, a representative result for a poor-quality mesh L11 is presented in Figure 3.25, from where we can see the difference between the various results is difficult to recognize. Therefore, different values of `remainDC` do not have significant impact on progressive performance for poor-quality meshes. Based on all of the test results, we recommend that the `remainDC` parameter be chosen as 1. We also repeated the above experiments with nondefault values for the other parameters in the framework (on page 52), and obtained the similar results. To summarize, the recommended values for the parameters `initialDC` and `remainDC` are $\lfloor \frac{\rho}{4} \rfloor$ and 1, respectively.

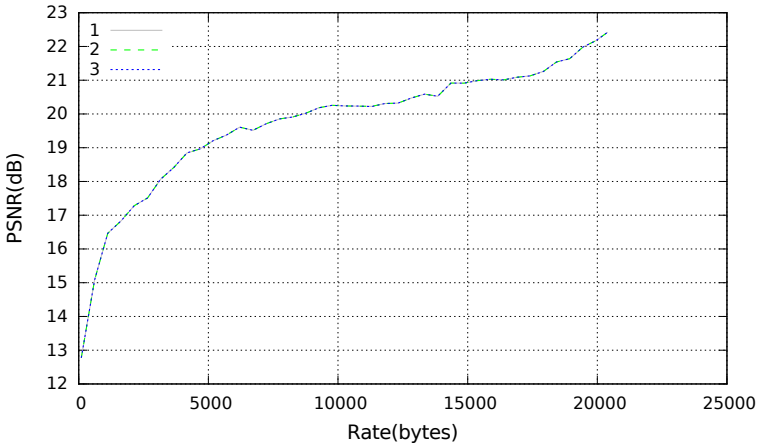
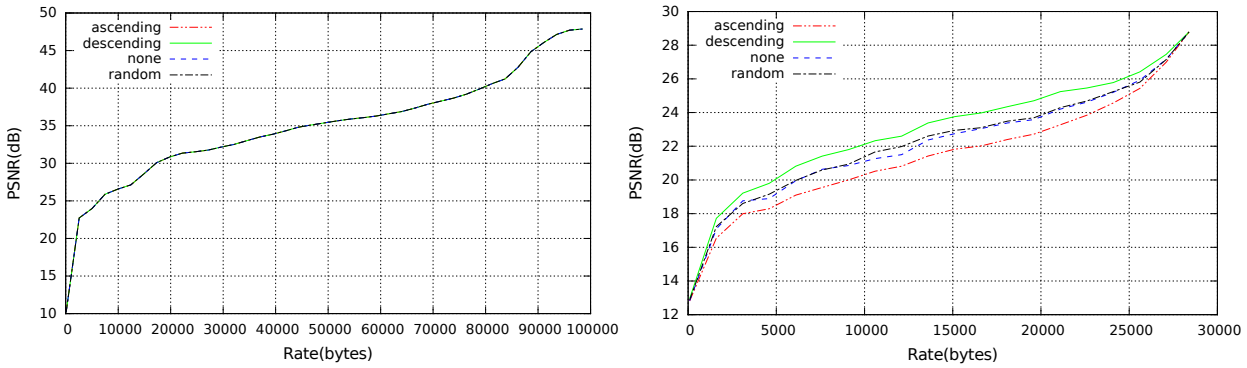


Figure 3.25: Progressive performance for poor-quality mesh L11 using different values of `remainDC` as 1, 2, and 3.



(a) (b)

Figure 3.26: Progressive performance using different orders for inserting constraints for (a) a good-quality mesh A4 and (b) a poor-quality mesh L12.

3.6.6 Choice of Insertion Order of Edge-Constraints

In what follows, we consider the influence of the choice of `edgeInsertion` on progressive coding performance. The values are considered for `edgeInsertion` are 0, 1, 2, and 3, which correspond to the following orders:

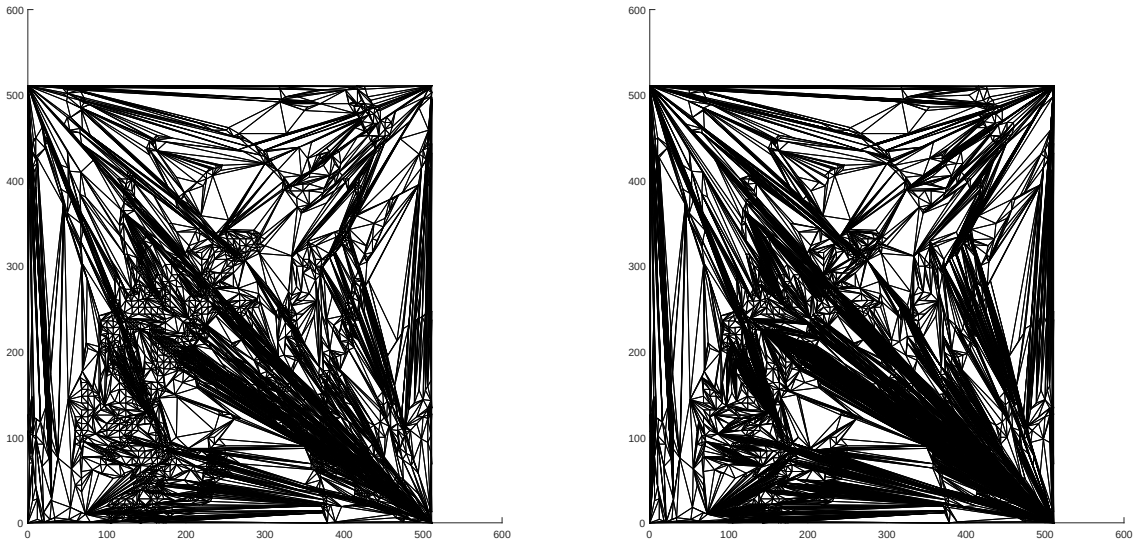
- edge-iteration order as determined by the data structure employed in the implementation;
- in order of descending edge length;
- in order of ascending edge length; and
- pseudo-random order.

As before, for experimental purposes, we use the 64 meshes listed in Tables 3.1, 3.2, and 3.3 in Section 3.5 (on page 47). Using each of the four orders described above, each mesh was losslessly encoded, and then decoded at many intermediate rates. In each case, the decoded dataset was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. The other parameters in the framework are again set to the default values described earlier (on page 52). A representative subset of these results involving two datasets is illustrated in Figure 3.26. Each of the two graphs in this figure shows the PSNR of the reconstructed image plotted against rate. On each graph, the far left corresponds to no information having been decoded, while the far right corresponds to the coded bitstream having been fully decoded. The results labeled with "none", "descending", "ascending", and "random" in the figure are obtained with the `edgeInsertion` parameter set to 0, 1, 2, and 3.

Table 3.17: Reconstruction quality at various (lossy) decoding rates, obtained with different insertion orders, for the mesh A4

	PSNR (dB)			
Decoded Bytes	none	descending	ascending	random
100	12.330	12.330	12.330	12.330
2542	**23.195**	23.202	23.212	23.203
4983	24.765	24.772	**24.757**	24.772
7423	26.275	26.278	**26.260**	26.271
9865	27.157	**27.154**	27.184	27.166
12308	28.216	28.214	28.207	**28.191**
14746	28.777	28.777	**28.759**	28.761
17188	30.139	**30.129**	30.133	30.140
19629	31.714	31.693	31.707	**31.656**
22071	**32.273**	32.289	32.310	32.274
24511	32.721	32.741	32.688	**32.677**
26952	33.135	33.130	33.131	**33.119**
29392	33.620	33.623	33.629	**33.603**
31834	33.799	33.813	**33.791**	33.795
34274	34.261	34.252	**34.251**	34.263
36716	**34.540**	34.542	34.553	34.581

Now, we analyze the results from above in detail. The representative results in Figure 3.26 are obtained for the meshes A4 and L12, which have the good and poor qualities, respectively. For good-quality meshes, we can see from Figure 3.26(a) that different insertion orders do not have a significant impact on the progressive performance. In order to examine the results more closely, we provide specific numerical values in Table 3.17 for the mesh A4. In this table, the PSNR values of the reconstructed images at different intermediate rates are given. At each decoding rate, the best result obtained with different insertion orders is typeset with bold font. From the data in the table, we can clearly see that the PSNR values at each stage are very close with no specific order consistently performing better than the others. Thus, we conclude that the choice of different insertion orders do not have a predictable impact on the progressive coding performance for the good-quality meshes. In terms of poor-quality meshes, we can see from the representative result, as illustrated in Figure 3.26(b), that the PSNR values for the image reconstructions generated using different insertion orders are significantly different. In particular, inserting the edge-constraints in the length-descending order can yield image reconstructions with higher PSNR (often by several dB) than in the length-ascending order. To better understand the influence of the different edge-insertion orders on the poor-quality meshes, we illustrate two lossy reconstructed meshes for L12 in Figure 3.27. The lossy meshes in Figure 3.27 are reconstructed by decoding 20000 bytes from



(a) (b)

Figure 3.27: Reconstructed meshes for L12 at lossy decoding rate (20000 bytes) using different edge insertion orders: (a) length-descending order and (b) length-ascending order.

the lossless coded bitstream, but with different insertion orders when generating the constrained Delaunay triangulation. As can be seen, the mesh in Figure 3.27(a) generated with the length-descending order has a better connectivity (showing inherent geometric structure more clearly) than the mesh in Figure 3.27(b), which is generated with the length-ascending order. The reason is, for poor-quality cases, the lossy reconstructed meshes will have even worse connectivity. Inserting the constrained edges in a length-descending order can avoid the long and bizarre edges to whatever extent is possible. Therefore, the recovered connectivity in Figure 3.27(a) is less messy compared to the connectivity in Figure 3.27(b). For the good-quality meshes, the connectivity of the decoded dataset is already in good shape, so different orders of inserting constraints do not have much influence on the quality of the mesh.

Since generating the constrained Delaunay triangulation is a post-processing step, so different orders of inserting the constraints do not have any impact on lossless coding bit rate. The computational time for the lengths of edges is also negligible. Therefore, we recommend setting `edgeInsertion` as descending order, which can also cover the poor-quality datasets without significant extra cost. The above experiments were repeated with different choices of the fixed parameters (on page 52) in the framework, and the conclusion remained the same.

3.7 Proposed Method

In the preceding sections, we have studied how various free parameters influence the coding performance and made a set of recommended choices for these parameters. Thus, our proposed method is defined to correspond to these best choices from earlier and are as follows: 1) `usePaddedRootCell` is set to 1 (i.e., use padded scheme); 2) `thresholdCI` and `thresholdRDC` are set to 512 and 128, respectively; 3) `usePriorityScheme` is set to 1 (i.e., use the prioritized order determined by importance values); 4) `valenceMax` is set to 12; 5) `initialDC` and `remainDC` are set to $\lfloor \frac{\rho}{4} \rfloor$ and 1, respectively; and 6) `edgeInsertion` is set to 1 (i.e., the order of descending edge length).

3.8 Differences Between Proposed Method and the ADIT and PK Methods

For the convenience of the reader, we now highlight the differences between the proposed method and the ADIT and PK schemes, upon which the proposed method is based. In particular, the proposed method borrows ideas from the ADIT method in terms of geometry and function value coding, and borrows ideas from the PK method in terms of connectivity coding.

3.8.1 Two Queues and DC Information

As explained earlier, the proposed method uses two queues, namely, the CI and RDC queues. The ADIT method also uses two queues, which play similar roles as the CI and RDC queues in the proposed method. The CI queue in the proposed method uses a priority calculation scheme that is dependent on the data set (as described on page 57), while the CI queue in the ADIT method uses a fixed priority scheme that causes nodes to be coded in breadth-first traversal order. Besides the different priority schemes, another difference between the CI queues of the two methods is the coded information for the nodes on the queues. In the ADIT scheme, the nodes from the CI queue are only coded for the geometry information without connectivity information, and all the DC information are left to be coded from the RDC queue. In the proposed method, however, the nodes from the CI queue are coded for the geometry and connectivity information, as well as the initial part of the DC information. Based on the experimental results shown in Section 3.6.5 (on page 70), we can see coding some initial DC information in the CI queue is better than leaving all DC information to the RDC queue for the proposed method.

In the ADIT method, the thresholding schemes used for the two queues, `thresholdCI` and `thresholdRDC`, are set to 512 and 256, respectively. In the proposed method, however, these parameters are set to 512 and 128. In the proposed method, more information needs to be coded for the nodes from the first queue compared to the ADIT method. Therefore, the two parameters `thresholdCI` and `thresholdRDC` in the proposed method should have a larger margin.

3.8.2 Geometry Information

The data structure used to represent the dataset in the ADIT method is a quadtree, generated by a quadtree partitioning of the root cell. The representation of the dataset in the proposed method with the QCP operations is similar as the quadtree in the ADIT method. Both of these two methods need to code the number (T) of nonempty subcells and the configuration of the nonempty subcells, but they code the value T in different ways.

In the ADIT method, T is arithmetically coded conditioned on M, P, and the level of the target cell on the quadtree. The quantity P is the prediction of T using the information of previously-coded neighbors, and the neighbor is defined based on the spatial closeness of the cells. In the proposed method, T is also coded based on the information of previously-coded neighbors, but the neighbor is defined based on the actual connectivity of the original mesh, which is more accurate than the spatial closeness.

3.8.3 Connectivity Information

The idea of coding the connectivity information is derived from the PK method. One of the differences between the proposed and the PK methods is that the proposed method uses the divide-and-conquer approach to handle large-valence vertices (as described in Section 3.4.3 on page 42). The large neighbor list is partitioned into several small sublists, and each sublist is handled in a practical efficient way without the risk of combinatorial blowup. Since the PK method does not have such a partitioning scheme, it would likely fail in practice when handling large-valence vertices.

Another difference in the coding of connectivity information relates to the updating of the face information. In the proposed method, the face information is not of concern in the 2.5-D dataset. We are only concerned with the connectivity (i.e., the triangulation) and the locations and function values of the sample points. If we consider the basic linear interpolation on the dataset, the linear faces can be determined by the constrained Delaunay triangulation generated in the post-processing step. In the PK method, however, the faces are updated from the edge-based connectivity after each of the vertex splits during the coding

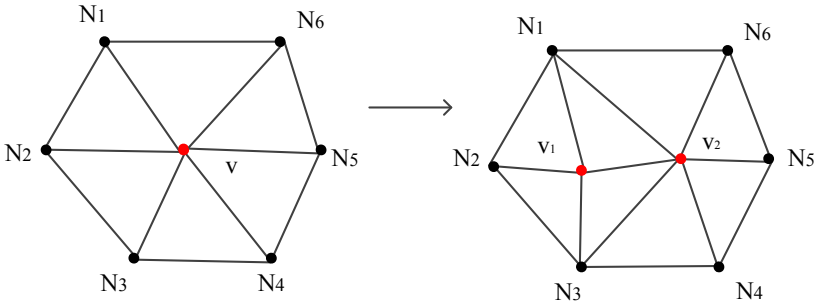


Figure 3.28: An example of vertex split. Vertex v is split into two new vertices v_1 and v_2.

procedure.

3.9 Additional Comments on the PK method

As stated before, the face information in the PK method is not explicitly coded, and can be updated from the edge-based connectivity. During the course of the work described herein, the author discovered that the rules used to update the face information in the PK method are not correct in certain circumstances. In what follows, we first introduce the face-updating rules in the PK method, and then use a simple 3-D triangle mesh as an example to illustrate the condition when the rules fail.

In the PK method, after each vertex split, the incident faces are updated as follows:

1. Suppose that A_1 is the vertex to be split in the incident face $\triangle A_1A_2A_3$, and the splitting generates two new vertices v_1 and v_2. The updating rules of the faces are given by

   (a) if both A_2 and A_3 are connected to v_1, add face $\triangle v_1A_2A_3$;

   (b) if both A_2 and A_3 are connected to v_2, add face $\triangle v_2A_2A_3$; and

   (c) delete $\triangle A_1A_2A_3$.

2. If v_1 and v_2 are connected, for each pivot N_i, add faces $\triangle v_1v_2N_i$, $i = 1, 2, \ldots, P$, where P is the number of pivots.

In the example of vertex split illustrated in Figure 3.28, the original vertex v has six local incident faces prior to the vertex split. After the vertex split, $\triangle N_1N_2v$ is replaced by $\triangle N_1N_2v_1$ because both N_1 and N_2 are connected to v_1, and $\triangle N_1N_6v$ is replaced by $\triangle N_1N_6v_2$ because both N_1 and N_6 are connected to v_2. Since v_1 and v_2 are connected to each other, and N_1, N_3 are pivots, extra faces $\triangle N_1v_1v_2$ and $\triangle N_3v_1v_2$ are added. Other incident faces are updated accordingly.

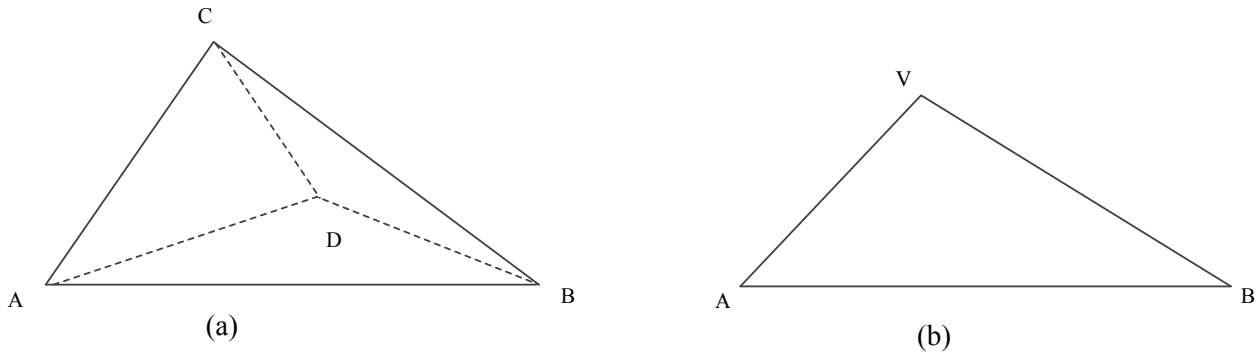


Figure 3.29: A simple 3-D triangle mesh. (a) Original mesh with four vertices (A, B, C, and D) and three faces ($\triangle ABC$, $\triangle ADC$, and $\triangle BCD$), and (b) the latest version of a coarser mesh.

In certain circumstances, however, using the preceding updating rules will cause extra faces to be added to the lossless reconstructed mesh that do not exist in the original. A simple 3-D triangle mesh is shown in Figure 3.29 to better illustrate the extra-face problem. The original mesh, as shown in Figure 3.29(a), has four vertices (i.e., A, B, C, and D) and three faces (i.e., $\triangle ABC$, $\triangle ADC$, and $\triangle BCD$). Essentially, this mesh looks like a tetrahedron but without the bottom face. Suppose that the vertices C and D are generated from the last vertex split of a previous vertex V. So the latest previous stage of a coarser mesh is shown in Figure 3.29(b). Based on the updating rules, the only face $\triangle ABV$ before vertex split can be updated in the following steps: 1) since both A and B are connected to C, face $\triangle ABC$ is added; 2) since both A and B are connected to D, face $\triangle ABD$ is added; 3) since C and D are connected, and both A and B are pivots, faces $\triangle ACD$ and $\triangle BCD$ are added; and 4) the original face $\triangle ABV$ is removed. Therefore, after the last vertex split, the reconstructed mesh has four vertices with four faces. The lossless reconstructed mesh has one extra face $\triangle ABD$ compared with the original mesh. In the proposed method, this extra-face problem does not occur, since the linear-interpolated faces of the mesh are determined implicitly by the constrained Delaunay triangulation.

Chapter 4

Evaluation of Proposed Mesh-Coding Method

Having introduced our proposed mesh-coding method, we now evaluate its performance by comparing with other coding methods. In particular, the proposed method is compared to the Gzip and Edgebreaker methods for lossless coding and the MSDC method for progressive coding.

4.1 Performance Comparison With Gzip

Gzip is general-purpose data compression method. Since Gzip cannot perform progressive coding, we only compare it with the proposed method in terms of lossless coding performance. For each of the 64 test meshes as listed earlier in Tables 3.1, 3.2, and 3.3 (on page 47), we losslessly encode the mesh using each of the Gzip and proposed methods and record the lossless bit rate. We list a representative subset of these lossless coding results, for nine individual datasets, in Table 4.1(a), and summarize the overall results for all 64 test meshes based on their categories in Table 4.1(b). The best result in each case in typeset with bold font.

Examining the individual results shown in Table 4.1(a), we can see the proposed method outperforms Gzip for meshes in all three categories, which is reasonable since Gzip is not designed to handle meshes, but as a string compression algorithm. The proposed method can exploit the structure inherent in meshes, leading to much higher coding efficiency. The overall results for all 64 meshes, as shown in Table 4.1(b), are consistent with the individual results. For the Delaunay meshes (i.e., category A), the average bit rate with the proposed method is 6.27 times lower than that obtained with Gzip. For the non-Delaunay meshes with good (i.e., category B) and bad (i.e., category C) qualities, the average bit rates with the

Table 4.1: Comparison of the lossless coding performance with Gzip. (a) Individual results for nine datasets, and (b) overall average results for all meshes in three categories.

(a)

Category	Individual Mesh	Original Rate (bits/vertex)	Rate (bits/vertex)		Gzip/Proposed Ratio
			Gzip	Proposed	
A	B4	393.05	120.28	**14.89**	8.08
A	L1	316.56	118.48	**22.15**	5.35
A	L4	356.55	119.45	**16.65**	7.17
A	P4	356.58	120.00	**17.46**	6.87
B	B5	363.12	144.71	**24.77**	5.84
B	B6	370.19	148.97	**22.72**	6.56
B	CT4	378.38	152.88	**23.37**	6.54
B	L16	367.18	147.51	**20.72**	7.12
C	L12	362.99	136.48	**30.86**	4.42

(b)

Category	Original Mean Rate (bits/vertex)	Mean Rate (bits/vertex)		Gzip/Proposed Ratio
		Gzip	Proposed	
A	349.64	119.52	**19.07**	6.27
B	351.77	144.32	**23.47**	6.15
C	348.52	136.27	**31.82**	4.28
Overall	350.30	140.30	**23.72**	5.92

Table 4.2: Comparison of the lossless coding performance with Edgebreaker. (a) Individual results for five datasets, and (b) overall average results for meshes with edge-flipping distances in different ranges.

(a)

Individual Mesh	Edge-flipping Distance	Rate (bits/vertex)	
		Edgebreaker	Proposed
P7	30.30%	23.29	**22.81**
CR1	34.29%	24.26	**23.75**
CR3	40.01%	**23.62**	24.60
CH3	57.53%	**20.18**	20.34
L12	150.51%	**27.22**	30.86

(b)

Edge-flipping Distance Range	Mean Rate (bits/vertex)	
	Edgebreaker	Proposed
0	20.51	**19.05**
$(0, 37.38\%]$	21.88	**20.87**
$(37.38\%, 80\%]$	**22.67**	23.29
$> 80\%$	**28.96**	31.83

proposed method are 6.15 and 4.28 times lower than those obtained with Gzip, respectively. From the overall results, the lossless bit rate with the proposed method is roughly 5.92 times better than that obtained with Gzip. Based on the above extensive experimental results, we conclude that the proposed method outperforms Gzip in terms of lossless coding performance. Moreover, the proposed method has the progressive coding capability, which Gzip does not.

4.2 Performance Comparison With the Edgebreaker Method

Now, the performance of the proposed method is compared to that of the Edgebreaker mesh-coding method, as implemented in [36]. This comparison was done as follows. For each of the 64 meshes listed in Tables 3.1, 3.2, and 3.3 (on page 47), the mesh was losslessly encoded using the two methods under consideration and the lossless coding bit rate was recorded.

We list a representative subset of these results involving five individual datasets in Table 4.2(a) and summarize the overall results for all 64 meshes in Table 4.2(b). In Table 4.2(a), for each mesh, we give the edge-flipping distance as well as the lossless bit rate for each method. The best result in each test case is typeset with bold font. As can be observed in Table 4.2(a), the proposed method outperforms the Edgebreaker for meshes with smaller

edge-flipping distances. By collecting the average results for the 64 meshes with different edge-flipping distances as shown in Table 4.2(b), we obtain the similar results as the individual datasets. Based on the average results and some calculations, we have the following observation. The proposed method outperforms Edgebreaker for all of the Delaunay meshes, with the lossless bit rate being 7.7% less on average. For the non-Delaunay meshes with good quality, if the edge-flipping distance is smaller than 37.38%, the proposed method is better than Edgebreaker, with the average lossless bit rate being 4.8% less. If the edge-flipping distance is greater than 37.38% and smaller than 80%, Edgebreaker outperforms the proposed method with the lossless bit rate being 2.7% less on average. For the non-Delaunay meshes with poor quality, which are not of practical interest, Edgebreaker outperforms the proposed method with the average lossless bit rate being 9.9% lower.

Although the proposed method only outperforms the Edgebreaker method for meshes with smaller edge-flipping distances in terms of lossless coding performance, it has the progressive coding capability, which the Edgebreaker does not. For each of the 64 meshes as listed in Section 3.5, after each mesh is coded losslessly using the proposed method, it is decoded at many intermediate rates. The decoded dataset at each case is interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image is computed. We illustrate a representative subset of these progressive coding results involving four datasets in Figure 4.1. The results in Figure 4.1 are obtained for the four meshes CH2, P9, K3, and CR2, which have the edge-flipping distances 46.26%, 30.80%, 37.38%, and 33.73%, respectively. On each graph, the trend line represents the PSNR of the reconstructed image plotted against rate, and the far right of the trend line corresponds to the lossless bit rate with the proposed method. The vertical bar on the right side of each graph denotes the corresponding lossless bit rate with the Edgebreaker method. As can be seen from Figure 4.1, the proposed method can achieve a progressive coding capability with smaller lossless bit rates for these four datasets compared with the Edgebreaker method.

Besides the lossless and progressive performance, we are also interested at the time cost using the two methods under consideration. For the purpose of making time measurements, we employed very modest hardware, namely, a 13-year-old computer with a 3.16 GHz Intel Core2 Duo CPU and 4.0 GB of RAM. From the results of the previous 64 meshes (as listed in Section 3.5 on page 47), the average time used by the Edgebreaker method for coding the meshes in categories A, B, and C, are 0.16 s, 0.25 s, and 0.19 s, respectively. The average time used by the proposed method, however, are 0.76 s, 1.33 s, and 1.90 s for these three categories. Therefore, the proposed method is less time-efficient relative to the Edgebreaker method.

To summarize, the proposed method outperforms the Edgebreaker method in terms of

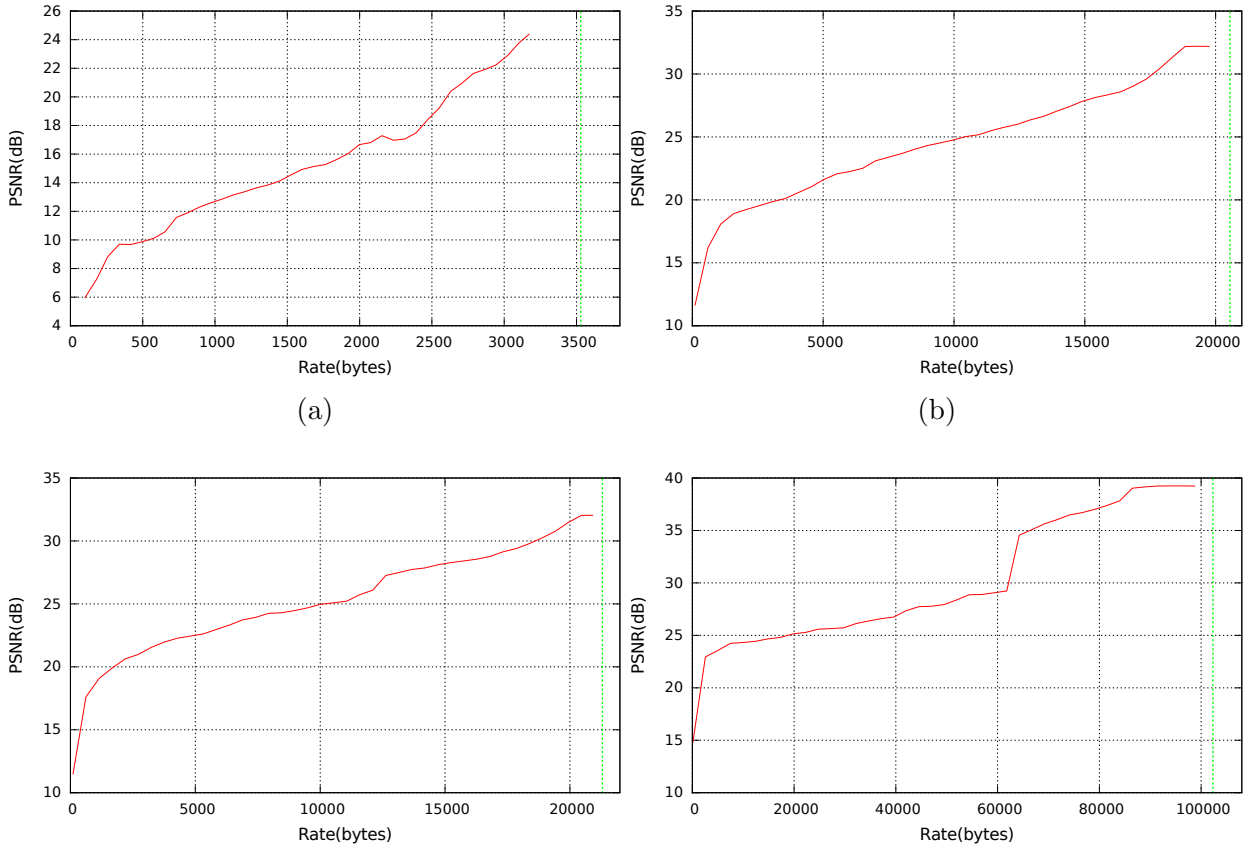


Figure 4.1: Progressive performance of the proposed method for meshes (a) CH2 (edge-flipping distance 46.26%), (b) P9 (edge-flipping distance 30.80%), (c) K3 (edge-flipping distance 37.38%), and (d) CR2 (edge-flipping distance 33.73%), with the vertical bar on the right side denoting the corresponding lossless bit rate with the Edgebreaker method.

progressive coding performance, and also outperforms the Edgebreaker in terms of lossless coding performance if the meshes do not deviate too far from (preferred-directions) Delaunay. Due to the progressive functionality, the proposed method is more computationally complex and therefore consumes more time.

4.3 Performance Comparison With the MSDC Method

So far, the evaluation of our method has been focused on lossless coding performance. Next, we consider progressive coding performance. Due to limited access to implementations of progressive mesh coders, the proposed method is compared to the MSDC method, which can only code meshes having Delaunay connectivity. So, the test datasets used in this comparison are all Delaunay meshes (i.e., from category A as listed in Table 3.1 on page 47).

For each of the twelve meshes as listed in Table 3.1, each mesh is losslessly coded, and then decoded at many intermediate rates. In each case, the decoded dataset was interpolated and rasterized to produce a lattice-sampled image and the PSNR relative to the original lattice-sampled image was computed. A representative subset of these progressive coding results involving several datasets is shown in Figure 4.2. Each of the four graphs in the figure shows the PSNR of the reconstructed image plotted against rate. On each graph, the far left corresponds to no information having been decoded, while the far right corresponds to the coded bitstream having been fully decoded. The maximum PSNR obtained corresponds to lossless reconstruction of the mesh.

Now, we examine the results in Figure 4.2 more closely. The results in Figure 4.2 are obtained for the four datasets B4, L1, L4, and P4. As can be seen from each graph, during most stages of the decoding procedure, the PSNR results obtained by the proposed method are higher (often by several dB) for the image reconstructions than those obtained with the MSDC method. Since the MSDC method does not code connectivity information, it has a smaller lossless bit rate than the proposed method. As shown in the graphs, after a certain stage when the lines for the two methods intersect, the MSDC method has higher PSNR values and the corresponding trend line terminates earlier. By studying the results for all Delaunay meshes, we have the similar observation that the proposed method has a better progressive performance at lower bit rates than the MSDC method, using 55% to 86% of the bit rate of that needed by the MSDC method to obtain similar-quality image approximations (with the PSNR value being 75% of the maximum PSNR obtained for the lossless reconstruction). At higher bit rates, both methods can generate image approximations with sufficiently high PSNR values, although the PSNR values of the MSDC method are higher than those of the proposed method.

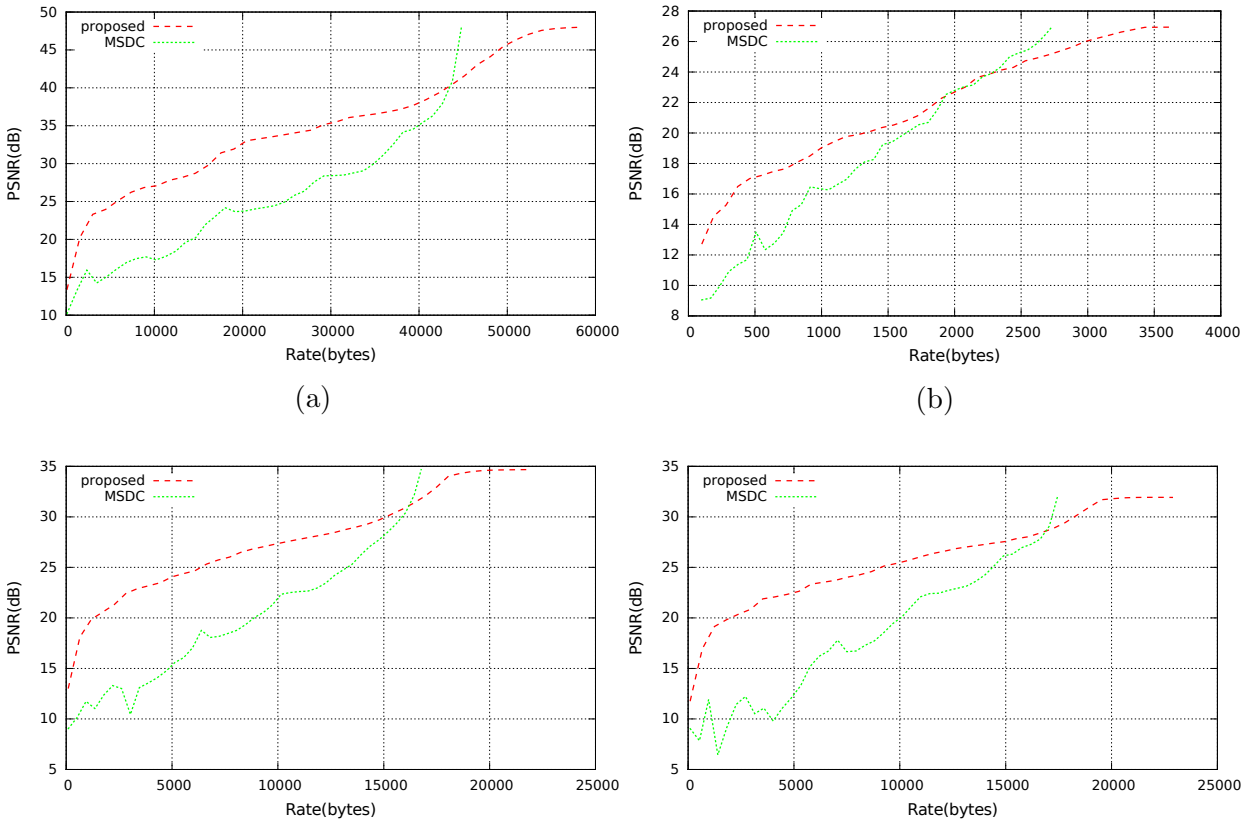
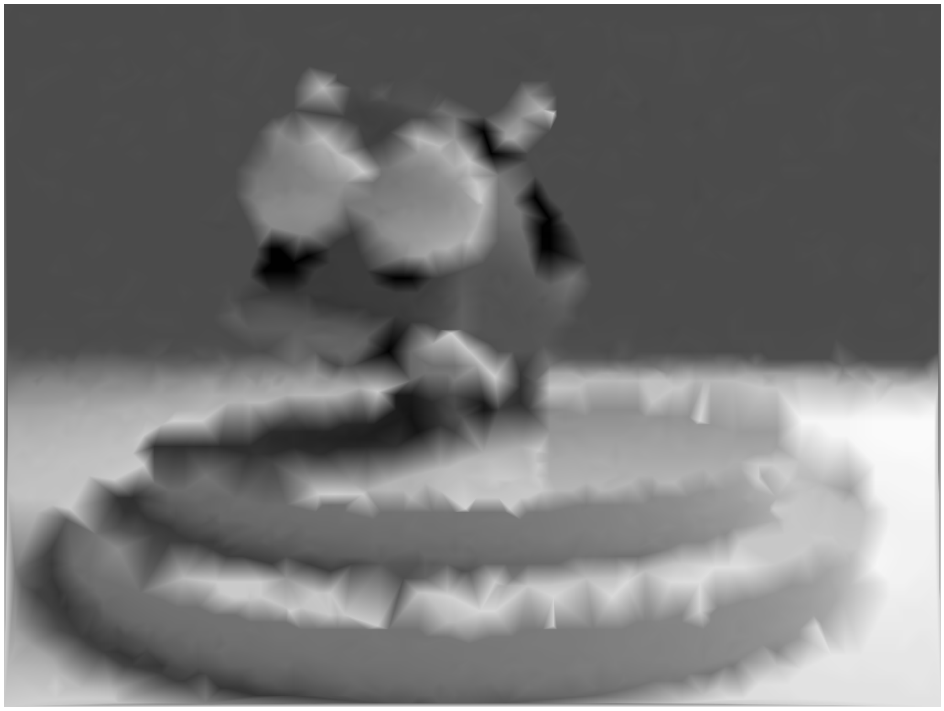


Figure 4.2: Comparison of the progressive performance with the MSDC method for individual meshes (a) B4, (b) L1, (c) L4, and (d) P4.

In our experiments, PSNR was found to correlate reasonably well with image quality as perceived by the human visual system. To demonstrate this, we now provide some examples of reconstructed images obtained with the methods under consideration. First, we study the quality of the reconstructed images at lower bit rates. We choose two representative test cases, B4 and L4, which are generated from the images `bull` and `lena`, respectively. Using each of the proposed and MSDC methods, each of the two meshes is losslessly coded once, and then decoded at a lossy bit rate. Then the decoded datasets are interpolated to produce the lattice-sampled images. For mesh B4 and L4, we choose the lossy bit rates as 17000 bytes and 10000 bytes, respectively. The reconstructed lattice-sampled images are shown in Figures 4.3 and 4.4. Examining the two images in Figure 4.3, it is clear that the proposed method can recover a reasonably good-quality image reconstruction (i.e., Figure 4.3(b)), while the MSDC method cannot (i.e., Figure 4.3(a)). We compute the PSNR of the two images relative to the original `bull` image, and the two values are 23.07 dB and 29.34 dB, for Figures 4.3(a) and (b), respectively. As we can see, the PSNR correlates reasonably well with the subjective quality. The similar phenomenon can also be observed in Figure 4.4. The PSNR values of these two images, Figures 4.4(a) and (b), are also computed relative to the original image `lena`. We obtain the two values as 21.97 dB and 26.94 dB. The image reconstruction generated by the proposed method in Figure 4.4(b) is reasonably good, while the one generated by the MSDC method in Figure 4.4(a) is not acceptable in most practical applications.

Next, we consider the quality of reconstructed images at higher bit rates. The same two meshes, B4 and L4, are decoded at a higher lossy bit rate after being losslessly encoded. For mesh B4 and L4, we choose the lossy bit rates as 45000 bytes and 17000 bytes, respectively. The reconstructed lattice-sampled images are shown in Figures 4.5 and 4.6. The PSNR values of the reconstructed images relative to the corresponding original images are computed as 47.99 dB, 41.08 dB, 34.68 dB, and 32.06 dB, for Figures 4.5(a), 4.5(b), 4.6(a), and 4.6(b), respectively. From the PSNR values, the image quality in Figure 4.5(a) should be better than Figure 4.5(b), and Figure 4.6(a) should be better that Figure 4.6(b). Examining the images in these two figures, however, we can barely see the difference in quality. The reason is, at higher bit rates, the image reconstructions already close to visually lossless. Therefore, the differences in quality between different images are difficult to recognize.

From the above, we conclude that the proposed method outperforms the MSDC method in terms of progressive performance. At lower bit rates, the proposed method can generate acceptable-quality image approximations with higher PSNR values, while the results from the MSDC method are not practically useful. At higher bit rates, although the MSDC method may obtain image approximations with higher PSNR values, the qualities of both



(a)

(b)

Figure 4.3: Reconstructed images obtained when 17000 bytes are decoded for mesh B4 using (a) the MSDC method (23.07 dB) and (b) the proposed method (29.34 dB).

(a)

(b)

Figure 4.4: Reconstructed images obtained when 10000 bytes are decoded for mesh L4 using (a) the MSDC method (21.97 dB) and (b) the proposed method (26.94 dB).

(a)

(b)

Figure 4.5: Reconstructed images obtained when 45000 bytes are decoded for mesh B4 using (a) the MSDC method (47.99 dB) and (b) the proposed method (41.08 dB).

(a)

(b)

Figure 4.6: Reconstructed images obtained when 17000 bytes are decoded for mesh L4 using (a) the MSDC method (34.68 dB) and (b) the proposed method (32.06 dB).

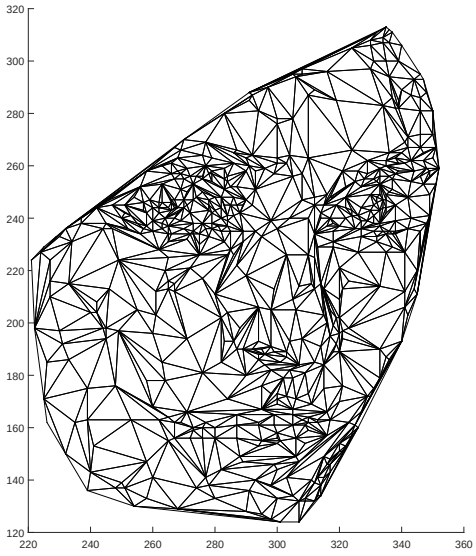


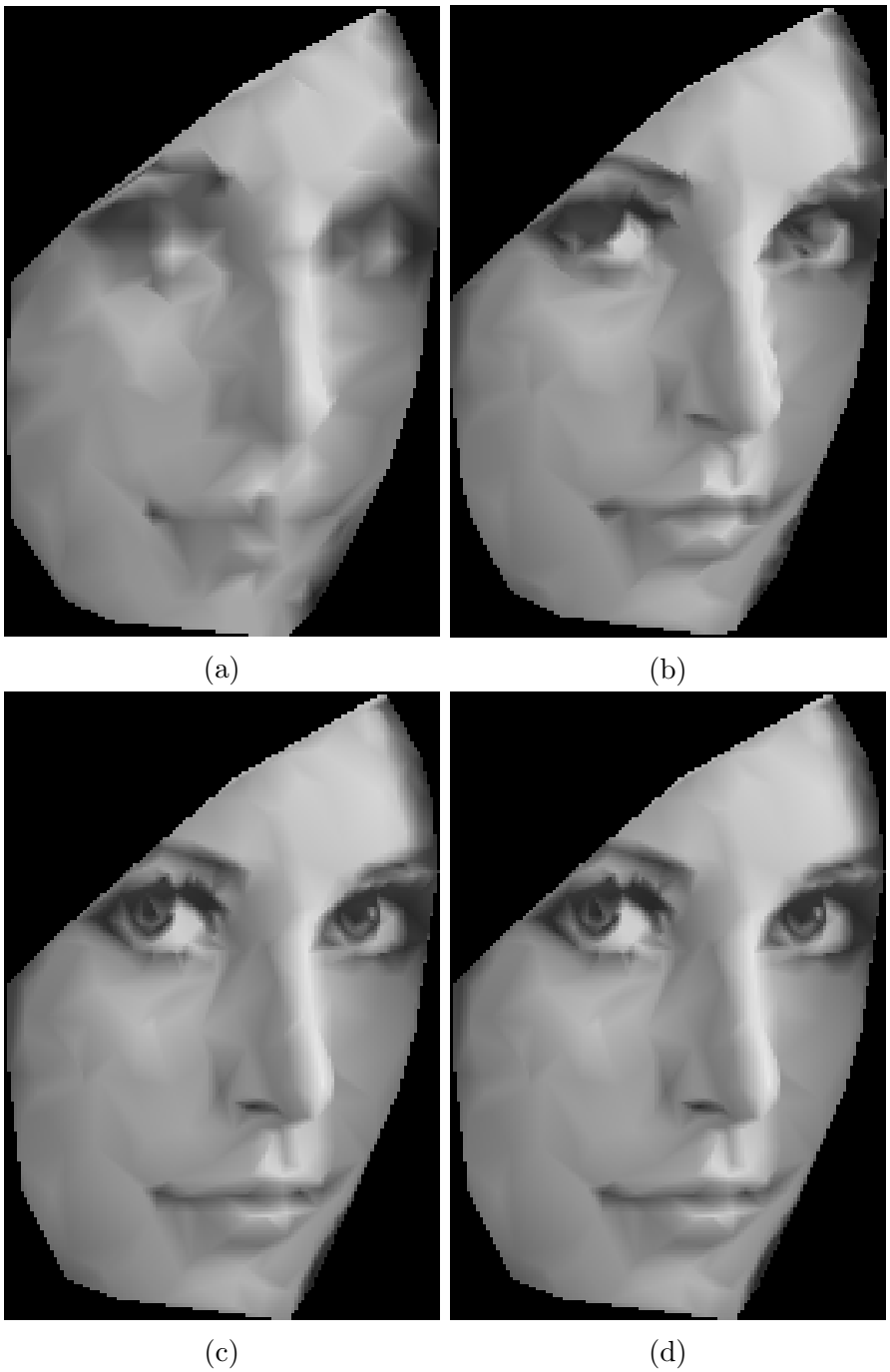
Figure 4.7: Triangulation of a mesh model of an image with an arbitrary convex domain.

results are sufficiently high that the difference in quality cannot be seen by the human visual system. Moreover, the proposed method can handle meshes with arbitrary connectivity, while the MSDC method cannot.

4.4 An Extended Application in Image Processing

In passing, we note that one advantage of using mesh models for images is that such models can directly represent images defined on arbitrary convex domains (as opposed to domains that are required to be isorectangles). Because of this, the proposed method can be used to code such images. For illustrative purposes, we now provide an example to illustrate this use of the proposed coder.

We consider a 2.5-D mesh model of part of the famous `lena` image with an arbitrary convex domain. The triangulation of the image domain is shown in Figure 4.7. The mesh is losslessly coded once, and then decoded at many intermediate rates. Four image approximations are generated from the decoded meshes at different intermediate rates and illustrated in Figure 4.8. From the subjective viewpoint, the qualities of the reconstructed images in Figures 4.8(a), (b), and (c) are incrementally better. The image shown in Figure 4.8(d) is the lossless reconstruction. This example demonstrates that the proposed method is effective for coding images defined on arbitrary convex domains.



(a) (b)

(c) (d)

Figure 4.8: Reconstructed image approximations obtained when (a) 500 bytes, (b) 1000 bytes, and (c) 1500 bytes are decoded, and (d) the lossless decoded image approximation when all 1598 bytes are decoded.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

In this thesis, a new progressive lossy-to-lossless coding framework was proposed for 2.5-D triangle meshes with arbitrary connectivity. The proposed framework has several free parameters and many variations were tried for each parameter in order to find an effective choice for good coding performance. After extensive experimentation, we recommended a particular set of choices for these parameters, leading to the specific mesh-coding method proposed herein.

Through experimental results, the proposed method was shown to be significantly better than the general-purpose compression method Gzip with the lossless bit rate of the proposed method being 5 to 6 times lower than that of Gzip. The proposed method also outperforms the Edgebreaker method, needing 8.1% less bits on average for lossless coding if the mesh connectivity does not deviate too far from the PDDT (i.e., the edge-flipping distance is less than 37.38%). Moreover, unlike Gzip and Edgebreaker, the proposed method provides progressive coding functionality. For progressive performance, the proposed method was compared with the MSDC scheme for meshes with Delaunay connectivity. From the experimental results, the proposed method was shown to outperform the MSDC method at lower bit rates by generating image approximations of much better quality, both in terms of the PSNR values and the subjective image quality. In particular, to achieve similar image quality (i.e., with a PSNR value being 75% of the maximum PSNR obtained for the lossless reconstruction), the proposed method needs 55% to 86% of the bit rate of that used by the MSDC method. At higher bit rates, although the images reconstructed by the MSDC method have higher PSNR, the difference in image quality obtained with the two methods is indistinguishable to the human visual system.

Besides achieving good performance, our work also makes contributions by solving the

two problems, combinatorial-blowup and extra-face, in the PK method, as described in Section 3.4.3 (on page 42) and Section 3.8 (on page 79), respectively. The first problem is addressed in the proposed framework by a divide-and-conquer approach. The proposed framework also avoids the second problem by generating linear-interpolated faces on the 2.5-D meshes implicitly using the constrained Delaunay triangulation.

5.2 Future Research

Although the coding method presented in this thesis has achieved fairly good performance, some aspects of this work are still worth exploring further. In what follows, we will briefly describe these areas.

When presenting the geometry coding in Section 3.4.3 (on page 37), we mentioned that after encoding the number T of nonempty subcells, the configuration specifying which T of the M subcells are nonempty is coded next. If the symbol representing the configuration is not binary, we need to employ a binarization scheme first to transform this symbol into a binary sequence and then code the resulting binary symbols in bypass mode, which is not an efficient coding method. Coding this configuration happens frequently during the entire coding procedure. So, if the efficiency of coding one configuration is increased slightly, the total coding efficiency may have a noticeable improvement. The above facts motivate us to consider a better way for coding the nonempty subcell configuration. Recall that the coding method of pivot-vertex tuple is efficient. If a similar efficient priority scheme is employed first to make the distribution of the encoded indices of the configurations more skewed, the coding efficiency may be increased with an adaptive arithmetic coder.

Another potential improvement is in the DC coding procedure. As stated before in Section 3.4.4 (on page 46), each DC is a $(\rho + 1)$-bit signed integer, and the sign bit is coded in bypass mode. If some effective sign prediction scheme could be devised, the coding efficiency could be improved.

Appendix A

Software User Manual

A.1 Introduction

As part of this work, a software implementation of the coding method proposed herein was developed. The software was written in `C++` and consists of about 11000 lines of code. The libraries utilized in this software include the Boost Library [1], the Computational Geometry Algorithm Library (CGAL) [2], the Signal Processing Library (SPL) [5], and the SPL Extensions Library (SPLEL).

Generally speaking, the software consists of two programs: 1), the `encoder` program, which performs encoding, and 2) the `decoder` program, which performs decoding. The `encoder` program reads a mesh from standard input in OFF format [4], encodes the mesh, and writes the coded bitstream to standard output. There are several options for `encoder` program that can be provided by the user to specify the parameters controlling the encoding procedure. The `decoder` program reads the coded bitstream from standard input, decodes the mesh, and writes the reconstructed mesh to standard output in OFF format.

The remainder of this appendix provides details on how to build and use the software. Several specific examples illustrating software usage are provided.

A.2 Build the Software

To build the software, the Make tool is utilized. Since the program utilizes several features from `C++11/14`, the compiler needs to be compatible with `C++11/14`. Hence, we choose GCC 6.1.0 as the compiler. As mentioned earlier, our software also utilizes the libraries such as CGAL, SPL, SPLEL, and Boost. Before building the software, users need to ensure that these libraries are installed. The versions of these libraries that were used for development

were:

- CGAL 3.8.2,
- SPL 1.1.24,
- SPLEL 1.1.26, and
- Boost-1.53.0.

In order to compile all of the source files and link the object files, one should set the directory to the top-level directory for the software. Then, to delete all the object files and executable files that were generated during the previous building process, run the command:

```
make clean
```

To compile all the source files and link the object files, run the command:

```
make
```

A.3 Detailed Program Descriptions

As mentioned earlier, the software consists of two programs, the `encoder` and `decoder` programs. In what follows, we provide detailed descriptions of these two programs and how to use them.

A.3.1 encoder

SYNOPSIS

```
encoder [OPTIONS]
```

DESCRIPTION

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

OPTIONS

The following options are supported:

`-h`		Prints the information of encoder usage.
`-p`	`$padding_mode`	Sets the padding mode to `$padding_mode`. There are two allowable values for the `$padding_mode` parameter: `none` and `power2`. The value `none` means unpadded mode; the value `power2` means padded mode. The default value is `power2`.
`-l`	`$thresholdRDC`	Sets the threshold value for the RDC queue to `$thresholdRDC`. The default value is 128.
`-Z`	`$initialDC`	Sets the number of times the DC coding procedure is invoked for each DC in the CI queue as `$initialDC`. The default value is 3.
`-z`	`$remainDC`	Sets the number of times for DC coding procedure is invoked for each DC in the RDC queue as `$remainDC`. The default value is 1.
`-v`	`$valenceMax`	Sets the threshold value for valence during the vertex split to `$valenceMax` to avoid the combinatorial blowup. The default value is 12.
`-r`	`$resultsFile`	Specifies the result file to be `$resultsFile`. This is a file to which to print all the necessary result information.

The program exits with status 0 if the software finishes normally, 1 if the software fails, and 2 if the options provided by the user are invalid. The result file has the following information in order:

- the number of vertices in the mesh
- the number of edges in the mesh
- the number of faces in the mesh
- the total number of bits of coded data (header + geometry + connectivity + function value)
- the total number of bits of coded geometry data and function values
- the total number of bits of coded connectivity data
- the time in seconds needed for encoding
- the maximum amount of memory used by the encoding program

For example, one basic way of using the `encoder` program with all default settings is as follows:

```
encoder < mesh.off > mesh.coded
```

The file `mesh.coded` will contain all the coded information.

A.3.2 decoder

SYNOPSIS

```
decoder [OPTIONS]
```

DESCRIPTION

This program reads a coded mesh from standard input and writes the decoded mesh to standard output in OFF format. The decoding process can be truncated to allow progressive decoding.

OPTIONS

The following options are supported:

`-h`	Prints out the information of decoder usage.
`-m $totalByte`	Sets the approximate number of bytes that will be decoded to `$totalByte`. Due to the manner of implementation, the actual number of bytes decoded may be slightly larger than `$totalByte`. By default the value is -1, which means the decoder will decode all bytes from the input file.
`-r $resultsFile`	Specifies the result file to be `$resultsFile`. This is a file to which to print all the necessary result information.
`-a`	Enable the automatic progressive decoding procedure.
`-i $interval`	Indicates meshes will be generated automatically each time when `$interval` extra number of bytes are decoded progressively until all information has been decoded. The first reconstructed mesh will be output when 100 bytes have been decoded. Other meshes will be output every time when `$interval` extra bytes being decoded. The default value for `$interval` is 100.

The program exits with status 0 if the software finishes normally, 1 if the software fails, and 2 if the options provided by the user are invalid. The result file has the following

information in order:

- the number of vertices in decoded mesh
- the number of edges in decoded mesh
- the number of faces in decoded mesh
- the time in seconds needed for decoding
- the maximum amount of memory used by the decoding program

The `-a` and `-i` options are provided for automatic progressive decoding purpose. The basic usage of the decoding program `decoder` for lossless decoding is as follows:

```
decoder < mesh.coded > recovered.off
```

The mesh is losslessly reconstructed and stored in the file `recovered.off`.

A.4 Examples of Software Usage

A few examples are provided in what follows to illustrate how to use the software with different options.

Example 1A. Suppose that we would like to encode the mesh in the file `mesh.off` to the file `mesh.coded` with the following requirements:

- Use the original unpadded root cell to start, even if the image domain does not have square power-of-two dimensions.
- For the nodes with detail coefficients processed from the CI queue, invoke two DC coding procedures for each DC generated during the CP coding procedure.
- When the nodes are processed from the RDC queue, invoke one DC coding procedure for each node to handle its DC.
- The threshold value for the valence is set to 10.

The above can be accomplished with the following command:

```
encoder -p none -Z 2 -z 1 -v 10 < mesh.off > mesh.coded
```

Example 1B. Suppose that we would like to encode the mesh in the file `mesh.off` to the file `mesh.coded` with the following requirements:

- Use the root cell as a padded rectangle with square power-of-two dimensions.

- Invoke four DC coding procedures for each node with a DC generated during the CP coding procedure in the CI queue.
- Invoke two DC coding procedures for each node handled from the RDC queue.
- The threshold value for the valence is set to 11.

The above can be accomplished with the following command:

```
encoder -Z 4 -z 2 -v 11 < mesh.off > mesh.coded
```

Example 2A. Suppose that we have a file `mesh.coded` containing encoded information, and we would like to decode the mesh in a lossless manner to the file `recover.off`, with the result file `decoded_info.txt`. The above can be accomplished with the following command:

```
decoder -r decoded_info.txt < mesh.coded > recover.off
```

Example 2B. Suppose that we have a file `mesh.coded` containing encoded information, and we would like to decode approximately 2000 bytes from `mesh.coded` and write the intermediate mesh to the file `recMesh_2000.off`. The above can be accomplished with the following command:

```
decoder -m 2000 < mesh.coded > recMesh_2000.off
```

Example 3. Suppose that we have a file `mesh.coded` containing encoded information and the size of this file is 10000 bytes. We would like to generate intermediate meshes whenever approximately 1000 extra bytes have been decoded during the entire decoding procedure, and write the lossless decoded mesh to the file `recovered.off`. The above can be accomplished with the following command:

```
decoder -a -i 1000 < mesh.coded > recovered.off
```

After the program terminates normally, ten intermediate meshes are generated during this procedure, namely, `recMesh_100.off`, `recMesh_1100.off`, ..., `recMesh_9100.off`, and the lossless reconstructed mesh is stored in `recovered.off`.

Bibliography

[1] Boost C++ Library. `http://www.boost.org`. Accessed: 2016-09-13.

[2] Computational Geometry Algorithms Library. `http://www.cgal.org`. Accessed: 2016-09-13.

[3] Kodak lossless true color image suite. `http://r0k.us/graphics/kodak/`. Accessed: 2016-08-23.

[4] Off, object file format. `http://segeval.cs.princeton.edu/public/off_format.html`. Accessed: 2016-09-13.

[5] SPL, Signal Processing Library. `http://www.ece.uvic.ca/~frodo/SPL/`. Accessed: 2016-09-13.

[6] USC-SIPI image database. `http://sipi.usc.edu/database/`. Accessed: 2016-08-23.

[7] JPEG-2000 test images. ISO/IEC JTC 1/SC 29/WG 1N 545, July 1997.

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

[9] M. D. Adams. Progressive lossy-to-lossless coding of arbitrarily-sampled image data using the modified scattered data coding method. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1017–1020, 2009.

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

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

[12] D. Cohen-Or, D. Levin, and O. Remez. Progressive compression of arbitrary triangular meshes. In *IEEE visualization*, volume 99, pages 67–72, 1999.

[13] S.A. Coleman, B.W. Scotney, and M. G. Herron. Image feature detection on content-based meshes. In *Proceedings of IEEE International Conference on Image Processing*, volume 1, pages 844–847. IEEE, 2002.

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

[15] L. Demaret and A. Iske. Scattered data coding in digital image compression. *Curve and Surface Fitting: Saint-Malo*, 2003:107–117, 2002.

[16] O. Devillers and M. Teillaud. Perturbations and vertex removal in a 3D Delaunay triangulation. In *14th ACM-Siam Symposium on Algorithms*, pages 313–319, 2003.

[17] C. Dyken and M. S. Floater. Preferred directions for resolving the non-uniqueness of Delaunay triangulations. *Computational Geometry*, 34(2):96–101, 2006.

[18] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[19] K. Fleischer and D. Salesin. Accurate polygon scan conversion using half-open intervals. In *Graphics Gems III*, pages 362–365. Academic Press Professional, Incorporated, 1992.

[20] P.-M. Gandoin and O. Devillers. Progressive lossless compression of arbitrary simplicial complexes. In *ACM Transactions on Graphics*, volume 21, pages 372–379. ACM, 2002.

[21] M. A. García and B. X. Vintimilla. Acceleration of filtering and enhancement operations through geometric processing of gray-level images. In *International Conference on Image Processing*, pages 97–100, 2000.

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

[23] K.-L. Hung and C.-C. Chang. New irregular sampling coding method for transmitting images progressively. *IEE Proceedings-Vision, Image and Signal Processing*, 150(1):44–50, 2003.

[24] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 279–286. ACM Press/Addison-Wesley Publishing Company, 2000.

[25] A. Khodakovsky, P. Schröder, and W. Sweldens. Progressive geometry compression. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 271–278. ACM Press/Addison-Wesley Publishing Company, 2000.

[26] J. Li and C.-C. J. Kuo. Progressive coding of 3-D graphic models. *Proceedings of the IEEE*, 86(6):1052–1063, 1998.

[27] 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, 2013.

[28] E. P. Mücke. A robust implementation for three-dimensional Delaunay triangulations. *International Journal of Computational Geometry & Applications*, 8(02):255–276, 1998.

[29] J. O'Rourke. *Computational geometry in C*. Cambridge University Press, 1998.

[30] J. Peng, C.-S. Kim, and C.-C. J. Kuo. Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 16(6):688–733, 2005.

[31] J. Peng and C.-C. J. Kuo. Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition. In *ACM Transactions on Graphics*, volume 24, pages 609–616. ACM, 2005.

[32] J. Popović and H. Hoppe. Progressive simplicial complexes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 217–224. ACM Press/Addison-Wesley Publishing Company, 1997.

[33] G. Ramponi and S. Carrato. An adaptive irregular sampling algorithm and its application to image coding. *Image and Vision Computing*, 19(7):451–460, 2001.

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

[35] M. Sarkis and K. Diepold. A fast solution to the approximation of 3D scattered point data from stereo images using triangular meshes. In *2007 7th IEEE-RAS International Conference on Humanoid Robots*, pages 235–241. IEEE, 2007.

[36] Y. Tang. Edgebreaker-based triangle mesh-coding method. Department of Electrical and Computer Engineering, University of Victoria. `http://www.ece.uvic.ca/~frodo/publications/yuetang_meng_project_report.pdf`, 2016.

[37] G. Taubin, A. Guéziec, W. Horn, and F. Lazarus. Progressive forest split compression. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 123–132. ACM, 1998.

[38] G. Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.