在藝術影像中作資料隱藏之研究

Data Hiding in Art Images

研 究 生：洪世結　　　　Student：Shi-Chei Hung

指導教授：蔡文祥　　　　Advisor：Wen-Hsiang Tsai

國 立 交 通 大 學

資 訊 科 學 研 究 所

碩 士 論 文

A Thesis
Submitted to Department of Computer and Information Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer and Information Science

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

藝術化影像上的資訊隱藏

研究生: 洪世結　　　　　　指導教授: 蔡文祥 博士

國立交通大學資訊科學系

摘要

藝術化影像(Art Images)是一種透過原始圖檔所產生出來的非相片般擬實(non-photorealistic)的影像。而馬賽克影像(Mosaic-Effect Images)則是藝術畫影像的一種，它由許多的幾何元件所構成，例如:磁磚、小圖和玻璃。在本論文中，我們研究了兩種馬賽克影像的產生與資訊隱藏技術。這兩種不同類型的馬賽克影像分別是磁磚畫與玻璃畫，前者是由許多大小、形狀相同的瓷磚組成，後者則是由不同形狀與大小的玻璃組成。在磁磚畫中，我們找到了三種可供資訊隱藏的屬性，分別是磁磚的旋轉角度、大小和材質。依據各個屬性的特性，我們將浮水印隱藏於磁磚的旋轉角度之中，以達到版權保護的目的。將秘密資訊隱藏於磁磚的大小之中以達到祕密傳輸的目的。最後將驗證資訊藏於磁磚的材質之中以達到影像與隱藏資訊完整性驗證的目的。由於此三種資訊隱藏的技術彼此獨立，所以使用者可以同時將這三種資料隱藏於磁磚畫中。在玻璃畫中，我們利用每片玻璃在生成時的樹狀資料結構(Tree Structure)來達到資訊隱藏的目的。我們利用的是樹的節點(Node)數目來作資訊隱藏，透過控制數的節點的數量，我們可以隱藏數個位元於其中。同樣的，我們也將此資訊隱藏的技術應用於玻璃畫的版權保護、秘密傳輸和影像完整性驗證。而對於各種不同的應用，我們會對資訊隱藏的流程作少許的修改。在本論文中，我們會對於磁磚畫和玻璃畫影像的生成和資訊隱藏提出完整的系統與流程，並透過實驗結果來證明此系統的實用性。

Data Hiding in Art Images

Student: Shi-Chei Hung Advisor: Dr. Wen-Hsiang Tsai

Department of Computer and Information Science

National Chiao Tung University

ABSTRACT

An art image is a non-photorealistic image created from a given image. A mosaic-effect image is a type of art image composed of geometric elements such as small images, tiles, and glass pieces. Two types of mosaic-effect images are investigated in this study, namely, tile mosaic images and stained glass images. Methods for automatic generation of these types of images and data hiding in them are proposed. In tile mosaic images, the tiles are squares of identical sizes, and are arranged regularly. Three different tile features, namely, orientation, size, and texture, are utilized for data hiding in the proposed methods. The orientations of tiles are used for copyright protection. The sizes of tiles are used for covert communication. And the textures of tiles are used for image authentication. A complete system to create tile mosaic images is also proposed, which may be employed to hide three kinds of data sequentially. In stained glass images, glass regions are generated by randomly sprinkled seeds and a region growing technique. Only one feature is used for data hiding in stained glass images, that is, the number of nodes of a tree constructed for pixel grouping in a glass region. A method for creating stained glass images and a system of three complete processes for watermarking, secret communication, and image authentication using the above-mentioned feature are also proposed. Experimental results show the feasibility of the proposed methods and systems.

ACKNOWLEDGEMENTS

The author is in hearty appreciation of the continuous guidance, discussions, support, and encouragement received from his advisor, Dr. Wen-Hsiang Tsai, not only in the development of this thesis, but also in every aspect of his personal growth.

Thanks are due to Mr. Chih-Jen Wu, Mr. Tsung-Yuan Liu, Mr. Shi-Yi Wu, Mr. Ming-Che Chen, Mr. Lien-Yi Weng and Mr. Yuei-Cheng Chuang for their valuable discussions, suggestions, and encouragement. Appreciation is also given to the colleagues of the Computer Vision Laboratory in the Department of Computer and Information Science at National Chiao Tung University for their suggestions and help during his thesis study.

Finally, the author also extends his profound thanks to his family for their lasting love, care, and encouragement. He dedicates this dissertation to his parents.

CONTENTS

ABSTRACT(in Chinese) i

ABSTRACT(in English)........ ii

ACKNOWLEDGEMENTS iii

CONTENTS........ iv

LIST OF FIGURES vii

Chapter 1 Introduction 1

1.1 Motivation........1

1.2 Review of Related Works 2

1.2.1 Previous Studies on Art Images 2

1.2.2 Previous Studies on Tile Mosaics 4

1.2.3 Previous Studies on Stained Glasses........5

1.2.4 Previous Studies on Data Hiding 6

1.2.5 Previous Studies on Image Mosaics 7

1.3 Overview of Proposed Methods........7

1.3.1 Definitions of Terms 7

1.3.2 Brief Descriptions of Proposed Methods for Tile Mosaic Images......9

1.3.3 Brief Descriptions of Proposed Methods for Stained Glass Images. 11

1.3.4 Contributions........12

1.4 Thesis Organization 13

Chapter 2 A New Tile Mosaic Image Creation Method for Information Hiding 14

2.1 Introduction........14

2.2 Review of Traditional Tile Mosaic Image Creation Process 15

2.3 Proposed Tile Mosaic Image Creation Process........19

2.3.1 Scheme of Creation Process........19

2.3.2 Use of Linear Regression Technique for Line Fitting........21

2.3.3 Creation of Visual Effects in Tiles for Information Hiding 22

2.3.4 Experimental Results and Discussions 23

Chapter 3 Three Methods for Data Hiding in Tile Mosaic Images .25

3.1 Introduction........25

3.1.1 Properties of Tile Orientations........25

3.1.2 Properties of Tile Sizes 26
3.1.3 Properties of Tile Textures 26
3.1.4 Concepts of Proposed Data Embedding and Extraction Techniques 27
3.2 Proposed Tile Feature Detection Techniques 29
3.2.1 Tile Scanning 30
3.2.2 Tile Region Detection 32
3.2.3 Tile Boundary Detection 36
3.2.4 Tile Feature Detection 37
3.3 Proposed Watermarking Method by Tile Orientation Modification 40
3.3.1 Core Concept 41
3.3.2 Data Embedding Process 42
3.3.3 Data Extraction Process 45
3.3.4 Experimental Results 46
3.4 Proposed Secret Hiding Method by Tile Size Modification 48
3.4.1 Core Concepts 48
3.4.2 Data Embedding Process 49
3.4.3 Data Extraction Process 51
3.4.4 Experimental Results 52
3.5 Proposed Authentication Method by Tile Texture Modification 54
3.5.1 Core Concepts 54
3.5.2 Authentication Signal Embedding Process 58
3.5.3 Authentication Signal Extraction Process 59
3.5.4 Applications for Image Verification 62
3.5.5 Applications for Secret Verification 62
3.5.6 Experimental Results 62

Chapter 4 A Stained Glass Image Creation Method for Information Hiding 68
4.1 Introduction 68
4.2 Review of Traditional Stained Glass Image Creation Process 69
4.3 Proposed Stained Glass Image Creation Process 70
4.3.1 Scheme of Creation Process 70
4.3.2 Preprocessing of Input Images 71
4.3.3 Tree Structure of Glass Regions and Region Growing Process 72
4.3.4 Glass Region Gap Filling Process 81
4.4 Experimental Results and Discussions 83

Chapter 5 Data Hiding in Stained Glass Images and Applications 85
5.1 Introduction 85

5.1.1 Concepts behind Proposed Technique 85
5.1.2 Concepts behind Proposed Data Embedding and Extraction Techniques 86
5.2 Glass Feature Detection Process 88
5.3 Data Hiding by Glass Boundary Cracking 92
5.4 Application to Secret Hiding 95
5.4.1 Data Embedding Process 95
5.4.2 Data Extraction Process 96
5.4.3 Experimental Results and Summary 96
5.5 Application to Watermarking for Copyright Protection 97
5.5.1 Data Embedding Process 97
5.5.2 Data Extraction Process 99
5.5.3 Experimental Results and Summary 100
5.6 Application to Authentication of Images 100
5.6.1 Authentication Signal Embedding Process 100
5.6.2 Authentication Signal Extraction Process 103
5.6.3 Experimental Results and Summary 103

Chapter 6 Conclusions and Suggestions for Future Works 107
6.1 Conclusions 107
6.2 Suggestions for Future Works 109

References 111

LIST OF FIGURES

Fig. 1.1 An oil painting created by Hertzmann [2, 3]. (a) A source picture. (b) The final oil painting. 2
Fig. 1.2 Samples of art images listed in Hertzmann [1]. (a) An image created by a trial-and-error painterly rendering algorithm described in Hertzmann [4]. (b) An image created by interactive painterly rendering process from Haeberli [5]. (c) An image created by a stippling algorithm from Secord [6] (d) Pen-and-ink illustration of a smooth surface, from Hertzmann and Zorin [7]. (e) A tile mosaic image from Hausner [8]. (f) An image computed by a trial-and-error algorithm from Haeberli [5]. 3
Fig. 1.3 Tile mosaic art work and images. (a) Detail from "Sea Creatures", National Museum of Naples, 1st century B. C. (b) An image mosaic created by Adobe Photoshop. (c) An image mosaic created by Haeberli's method. 5
Fig. 1.4 Stained glass window and images. (a) Detail of The Crucifixion, St. James Church, Staveley, UK by Neil Ralley [23]. (b)(c) Some results of Mould's method. ..6
Fig. 1.5 Image mosaics created by Lin and Tsai [21, 22]. (a) An image mosaic of Lena. (b) An image mosaic of Albert Einstein. (c) An image mosaic of the campus of National Chaio Tung University, Taiwan. 9
Fig. 1.6 Proposed framework of data hiding in a tile mosaic image. 10
Fig. 1.7 Proposed framework of data extraction from a tile mosaic images. 11
Fig. 1.8 Proposed framework of data hiding in a stained glass image. 12
Fig. 1.9 Proposed framework of data extraction from a stained glass image. 12
Fig. 2.1 Voronoi diagrams from [8]. (a) A voronoi diagram. (b) A centroidal voronoi diagram. 15
Fig. 2.2 Some intermediate images and a tile mosaic image created by Hausner [8]. (a) The original Taiji image. (b) The direction field of (a). (c) Sprinkled seeds and the corresponding voronoi diagram. (d) A centroidal voronoi diagram of (a) after 20 iterations. (e) A centroidal voronoi diagram of (a) after 20 iterations with edge avoidance. (f) The final tile mosaic image. 18
Fig. 2.3 Illustration of minimum distance between neighboring tiles. 20
Fig. 2.4 3x3 Sobel mask. 20
Fig. 2.5 Flowchart of proposed line fitting procedure for the aligning vector. 21
Fig. 2.6. Tiles with visual effects. (a) Tiles without texture effect. (b) Tiles with texture effect. (c) Tiles without border and texture effect. 22
Fig. 2.7 Tile mosaic image creation. (a) A tile mosaic image created by Hausner's creation process [8]. (b) A tile mosaic image created by proposed creation process..23
Fig. 2.8 Some created tile mosaic images. (a) A tile mosaic image of the word "Taiwan". (b) A tile mosaic image with a painting of Monet as input. (c) A tile mosaic image with

a painting of Van Gogh as input......24
Fig. 3.1 Properties of tile orientations......25
Fig. 3.2 Effective range of θ. 26
Fig. 3.3 Flowchart of data hiding in a tile mosaic image......28
Fig. 3.4. Flowchart of data extraction from a tile mosaic image. 28
Fig. 3.5 Proposed tile feature detection process. 31
Fig. 3.6 Finding grid lines with local maximum accumulations. 32
Fig. 3.7 Tile region detection. (a) A tile region map with erroneous points at the left. (b) a detected tile boundary with an erroneous apex at the left. (c) a detected tile boundary with accurate apexes. 33
Fig. 3.8 Removing erroneous corners of adjacent tiles (a) A tile region map. (b) Scan from the left grid edge for minimum accumulation value. (c) Details of a tile region map after error corner elimination. 34
Fig. 3.9 Experimental results of tile boundary detection. (a) A result without utilizing step 2 of algorithm 3.1.2. (b) A result with utilizing step 2 of algorithm 3.1.2. 35
Fig. 3.10 Illustrations of boundary detection. (a) Rotated tile with the four detected lists; (b) rotated tile with the detected four apexes and tile boundary; (c) non-rotated tile with the four lists; (d) non-rotated tile with the detected four apexes and tile boundary....38
Fig. 3.11 Tile orientation detection 39
Fig. 3.12 Data hiding by tile orientation modification. (a) Two adjacent tiles. (b) Tiles with bit 0 embedded in $Tile_{i+1}$. (c) Data hiding strategy. 43
Fig. 3.13 Watermark embedding process......45
Fig. 3.14 Watermark extraction process......46
Fig. 3.15 Experimental results. (a) A Taiji image; (b) a tile mosaic image of Taiji without watermark embedded; (c) a tile mosaic image with watermark (e) embedded; (d) a damaged image of (c), (e) an input watermark; (f) a watermark extracted from (c); (g) a watermark extracted from (c) with a wrong key; (h) a watermark extracted from (d). 47
Fig. 3.16 Data hiding by tile size modification. (a) Concept of data embedding. (b) Data of data extraction......49
Fig. 3.17 Secret message embedding process......51
Fig. 3.18 Secret message extraction process......52
Fig. 3.19 Experimental results. (a) A tile mosaic image without secret message embedded. (b) a tile mosaic image with secret message embedded, (c) Some details of (b), (d) The secret message extracted from (b), (e) The secret message extracted from (b) with a wrong key. 53
Fig. 3.20 Data hiding by tile texture modification......55
Fig. 3.21 Authentication signal embedding process 60

Fig. 3.22 Tile mosaic image verification process 61
Fig. 3.23 Experimental results. (a) A tile mosaic image without data embedded. (b) A tile mosaic image with a watermark, a secret message and authentication signals embedded. (c) Some details of (a). (d) Some details of (b). (e) A watermark extracted from (b). (f) A secret message extracted from (b). (g) Verification result of (b). (h) A copy of (b) with some damaged tiles. (i) A verification result of (h). (j) A watermark extracted from (i). (k) A secret message extracted from (i). 64
Fig. 4.1 Some intermediate images and stained glass images created by Mould [13]. (a) An original image Gretzky. (b) Segmented regions of (a). (c) Region boundaries of (b). (d) Smoothed region boundaries of (b). (e) Associated stained glass image. (e) Another stained glass image with different background from (e)..70
Fig. 4.2 Proposed stained glass image creation process. 71
Fig. 4.3 Preprocessing of an original image. (a) The original image, (b) A quantized image of (a). (c) A filtered image of (b). (d) A voting filter of size 11x11 74
Fig. 4.4 Tree structure of glass regions. (a) A tree of a glass region; (b), (c), (d) and (e) An example of Steps 2 and 3 in Algorithm 4.1; (g) Figure descriptions. 77
Fig. 4.5 Proposed region growing process. (a) A version of linking the leaf nodes. (b) A version of searching the last expanding nodes. (c) Figure descriptions. 80
Fig. 4.6 Illustration of gap filling process. 82
Fig. 4.7 Experimental results. (a) An original image. (b) A stained glass image of (a). 83
Fig. 4.8 Some experimental results. 84
Fig. 5.1 An example of removing nodes for data hiding. (a) A glass region copied from Fig. 4.4(b). (b) A glass region with data embedded. 86
Fig. 5.2 Proposed data embedding process. 87
Fig. 5.3 Proposed data extraction process. 88
Fig. 5.4 Proposed glass feature detection process. 89
Fig. 5.5 Detection result of a stained glass image. 91
Fig. 5.6 Core concept of data hiding in stained glass images. 93
Fig. 5.7 Illustration of acquiring effective trees in data embedding and extraction process. 95
Fig. 5.8 Experimental results of secret hiding in stained glass image. (a) A stained glass image without hidden data. (b) A stained glass image with secret message embedded. (c) Details of (a). (d) Details of (b). (e) The secret message extracted from (b). (f) The extraction result of (b) with a wrong key. 98
Fig. 5.9 Experimental results of watermarking a stained glass image. (a) A stained glass image with watermark, (c), embedded. (b) A damaged image of (a). (c) An input watermark. (d) A watermark extracted from (a). (e) A watermark extracted from (a)

with a wrong key. (f) A watermark extracted from (b). 101
Fig. 5.10 Experimental results of authentication. (a) A stained glass image with authentication signals embedded. (b) An authentication image of (a). 104
Fig. 5.11 A damaged image of Fig. 5.10 (a). 105
Fig. 5.12 An authentication image of Fig. 5.11 106

Chapter 1 Introduction

1.1 Motivation

In recent years, people often show themselves on the networks. More and more people have their own web sites, web logs (blogs), web albums, guest books, etc. They exchange any interesting things with others all over the world through these media. Digital image is the most popular format transferred through these media. They may be used for decorations or as photos on these media. No matter what digital images are used for, people do often make digital images look more artistic before publishing them. We name this kind of digital images *art images*. However, when we publish art images on networks, people can duplicate or tamper them easily. Thus, the issues of copyright protection and authentication of art images must be taken into consideration more seriously. Many researches about art image creation and data hiding have been carried out. But rare of them can integrate the two topics of art image creation and copyright protection into one single approach. If we want to create an art image to decorate our web sties and don't expect this image to be modified and downloaded without our permission, additional watermarking algorithms need to be performed after a creation process. This extra procedure of watermarking seems too complicated to a common user. Therefore, it is desired to investigate art image creation processes which are integrated with data hiding techniques. It is hoped to hide watermarks and authentication signals as soon as an art image is created. The copyright and integrity of the art image can thus be protected easily. Furthermore, it is also desired to conduct secret communication by similar techniques.

In this study, we will deal with two different formats of art images. One is *tile* *mosaic image* and the other is *stained glass image*. Each of them will be introduced in the next section. We will propose processes for creation of them and achieve the data hiding propose by utilizing their image features.

1.2 Review of Related Works

1.2.1 Previous Studies on Art Images

Many researches investigated the problem of how to create art images in recent years. Hertzmann [1] surveys the ideas of creating art images. He defined automatic approaches to creating non-photorealistic imagery by placing discrete elements such as paint strokes and stipples, and called them *stroke-based rendering* (SBR). Here we define art images as the final products of SBR. Hertzmann [1] also surveyed many SBR algorithms and styles such as painting, pen-and-ink drawing, tile mosaics, stippling and streamline visualization. Although there are plenty algorithms of SBR, one important goal of these algorithms is to make art images look like some other types of images such as oil painting. Fig. 1.1 shows an example of oil painting from [2] and [3]. Fig. 1.1(a) is the source picture and Fig. 1.1(b) is the final oil painting. Some other samples of art images are shown in Fig. 1.2.

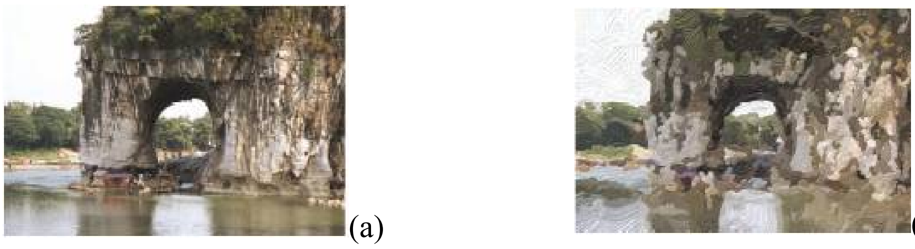
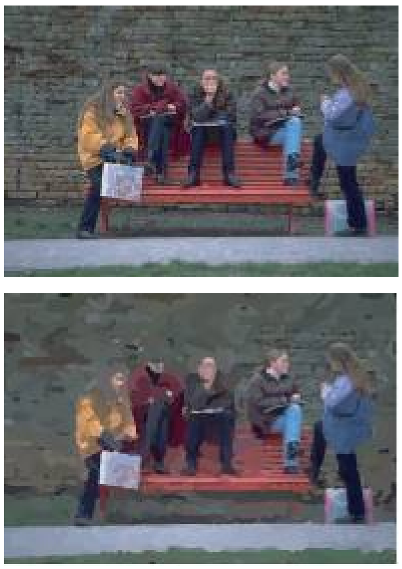
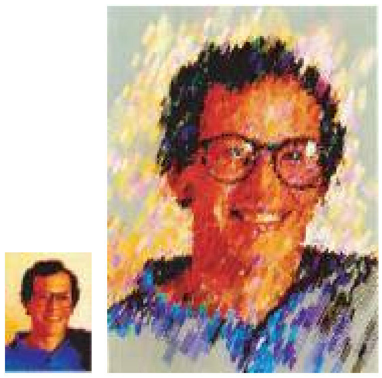


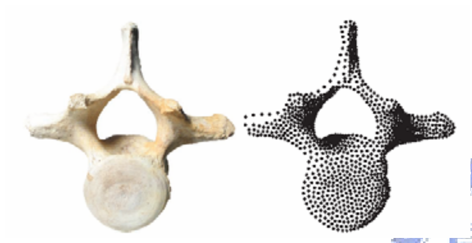
Fig. 1.1 An oil painting created by Hertzmann [2, 3]. (a) A source picture. (b) The final oil painting.



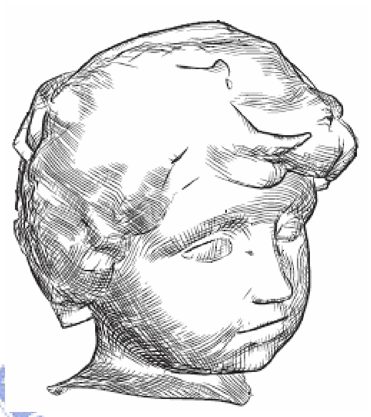
(a)



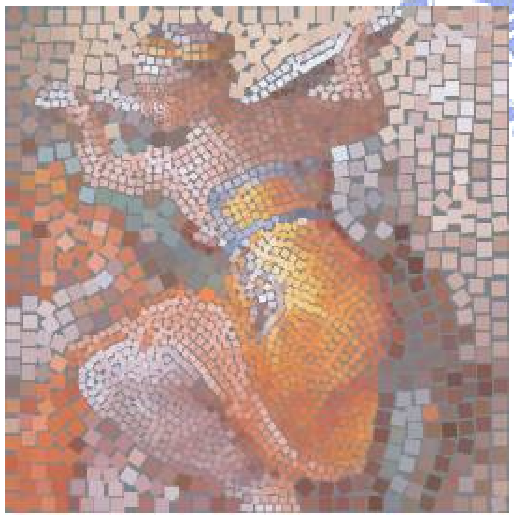
(b)



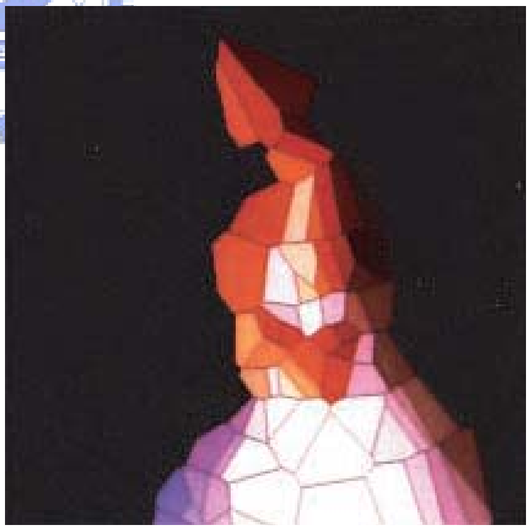
(c)



(d)



(e)



(f)

Fig. 1.2 Samples of art images listed in Hertzmann [1]. (a) An image created by a trial-and-error painterly rendering algorithm described in Hertzmann [4]. (b) An image created by interactive painterly rendering process from Haeberli [5]. (c) An image created by a stippling algorithm from Secord [6] (d) Pen-and-ink illustration of a smooth surface, from Hertzmann and Zorin [7]. (e) A tile mosaic image from Hausner [8]. (f) An image computed by a trial-and-error algorithm from Haeberli [5].

In this study, we will focus on *mosaic-effect* images. A mosaic-effect image is a type of art images composed of geometric elements such as small images, tiles, and glass pieces. These elements may or may not have identical shapes or sizes. We will aim at creating mosaic-effect images composed of *non-overlapping elements*. With non-overlapping elements, hidden data can be extracted so that we can achieve the goal of data hiding. Two types of mosaic-effect images are investigated in this study. They are *tile mosaic images* and *stained glasses images*, as shown in Fig. 1.2(e) and Fig. 1.2(f), respectively.

1.2.2 Previous Studies on Tile Mosaics

Tile mosaics are surface decorations composed of numerous small tiles, which are often of similar shapes or sizes, but in different colors. Tile mosaics appeared in Greek and Roman times over 2000 years ago, and are still widely used today. Fig. 1.3(a) shows an example of a tile mosaic artwork. Creating tile mosaic images by computing is a new research topic in recent years. Adobe PhotoShop provides a filter to create mosaic-effect images, but it does so merely by reducing the resolution of an input image, as shown in Fig. 1.3(b). Haeberli [5] used voronoi diagrams to create mosaic-effect images by placing tile sites randomly and filling the voronoi regions, as shown in Fig. 1.3(c). However, Haeberli's method does not attempt to follow the edge features in the image and the tiles all have different shapes. Hausner [8] proposed a method to create tile mosaic images by utilizing centroidal voronoi diagrams, as shown in Fig. 1.2(e). In Hausner [8], the method proposed by Hoff [9] is extended to draw a voronoi diagram efficiently, and Lloyd's algorithm [10] was utilized to produce centroidal voronoi diagrams by moving each seed to the centroid of its voronoi region. Hausner's method will be described with more details in Chapter 2.

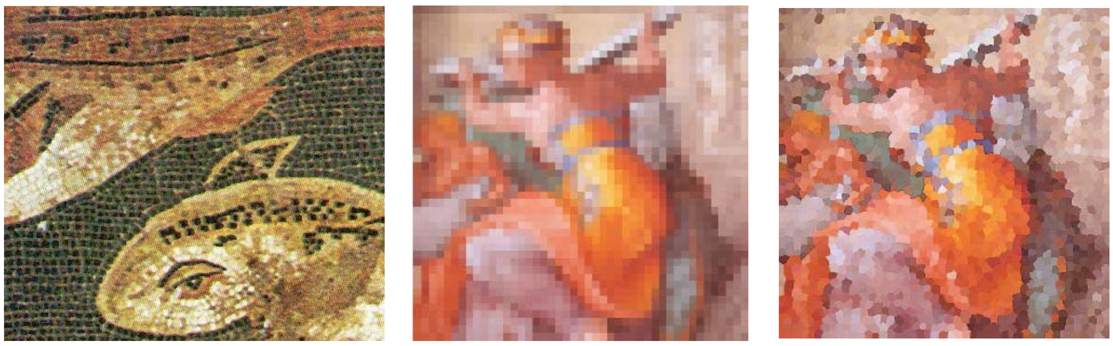


Fig. 1.3 Tile mosaic art work and images. (a) Detail from "Sea Creatures", National Museum of Naples, 1st century B. C. (b) An image mosaic created by Adobe Photoshop. (c) An image mosaic created by Haeberli's method.

1.2.3 Previous Studies on Stained Glasses

Stained glass windows are composed of glass pieces, which are of different shapes, colors, and sizes. According to the investigation results by Armitage and Osborne in [11, 12], stained glass windows first appeared in the 7th century, had heyday at the 16th century, and still being built today. Fig. 1.4(a) shows the detail of a stained glass window. Although creating mosaic-effect images is a new research topic in recent years, little attention has been paid to the historical successor to the mosaic, the stained glass image. Mould [13] proposed an algorithm for stained glass image creation. He started with an initial segmentation of a source image by the image processing system, EDISON, proposed in [14, 15, 16]. Then he used erosion and dilation operators [17, 18] to manipulate and smooth the initial segmented regions. Finally, he applied a displacement map representing imperfections in the glass and applied leading between tile boundaries. Fig. 1.4(b) and (c) are some results of Mould's method.

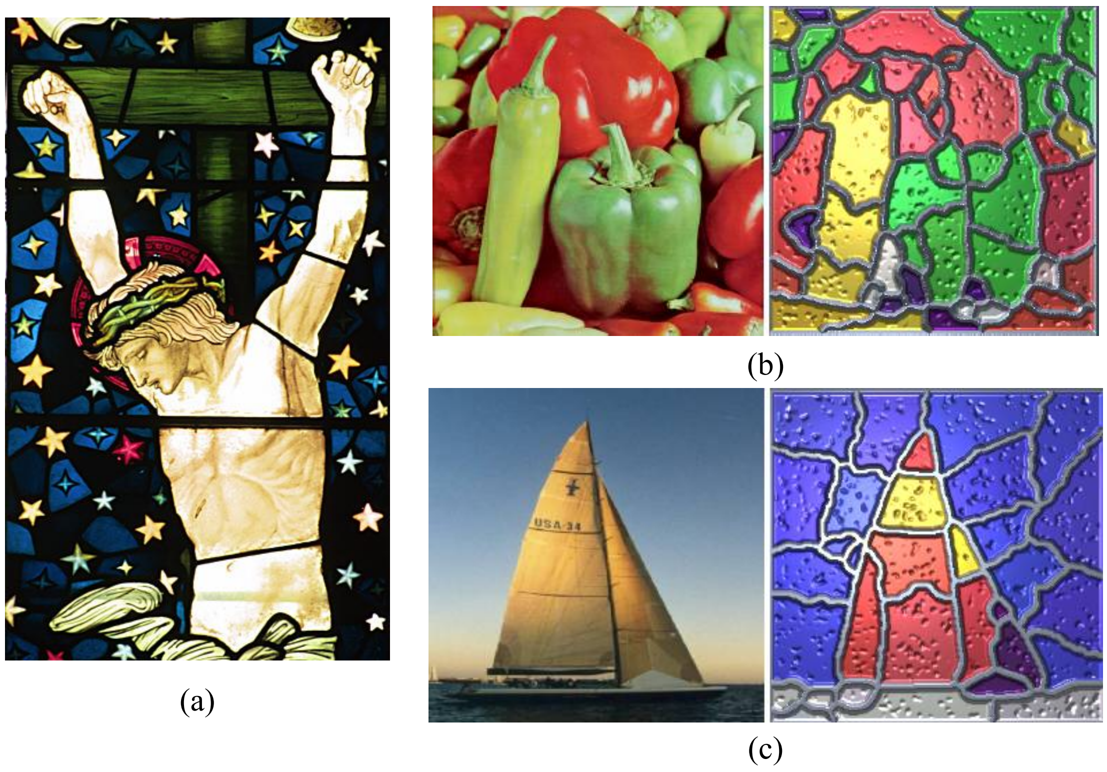


Fig. 1.4 Stained glass window and images. (a) Detail of The Crucifixion, St. James Church, Staveley, UK by Neil Ralley [23]. (b)(c) Some results of Mould's method.

1.2.4 Previous Studies on Data Hiding

The research on data hiding aims to embed information imperceptibly into a given media. Data hiding in images is mostly cultivated on the weaknesses of the human visual system, for example, by changing the least significant bits of the pixels of a cover image to embed information [19]. The information embedded in an image can be used to protect the copyright of the image, verify the authenticity of the image, and convey a secret message in the image, and so on.

Data hiding in images is a popular research topic in recent years. Researches on this topic can be classified into three approaches, namely, the spatial-domain approach, the frequency-domain approach, and the combination of the first two [20]. No matter what classifications they belong to, most of these researches are based on pixelwise or blockwise operations and few image features are used in these researches. In this study, data hiding methods correspond to individual features of art images will be proposed. We will focus on two types of mosaic-effect images, that is, tile mosaic image and stained glass image, along with their creation processes. Unlike the traditional methods of data hiding in images, we will hide data in the orientations, sizes, and textures of tile mosaic images and hide data by slight glass cracking in stained glass images. All the details will be discussed in the following Chapters.

1.2.5 Previous Studies on Image Mosaics

Image mosaics (or *photomosaics*) are another common format of mosaic-effect image. An image mosaic is composed of a large number of small images, called *tile images*. When viewing the resulting image mosaic from a distance, the grid tile images combine to yield an impressive integrated painting. Fig. 1.5 shows some examples of image mosaics. Image mosaics are good examples which prove that data can be embedded into individual features of art images. Lin and Tsai [21, 22] proposed a method to hide information in image mosaics by manipulating the four borders and the histogram of a tile image.

1.3 Overview of Proposed Methods

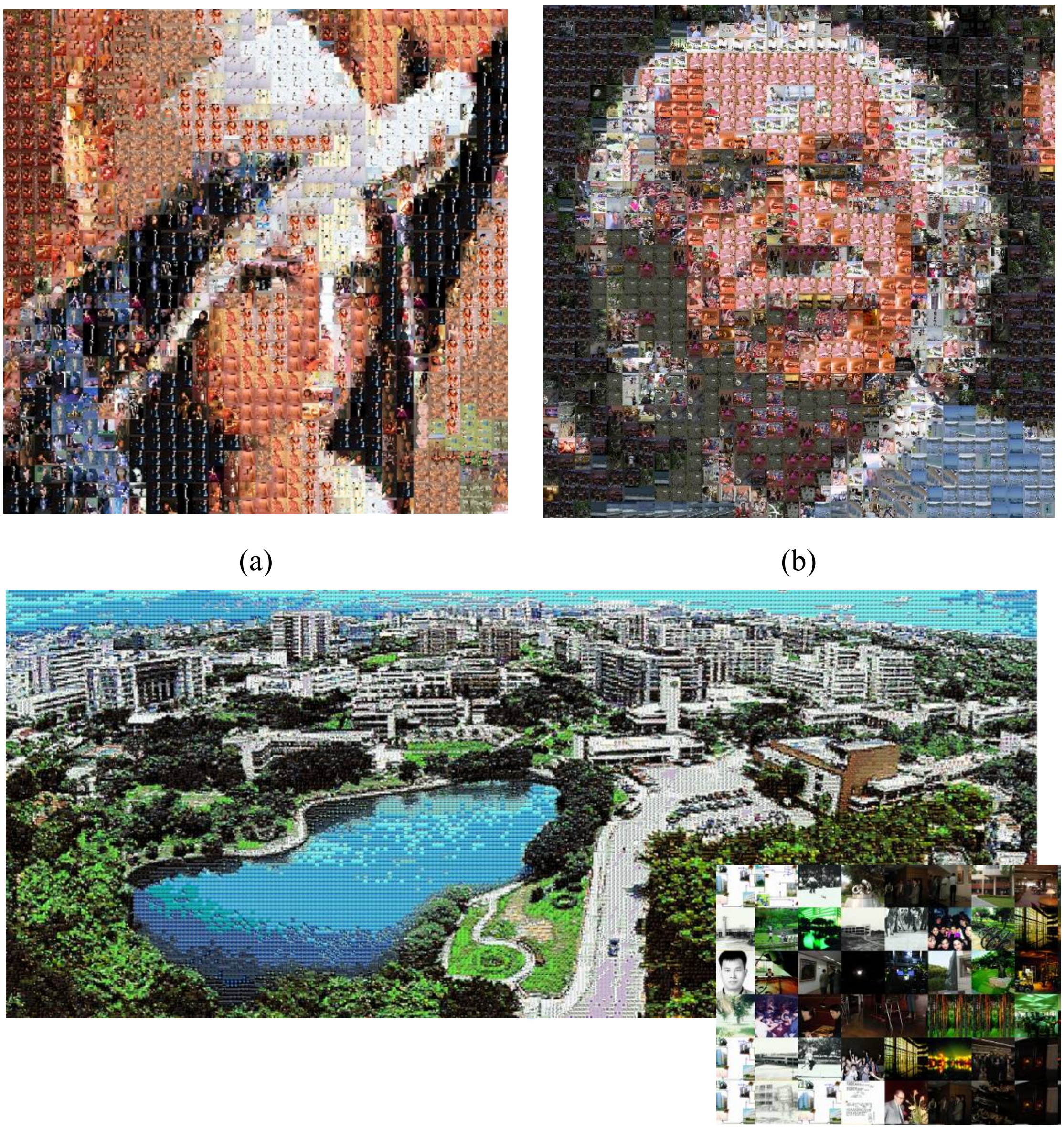
1.3.1 Definitions of Terms

Before describing the proposed methods, some definitions of terms used in this thesis are introduced as follows.

1. *Original image*: An original image is an image chosen to produce a tile mosaic image or a stained glass image.
2. *Art image*: An art image is a non-photorealistic image created from an original image. In this thesis an art image may be a tile mosaic image or a stained glass

image.

3. *Creation process*: A creation process will create an art image from an original image.
4. *Embedding process*: An embedding process is a process to embed data in an art image.
5. *Extraction process*: An extraction process is a process to extract hidden data from an art image.
6. *Tile mosaic image*: A tile mosaic image is an image composed of numerous small tiles, which are often of similar shapes or sizes, but in different colors. The orientations of the tiles align with the nearby edges of the original image.
7. *Aligning vector*. An aligning vector is derived by the edge points within a computing range of a tile in a tile mosaic image. It is used to re-orient the tile so that the tile can align the nearby edges.
8. *Stained glass image*: A stained glass image is an image composed of glass pieces, which are of different shapes, colors, and sizes. The glass pieces reveal the edges of the original image. And there exist *leading* between glass pieces.
9. *Leading*: The black and thin area between glass pieces.
10. *Authentication signal*: An authentication signal is embedded into an image. It is fragile such that any alteration to the image can be detected.
11. *Authentication image*: An authentication image is obtained by making certain indication marks in the cover image after checking the embedded authentication signals. By the indication marks, people are aware of which parts of the cover image have been tampered.



(a) (b)

(c)

Fig. 1.5 Image mosaics created by Lin and Tsai [21, 22]. (a) An image mosaic of Lena. (b) An image mosaic of Albert Einstein. (c) An image mosaic of the campus of National Chaio Tung University, Taiwan.

1.3.2 Brief Descriptions of Proposed Methods for Tile Mosaic Images

The proposed framework of data hiding in a tile mosaic image is shown in Fig. 1.6. First, we create a tile mosaic image by a creation process proposed in this study, which will be discussed in Chapter 2. Then we apply a data embedding process by utilizing a secret key for protecting the embedded data. Before the embedding process, we will transform the given data into a bit sequence. The given data may be a watermark, secret information, or authentication signals. Each of them will be embedded into different tile features. Due to the different features we choose for embedding the different types of given data, we can embed them simultaneously. Of course, not all of the three types of the given data need to be embedded at the same time in all applications. We can embed only one or two of them as needed. A detailed description will be given in Chapter 3.

The proposed framework of data extraction from a tile mosaic image is shown in Fig. 1.7. First, we apply a tile feature detection process proposed in this study, which will be discussed in Section 3.2. Then we can extract the embedded data by utilizing the proposed data extraction process and a secret key. The data embedding and extraction process will be discussed in Section 3.3 through Section 3.5.

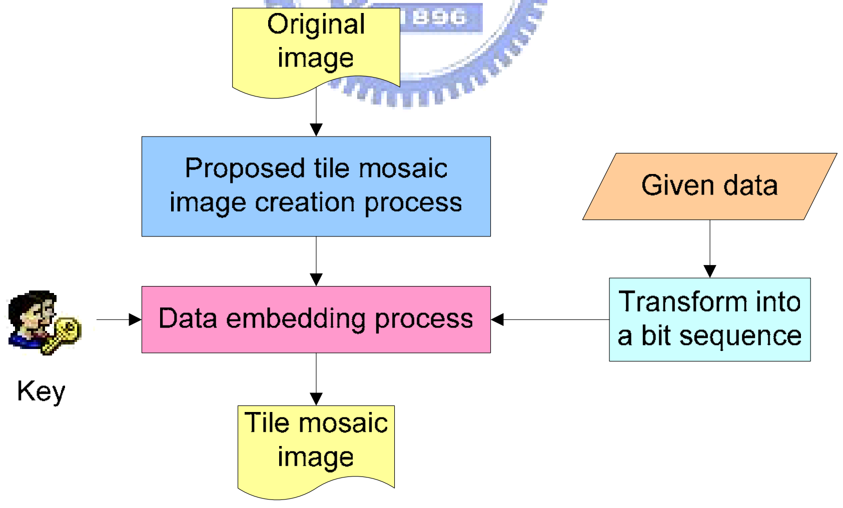

Original image
Proposed tile mosaic image creation process
Given data
Key
Data embedding process
Transform into a bit sequence
Tile mosaic image

Fig. 1.6 Proposed framework of data hiding in a tile mosaic image.

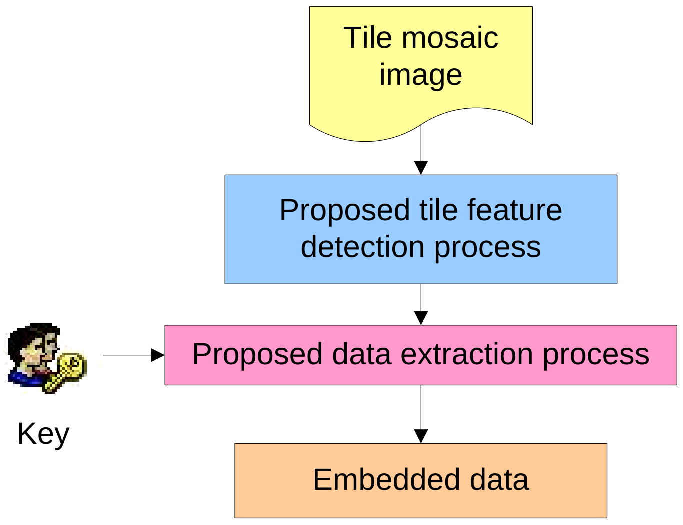

Tile mosaic image
Proposed tile feature detection process
Key
Proposed data extraction process
Embedded data

Fig. 1.7 Proposed framework of data extraction from a tile mosaic images.

1.3.3 Brief Descriptions of Proposed Methods for Stained Glass Images

The proposed framework of data hiding in a stained glass image, which is shown in Fig. 1.8, is similar to the one of a tile mosaic image. But there are two differences between them. The first is that only one feature is used for data hiding in a stained glass image. So, only one kind of data can be embedded at a time (i.e., only one of a watermark, a secret message, and a set of authentication signals can be embedded). The second difference is that we utilize a secret key for protecting embedded data while we create a stained glass image. In other words, the secret key is utilized when the data is to be embedded in a tile mosaic image.

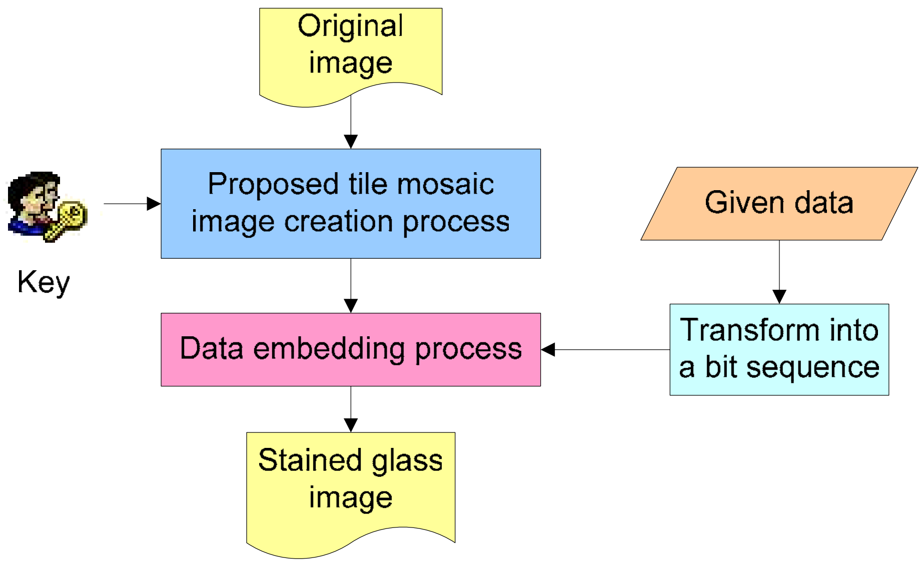

Original image
Key
Proposed tile mosaic image creation process
Given data
Transform into a bit sequence
Data embedding process
Stained glass image

Fig. 1.8 Proposed framework of data hiding in a stained glass image.

The data extraction process is simply an inverse version of the data hiding process as shown in Fig. 1.9.

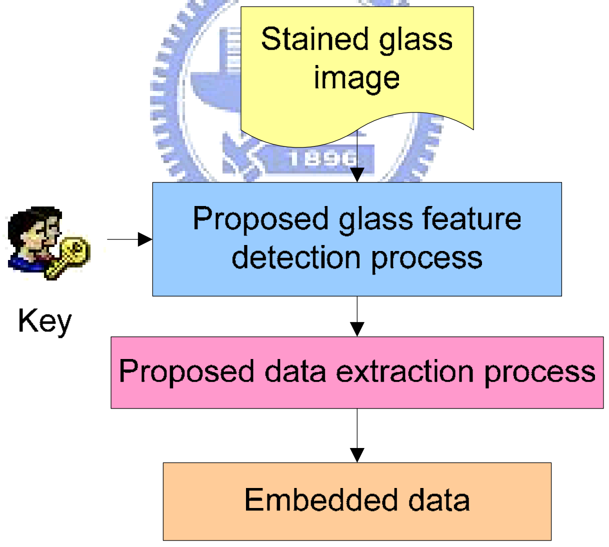

Stained glass image
Key
Proposed glass feature detection process
Proposed data extraction process
Embedded data

Fig. 1.9 Proposed framework of data extraction from a stained glass image.

1.3.4 Contributions

Some major contributions of this study are listed as follows.

1. A method to create tile mosaic images made of square and non-overlapping tiles is proposed.
2. A method to hide data in the orientation, size and texture of a tile in a tile mosaic

image is proposed.

3. A method to detect the orientation, size, and texture of a tile in a tile mosaic image is proposed.
4. Methods to embed watermarks, secret information and authentication signals respectively into the orientations, sizes, and textures of the tiles in a tile mosaic image are proposed.
5. A method to create stained glass images is proposed.
6. A method to hide data in the tree structures of glass regions is proposed.
7. A method to detect the tree structures of glass regions is proposed.
8. Methods to embed a watermark, a secret message, or a set of authentication signals one at a time into a stained glass image are proposed.

1.4 Thesis Organization

This thesis is organized as follows. In Chapter 2, the proposed system of tile mosaic image creation for data hiding is described. In Chapter 3, the proposed data hiding methods by utilizing the three features of tiles will be introduced. The three corresponding applications (i.e. watermarking, secret communication and authentication) and the proposed tile feature detection process are also discussed in Chapter 3. In Chapter 4, the proposed stained glass image creation method is described. The proposed tree structure and region growing techniques of glass regions are also be discussed. Chapter 5 describes the proposed data hiding method by glass cracking and three applications. The proposed glass feature detection process is also described. Conclusions of our works as well as discussions on future works are included in Chapter 6.

Chapter 2
A New Tile Mosaic Image Creation Method for Information Hiding

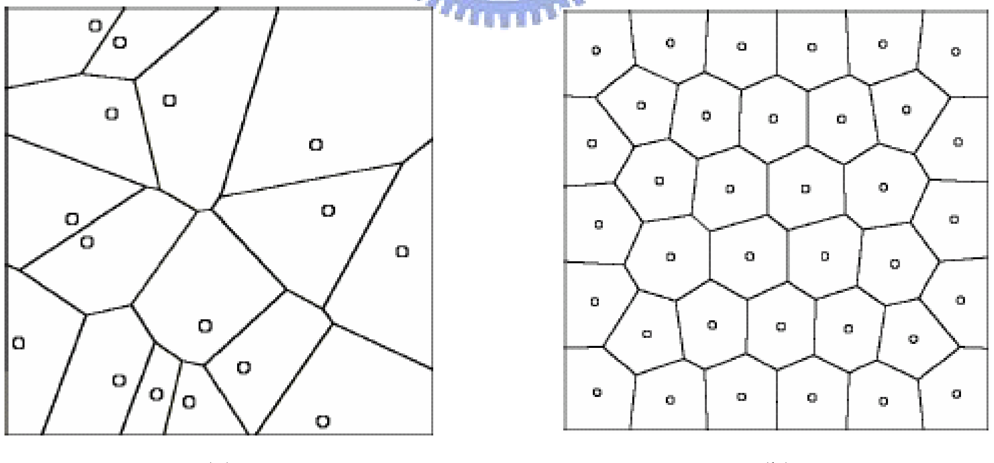
2.1 Introduction

In this chapter we will discuss how to create a tile mosaic image automatically with an image as input. A crucial issue of tile mosaic image creation is how to make the edges in the tile mosaic image clear. There are two concepts in dealing with this issue, that is, *tile rotation* and *edge avoidance*. The former is to rotate the tiles to align with the nearby edges in the input image. The latter is to prevent the tiles from straddling the edges by moving the tiles away from the edges, that is, by what we call *tile displacement*. By applying these two concepts, no edge will overlap the tiles. In Section 2.2, we will give a brief description of one traditional tile mosaic image creation procedure which includes these two concepts. This method was proposed by Hausner [8].

However, in order to achieve the purpose of information hiding, we will propose a simpler method for tile mosaic creation. The proposed creation process results in no overlapping tiles and the tiles will be positioned in regular grids. We will utilize the linear regression technique for line fitting to derive the *aligning vector*, which is used to re-orient the tile so that the tile can align the nearby edges. Furthermore, without tile displacement and overlapping, tile features such as orientation, size, and texture can be detected much easier. We name the process *tile feature detection process* and will give detailed descriptions in Chapter 3. In Section 2.3, the proposed tile mosaic image creation process for the information hiding purpose will be described in detail. Finally, some experimental results will be presented in Section 2.4.

2.2 Review of Traditional Tile Mosaic Image Creation Process

Before we introduce the whole process of tile mosaic image creation, we will review the concepts of voronoi diagram and centroidal voronoi diagram. First, a set of randomly generated seeds is given on a plane. A voronoi diagram is a graph that divides the plane into several regions (called voronoi regions), such that all points within a region is closest to its associated seed [8]. Fig. 2.1 (a) shows a number of randomly generated seeds and the corresponding voronoi diagram. To generate a centroidal voronoi diagram, we just move each seed to the centroid of its voronoi region and re-compute the voronoi diagram. Fig. 2.1(b) shows a centroidal voronoi diagram after several iterations.



(a) (b)

Fig. 2.1 Voronoi diagrams from [8]. (a) A voronoi diagram. (b) A centroidal voronoi diagram.

Now, we will present the tile mosaic creation process proposed by Hausner [8] by utilizing the voronoi diagram and a *direction field*. A direction field is a collection of vectors, each of which is perpendicular to an edge found in the input image.

The whole process is briefly described in Algorithm 2.1. Initially, for the use of edge avoidance and tile rotation, Hausner [8] derived the direction field D of the original image I. Then he sprinkled seeds randomly and computed the corresponding voronoi diagram. Hoff [9] presented a method (first proposed by Haeberli [5]) that can be extended to draw a voronoi diagram efficiently. In Steps 4 and 5, Hausner utilized Lloyd's algorithm [10] to produce the centroidal voronoi diagram by moving each seed to the centroid of its voronoi region. In Step 6, for edge avoidance, Hausner moved the seeds away from the edges in the directions specified in D and then run Steps 3 through 6 iteratively. Finally, he generated tiles which are located at the seeds' position and rotated to the directions that specified in D in Step 6.

Algorithm 2.1: Traditional tile mosaic image creation process.

Input: an original image T.

Output: a tile mosaic image M.

Steps:

```
Step 1    Derive the direction field D of T.
Step 2    Sprinkle seeds randomly on a two-dimensional plane.
Step 3    Compute the voronoi diagram of the sprinkled seeds.
Step 4    Compute the centroid of each voronoi region.
Step 5    Move each seed to the centroid of its voronoi region.
Step 6    Move the seeds away from the edges.
Step 7    Repeat Steps 3 through 6 in specified iterations.
Step 8    Generate tiles which are located at the seeds' position and rotated to the
```

directions derived in Step 1.

Step 9 Paint the generated tiles on M.

Fig. 2.2 shows some intermediate images and the final tile mosaic image created by Hausner [8]. Fig. 2.2(a) is a Taiji image taken as an input and Fig. 2.2(b) is the corresponding direction field. Fig. 2.2(c) shows a number of randomly generated seeds and the corresponding voronoi diagram. Fig. 2.2(d) shows the centroidal voronoi diagram after twenty iterations without edge avoidance. Fig. 2.2(e) shows the centroidal voronoi diagram after twenty iterations with edge avoidance. Fig. 2.2(f) shows the final tile mosaic image of Fig. 2.2(a).

The method proposed by Hausner [8] looks perfect for tile mosaic image creation. However, there are some weaknesses in Hausner's method for data hiding applications. First, we can observe that some tiles overlap each other in Fig. 2.2(f). Overlapping tiles will cause loss of embedded information and make the tile feature detection work much more complicated. Second, the positions of the tiles are changed. Displaced tiles will disturb the data embedding sequence of the tiles. In Section 2.3, we will propose a new method to create tile mosaic images that are suitable for data hiding. In the proposed method, tiles are squares of identical sizes, and are arranged regularly in the image.

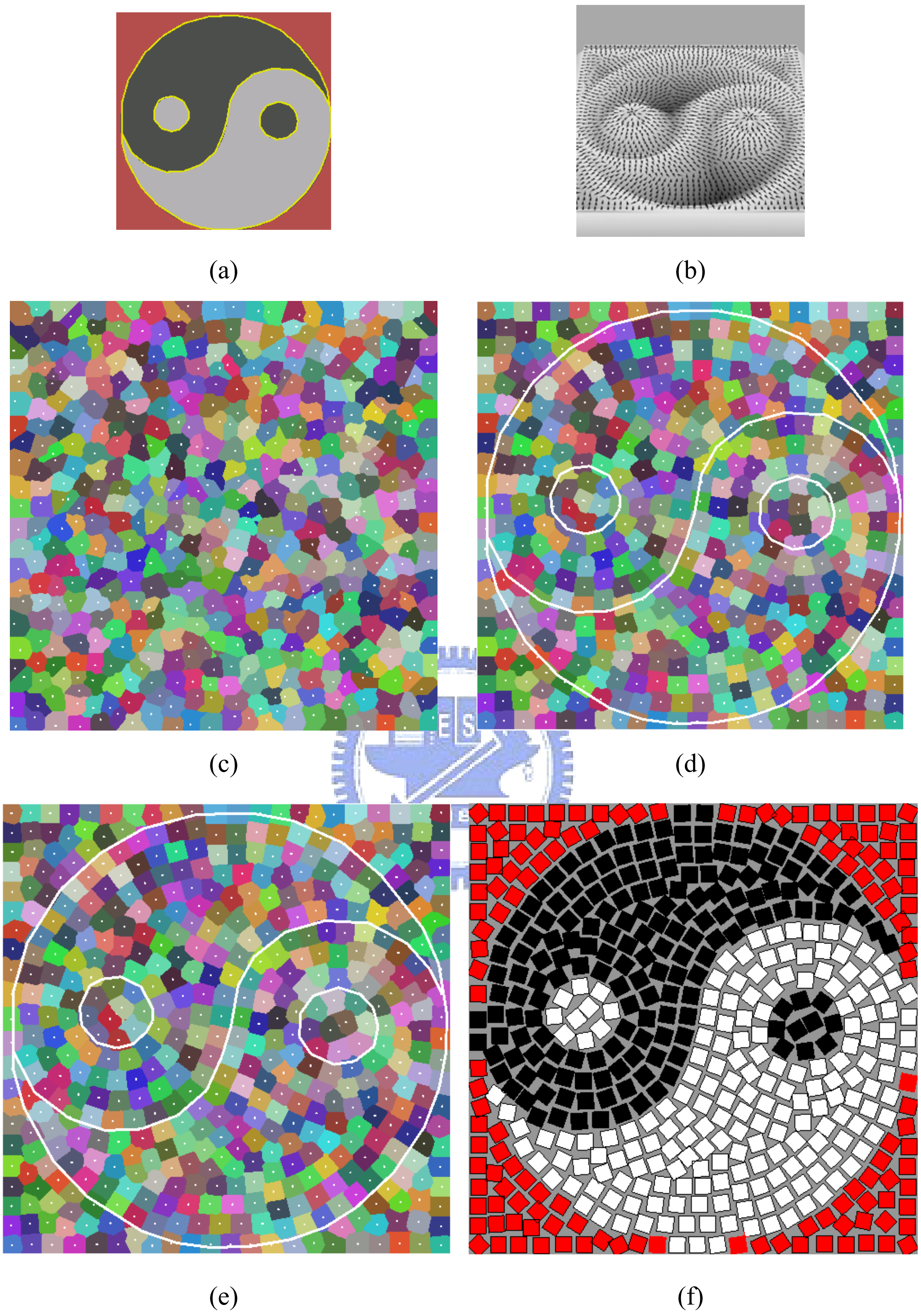


Fig. 2.2 Some intermediate images and a tile mosaic image created by Hausner [8]. (a) The original Taiji image. (b) The direction field of (a). (c) Sprinkled seeds and the corresponding voronoi diagram. (d) A centroidal voronoi diagram of (a) after 20 iterations. (e) A centroidal voronoi diagram of (a) after 20 iterations with edge avoidance. (f) The final tile mosaic image.

2.3 Proposed Tile Mosaic Image Creation Process

2.3.1 Scheme of Creation Process

The proposed tile mosaic creation process is based on Hausner's method [8]. The enhancement in the process allows simple detection of tile features so that the embedded information can be extracted. The proposed creation process results in non-overlapping tiles positioned in regular grids. The overall process is described as an algorithm below.

Algorithm 2.2: Proposed tile mosaic creation process for data hiding.

Input: an original image T and a tile size.

Output: a tile mosaic image M.

Steps:

Step 1 Compute an appropriate inter-tile distance for the input tile size and create accordingly initial upright tiles in M with a chessboard arrangement.

Step 2 Find edges in T by the Sobel operator and threshold the resulting edge-value image into an edge-point image E.

Step 3 For each initial tile L in M, collect a sufficient number of edge points around L and line-fit them to get a *tile aligning vector* V.

Step 4 Rotate L according to the direction of V and fill L with the color of the corresponding tile region center in the original image T.

In Step 1, we compute the inter-tile distance to prevent tiles from overlapping for all possible tile rotation angles, and create a tessellation of upright square tiles in the output image. Let the tile size be X, which is the length of one side of a square shape, as shown in Fig. 2.3. We keep the minimum distance between adjacent tiles to be at least $\sqrt{2}$ X, so that no matter how the tiles are rotated, there will be no overlapping.

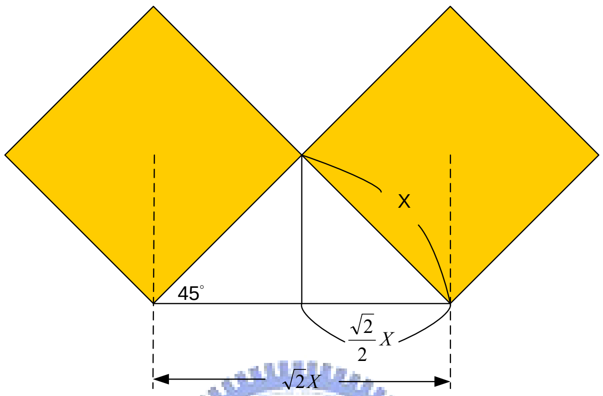


Fig. 2.3 Illustration of minimum distance between neighboring tiles.

In order to rotate the tiles to approximate *T* suitably, we need the edge information of *T*. For this purpose, in Step 2 we apply the 3×3 Sobel operator (as shown in Fig. 2.4) to *T* and threshold the result into a binary edge-point image.

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Fig. 2.4 3x3 Sobel mask

After obtaining the edge points of *T*, we take first in Step 3 an initial neighborhood range of *L* and count the number of edge points within the range. We then keep enlarging the neighborhood range until a pre-determined number of edge points are collected. Finally, we fit the collected edge-points by a line which is then taken as the *aligning vector* for *L*. The flowchart of collecting edge points within a neighborhood range is shown in Fig. 2.5. We will review the utilized linear regression technique for line fitting in Section 2.3.2.

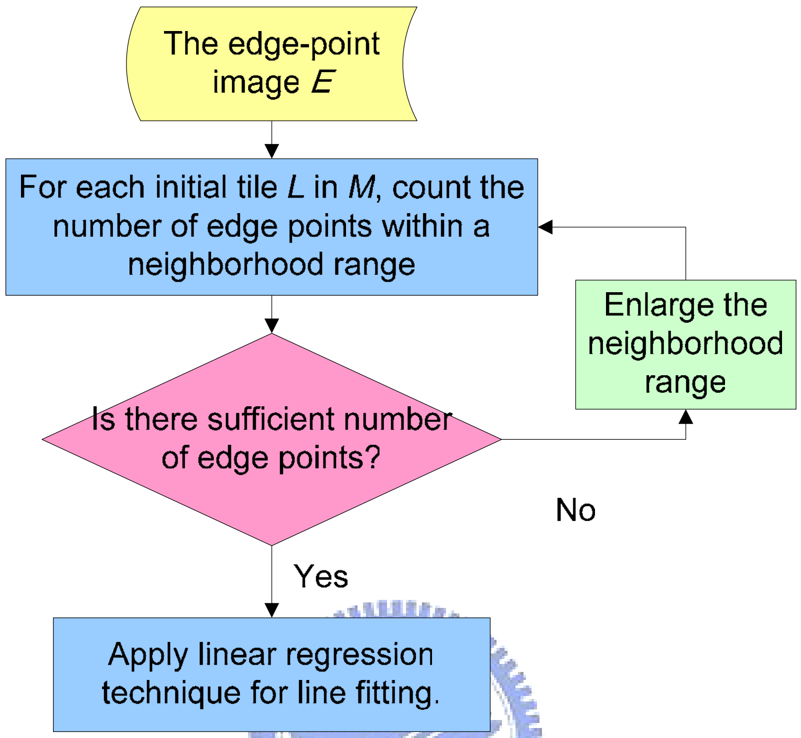

The edge-point image E
For each initial tile L in M, count the number of edge points within a neighborhood range
Is there sufficient number of edge points?
Enlarge the neighborhood range
No
Yes
Apply linear regression technique for line fitting.

Fig. 2.5 Flowchart of proposed line fitting procedure for the aligning vector.

After getting the aligning vectors, we rotate the tiles so that their sides are in line with the aligning vectors and fill in each rotated tile the color at the corresponding region center in the original input image T, as described in Step 4.

2.3.2 Use of Linear Regression Technique for Line Fitting

In this section, we will review the linear regression technique for line-fitting a set of edge points within a neighborhood range. Assume that there are n edge points within the neighborhood range. Let $P_i(x_i, y_i)$ denote the edge point (where $0<i<n$), and $y = b_0 + b_1 x$ denote the formula of the fitted line. In the formula, b_1 stands for the

orientation of the fitted line and b_0 stands for the line position. They can be computed as follows:

$$b_1 = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}, \quad b_0 = \overline{y} - b_1 \overline{x}$$

Because we take the orientation of the line as the aligning vector, only b_1 is computed in our process.

2.3.3 Creation of Visual Effects in Tiles for Information Hiding

There are two visual effects to be handled in our creation process. The first is tile texture and the other is tile border. Here, tile texture means the noise within a tile. The tile border, as implied by the name, is the border of a tile. Both of them can affect our feeling of sight while seeing a tile mosaic image. Fig. 2.6 shows tiles with and without these two effects.

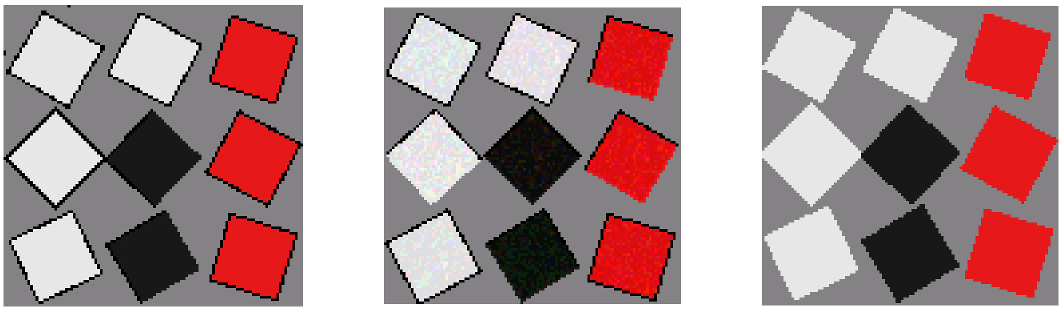
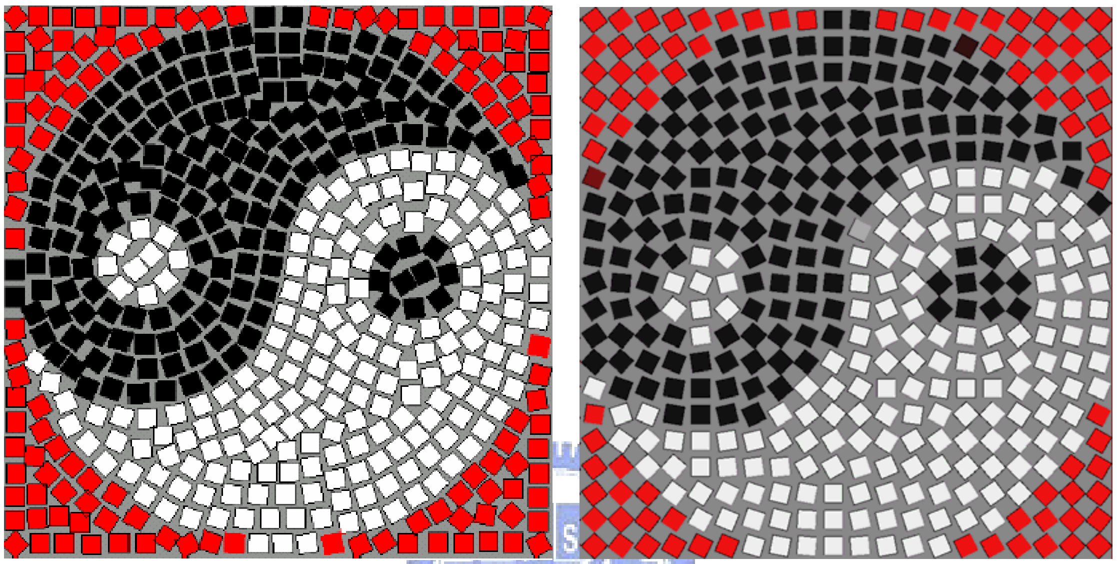

(a) (b) (c)

Fig. 2.6. Tiles with visual effects. (a) Tiles without texture effect. (b) Tiles with texture effect. (c) Tiles without border and texture effect.

2.3.4 Experimental Results and Discussions

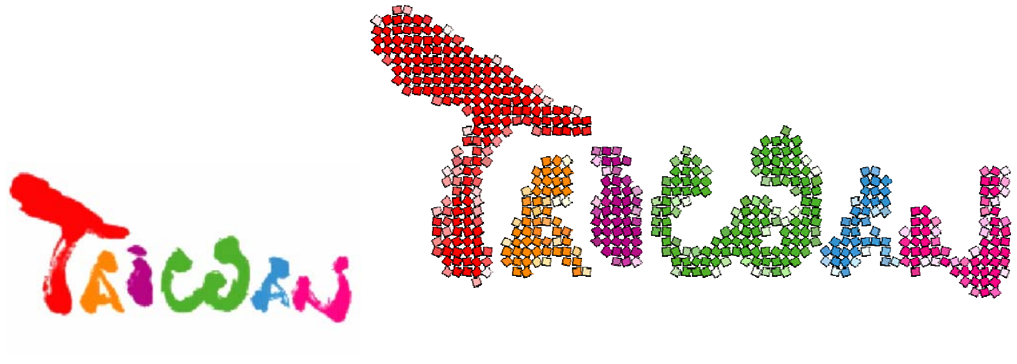
In this chapter we reviewed a traditional tile mosaic image creation process and proposed a new approach of it. Fig. 2.7 shows the tile mosaic images created by these two methods.



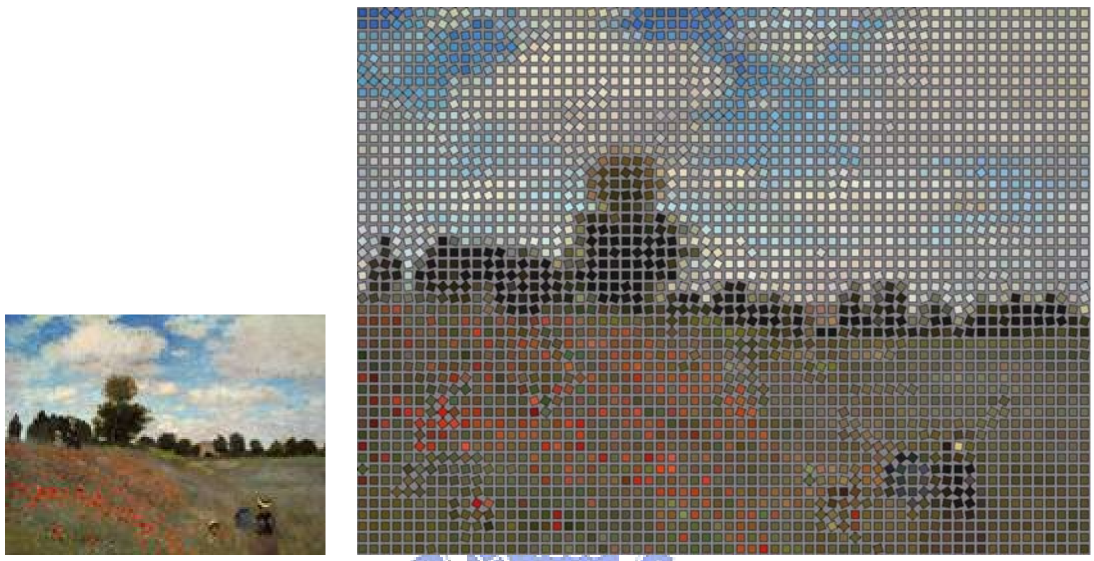
(a) (b)

Fig. 2.7 Tile mosaic image creation. (a) A tile mosaic image created by Hausner's creation process [8]. (b) A tile mosaic image created by proposed creation process.

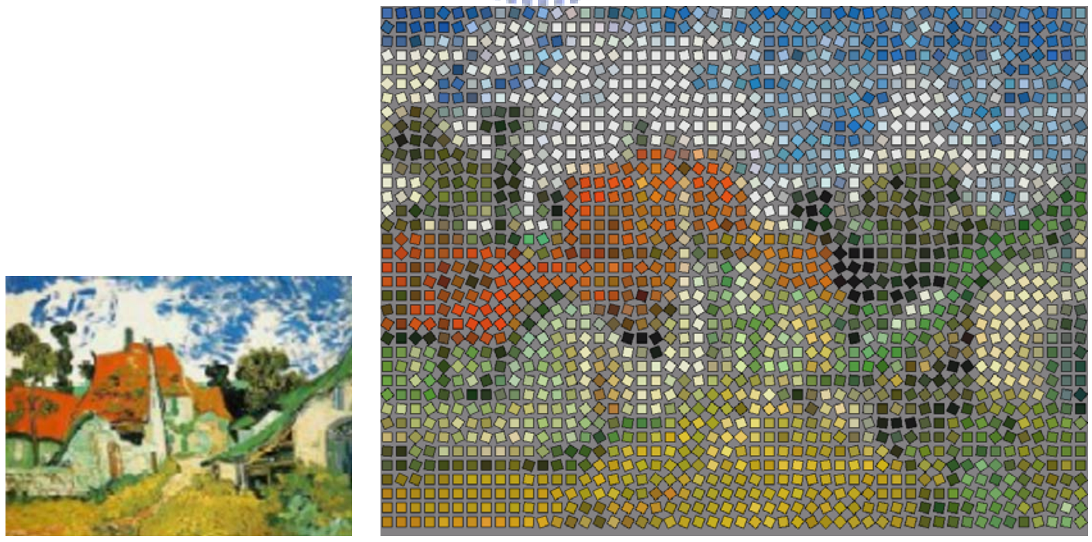
As seen from the examples shown in Fig. 2.7, our creation process creates looser tiles than those created by Hausner [8]. With looser arrangements of tiles, we can guarantee that there will be no overlapping tile. Furthermore, we notice that the tiles in Fig. 2.7(b) just rotate at where they are in the chessboard grid without any displacement, so that we can get the tile positions easily. With these two features different from those of Hausner's method, information embedding and extraction become possible and easy, and will be introduced in the next chapter. Some other tile mosaic images created by the proposed method is shown in Fig. 2.8.



(a)



(b)



(c)

Fig. 2.8 Some created tile mosaic images. (a) A tile mosaic image of the word "Taiwan". (b) A tile mosaic image with a painting of Monet as input. (c) A tile mosaic image with a painting of Van Gogh as input.

Chapter 3
Three Methods for Data Hiding in Tile Mosaic Images

3.1 Introduction

As we mentioned in Chapter 1, we embed data into certain image features of a host art image. We will introduce how to hide data in the orientations, sizes, and textures of tile mosaic images in this chapter. Of course, the tile feature detection process will also be discussed.

3.1.1 Properties of Tile Orientations

The properties of tile angles are illustrated in Fig. 3.1 where we can see that a tile rotated through an angle of θ has the same rotation effect as through an angle of θ+90^{o}. Thus, the effective range of rotation angles may be limited to be 0 ≦ $\theta < 90^{o}$. Fig. 3.2 shows the effective range of θ. We paint it with darker blue (i.e., the second quadrant) in Fig. 3.2.

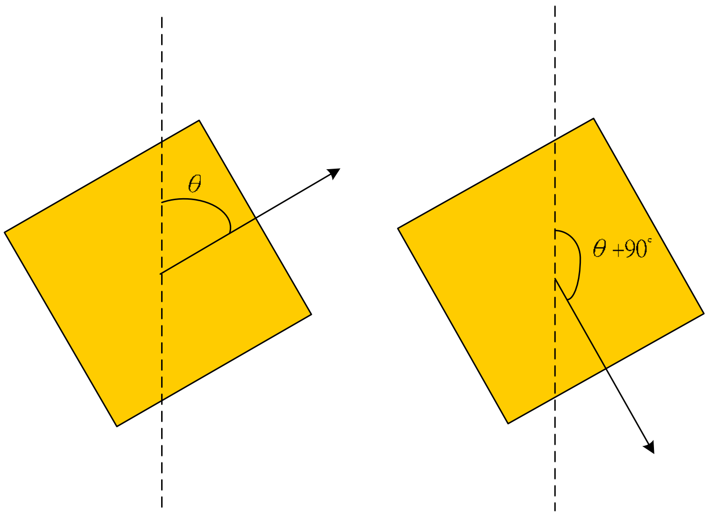


Fig. 3.1 Properties of tile orientations.

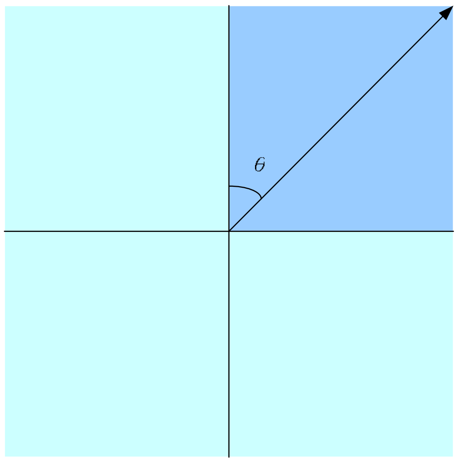


Fig. 3.2 Effective range of *θ*.

3.1.2 Properties of Tile Sizes

In Section 3.4, an application of data hiding by tile size modification will be described. We will utilize the number of pixels (i.e., tile pixel number, *TPN*) within a tile instead of the edge length to stand for the tile size. Because *TPN* can provide us more precise information about tile size than edge length can do. All the details about data hiding by tile size modification and its application will be discussed in Section 3.4.

3.1.3 Properties of Tile Textures

Data hiding by tile texture modification is another interesting way of hiding data in a tile mosaic image. Here, tile texture means the noise within a tile. As discussed in Section 2.3.3, tile texture can affect our sense of sight while seeing a tile mosaic image. The way we utilize tile texture for data hiding is to modify the *noise ratio*. The term, noise ratio, means the ratio of noise pixels within a tile. By detecting the noise ratio, we can get the data hidden in a tile.

We will limit the range of noise ratio, *Nr*, from 0 to 0.5, that is, $0 \le Nr \le 0.5$. The reason for confining the noise ratio to be smaller than 0.5 is described as follows.

When processing data extraction, in each RGB channel of a tile, we try to find two different values of colors (one is the original tile color and the other is the noise color). We then compute the corresponding ratios of these two values and view the one which has the lower ratio as the noise color and view the higher one as the original tile color. So, in the data embedding process, if we add noise pixels with ratio greater than 0.5, we will view the noise pixels as original tile pixels. In such a condition, we will extract data erroneously.

3.1.4 Concepts of Proposed Data Embedding and Extraction Techniques

In this section, we will give an overall concept and the flowcharts of data hiding in tile mosaic images after having a glance at the three tile features. As shown in Fig. 3.3, initially we apply the tile mosaic image creation process proposed in Section 2.3, and embed data by modifying the three features (i.e., orientation, size, and texture). Note that we can sequentially embed data into the three features. First, we embed a watermark by modifying the tile orientations. Second, we embed a secret message by tile size modification. Finally, we generate authentication signals and embed them by tile texture modification. By authentication signals, we can verify the integrity of a tile mosaic image together with the embedded data. Of course, not all of the three types of the given data need to be embedded at the same time in all applications. We can embed only one or two of them as needed. All the details of the procedures for embedding the three types of features will be discussed in detail in Sections 3.3, 3.4, and 3.5. The way we extract the embedded data from a tile mosaic image is simply the inverse of how we embed it. Fig 3.4 shows a flowchart of extracting data from a tile mosaic image. Initially, we apply the proposed tile feature detection process to extract the tile features and then derive the embedded messages by analyzing the extracted features.

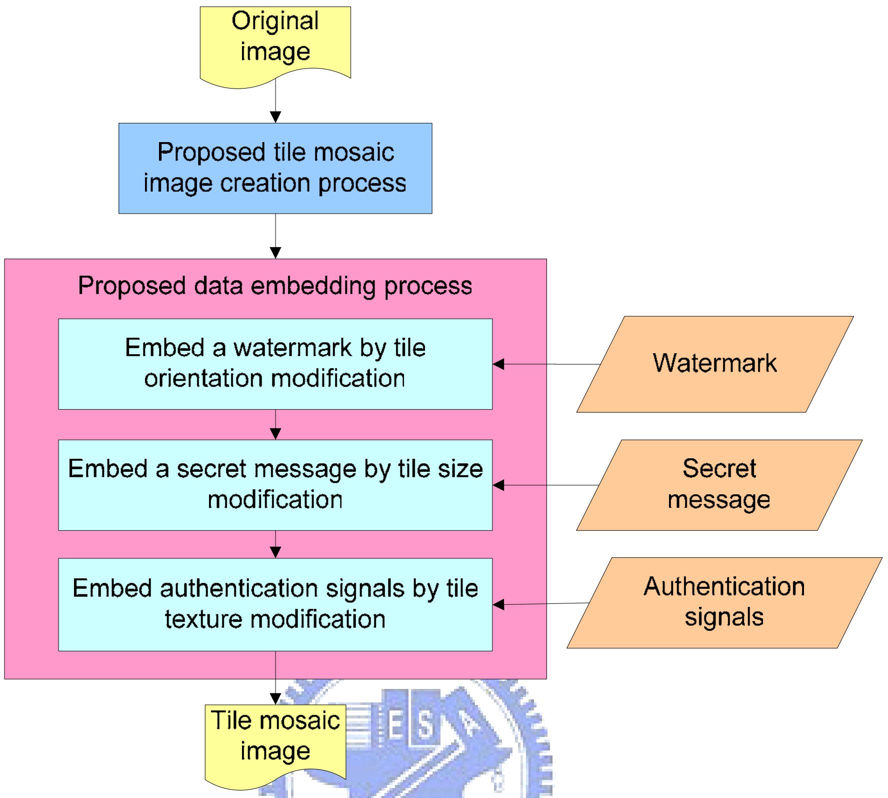

Original image
Proposed tile mosaic image creation process
Proposed data embedding process
Embed a watermark by tile orientation modification
Embed a secret message by tile size modification
Embed authentication signals by tile texture modification
Watermark
Secret message
Authentication signals
Tile mosaic image

Fig. 3.3 Flowchart of data hiding in a tile mosaic image.

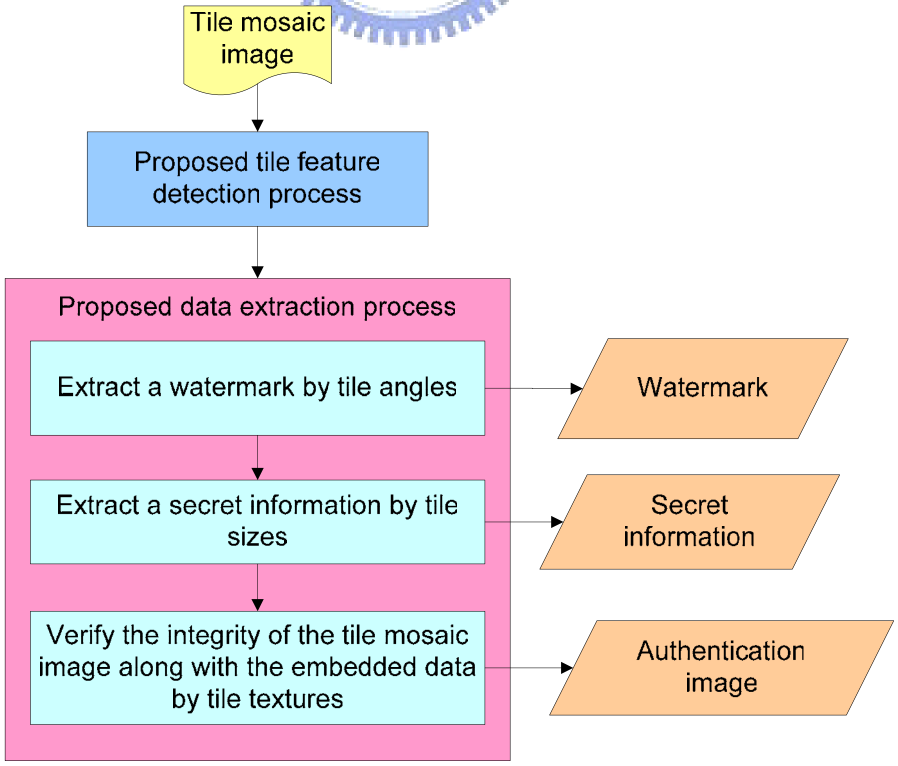

Tile mosaic image
Proposed tile feature detection process
Proposed data extraction process
Extract a watermark by tile angles
Extract a secret information by tile sizes
Verify the integrity of the tile mosaic image along with the embedded data by tile textures
Watermark
Secret information
Authentication image

Fig. 3.4. Flowchart of data extraction from a tile mosaic image.

3.2 Proposed Tile Feature Detection Techniques

Before introducing the proposed data hiding method, we first introduce the proposed tile feature detection method. As shown in Algorithm 3.1 and Fig. 3.5, first we derive the approximate tile positions by *tile scanning*. After tile scanning, we can get a two-dimensional array of grids, and in each grid there is a tile. We will have a detailed discussion about tile scanning in Section 3.2.1. Second, we apply a *tile region detection process* to each of the grids to derive a *tile region map* which indicates whether the pixels in a grid belong to the associated tile or not. We will discuss the tile region detection process in detail in Section 3.2.2. In Step 3, in order to derive the tile orientations, we need the *four apexes* of each tile. We apply the proposed *tile boundary detection* algorithm to achieve that purpose. And the details will be described in Section 3.2.3.

With the *four apexes* and the *tile region map* of a tile, we can get the tile orientation *To*, the tile size (i.e., the number of the tile pixels), *TPN* and the tile noise ratio *Tnr*. Furthermore, we can get the average of *TPN* which is denoted by *TPNavg* by testing whether the detected boundary is rectangular. If not, it means that there is something wrong with the tile. When calculating the value of *TPNavg*, we need to skip the non-rectangular tiles. By doing so, we can derive a value of *TPNavg* with higher accuracy. All the details will be discussed in Section 3.2.4.

Algorithm 3.1: Tile feature detection process

Input: a tile mosaic image *M*.

Output: tile orientation *To*, the number of tile pixels *TPN*, average *TPN* values, *TPNavg*, tile noise ratio *Tnr*.

Steps:

Step 1 Derive tile positions by tile scanning.

Step 2 For each tile perform the following steps.

2.1 Derive the tile region map by tile region detection.

2.2 Derive the four apexes by tile boundary detection.

2.3 Get the tile orientation, *To*, by the four apexes.

2.4 Detect the tile noise ratio, *Tnr*, by the tile region map.

2.5 Get the number of tile pixels, *TPN*, by the tile region map

2.6 Check if the detected boundary is rectangular.

if it is rectangular, then accumulate the value of *TPN*.

Step 3 Compute the value *TPNavg* by the accumulation *TPN* value calculated in Step 2.6.

3.2.1 Tile Scanning

In Step 1 of Algorithm 3.1, we derive the tile positions by tile scanning. The result is a two-dimensional array of grids, and in each grid there is a tile. In this step, we compute first the horizontal and vertical projections of the background pixels (i.e., the non-tile pixels) of the input image *M*, as illustrated in Fig. 3.6. We then search the accumulation values of the projections for local maxima. The scan lines with local maximum accumulation values are taken as the desired grid lines, shown as white lines in Fig. 3.6. With accumulation histograms, we can see clearly in the figure the correspondence between the local maximum accumulation values and the grid lines. The algorithm of tile scanning is shown in Algorithm 3.2.

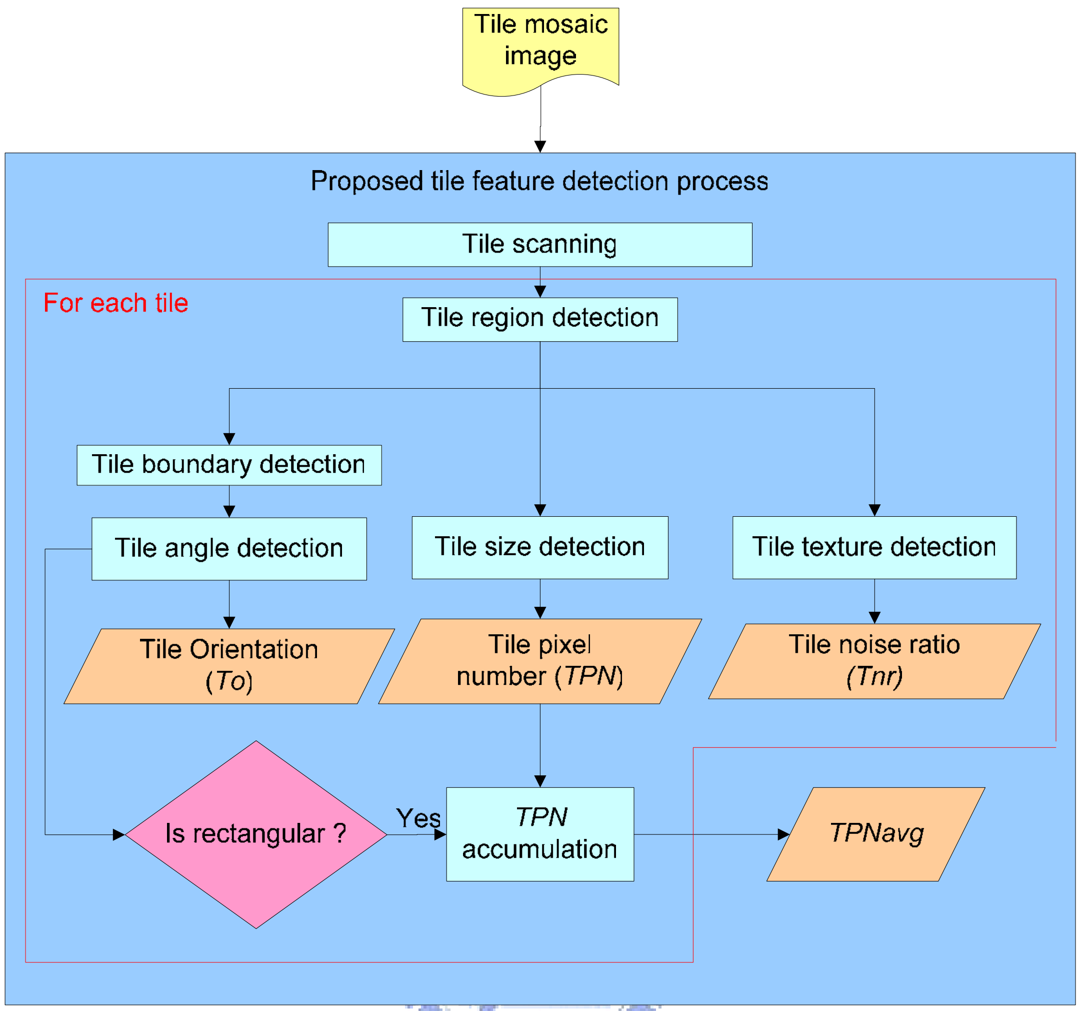


Fig. 3.5 Proposed tile feature detection process.

Algorithm 3.2: Proposed tile scanning process

Input: a tile mosaic image *M*

Output: two-dimensional grids

Steps:

Step 1 Compute the horizontal and vertical projections of the background pixels in *M*.

Step 2 If the accumulation value is local maximum, output this scan line as a grid line.

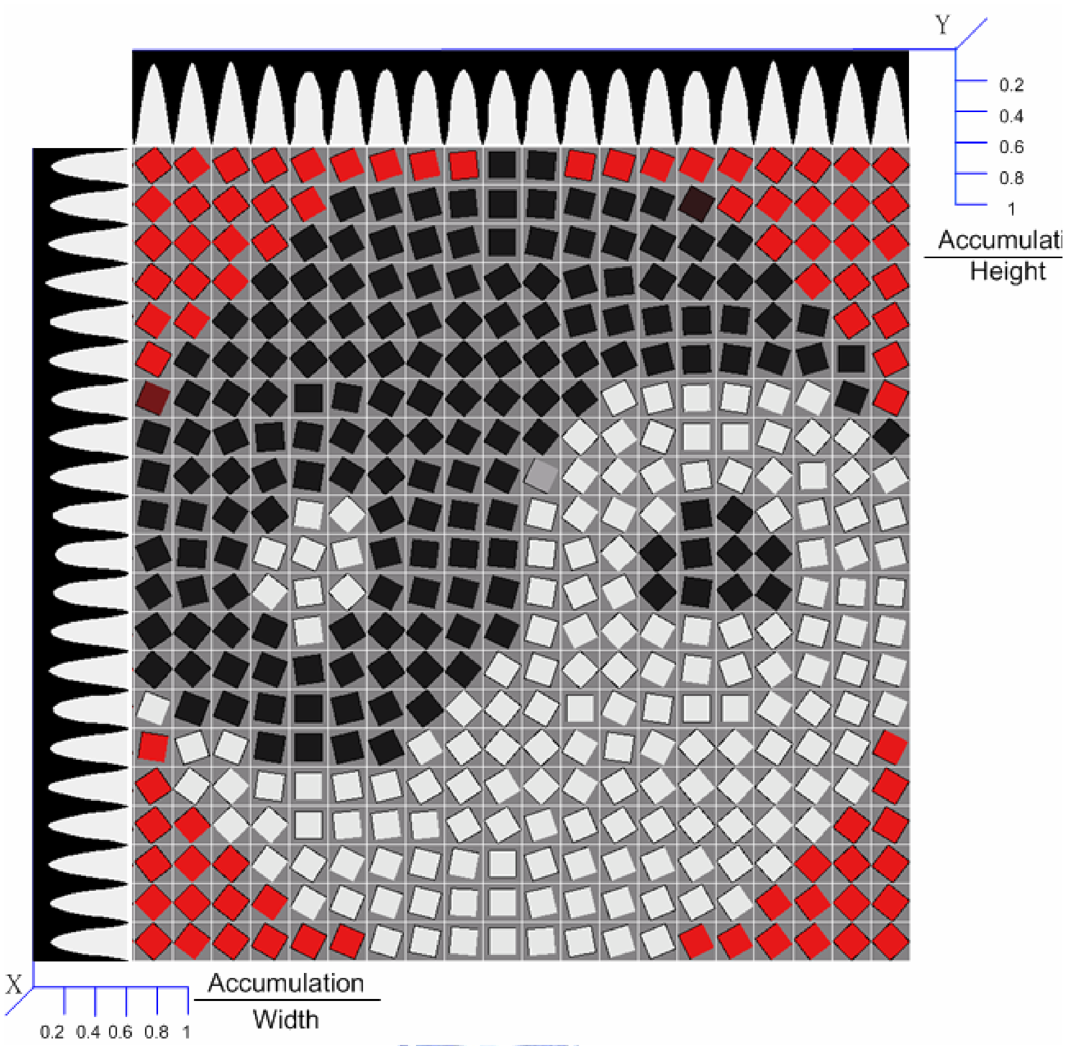
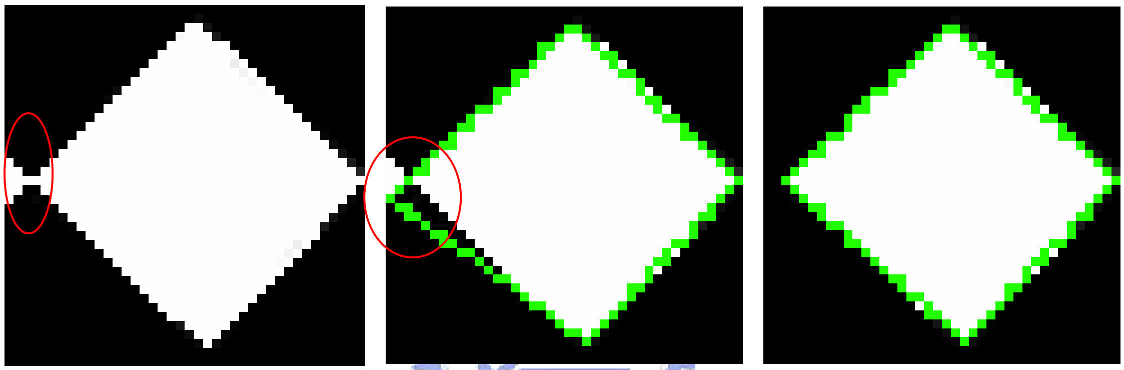


Fig. 3.6 Finding grid lines with local maximum accumulations.

3.2.2 Tile Region Detection

After obtaining the approximate positions and ranges of the tiles by tile scanning, we determine the tile regions for further analysis in Step 2.1 of Algorithm 3.1. We use a region growing technique to create a tile region map, *Mp*, and remove the untouched pixels in the growing process from the tile region map. In more details, it is mentioned first that the two-dimensional grids derived in the previous section only provide the approximate ranges of the tiles. As a result, given a grid *G*, the corners of the tiles of the neighboring grids of *G* may intrude into *G*, in touch with the tile in *G*. An example is shown in Fig. 3.7(a) where this type of intrusion takes place on the left-hand side of the grid. Such touching will result in an error as shown in Fig. 3.7(b) after we perform *tile boundary detection* in Step 2.2 of Algorithm 3.1. The way we detect such erroneous corners is to scan each grid from the grid edges to the grid center, and search for the local minimum accumulation values of the projections of the tile pixels. We then remove such erroneous corners of adjacent tiles before we detect the tile boundary. The result will then be more accurate, as shown in Fig. 3.7(c). The details of tile region detection are described in Algorithm 3.3.



(a) (b) (c)

Fig. 3.7 Tile region detection. (a) A tile region map with erroneous points at the left. (b) a detected tile boundary with an erroneous apex at the left. (c) a detected tile boundary with accurate apexes.

Algorithm 3.3: Tile region detection process

Input: a grid *G*.

Output: a tile region map *Mp*.

Steps:

Step 1 Compute the central point, *C*, of *G*.

Step 2 Choose C as the starting point. Apply a region growing technique on *C* for deriving *Mp*.

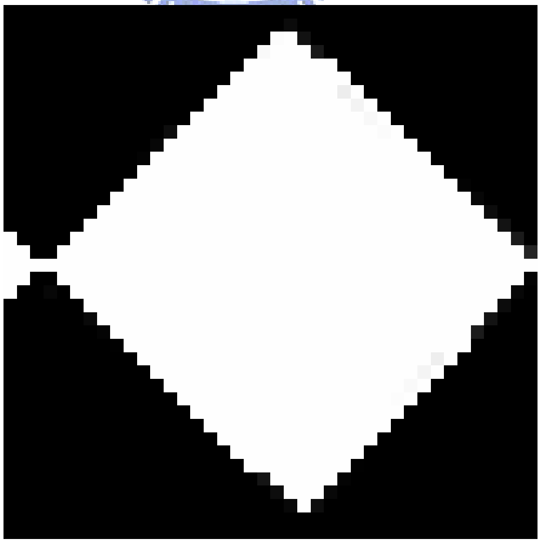
Step 3 Scan *Mp* from the grid edges to the grid center, and search for the local minimum accumulation values of the projections of the tile pixels.

Step 4 Remove tile pixels, which are outer than the scan lines found in Step3, from *MP*.

Step 5 Output the tile region map *Mp*.

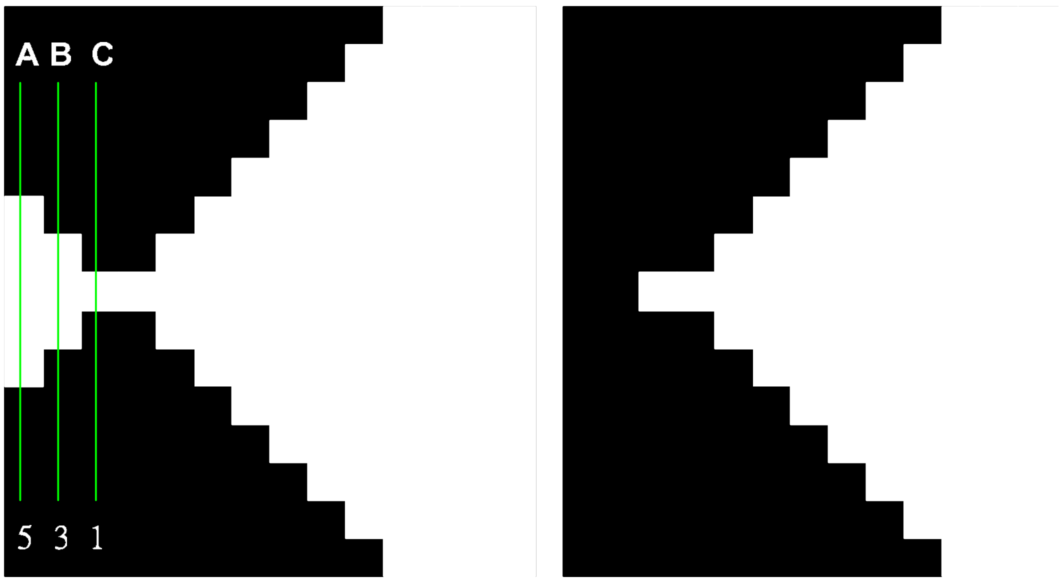
Fig. 3.8 is an illustration of Steps 3 and 4 of Algorithm 3.3. In Fig. 3.8(b), the green lines A, B, and C are the scan lines of the left grid edge. C is the scan line of the local minimum accumulation value. In Fig. 3.8(c), we can see that the erroneous corner is removed by removing the pixels which are outer than scan line C.

After deriving the tile region map, *MP*, we can detect the tile boundary of each tile by the proposed tile boundary detection process. Fig. 3.9 shows some experimental results of tile boundary detection. Fig. 3.9(a) is the result of tile boundary detection without utilizing Steps 3 and 4 in Algorithm 3.3 when deriving *MP*. Fig. 3.9(b) is the result with the two steps conducted. It's clear that if we apply the two steps to the tile region map, we can get more accurate result in tile boundary detection.



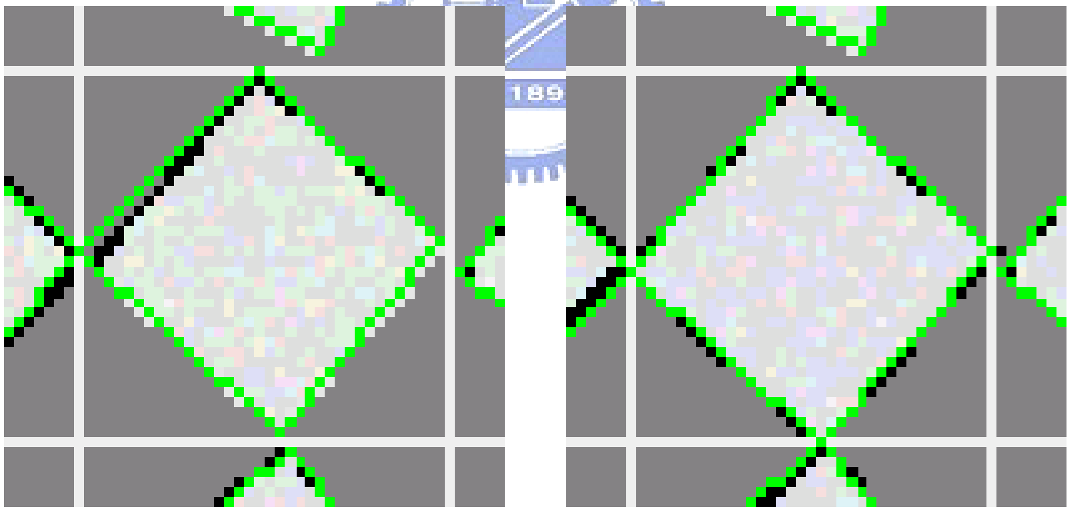
(a)

Fig. 3.8 Removing erroneous corners of adjacent tiles (a) A tile region map. (b) Scan from the left grid edge for minimum accumulation value. (c) Details of a tile region map after error corner elimination.



(b) (c)

Fig. 3.8 Removing erroneous corners of adjacent tiles (a) A tile region map. (b) Scan from the left grid edge for minimum accumulation value. (c) Details of a tile region map after error corner elimination(continued).



(a) (b)

Fig. 3.9 Experimental results of tile boundary detection. (a) A result without utilizing step 2 of algorithm 3.1.2. (b) A result with utilizing step 2 of algorithm 3.1.2.

3.2.3 Tile Boundary Detection

The next step of tile feature detection is the tile boundary detection process, as described in Step 2.2 of Algorithm 3.1. By this process, we can derive the four apexes of a tile. The orientation of the tiles can then be computed easily from these four apexes. Algorithm 3.4 presents the proposed tile boundary detection process.

Algorithm 3.4: Tile boundary detection process.

Input: a tile region map, *Mp*.

Output: the four apexes of each tile, *n*, *e*, *s* and *w*, respectively, and the number of the tile pixels, *TPN*.

Steps:

Step 1 Perform a raster scan of the tile region map, get the lists of the highest, rightmost, lowest, and leftmost tile pixels, and denote them by *N*, *E*, *S* and *W*, respectively.

Step 2 Compute the number of tile pixels, *TPN*, in the tile region map, and the approximate tile size in pixels, *Ts*, by the formula $Ts = \sqrt{TPN}$.

Step 3 Compute the centroid of the tile region map and denote it by $C(cenX, cenY)$.

Step 4 For each tile *L*, check if *L* is rotated or not in the following way: if the number of pixels in at least two of *N*, *E*, *S*, and *W* are larger than $Ts - 3$, then

4.1 regard the tile as non-rotated and compute the four apexes as $n(cenX - Ts/2, cenY - Ts/2)$, $e(cenX + Ts/2, cenY - Ts/2)$, $s(cenX + Ts/2, cenY + Ts/2)$ *and* $w(cenX - Ts/2, cenY + Ts/2)$; otherwise,

4.2 regard the tile as rotated and choose from *N*, *E*, *S*, and *W* the pixels which are farthest from the centroid, $C(cenX, cenY)$, as the four apexes.

Step 5 Output the four apexes of *L*.

In the first step, we scan the tile region map horizontally and vertically to get the four lists of the outmost tile pixels, *N*, *E*, *S*, and *W*. As shown in Fig. 3.10(a) and (c), the hollow rectangles are the detected four lists. In Step 2 we compute the approximate tile size. In Step 3, we compute the centroid of the white pixels in the region map. In Step 4, we determine whether the tile is rotated or not by checking if there are at least two lists whose sizes are larger than *Ts* – 3. It can be figured out that if the tile is not rotated, then there will be a sufficient number of vertically or horizontally aligned pixels, collected into *N*, *E*, *S*, and *W*. And we take this number to be *Ts* – 3 according to our experimental experience, which provides the best accuracy. As shown in Fig. 3.10(a) and (b), if the tile is rotated, we derive the four apexes by Step 4.1; otherwise, we derive the four apexes by Step 4.2, as shown in Fig. 3.10(c) and (d). In Fig. 3.10(b) and (d), the solid squares are the chosen four apexes, which are denoted by *n*, *e*, *s* and *w*, respectively, and the blue quadrangles are the detected tile boundaries.

3.2.4 Tile Feature Detection

After deriving the tile region map and the four apexes of each tile, we can now compute the remaining tile features. The detection processes are described as follows.

A. Tile Orientation Detection

The way we detect the tile orientation, θ, is to compute the directions of the four boundaries derived from the tile boundary detection process. As shown in Fig. 3.11, we compute the tile orientation by θ_{wn} = arctan(ΔY/ΔX), and the other three tile orientations θ_{ne}, θ_{es}, θ_{sw} similarly. In order to determine the tile orientation with more accuracy, the directions of the four boundaries are averaged as the desired result, that is, the final tile orientation θ is obtained by the formula: $\theta = (\theta_{wn} + \theta_{ne} + \theta_{es} + \theta_{sw})/4$.

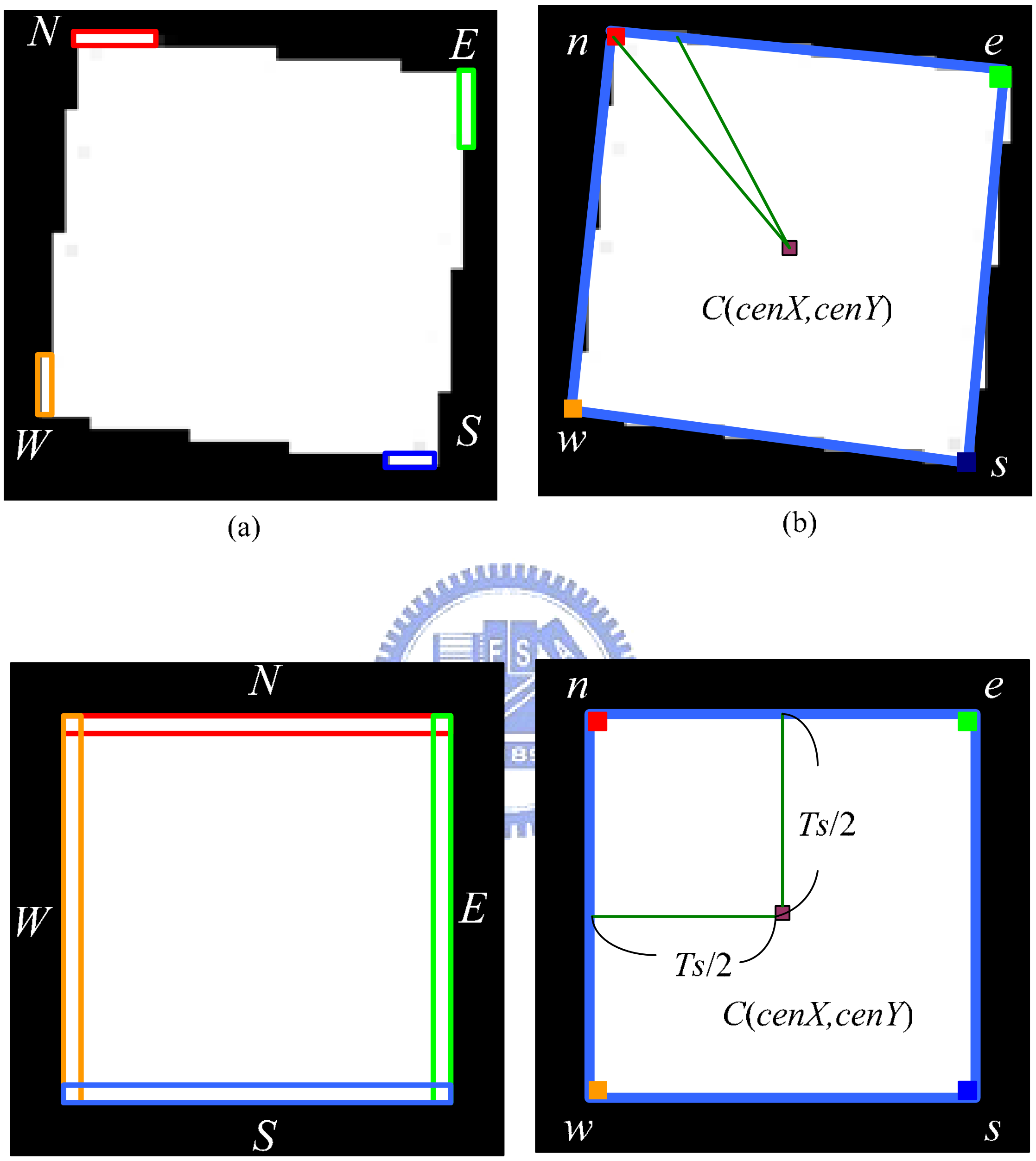


Fig. 3.10 Illustrations of boundary detection. (a) Rotated tile with the four detected lists; (b) rotated tile with the detected four apexes and tile boundary; (c) non-rotated tile with the four lists; (d) non-rotated tile with the detected four apexes and tile boundary.

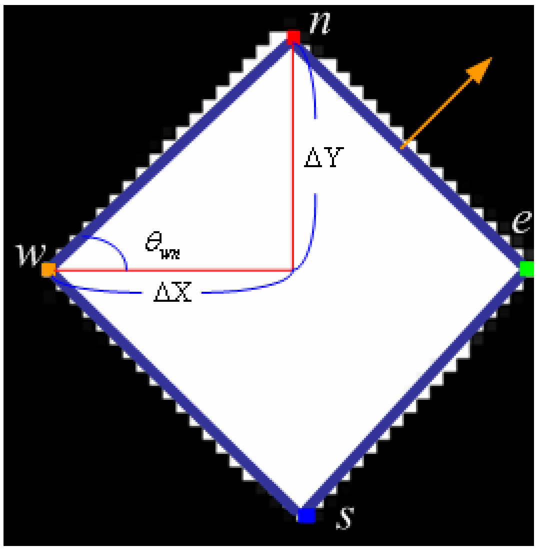


Fig. 3.11 Tile orientation detection

B. Tile Size Detection

As mentioned in Section 3.1.2, for more accuracy information of tile sizes, we utilize the number of the tile pixels, *TPN*, instead of the tile size, *Ts*. In fact, the value of *TPN* is computed in Step 2 of Algorithm 3.4 while we process tile boundary detection. Note that *TPN* is the number of white pixels in the tile region map derived from the tile region detection process.

C. Average Tile Size Detection

As implied by the name, the average tile size, *TPNavg*, is the average of all the values of *TPN* in a tile mosaic image. To compute the average value of *TPN* is simple, but we have to notice if the tile has been damaged or not. If a tile has been damaged, the detected value of *TPN* will be an enormous error. So, in order to keep *TPNavg* from being affected by damaged tiles, we must ignore the *TPN* which are computed from damaged tiles. In other word, we compute the value of *TPNavg* by averaging the values of *TPN* of all tiles except the damaged ones.

The way we check whether a tile has been damaged or not is to check the directions of the four boundaries (i.e., θ_{wn}, θ_{ne}, θ_{es}, and θ_{sw}), which are derived in the tile boundary detection process. If there is any adjacent boundary pair whose directions differ in an amount greater than 10^{o}, we judge the associated tile as a damaged tile.

D. Tile Texture Detection

By consulting a tile region map derived from the tile region detection process, we can compute the noise ratio, *Nr*, of a tile. All the details of tile texture detection have been discussed in section 3.1.3. Note that, we compute the noise ratios in the RGB channels individually. That is, we will have three noise ratios of each R, G and B channel after tile texture detection process.

3.3 Proposed Watermarking Method by Tile Orientation Modification

We have described the details of tile feature detection in tile mosaic images in the previous sections. In this section, we will describe the proposed technique for data hiding in tile mosaic images, and an application, watermarking, for copyright protection of images. The tile feature we utilize for data hiding is tile orientation. We use the idea of slight modification of tile orientations to achieve our purpose of hiding data in tile mosaic images. Due to the characteristic of tile orientation detection, errors may occur when we conduct tile orientation detection. Most of the errors are smaller than 3^{o}, but it may still result in bit errors in the result of data extraction. Although the bit error rate is low (around 2 or 3 error bits in 700 tiles), it can cause some damages in the extracted data. In the case of Big5 text, for example, one error bit can destroy a character containing 16 bits. There are two ways for solving this problem. The first is to enlarge the degree of tile orientation modification, but it will cause the tiles to rotate farther away from their original alignments in positions and so damage more of the edges of the tile mosaic image. The second way is to enlarge the size of the tiles, but this will cause more loss of the details of the original image. Thus there is a tradeoff between bit accuracy and aesthetic appearance of the tile mosaic image. However, in the application of watermarking we will do nothing about it. The error bits are still there and will cause salt-and-pepper noise in the extracted watermark, but the watermark can still be recognized visually under the condition of two or three erroneous pixels. Furthermore, we can apply certain more effective data extraction skills like the voting strategy to reduce the salt-and-pepper noise in the extracted watermark. The proposed process of data hiding by tile orientation modification is described in the following.

3.3.1 Core Concept

The core concept of data hiding by tile orientation modification is illustrated in Fig. 11. In principle, we hide data by encoding the angle between the orientations of every two adjacent tiles. Assume the orientation of $Tile_i$ to be θ_i and that of $Tile_{i+1}$ to be θ_{i+1}. By modifying θ_{i+1}, we can adjust the absolute value of the difference $\theta_i - \theta_{i+1}$ for data hiding. More specifically, as shown in Fig. 3.12(a) and (c) with a right angle which centers on θ_i, we divide the right angle into several sectors (four in the figure), and divide each sector into several sub-sectors, with each sub-sector representing a specific bit code. The number of sub-sectors, RN, in each sector is based on the number of bits, $bitN$, we want to embed in the sector. That is, $RN = 2^{bitN}$. When the value of $bitN$ is one, there are two sub-sectors in each sector, representing bit codes 0 and 1, respectively. When $bitN$ equals two, RN will be four, and the four sub-sectors in a sector represent bit pair codes 00, 01, 10, and 11, respectively.

Assume now that we want to embed a bit 0 into $Tile_{i+1}$. Since θ_{i+1} falls in the sub-sector which represents bit 1, we have to adjust θ_{i+1} to fall in the sub-sector which represents bit 0 by rotating $Tile_{i+1}$ through a certain angle. As shown in Fig. 3.12(c), the closest sub-sector which represents bit 0 is the right one. So we adjust θ_{i+1} to align it with the center of that sub-sector to get the new angle of θ_{i+1}'. In Fig. 3.12(c), the red dashed arrow is the θ_{i+1}' with bit 0 embedded. On the other hand, if we want to embed a bit 1 into $Tile_{i+1}$, since θ_{i+1} falls in the sub-sector which represents bit 1, we just adjust θ_{i+1} to align with the center of the sub-sector. By doing so, we can have more accuracy in the later process of data extraction. If θ_{i+1} initially does not fall in the right angle which centers on θ_i, we will adjust θ_{i+1} by adding or subtracting 90^o to make θ_{i+1} fall in the right angle. As described in Section 3.1.1, this operation will not affect the orientation of a tile.

We can see that the average angle of tile rotations for data hiding is half of the span angle of a sector. By increasing the number of sectors (reducing the span angles of sectors), the degree of tile re-orientation can be reduced, but this will cause more errors when performing tile orientation detection in data extraction. On the contrary, we can reduce the number of sectors to increase the accuracy of tile orientation detection, but this will cause the tiles to be rotated farther away from their original alignments, causing the edges in the image to be damaged more seriously.

3.3.2 Data Embedding Process

The procedure of embedding a watermark into a tile mosaic image is shown in Fig. 3.13 and described in Algorithm 3.5 below.

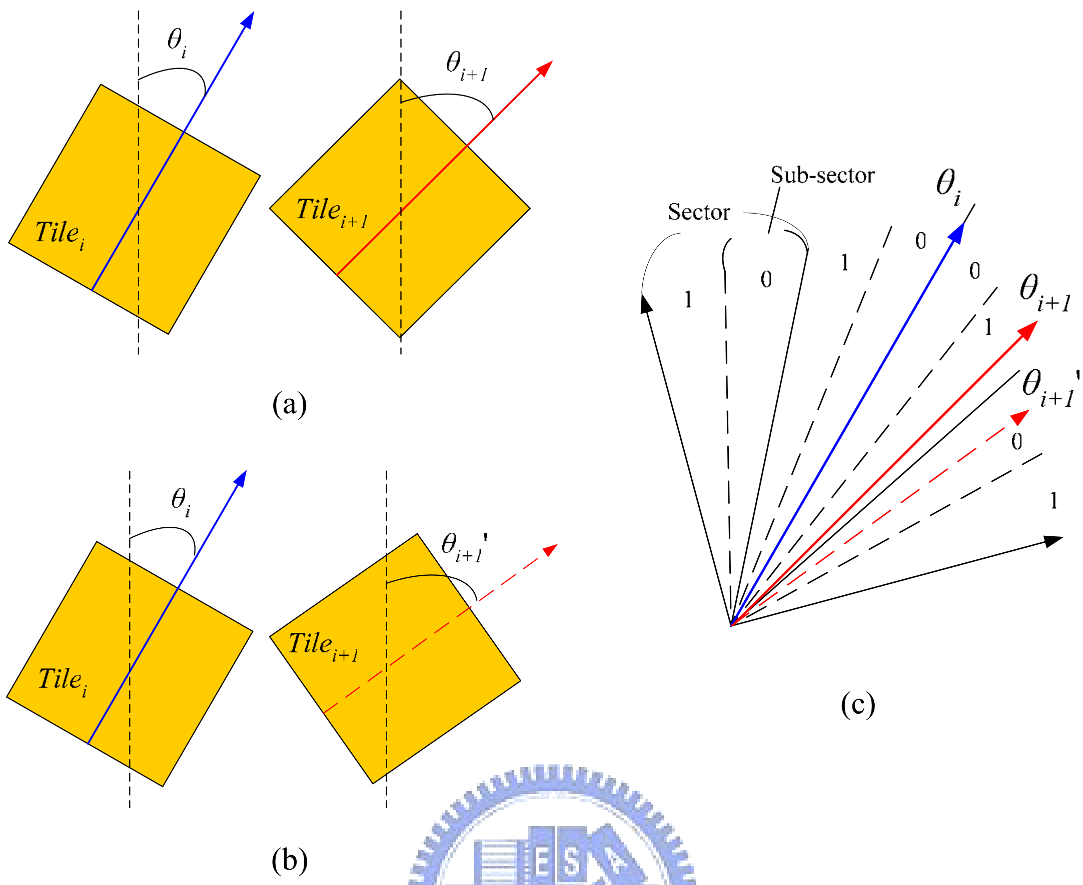


Fig. 3.12 Data hiding by tile orientation modification. (a) Two adjacent tiles. (b) Tiles with bit 0 embedded in $Tile_{i+1}$. (c) Data hiding strategy.

Algorithm 3.5: Watermarking in a tile mosaic image.

Input: a tile mosaic image M, a watermark image W, and a key K.

Output: a tile mosaic image with the watermark embedded.

Steps:

Step 1 Link the tiles in M into a 1-D sequence $Tile_0$, $Tile_1$, …, $Tile_m$ by a raster scan of the tiles.

Step 2 Resize the input watermark image W into 25×25 pixels and transform it into a bit sequence $B = B_0, B_1, \ldots, B_n$.

Step 3 Generate a sequence of sub-sector code mappings, $CM = CM_0, CM_1, \ldots, CM_{m-1}$, by the input key K.

Step 4 For each adjacent tile pair $Tile_i$ and $Tile_{i+1}$, embed B_i into M by modifying the orientation θ_{i+1} of $Tile_{i+1}$ according to the code mapping CM_i.

Step 5 If the embedding capacity, $m - 1$, is larger than the length of the bit sequence B, embed B repeatedly by Step 4 until the embedding capacity is exhausted.

In the first step of the above algorithm, we link the tiles into a 1-D sequence. Then, in Step 2 we resize the input watermark image into an image of 25×25 pixels and transform it into a bit sequence B. Before modifying the orientations of the tiles, we decide in Step 3 what code mappings the sub-sectors will represent in the subsequent data embedding. The result is a sequence of code mappings, CM. As shown in Fig. 3.12(c), the code mapping CM_i for the right angle (used by a tile $Tile_i$), as seen from the center to the two sides, is (0, 1). The alternative choice is (1, 0). We generate the code mappings in CM randomly for all the tiles by the input key for the purpose of protecting the embedded data. After generating the sequence CM, for each adjacent tile pair $Tile_i$ and $Tile_{i+1}$, we embed B_i by modifying the orientation of $Tile_{i+1}$, i.e., by enlarging or reducing the magnitude of θ_{i+1} , based on the corresponding CM_i as described in the last section.

According to the above way of data embedding, assume that there are m tiles in a tile mosaic image, then only $m - 1$ bits can be embedded. And that is why the sequence of CM is of the size of $m - 1$. So, if the embedding capacity, $m - 1$, is smaller than the length of the bit sequence, n, some bits will be discarded due to the insufficiency of tiles to embed data. On the other hand, if $m - 1$ is larger than n, we will embed the bit sequence repeatedly for additional robustness.

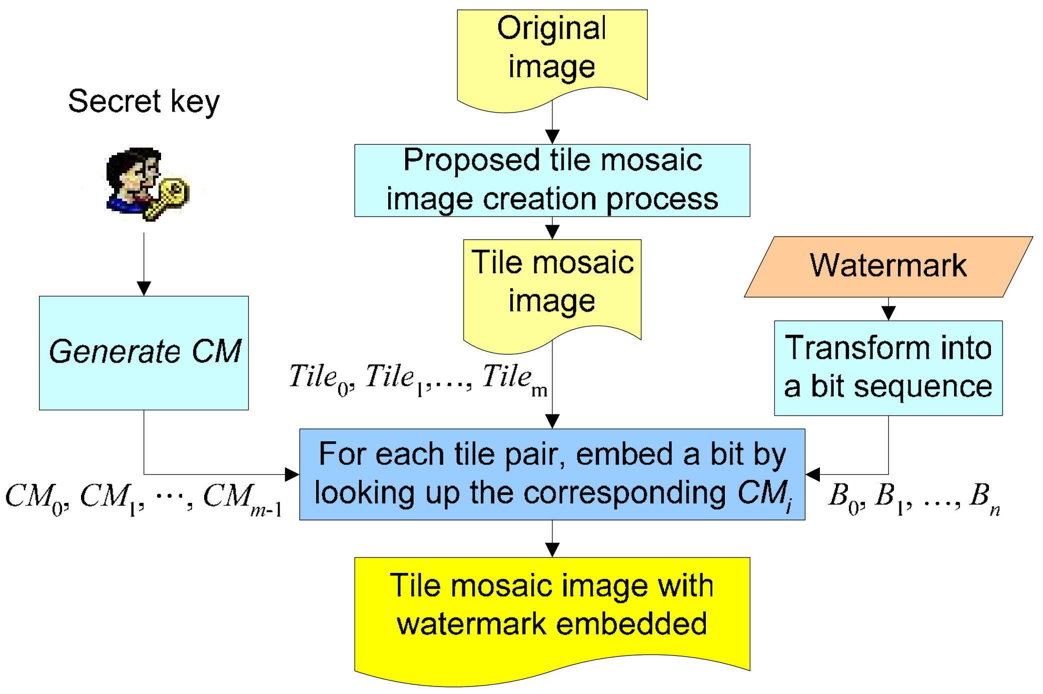

Original image
Proposed tile mosaic image creation process
Tile mosaic image
Secret key
Watermark
Transform into a bit sequence
Generate CM
$Tile_0, Tile_1, \ldots, Tile_m$
For each tile pair, embed a bit by looking up the corresponding CM_i
$CM_0, CM_1, \cdots, CM_{m-1}$
$B_0, B_1, \ldots, B_n$
Tile mosaic image with watermark embedded

Fig. 3.13 Watermark embedding process.

3.3.3 Data Extraction Process

The watermark extraction process is simply an inverse version of the embedding process. A flowchart of the watermark extraction process is shown in Fig. 3.14. First, we derive the tile orientations, $\theta_0, \theta_1, \ldots, \theta_m$, of the tiles by the tile feature detection process, as mentioned in Section 3, and generate the sequence *CM* of code mappings by an input key. Then, by utilizing *CM* and the tile orientations, we extract the embedded watermark. In more details, for each adjacent tile pair $Tile_i$ and $Tile_{i+1}$ with orientations θ_i and θ_{i+1}, we create a right angle which centers on θ_i and find the sub-sector where θ_{i+1} falls. In this way we can get the corresponding bit code B_i by looking up the corresponding code mapping CM_i in *CM*. If the extracted bit sequence is larger than 25×25, we apply a voting strategy to recover the watermark with more robustness.

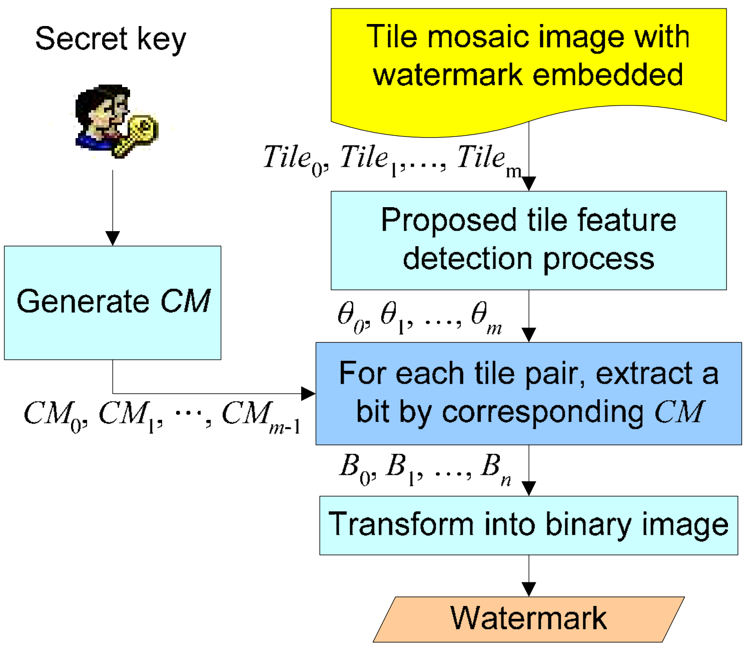
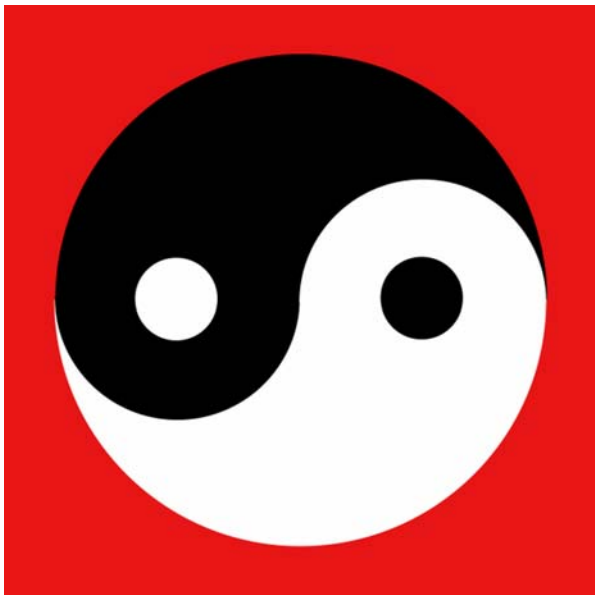
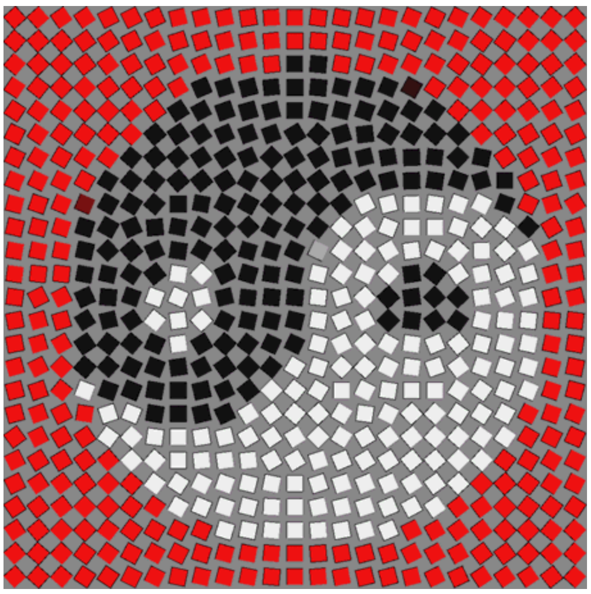

Secret key
Tile mosaic image with watermark embedded
$Tile_0$, $Tile_1$,…, $Tile_m$
Proposed tile feature detection process
Generate *CM*
θ_0, θ_1, …, θ_m
For each tile pair, extract a bit by corresponding *CM*
CM_0, CM_1, ⋯, CM_{m-1}
B_0, B_1, …, B_n
Transform into binary image
Watermark

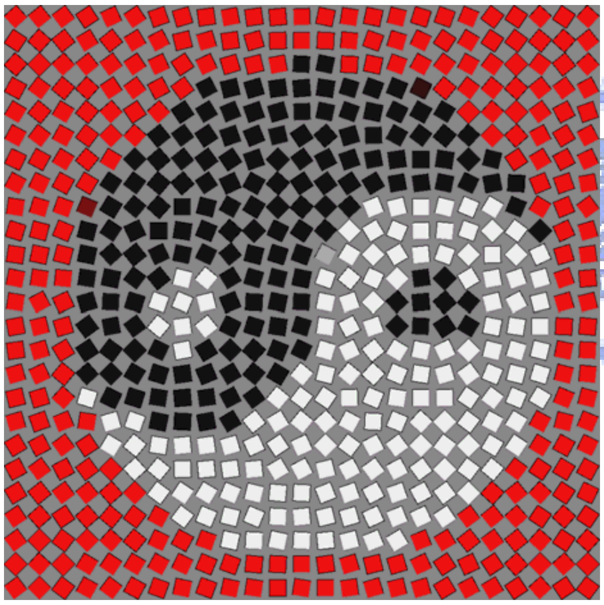
Fig. 3.14 Watermark extraction process.

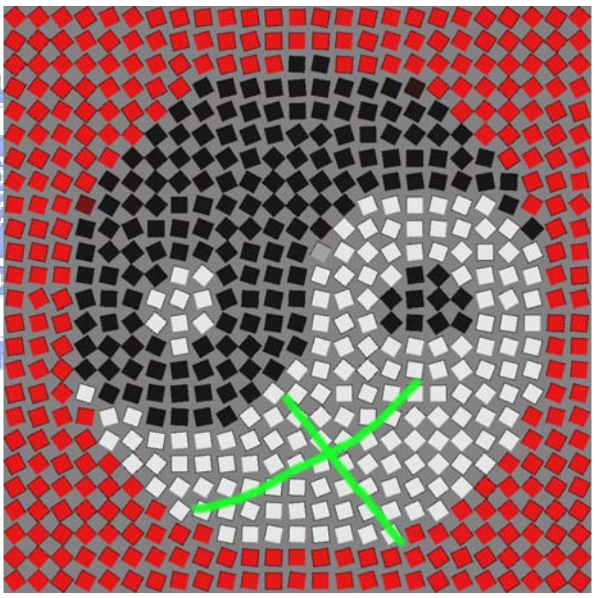
3.3.4 Experimental Results

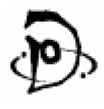
The proposed method was tested on a series of images. Some experimental results are shown in this section. In Fig. 3.15, (a) and (e) are an input image and a watermark, respectively, (b) is a tile mosaic image of (a) without the watermark embedded, and (c) is a tile mosaic image of (a) with the watermark (e) embedded. Because the tiles of (c) have been rotated slightly for embedding the watermark (e), (c) is a little bit more distorted than (b), but still acceptable. If we hide more bits in each tile, the generated tile mosaic image will be more distorted than (c). Fig. 3.15(f) is the watermark extracted from (c). Because we resize the watermark before embedding it, Fig. 3.15(f) is a scaled-down version of Fig. 3.15(e). Fig. 3.15(g) is the watermark extracted from Fig. 3.15(c) with a wrong key. From Fig. 3.15(g), we can see that the watermark cannot be extracted with a wrong key. Fig. 3.15(d) is a damaged image of Fig. 3.15(c) with a mark on it, and Fig. 3.15(h) is the extracted watermark from Fig. 3.15(d). From Fig. 3.15(d) and Fig. 3.15(h), we can see that the watermark can still be recognized even subject to a certain degree of damaging.

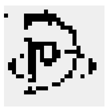

(a)

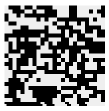

(b)


(c)


(d)


(e)


(f)


(g)

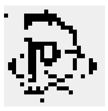

(h)

Fig. 3.15 Experimental results. (a) A Taiji image; (b) a tile mosaic image of Taiji without watermark embedded; (c) a tile mosaic image with watermark (e) embedded; (d) a damaged image of (c), (e) an input watermark; (f) a watermark extracted from (c); (g) a watermark extracted from (c) with a wrong key; (h) a watermark extracted from (d).

3.4 Proposed Secret Hiding Method by Tile Size Modification

In this section, we will propose a method for hiding a secret message into a tile mosaic image by tile size modification. In order to avoid tile overlapping, the way we modify the tile sizes is shrinking them instead of magnifying them. By shrinking tiles, we can hide bits into the tiles. Since the degree of tile shrinking is slight, we can not tell the size differences among tiles by naked eyes. Furthermore, we utilize the number of tile pixels, *TPN*, instead of the tile sizes. By utilizing *TPN*, we can detect tile sizes with better accuracy during data extraction.

3.4.1 Core Concepts

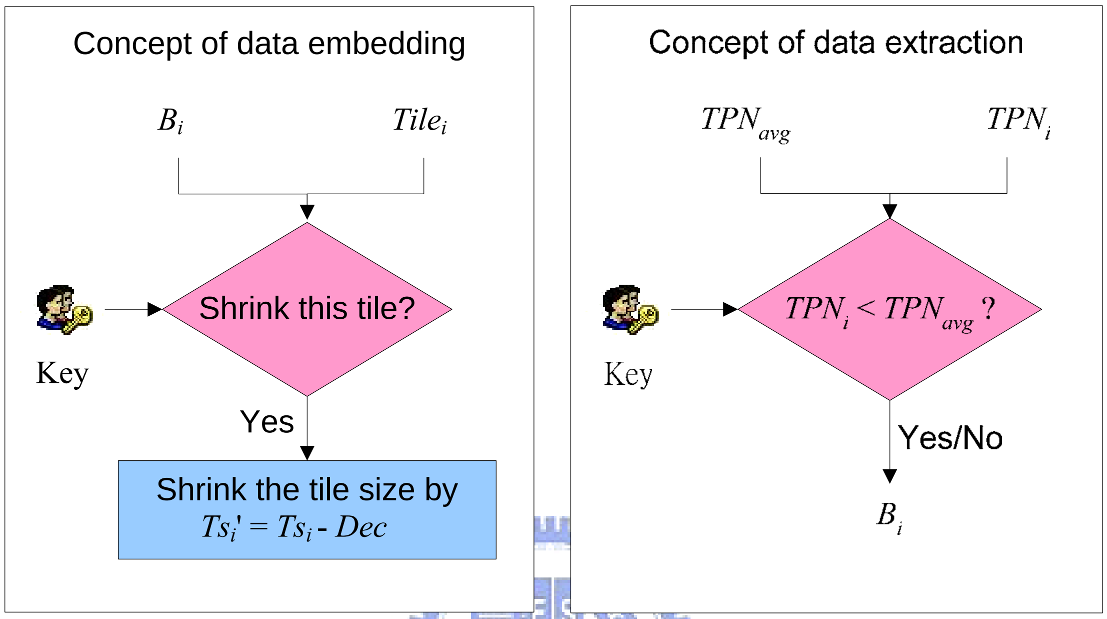
The core concept of data hiding by tile size modification is shown in Fig. 3.16. Assume that we want to embed a bit code, B_i, into $Tile_i$. As shown in Fig. 3.16(a), by checking the input key, we can know whether we need to shrink this tile or not. If the answer is 'Yes', we will shrink the tile size, Ts_i, of $Tile_i$ by the formula, $Ts_i' = Ts_i - Dec$ where the parameter, *Dec*, is a predefined value standing for the degree of tile shrinking.

In the data extracting process as shown in Fig. 3.16(b), for each $Tile_i$, we will check whether TPN_i is smaller than TPN_{avg} or not. By the checking result, we can recognize whether this tile has been shrunk or not. If TPN_i is smaller than TPN_{avg}, we will view $Tile_i$ as a shrunk tile. Otherwise, we will view $Tile_i$ as a non-shrunk tile. Finally, we can get the embedded bit code, B_i, by the checking result and the input key

In order to keep high detection accuracy, we have to adjust the value of *DEC* while the value of *Ts* varies. The relation between *DEC* and *Ts* is shown as follows:

$$Dec = \left\lceil \frac{Ts}{30} \right\rceil$$

By varying the value of *Dec* by the previous formula, detection results without bit errors can be yielded if the value of *Ts* is larger than or equal to eight.



(a) (b)

Fig. 3.16 Data hiding by tile size modification. (a) Concept of data embedding. (b) Data of data extraction.

3.4.2 Data Embedding Process

The procedure of secret hiding in a tile mosaic image is shown in Fig. 3.17 and described in Algorithm 3.6 below.

Algorithm 3.6: Secret hiding in a tile mosaic image.

Input: a tile mosaic image *M*, a secret message *Mes*, tile size shrinking degree *Dec*, and a key *K*.

Output: a tile mosaic image with secret message embedded.

Steps:

Step 1 Link the tiles in *M* into a 1-D sequence $Tile_0$, $Tile_1$, …, $Tile_m$ by a raster scan of the tiles.

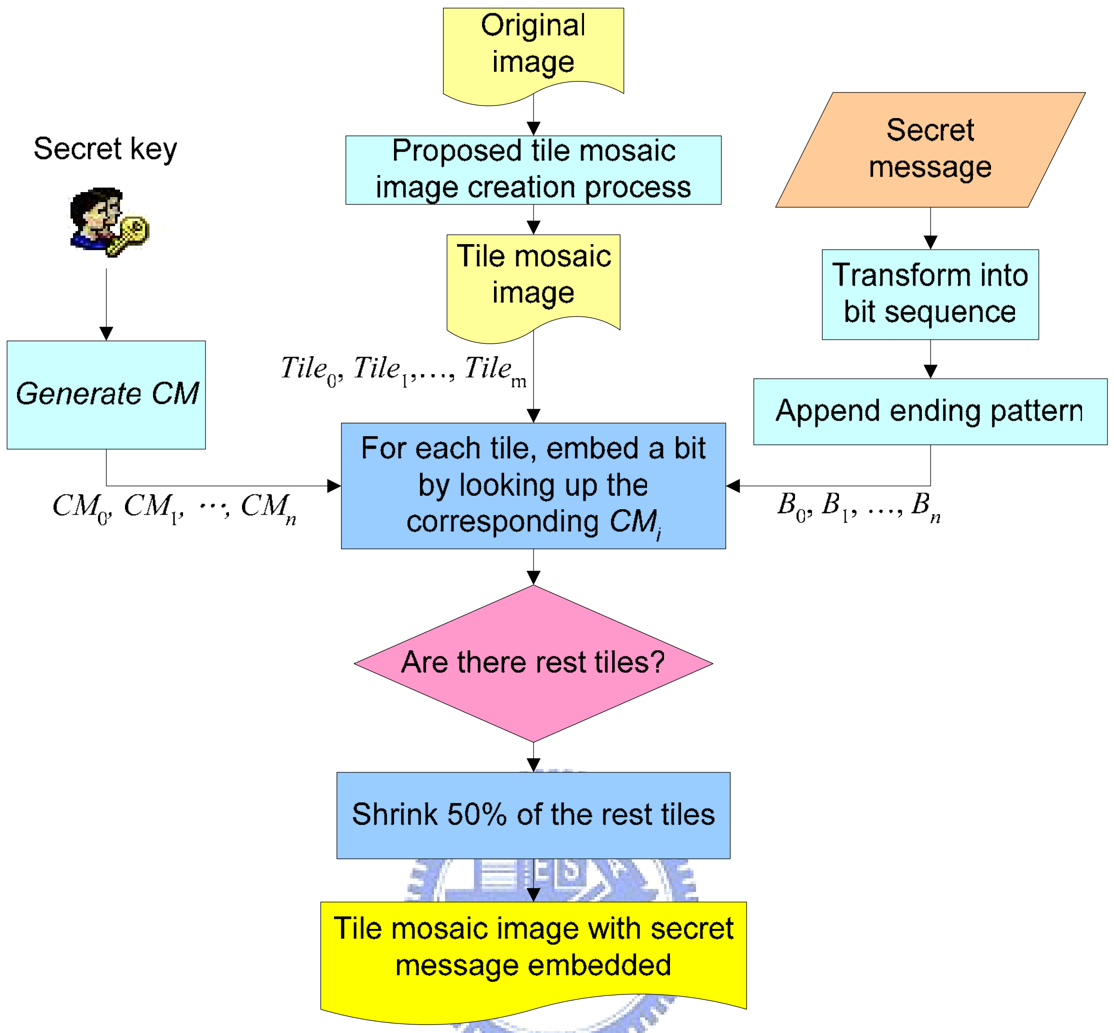
Step 2 Transform *Mes* into a bit sequence $B = B_0, B_1, \ldots, B_n$, with ending pattern appended in the rear of *B*.

Step 3 Generate a sequence of code mappings, $CM = CM_0, CM_1, \ldots, CM_n$, by the input key *K*.

Step 4 For each B_i, decide whether to shrink $Tile_i$ or not by checking CM_i. If so, shrink $Tile_i$ by the formula: $Ts_i' = Ts_i - Dec$.

Step 5 If *m* is larger than *n*, shrink half of the rest tiles (i.e., $Tile_i$, where $n+1 \leqq i \leqq 0$) by the same formula in Step 4.

Initially, we transform the secret message into a bit sequence, *B*, and append an ending pattern (sixteen successive 0s) at the end of *B*, that is, $B_i = 0$ if $n-16<i\leqq n$. By the ending pattern, we can determine where the message ends in a sequence of extracted bits in the detection process. Furthermore, in Step 2, we check the capacity (including the ending pattern) of *M* and truncate the data out of the capacity in *Mes*. So, the size of *B* is not greater than the size of the tile sequence after appending the ending pattern. In Step 3, as what we do in Algorithm 3.5, we generate a sequence of code mapping, *CM*. In this application, a code map, CM_i, is used for deciding whether to shrink $Tile_i$ or not when embedding B_i into $Tile_i$. In Step 4, we then embed B_i in $Tile_i$ by checking CM_i. Finally, in order to keep the value of *TPNavg* to be $\left(Ts - \frac{Dec}{2}\right)^2$ approximately, so that, a better detection accuracy can be achieved, we need to shrink half of the rest tiles by the formula used in Step 4.

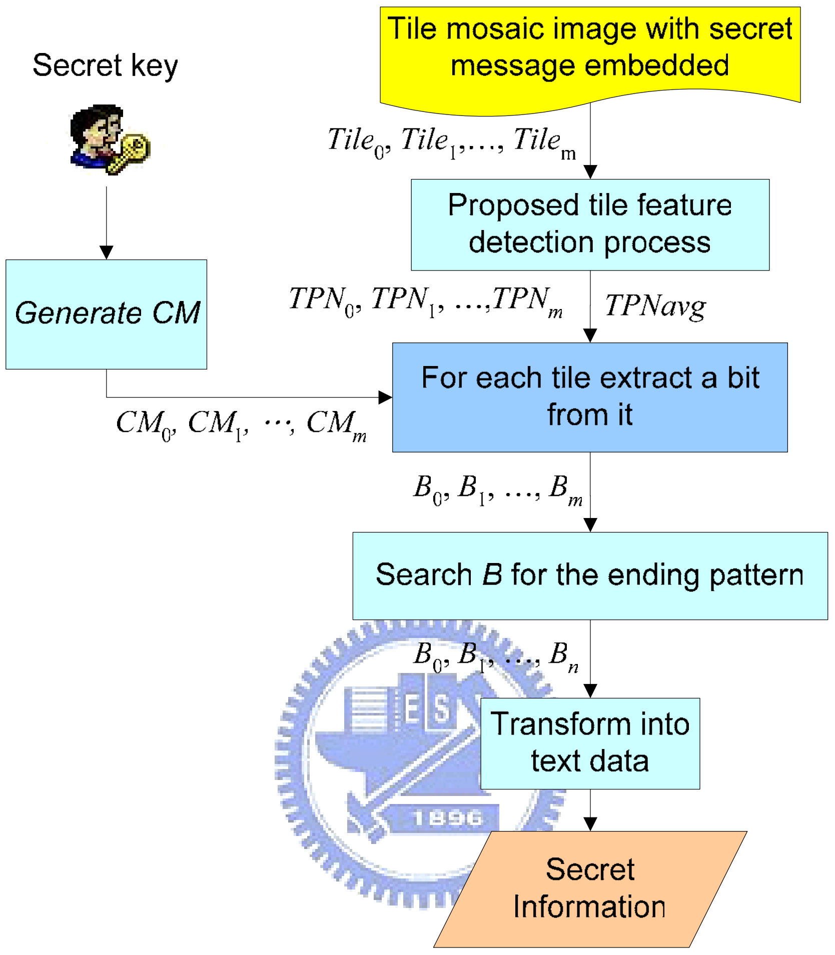


Original image
Secret key
Proposed tile mosaic image creation process
Secret message
Tile mosaic image
Transform into bit sequence
Generate CM
$Tile_0$, $Tile_1$,…, $Tile_m$
Append ending pattern
For each tile, embed a bit by looking up the corresponding CM_i
CM_0, CM_1, ⋯, CM_n
B_0, B_1, …, B_n
Are there rest tiles?
Shrink 50% of the rest tiles
Tile mosaic image with secret message embedded

Fig. 3.17 Secret message embedding process

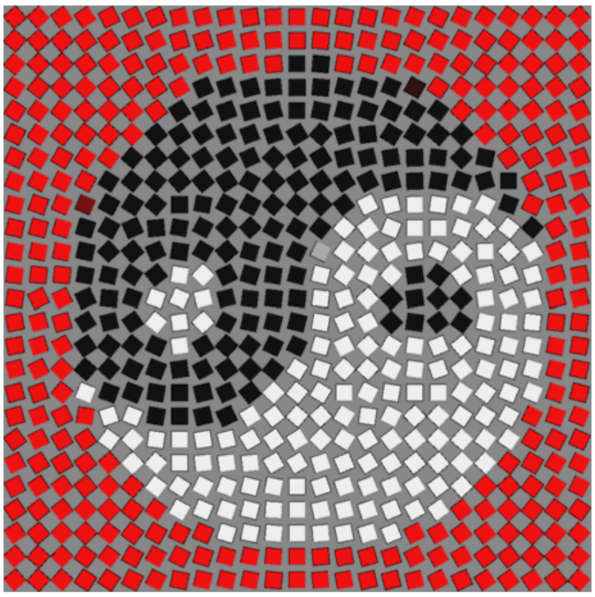
3.4.3 Data Extraction Process

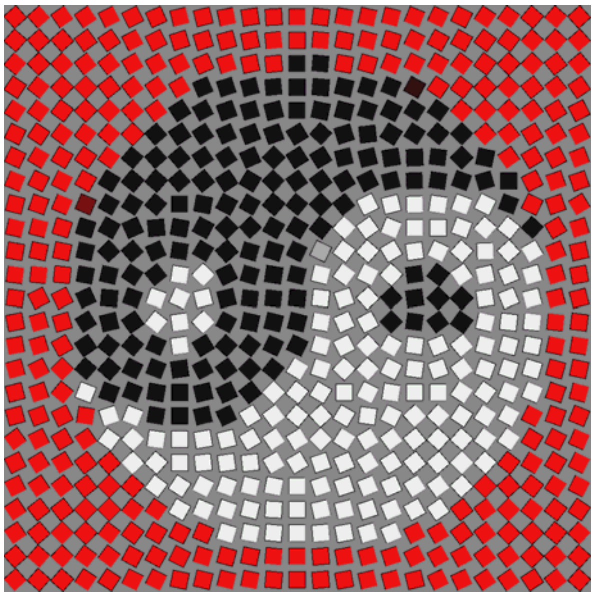
The secret message extraction process is also an inverse of the embedding process. A flowchart of the secret message extraction process is shown in Fig. 3.18. First, we derive the tile pixel numbers, TPN_0, TPN_1, …, TPN_m, of the tiles and their average ,*TPNavg* , by the tile feature detection process, and generate the *CM* sequence by an input key. Then, for each TPN_i, we extract a bit B_i according to CM_i, by checking whether TPN_i is larger than*TPNavg* or not. In the following step, we search *B* for the ending pattern (16 successive 0s) and truncate the redundant bits at the rear of B. Finally, we transform the bit sequence into text format, and then the embedded secret message is thus extracted.

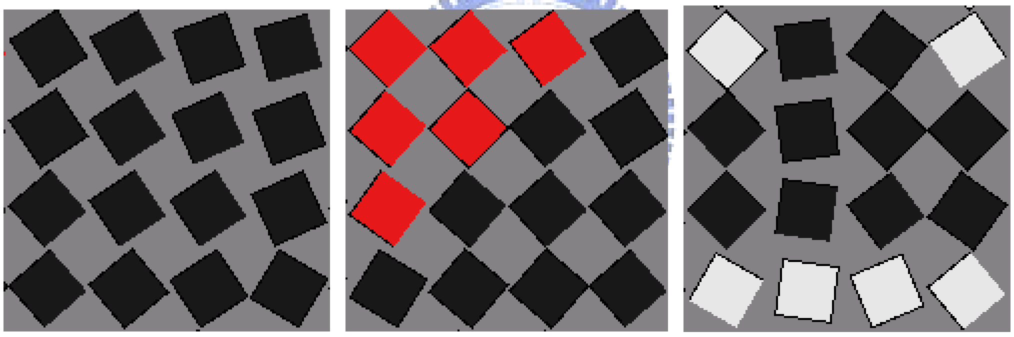

Tile mosaic image with secret message embedded
Secret key
$Tile_0, Tile_1, \ldots, Tile_m$
Proposed tile feature detection process
Generate CM
$TPN_0, TPN_1, \ldots, TPN_m$
$TPNavg$
For each tile extract a bit from it
$CM_0, CM_1, \cdots, CM_m$
$B_0, B_1, \ldots, B_m$
Search B for the ending pattern
$B_0, B_1, \ldots, B_n$
Transform into text data
Secret Information
Fig. 3.18 Secret message extraction process

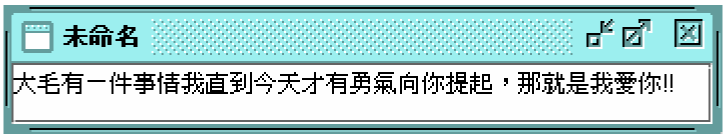
3.4.4 Experimental Results

Some experimental results are shown in Fig. 3.19. Fig. 3.19(a) is a tile mosaic image of TaiJi without secret information embedded. Fig. 3.19(b) is a tile mosaic image with the secret message, "大毛有一件事情我直到今天才有勇氣向你提起,那就是我愛你!!," embedded. Fig. 3.19(c) presents some details in (b). We can not figure out exactly which tiles have been shrunk, because the degree we shrank a tile is only one bit wide in Fig. 3.19(b). Fig. 3.19(d) is the secret information extracted from (b). We can find out that there is no error bits within it even though we apply no data recovery technique on the extracted data. Fig. 3.19(e) is the secret information extracted from (b) with a wrong key. So, Fig. 3.19(e) shows the secret information is protected by the key properly.


(a)

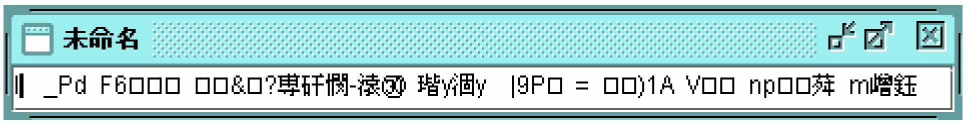

(b)


(c)



大毛有一件事情我直到今天才有勇氣向你提起，那就是我愛你!!

(d)



_Pd F6□□□ □□&□?専矸憪-禯㉚ 揩y僴y |9P□ = □□)1A V□□ np□□莾 m嶒鈺

(e)

Fig. 3.19 Experimental results. (a) A tile mosaic image without secret message embedded. (b) a tile mosaic image with secret message embedded, (c) Some details of (b), (d) The secret message extracted from (b), (e) The secret message extracted from (b) with a wrong key.

3.5 Proposed Authentication Method by Tile Texture Modification

In this section we will propose a tile mosaic image authentication method by tile texture modification. We will propose first a hash function which transforms the tile features of a tile into three binary bits. We then hide the three bits into the RGB channels by adding noise (modifying tile textures of each RGB channels) into a tile. The reason for embedding authentication signals into tile textures is due to the high capacity contained in the tile textures. For each tile, the capacity of tile texture is three bits, however, the capacities of tile orientation and tile size are only one bit. Although we can embed more than one bit into tile orientations and tile sizes, it will cause more loss of the details of the original image. In the case of using three-bit authentication signals, there is a probability of 1/8 that we may miss to find out a tampered tile that has been modified. In the case of using one-bit authentication signal, the probability is $\frac{1}{2}$ which makes the authentication process impractical. So, in order to make the authentication process more practical, tile textures are chosen as the host feature for hiding the authentication signals.

3.5.1 Core Concepts

As we mentioned in Section 3.1.3, in this study we define tile texture as noise pixels in tiles. By adding noise pixels in the R, G, and B channels we can achieve the goal of data hiding. In order to have high detection accuracy in the tile texture detection process, we embed only one bit into each of the R, G, and B channels. The core concept of data hiding in tile texture is similar to the concepts of data hiding in tile orientation, as shown in Fig. 3.20.

The range of noise ratio, Nr, is limited to be within 0 to 0.5, that is, $0 \le Nr \le 0.5$. The reason for confining noise ratios to be under 0.5 has been discussed in Section 3.1.3. In order to verify whether the authentication signals have been embedded or not, we retain $0 \le Nr \le 0.05$ as the *rate base*. If the value of Nr of a tile is under the condition $0 \le Nr \le 0.05$, we will determine that no authentication signal is embedded in the tile. So, the effective rate range of Nr is $0.05 \le Nr \le 0.5$ and will be divided into two sub-ranges, one representing code 0 and the other code 1.

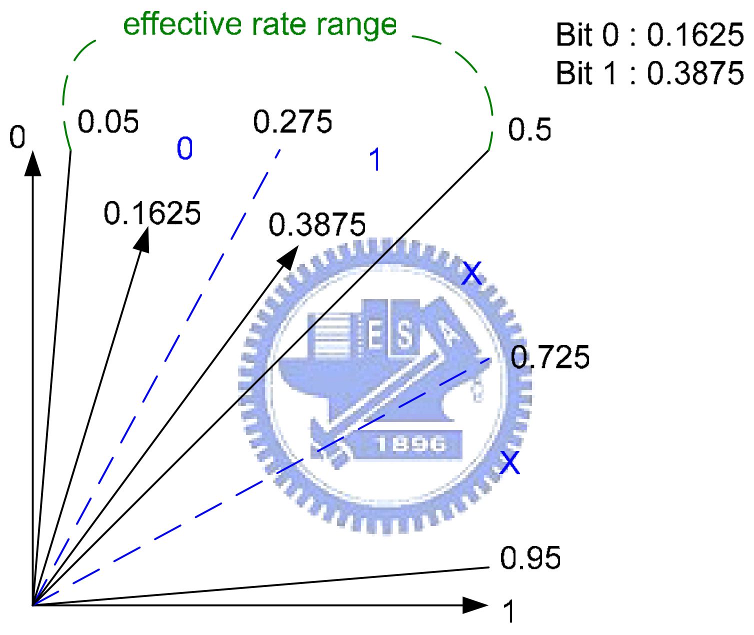


Fig. 3.20 Data hiding by tile texture modification

As shown in Fig. 3.20, for each color channel of a tile, if we want to embed code 0 we will add noise with ratio 0.1625. On the other hand, if we want to embed code 1 the noise ratio will be 0.3875.

Before we embed bit codes into the tile textures, we have to transform the tile features into a three-bit code for each single tile. And then, we embed each bit of the code into one of the color channel. We propose hash functions to do that. The hash functions and the associated parameters are defined as follows. Let the tile color be denoted as *Tc*, the tile size as *Ts*, and the tile orientation as *To*. Denote the number of sectors defined as Fig. 3.12(c) as *BN*, the number of sub-sectors defined as Fig. 3.12(c) as *RN*. Define three terms as follows:

$$RGBhash = (Tc.red \times Tc.green \times Tc.blue) \bmod 1013 \ldots\ldots(1)$$

$$Sizehash = \text{The bit code embedded by tile size modification} \ldots\ldots(2)$$

$$ORIhash = \frac{To}{AQL}, \text{ where } AQL = \frac{2 \times BN \times RN}{2} \ldots\ldots(3)$$

Then the hash function is defined as follows:

$$Hash = ((RGBhash+1) \times (Sizehash+1) \times (ORIhash+1) \times Random(key)) \bmod 2^3 \ldots\ldots(4)$$

There are three tile features that we are going to authenticate. They are tile color, tile size and tile orientation which we denote by *Tc*, *Ts* and *To*, respectively. First, we compute the hash value of *Tc* by taking a product of the values of the R, G, and B channels of *Tc*, as shown in Formula (1). Then we compute the remainder of dividing the product by 1013. 1013 is an arbitrarily chosen prime number. We denote the hashing value of *Tc* by *RGBhash*.

Second, the hash value of *Ts* is simply the bit code embedded by the proposed tile size modification method. (As mentioned above, we hide secret messages by tile size modification) If no secret message is embedded by a user, we will embed a text, “no data embedded,” to indicate that there is no data embedded in this tile mosaic image, and then shrink 50% of the rest tiles as what we do in Section 3.4.2. So, no matter the user has embedded a secret message or not, the tile size, *Ts*, of each tile will represent a corresponding bit code. The bit code may be part of a text embedded by a user or part of the text “no data embedded”. Even though the tile is shrunk as the rest tiles in the proposed secret hiding process, the size of this tile will also represent a corresponding bit code by utilizing a secret key. In this condition, the corresponding bit codes are composed of arbitrary bits. The hash value of *Ts* is simply the corresponding bit code. We denote the hashing value of *Ts* by *Sizehash*.

The way we hash a tile orientation is to divide the effective range of tile orientations (i.e., from 0^o through 90^o) into several sectors. Unlike what we do in watermarking, we *do not* divide a sector into sub-sectors. The hash value of a tile orientation, *To*, is the serial number of the *sector* where *To* falls. As we mentioned in Section 3.3, there may be some bit errors when detecting tile orientations. In watermarking, we can ignore the salt-and-paper noise caused by the detection errors. But in the application of authentication, we do not want any authentication error to take place. So, the number of sectors is half of the number of sub-sectors we used in watermarking. In other words, the span of the sector we use in authentication is a double of the span of the sub-sectors in watermarking. By enlarging the span of each sector, we can eliminate the errors that may occur in the detection process. The number of sub-sectors used in watermarking is $2 \times BN \times RN$. Thus, the number of sectors used in authentication is $\frac{2 \times BN \times RN}{2}$ and will be denoted by *AQL*. Finally, the hashing value of tile orientation which we denote by *ORIhash* is $\frac{To}{AQL}$, as shown in Formula (3).

In the final step, we transform the three hashing values into a three-bit code by Formula (4). As shown in (4), we apply the modulus operation with the modulo of 2^3. The hashing value will thus be shrunk into three bits by the modulus operation. Then we embed the three bits by adding noise into the R, G and B channels in a tile. As an example, if we want to embed a code 0 into the R channel of a tile, we add noise with rate 0.1625 into the R channel as shown in Fig. 3.20. Unlike the proposed watermarking method, we do not generate *code maps* with a secret key to indicate what code a sub-range represents. In watermarking, a secret key is used for generating the *code map*s. In authentication, we apply a secret key in generating bit codes by the hash functions (i.e., by Formula (4)). In Formula (4), we use a random number generator with a secret key (i.e., *Random*(*key*)) as a seed, for protecting the embedded authentication signals. The function, *Random*(*key*), will generate two totally different sequences of numbers, if these sequence are generated by two different key values. The authentication signal embedding and extraction processes will be described in the following two sections. The *code map* we used in authentication is defined in Fig. 3.20 and will not be changed.

3.5.2 Authentication Signal Embedding Process

The procedure of embedding authentication signals in a tile mosaic image is shown in Fig. 3.21 and described in Algorithm 3.7 below. After linking the tiles into an 1-D sequence, we transform the three tile features into a sequence of bit codes $B = B_0, B_1, \ldots, B_m$. Each bit code, B_i, contains three bits, namely, $B_i = B_{i2}\ B_{i1}\ B_{i0}$. In Step 3, for each B_{ij}, we look up the code map defined in Fig. 3.20 for the corresponding noise ratio R_{ij}. Finally, we add noise into the R, G and B channels by the corresponding noise ratios. For example, if we want to embed noise pixels in the R channel of a tile, $Tile_i$, with a noise ratio, R_{i0}, first, we pick pixels randomly within $Tile_i$ as noise pixels with a picking ratio of R_{i0}. We then increase or decrease the intensities of all the picked pixels in the R channel. We *randomly* make the decision of generating noise by increasing or decreasing the intensities of the picked pixels. At the end, the authentication signals are embedded into the tile texture of $Tile_i$.

Algorithm 3.7: Authentication signals generation and embedding process.

Input: a tile mosaic image *M*, and a key *K*.

Output: a tile mosaic image with authentication signals embedded.

Steps:

Step 1 Link the tiles in M into a 1-D sequence $Tile_0$, $Tile_1$, …, $Tile_m$ by a raster scan of the tiles.

Step 2 Generate a sequence of authentication signals $B = B_0, B_1, \ldots, B_m$, by the hashing functions proposed in the last section. Each B_i in B contains three bits, which is denoted by: $B_i = B_{i2}\, B_{i1}\, B_{i0}$.

Step 3 Generate a sequence of sets of noise ratios $R = R_0, R_1, \ldots, R_m$, by looking up the code map defined in Fig. 3.20. Each R_i in R contains three different ratios, which is denoted by: $R_i = \{R_{i2}, R_{i1}, R_{i0}\}$.

Step 4 For each $Tile_i$ embed B_i into M by the following steps.

4.1 Embed B_{i0} into $Tile_i$ by adding noise into the R channel with noise ratio R_{i0}.

4.2 Embed B_{i1} into $Tile_i$ by adding noise into the G channel with noise ratio R_{i1}.

4.3 Embed B_{i2} into $Tile_i$ by adding noise into the B channel with noise ratio R_{i2}.

3.5.3 Authentication Signal Extraction Process

The proposed authentication method is shown in Fig. 3.22. After the proposed tile feature detection process is completed, we can derive the tile map (i.e., Mp_i) and the three tile features Tc, Ts and To of each tile. We can also detect whether the tile is a rectangle or not by the average tile size detection process discussed in Section 3.2.4.C. If the tile is not a rectangle, it indicates that the tile has been damaged, and the verification process of this tile is terminated. On the other hand, if the tile is a rectangle, we apply further a verification process on this tile. The further verification process is described as follows.

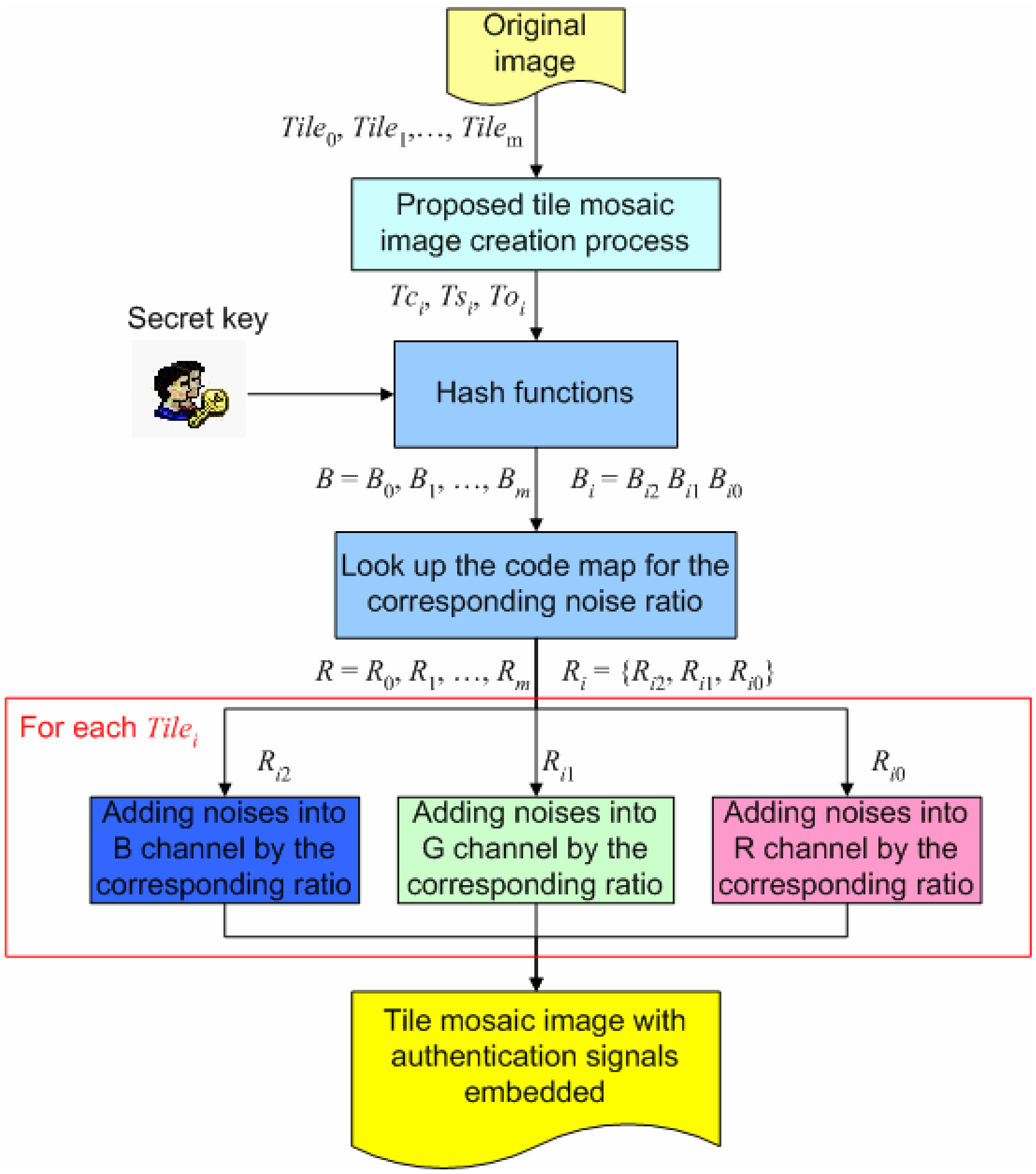
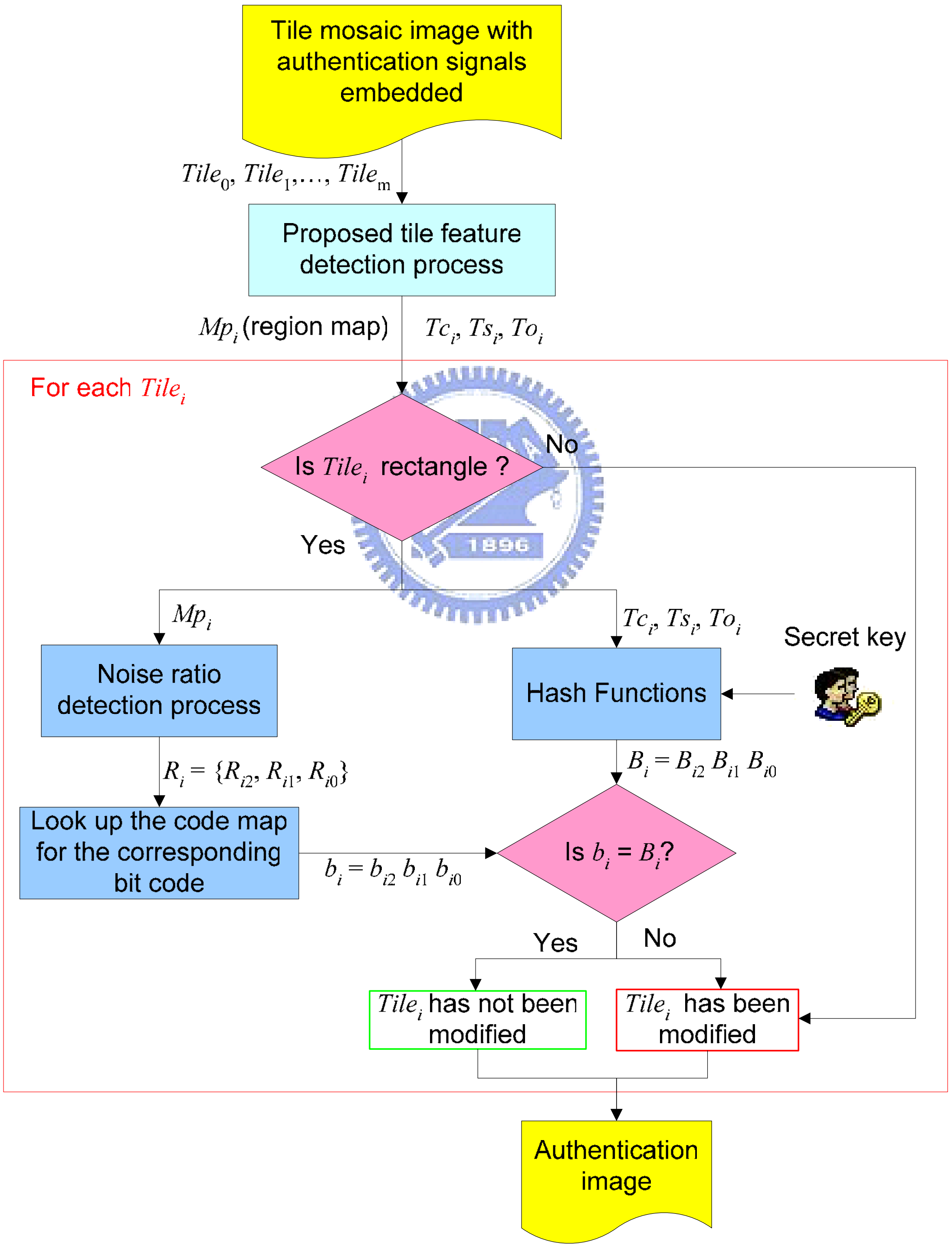

Original image
$Tile_0, Tile_1, \ldots, Tile_m$
Proposed tile mosaic image creation process
Tc_i, Ts_i, To_i
Secret key
Hash functions
$B = B_0, B_1, \ldots, B_m$
$B_i = B_{i2}\, B_{i1}\, B_{i0}$
Look up the code map for the corresponding noise ratio
$R = R_0, R_1, \ldots, R_m$
$R_i = \{R_{i2}, R_{i1}, R_{i0}\}$
For each $Tile_i$
R_{i2}
R_{i1}
R_{i0}
Adding noises into B channel by the corresponding ratio
Adding noises into G channel by the corresponding ratio
Adding noises into R channel by the corresponding ratio
Tile mosaic image with authentication signals embedded

Fig. 3.21 Authentication signal embedding process

First, we get the hashing value, which we denote by B_i, by the hash functions listed in section 3.5.1. Second, for each R, G, or B channel of a tile, pixels have two different intensities (one is the original tile color and the other the noise color). We then accumulate the number of pixels of the two different intensity values and view the one having lower accumulation value as the noise color. The ratio of pixels of the noise color is what we call noise ratio. After deriving the three noise ratios in the R, G and B channel, we look up the code map as defined in Fig. 3.20 for the corresponding three bits. In the following, we compose the three bits into a code, b_i. Finally, we compare B_i and b_i. If they are of the same value, we claim that the tile has not been modified. If they are with different values, we instead claim that the tile has been modified. After applying the authentication process on all the tiles, we will have an authentication image of the input tile mosaic image as a result of authentication.



Tile mosaic image with authentication signals embedded
$Tile_0$, $Tile_1$,…, $Tile_m$
Proposed tile feature detection process
Mp_i (region map)
Tc_i, Ts_i, To_i
For each $Tile_i$
Is $Tile_i$ rectangle ?
No
Yes
Mp_i
Tc_i, Ts_i, To_i
Secret key
Noise ratio detection process
Hash Functions
$R_i = \{R_{i2}, R_{i1}, R_{i0}\}$
$B_i = B_{i2}\ B_{i1}\ B_{i0}$
Look up the code map for the corresponding bit code
$b_i = b_{i2}\ b_{i1}\ b_{i0}$
Is $b_i = B_i$?
Yes
No
$Tile_i$ has not been modified
$Tile_i$ has been modified
Authentication image

Fig. 3.22 Tile mosaic image verification process

3.5.4 Applications for Image Verification

As discussed in previous sections, we can verify the integrity of tile color, tile size, tile angle, and tile texture by the proposed tile mosaic image authentication system. In order to eliminate the errors caused by the tile feature detection process. We have to have more tolerance of errors. If the error is above the defined tolerance, we will claim that this tile has been modified. Because we embed only three bits into a tile texture, it is $\frac{1}{8}$ in probability that we may miss to detect a tile that has been damaged. But in most cases damaged tiles are not in the shape of rectangles. And we can always indicate a tile which is non-rectangular as being damaged. Some experimental result is given in Section 3.5.6.

3.5.5 Applications for Secret Verification

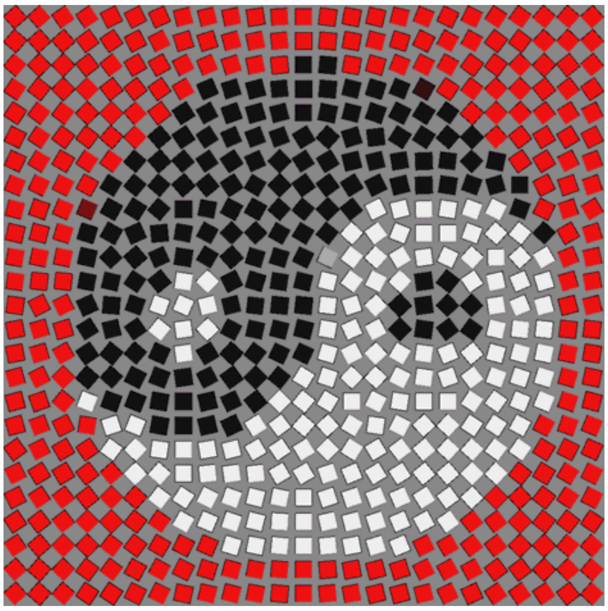
Because we can verify the integrity of tile features, we can also verify the fidelity of embedded data indirectly. An example is secret hiding in tile sizes. By verifying the integrity of tile sizes in a tile mosaic image, we can also verify the fidelity of embedded secret message. Although we can not indicate which word or character is damaged, we can verify the fidelity of the whole secret message. Some experimental result is given in Section 3.5.6.

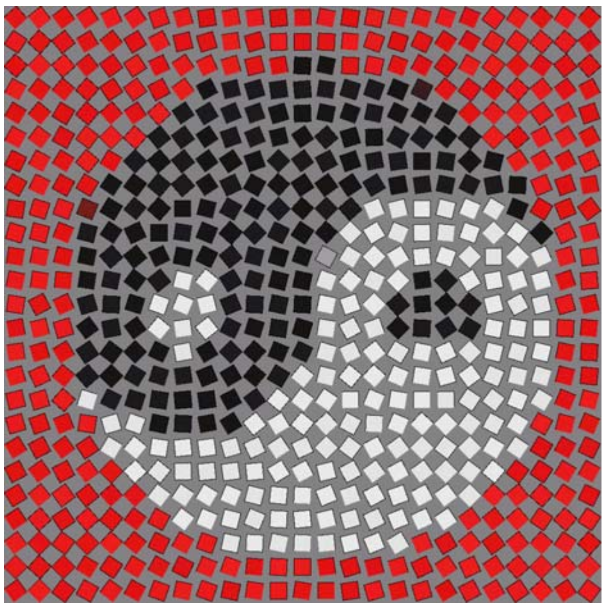
3.5.6 Experimental Results

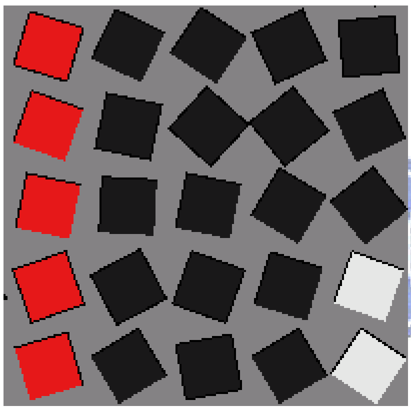
Some experimental results are shown in Fig. 3.23. In Fig. 3.23, (a) is a tile mosaic image of Taiji without data embedded and (b) is a tile mosaic image with a watermark, certain secret information, and authentication signals embedded. Fig. 3.23(c) and Fig. 3.23(d) are some details of Fig. 3.23(a) and (b), respectively. We can see that some noise is added to (d) and that the tile angles and tile sizes of (d) have slight difference from the ones of (c). Fig. 3.23(e) and (f) are the watermark and secret information extracted from Fig. 3.23(b). Fig. 3.23(g) is the verification result of Fig. 3.23(b). In Fig. 3.23(g), we can see that all the tiles are bounded by green squares which mean that the tiles are not modified. By Fig. 3.23(e), (f) and (g), we can see that a watermark, a secret message, and authentication signals can be embedded in a tile mosaic image simultaneously by utilizing the proposed methods. The experimental results also show that the different types of embedded data will not interfere with one another.

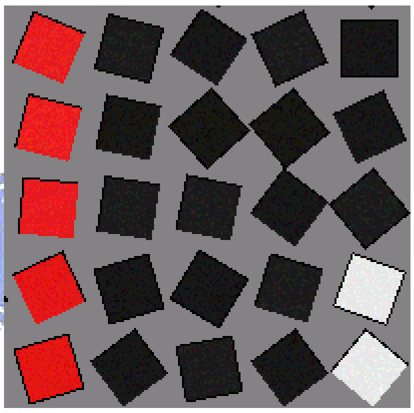
Fig. 3.23(h) is a copy of (b) with three groups of damaged tiles. We bound them with yellow squares. The tile in square 1 is magnified and the tile in square 2 is rotated. The four tiles in square 3 are copied from some other tiles. Furthermore, some blue strokes are painted at the bottom-right corner of Fig. 3.23(h). Fig. 3.23(i) is the authentication image of (h). The magenta quadrangles are the found damaged tiles. In Fig. 3.23(i), we can see that the damaged tiles are found by our authentication process correctly. In the bottom-right corner of (i), we can see that the magenta quadrangles are not rectangular. So, we claim that these tiles are damaged without any further verification process (i.e., without checking the embedded hash values). Fig. 3.23(j) is a watermark extracted from (h). Since (h) has been damaged, some salt-and-pepper noise appears on (j) but the watermark is still recognizable visually. Fig. 3.23(k) is the secret information extracted from (h). Since (h) is damaged, there are some arbitrary bits appear in (k).

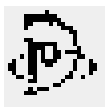
Because some damaged tiles are found by the proposed authentication process, the user can decide whether to believe the extracted secret message and watermark or not by the authentication image shown in Fig. 3.23(i).

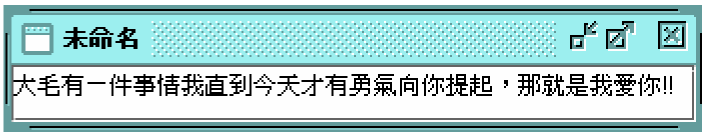

(a)


(b)


(c)


(d)

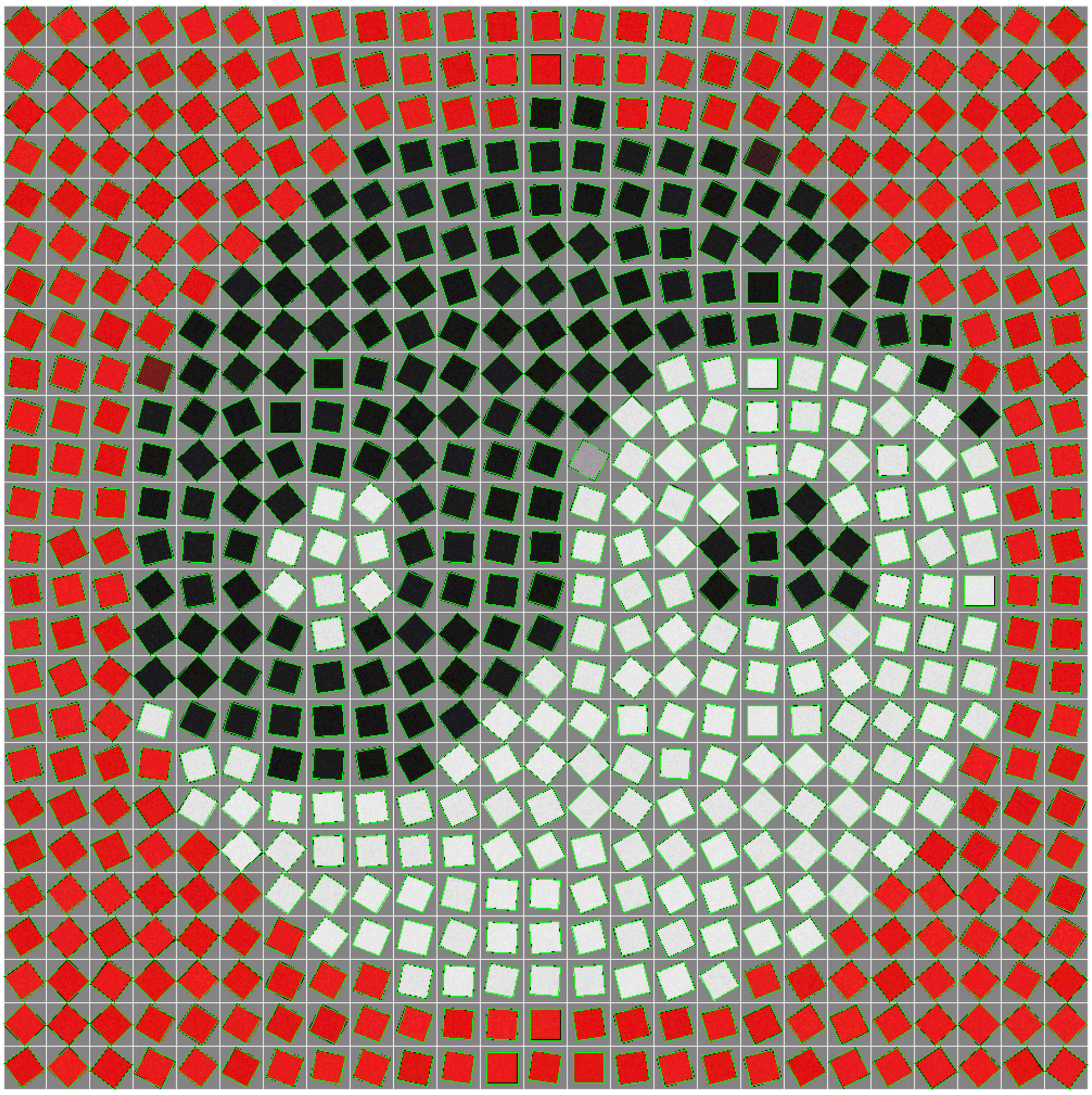

(e)


未命名

大毛有一件事情我直到今天才有勇氣向你提起，那就是我愛你!!

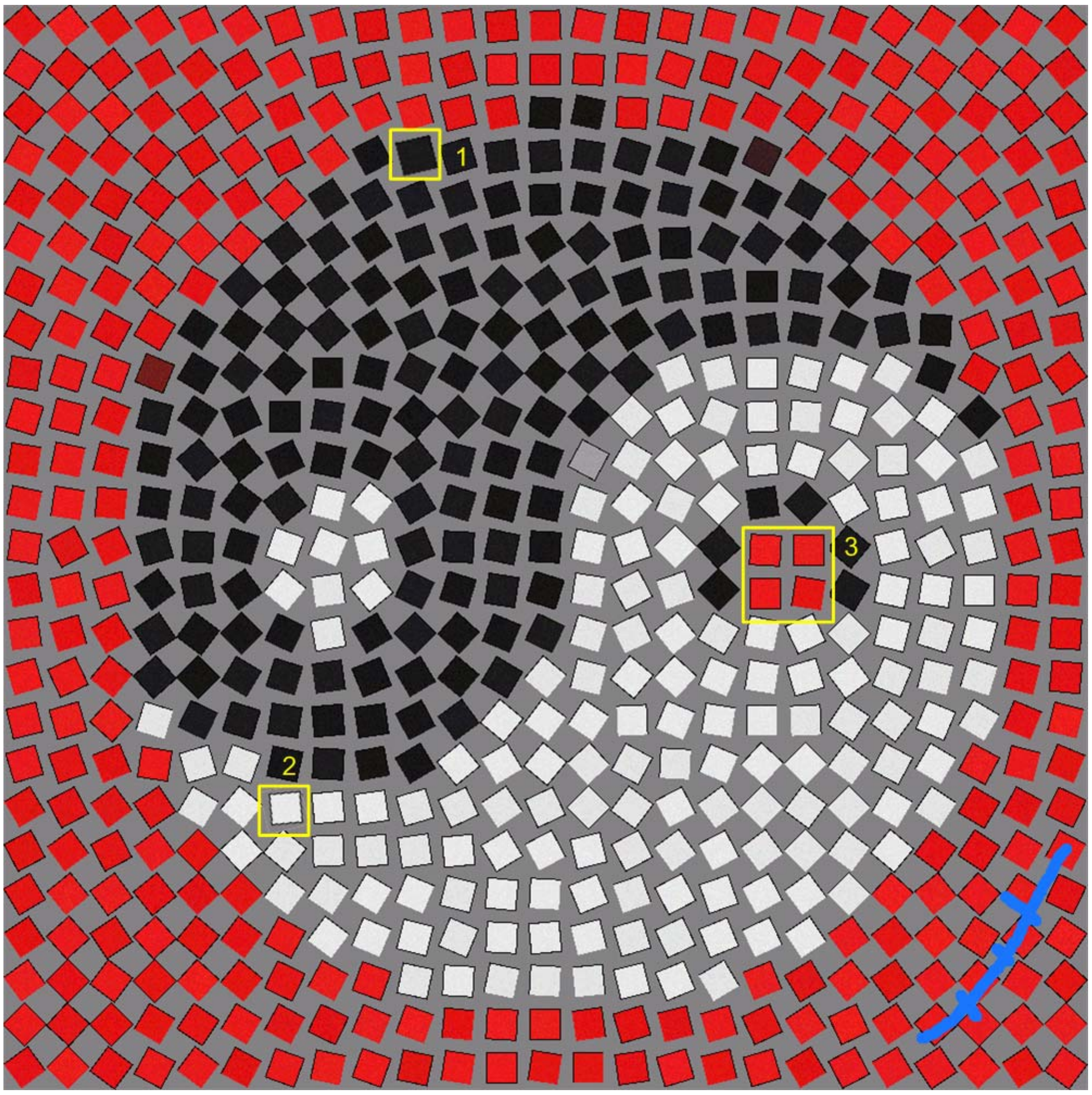
(f)

Fig. 3.23 Experimental results. (a) A tile mosaic image without data embedded. (b) A tile mosaic image with a watermark, a secret message and authentication signals embedded. (c) Some details of (a). (d) Some details of (b). (e) A watermark extracted from (b). (f) A secret message extracted from (b). (g) Verification result of (b). (h) A copy of (b) with some damaged tiles. (i) A verification result of (h). (j) A watermark extracted from (i). (k) A secret message extracted from (i).



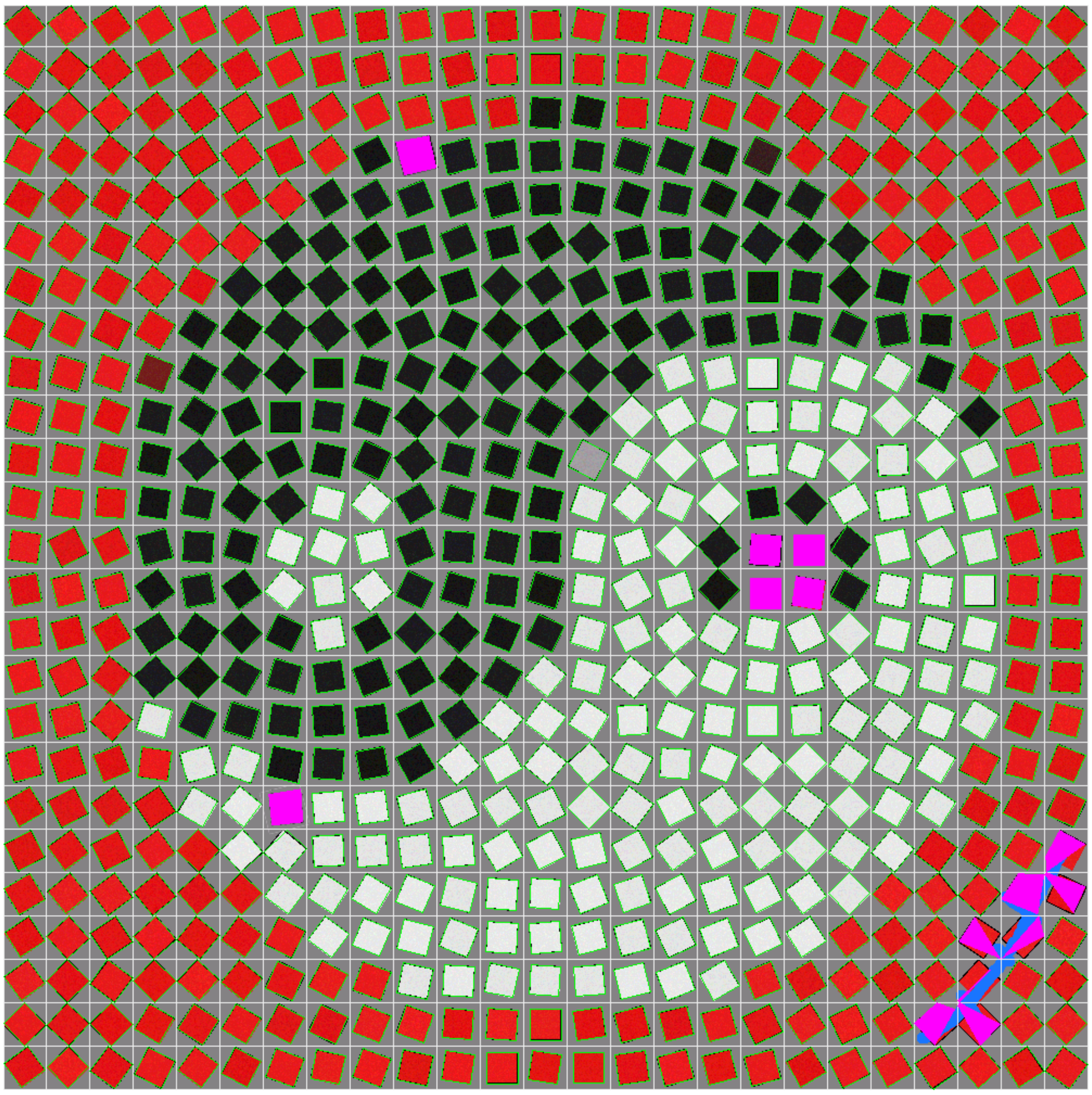
(g)

Fig. 3.23 Experimental results. (a) A tile mosaic image without data embedded. (b) A tile mosaic image with a watermark, a secret message and authentication signals embedded. (c) Some details of (a). (d) Some details of (b). (e) A watermark extracted from (b). (f) A secret message extracted from (b). (g) Verification result of (b). (h) A copy of (b) with some damaged tiles. (i) A verification result of (h). (j) A watermark extracted from (i). (k) A secret message extracted from (i) (continued).

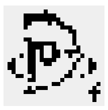


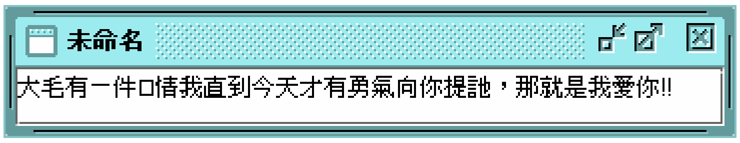
(h)

Fig. 3.23 Experimental results. (a) A tile mosaic image without data embedded. (b) A tile mosaic image with a watermark, a secret message and authentication signals embedded. (c) Some details of (a). (d) Some details of (b). (e) A watermark extracted from (b). (f) A secret message extracted from (b). (g) Verification result of (b). (h) A copy of (b) with some damaged tiles. (i) A verification result of (h). (j) A watermark extracted from (i). (k) A secret message extracted from (i) (continued).



(i)





大毛有一件□情我直到今天才有勇氣向你提訑，那就是我要你!!

(j) (k)

Fig. 3.23 Experimental results. (a) A tile mosaic image without data embedded. (b) A tile mosaic image with a watermark, a secret message and authentication signals embedded. (c) Some details of (a). (d) Some details of (b). (e) A watermark extracted from (b). (f) A secret message extracted from (b). (g) Verification result of (b). (h) A copy of (b) with some damaged tiles. (i) A verification result of (h). (j) A watermark extracted from (i). (k) A secret message extracted from (i) (continued).

Chapter 4
A Stained Glass Image Creation Method for Information Hiding

4.1 Introduction

In this chapter, we will investigate another format of mosaic-effect image which is named stained glass image. A stained glass image is composed of numerous glass regions which are divided by black and thin gaps, which are named *leading*. In Section 0, we will give a brief description of a traditional stained glass image creation process proposed by Mould [13]. In Section 4.3, a new approach to automatic creation of stained glass images from an input image for data hiding applications will be proposed. These two creation processes both start from image segmentation of an input image. However, the method proposed by Mould has no glass feature that can be used for data hiding. On the other hand, there will be a glass feature utilized for data hiding in the proposed method.

The proposed creation process is described as follows. First, we divide an input image into several color regions. Then we apply region growing techniques to glass regions. Each glass region will result in a glass piece in the final stained glass image. After creating the glass regions, we apply a data hiding technique by slight glass cracking to achieve the purpose of hiding data in the image. The details of data hiding and the corresponding applications will be discussed in Chapter 5.

4.2 Review of Traditional Stained Glass Image Creation Process

Before we propose the procedure of stained glass image creation, we will review first a traditional creation process proposed by Mould [13]. The process is briefly described in Algorithm 4.1 in the following.

Algorithm 4.1: a traditional tile mosaic image creation process.

Input: an original image *I*.

Output: a stained glass image *T*.

Steps:

Step 1 Segment *I* into several regions by the image processing system, EDISON, proposed in [14, 15, 16].

Step 2 Use erosion and dilation operators proposed in [17, 18] to manipulate and smooth the initial segmented regions (glass regions).

Step 3 Apply a displacement map representing imperfections in the glass regions and apply leading between tile boundaries.

Mould's method first segments the original image into several regions. As shown in Fig. 4.1, (a) is an initial image *Gretzky* and (b) is the segmented regions of (a) after applying Step 1. Fig. 4.1(c) is the region boundaries of Fig. 4.1(b). By applying Step 2, we can get a smoothed version of Fig. 4.1(c) as shown in (d). Fig. 4.1(e) is the final stained glass image by applying Step 3 on Fig. 4.1(d). And Fig. 4.1(e) is another version of Fig. 4.1(e) whose background is different from that of Fig. 4.1(e).

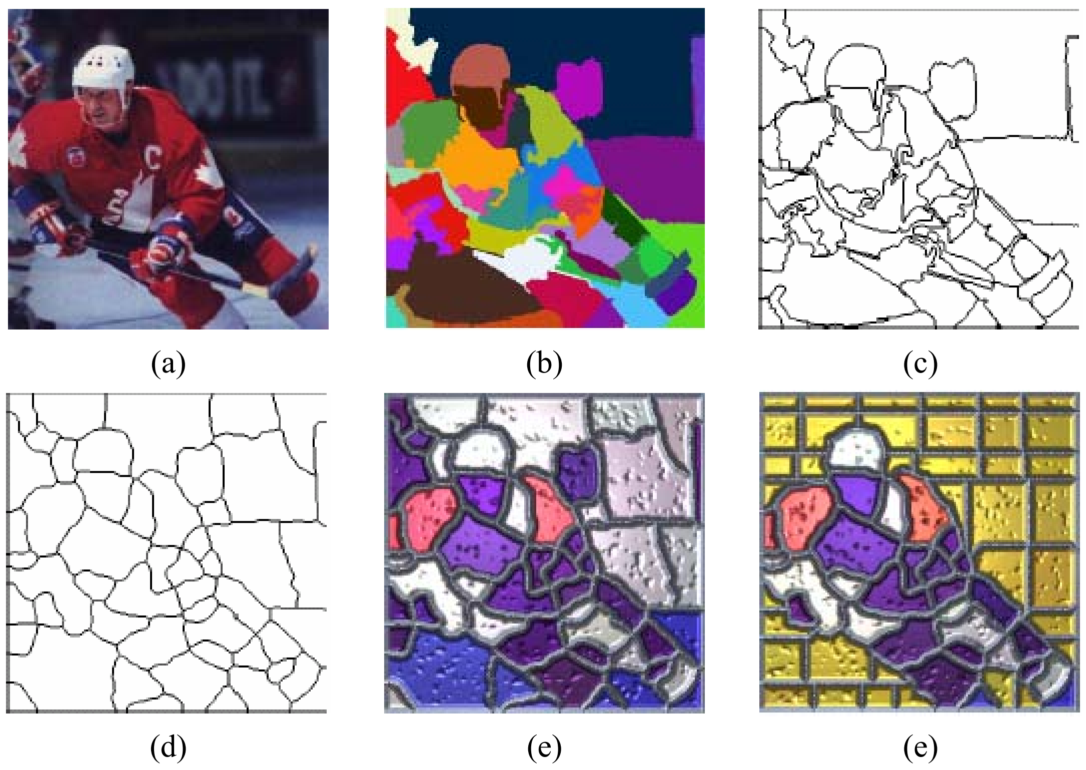


Fig. 4.1 Some intermediate images and stained glass images created by Mould [13]. (a) An original image Gretzky. (b) Segmented regions of (a). (c) Region boundaries of (b). (d) Smoothed region boundaries of (b). (e) Associated stained glass image. (e) Another stained glass image with different background from (e).

4.3 Proposed Stained Glass Image Creation Process

4.3.1 Scheme of Creation Process

Fig. 4.2 shows the scheme of the proposed stained glass image creation process. First, we quantize each pixel value of an input image into three bits, that is, one bit per R, G and B channel of each single pixel. And then we filter off the noise appearing during quantization by a voting filter. After applying the previous two steps, the input image will be divided into several color regions. The previous two steps are what we call preprocessing of an input image and will be discussed in detail in Section 4.3.2. Second, we sprinkle seeds on the color regions and apply the proposed region growing technique on each seed. A random number generator is used for seed sprinkling. We will use a secret key as a seed of the random number generator, if we want to embed data in the stained glass image. By the proposed region growing technique, we will have glass regions grown from each of the sprinkled seeds. In Section 4.3.3, a tree structure of glass regions and the proposed region growing technique will be discussed. In the final step, we search for gaps which are of large areas among created glass regions and fill the gaps with extra glass regions. We name it *glass region gap filling process* and will discuss it in Section 4.3.4. Finally, a stained glass image is created with glass color specified by the seed positions and the input image.

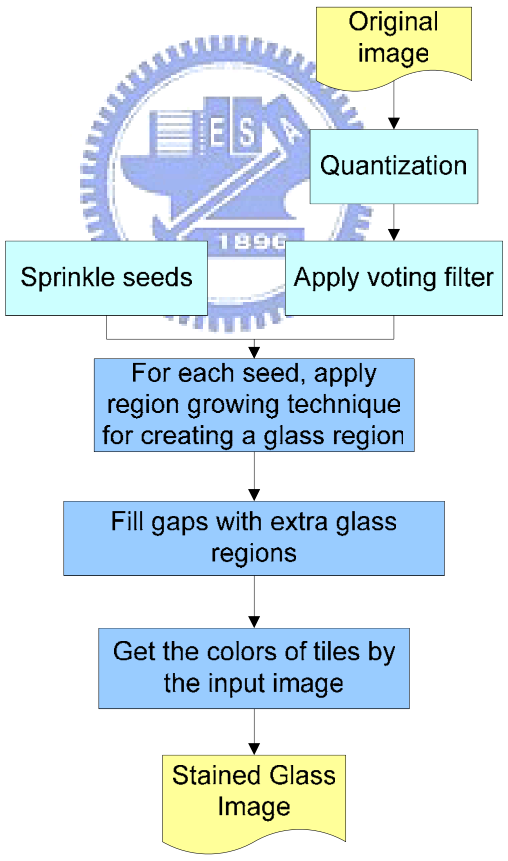

Original image
Quantization
Sprinkle seeds
Apply voting filter
For each seed, apply region growing technique for creating a glass region
Fill gaps with extra glass regions
Get the colors of tiles by the input image
Stained Glass Image

Fig. 4.2 Proposed stained glass image creation process.

4.3.2 Preprocessing of Input Images

Before applying the region growing technique for glass region creation, we have to apply some operations on the input image. They are quantization and filtering. As shown in Fig. 4.3, (a) is an input image and (b) is the quantization result of (a). We quantize Fig. 4.3(a) into three bits per pixel, that is, one bit per R, G and B channel of each single pixel. In Fig. 4.3(b), we can see that these *color regions* are shattered and not smooth. It will affect the results of region growing. So we have to smooth the initial segmentation before applying the region growing technique. We apply a voting filter to do that. Fig. 4.3(d) is the applied voting filter. For each pixel in (b), we accumulate the numbers of pixel colors within a square which is centered at that pixel. And then we set the color of that pixel as the color which has the maximum accumulation value. Here the square size is 11*11. Fig. 4.3(c) is the result after filtering. We can see the noise in Fig. 4.3(c) is removed. We name Fig. 4.3(c) a *color region image*. After removing the noise, we can now apply the region growing technique to these color regions.

4.3.3 Tree Structure of Glass Regions and Region Growing Process

In this section, we describe the propose tree structure of glass regions for region growing. The proposed tree structure is shown in Fig. 4.4(a). A tree node is either an interior node or a leaf node. An interior node has child nodes and a leaf node does not. We classify the tree nodes into two types, *succeeding node* and *expanding node*. Expanding nodes are the first or last nodes in a node level. As shown in Fig. 4.4, we bound expanding nodes by red borders and bound succeeding nodes by black borders. As shown in Fig. 4.4(a), there will be two child nodes of an expanding node, if any. One is an expanding node and the other is a succeeding node. On the other hand, a succeeding node will have only a child node, if any. And the child node is also a succeeding node.

A. Proposed Tree Growing Process

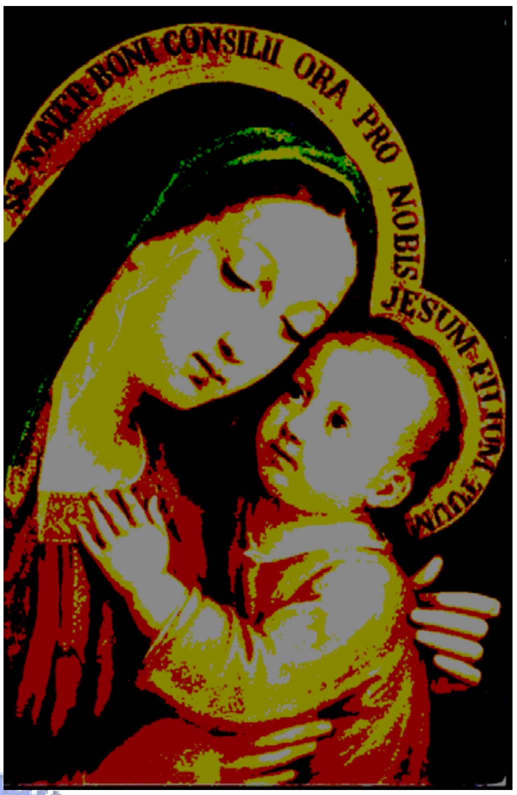
The proposed tree growing process is illustrated in

Algorithm 4.2. We will explain all the steps by taking Fig. 4.4 as an example. In Fig. 4.4, a dashed green line represents a color region derived from Section 4.2.2. Initially, we add two expanding nodes in L_1 as the child nodes of the root, R, and set them as interior nodes. In Step 2.1, we generate a child node, which is interior and a succeeding node, for each of the interior nodes in L_i ($i \geq 2$). And then in Step 2.1.1, we discard the generated child nodes, which are out of bound, from L_i and transfer the associated parent nodes from an interior node to a leaf node. Fig. 4.4(b) and (c) is an example of Step 2.1.1. As shown in Fig. 4.4(b), four succeeding nodes are produced in L_{i+1}. The right-most one is located in a color region different from where its parent node is located. So, we regard this node as out of bound, discard it from its parent node, and transfer its parent node from an interior node to a leaf node.

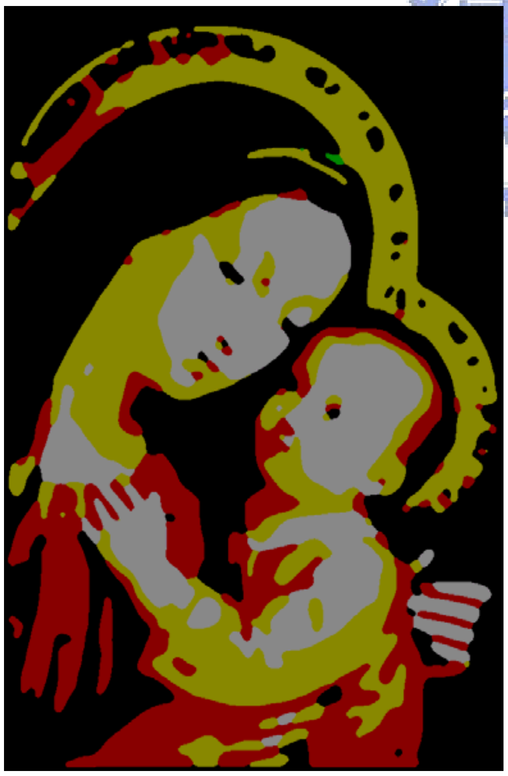
In Step 2.2, we generate a child node, which is interior and an expanding node, for each interior node which is also an expanding node in L_i. And then in Step 2.2.1, we discard the generated child nodes, which are out of bound, from L_i *without transferring the associated parent nodes from an interior node to a leaf node* (unlike what we do in Step 2.1.1). Fig. 4.4(d) and (e) is an example of Step 2.2.1. As shown in Fig. 4.4(d), only one node is produced in L_{i+1}. Because the right-most expanding node in L_i has been transferred into a leaf node in Step 2.1.1, we will not extend this node any more. As shown in Fig. 4.4(e), the generated child node is located in a color region different from where its parent node is located, so we discard it without transferring its parent node to a leaf node.

SS. MATER BONI CONSILII ORA PRO NOBIS JESUM FILIUM TUUM

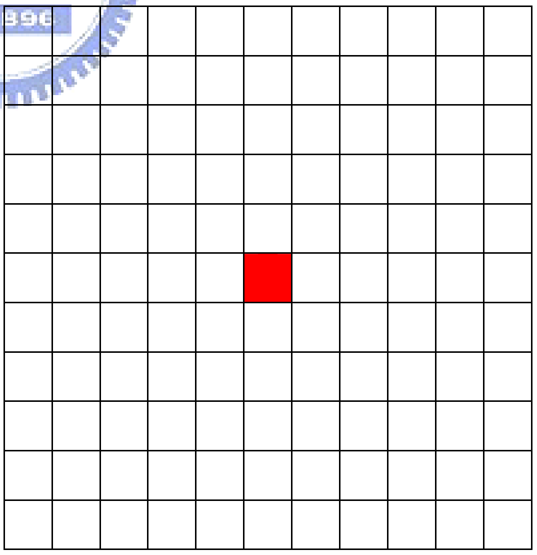
(a)



(b)



(c)



(d)

Fig. 4.3 Preprocessing of an original image. (a) The original image, (b) A quantized image of (a). (c) A filtered image of (b). (d) A voting filter of size 11x11

In this algorithm, Steps 2.1.2 and 2.2.2 are used for keeping the depth differences of neighboring leaf nodes less than or equal to one. The reason for doing that is shown in Fig. 4.4(f). In Fig. 4.4(f), the depth difference of the neighboring two leaf nodes is four. The blue line in Fig. 4.4(f) is the edge defined by the two leaf nodes. We can see that this edge overlaps another color region. It will result in overlapping glass regions when the growing process terminates. That is why we have to keep the depth differences of neighboring leaf nodes to be less than or equal to one by Step 2.1.2 and 2.2.2

Finally, in Step 2.3, we decide whether to terminate the growing process or not by checking if there are child nodes generated in Steps 2.1 and 2.3. If the process terminates, the derived tree will be part of a glass region in the stained glass image we created.

Algorithm 4.2: Tree growing process

Input: a tree root *R*.

Output: a tree, *GRT*, which is grown from *R*.

Steps:

```
Step 1   Add two expanding nodes in $L_1$ as the child nodes of $R$. Set the states of
         them as interior nodes.
Step 2   For each node level $L_i$, where i ≧ 2, perform the following steps:
    2.1   For each interior node, $N_{int}$, in $L_i$, generate a child node which
          is interior and succeeding. Denote it by $C_{is}$.
        2.1.1 Check whether $C_{is}$ is out of bound by the following way:
              if $N_{int}$ and $C_{is}$ are not in the same color region, regard $C_{is}$
              as out of bound, and then
              A    discard $C_{is}$ from $N_{int}$;
              B    trasfer $N_{int}$ from an interior node to a leaf node.
```

2.1.2 If the depth of C_{is} is greater than the depth of its neighboring sibling, then

A discard C_{is} from N_{int};

B trasfer N_{int} from an interior node to a leaf node.

2.2 For each interior node which is also an expanding node, N_{intE}, generate diagonally a child node which is interior and expanding. Denote it by C_{ie}.

2.2.1 Check whether C_{ie} is out of bound by the following way:

if N_{intE} and C_{ie} are not in the same color region, regard C_{ie} as out of bound, and then

discard C_{ie} from N_{intE}.

2.2.2 If the depth of C_{ie} is greater than the depth of its neighboring sibling, then

discard C_{ie} from N_{intE}.

2.3 Check whether the growing process is converged by the following way:

if there are child nodes generated in Steps 2.1 and 2.2, then

2.3.1 regard *GRT* as non-converged, accumulate the value of *i* by 1, and go back to the beginning of Step 2; otherwise,

2.3.2 regard *GRT* as converged, output *GRT*, and terminate the growing process.

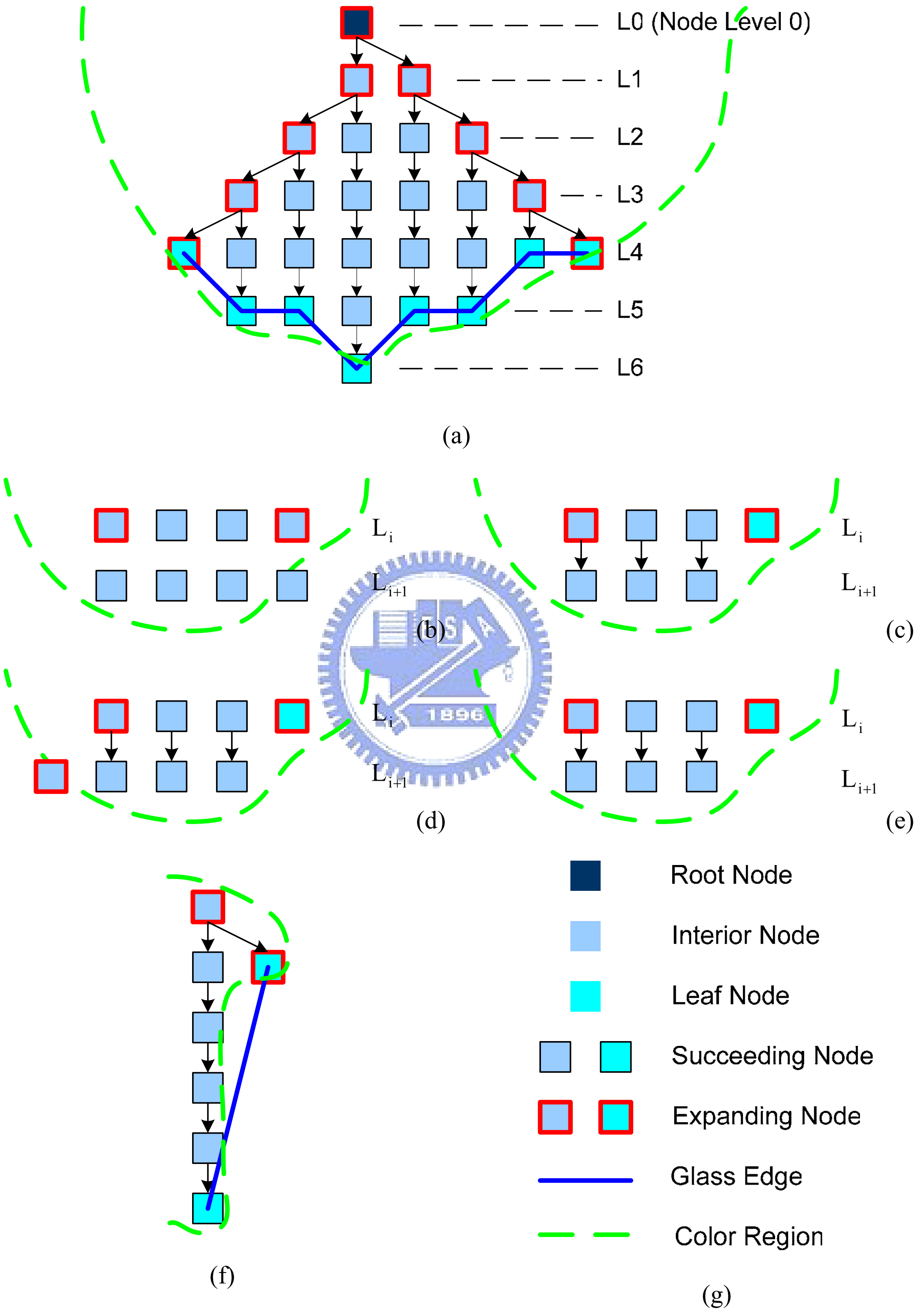


Fig. 4.4 Tree structure of glass regions. (a) A tree of a glass region; (b), (c), (d) and (e) An example of Steps 2 and 3 in Algorithm 4.1; (g) Figure descriptions.

B. Proposed Region Growing Process for Glass Region

After preprocessing the original image, we derive a corresponding color region image (as shown in Fig. 4.3(c)). We then sprinkle seeds on the color region image, and start the region growing process for creating the glass regions. Note that each glass region in a stained glass image represents a glass piece in a stained glass window. Each seed will result in a glass region after applying the proposed region growing process. The proposed region growing process is described as follows.

As shown in Fig. 4.5, first we root four trees in each seed, which is denoted by *Nt*, *Et*, *St*, and *Wt*, respectively. We then grow the four trees in four directions, namely, north, east, south, and west, respectively, by applying Algorithm 4.2. We will grow all the trees of all seeds simultaneously level by level by Algorithm 4.2. After the tree growing processes of these four trees terminate, we link the leaf nodes and the four trees to form the boundary of the associated glass region. The links of the leaf nodes are drawn by blue and the links of the trees are drawn by yellow in Fig. 4.5. We derive a leaf node link simply by linking the adjacent leaf nodes of a tree. However, if we link the four trees simply by linking the rightmost and leftmost leaf nodes of two adjacent trees, the link of trees may be out of the color region. As shown in Fig. 4.5(a), the link between *Nt* and *Et* is out of the color region, and it will result in overlapping glass regions. In order to keep the generated glass regions from overlapping, we need to search the *first expanding nodes* in the *node expanding pass* of the rightmost and leftmost leaf nodes of a tree. A node expanding pass is an upward pass which starts from a leaf node to the tree root. As shown in Fig. 4.5(b), starting from the leftmost leaf node of *Nt*, we can find an expanding node behind two succeeding nodes in the node expanding pass. By linking the first expanding nodes, the leftmost leaf node, and the rightmost leaf node of two adjacent trees, we can derive a tree link within the associated color region. As shown in Fig. 4.5(b), the tree link between *Nt* and *Et* is bound into the color region by adding the expanding node searching pass into the tree link.

Two points need to be noticed in the stained glass image creation process. First, according to the proposed tree growing process, each glass region has a minimum size of nine nodes, which is shown by pink squares in Fig. 4.5(a) and (b). In order to keep the glass regions from overlapping, we have to keep the inter-seed distance at least three nodes in magnitude. In this study, we let the distance between nodes to be two pixels. So, the minimum inter-seed distance must be six pixels. The last point to be noticed is that more than one seed may be located in a color region. In order to keep the created glass from overlapping, we have to keep a global *tree map* which is use to record whether a node position has been occupied by other trees or not. We will transfer the parent node, which wants to extend a child node to an occupied node position, from an interior node to a leaf node. The *tree map* will also be used in the next section for the gap filling process.

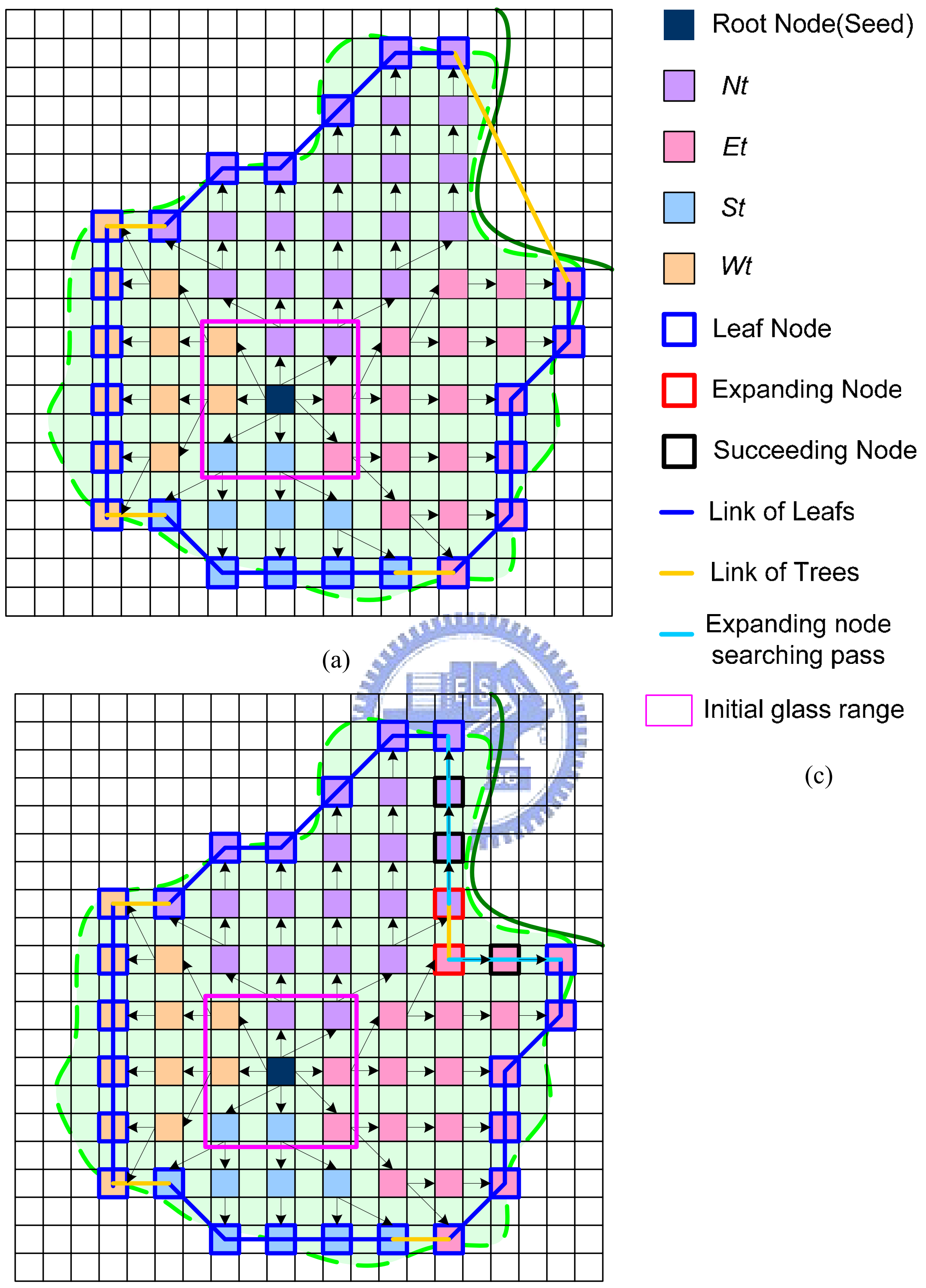


Fig. 4.5 Proposed region growing process. (a) A version of linking the leaf nodes. (b) A version of searching the last expanding nodes. (c) Figure descriptions.

4.3.4 Glass Region Gap Filling Process

At the beginning of the proposed glass growing process, we sprinkle seeds as tree roots randomly. It will result in more than one seed sprinkled in a color region. On the contrary, it might also happen that no seed is sprinkled in a color region, causing some gaps among glass regions. So, we have to apply a further step for filling these gaps. We name the step the gap filling process.

As shown in Fig. 4.6, the blue polygons are the glass regions grown by the previous steps. Among the blue polygons are the gaps we need to fill. Initially, we scan the tree map. If there is no tree node within a square range of size 15x15 pixels, we put an additional seed at the center of that square and apply the proposed glass region growing process on it. The yellow polygons in Fig. 4.6 are the additional glass regions for filling the gaps.

In Chapter 5 we will propose a data hiding technique along with three applications, namely, watermarking, secret communication, and image authentication. The data is embedded in the glass region with randomly generated seeds. We will not embed data into the glass regions which are used for gap filling except in the application of image authentication.

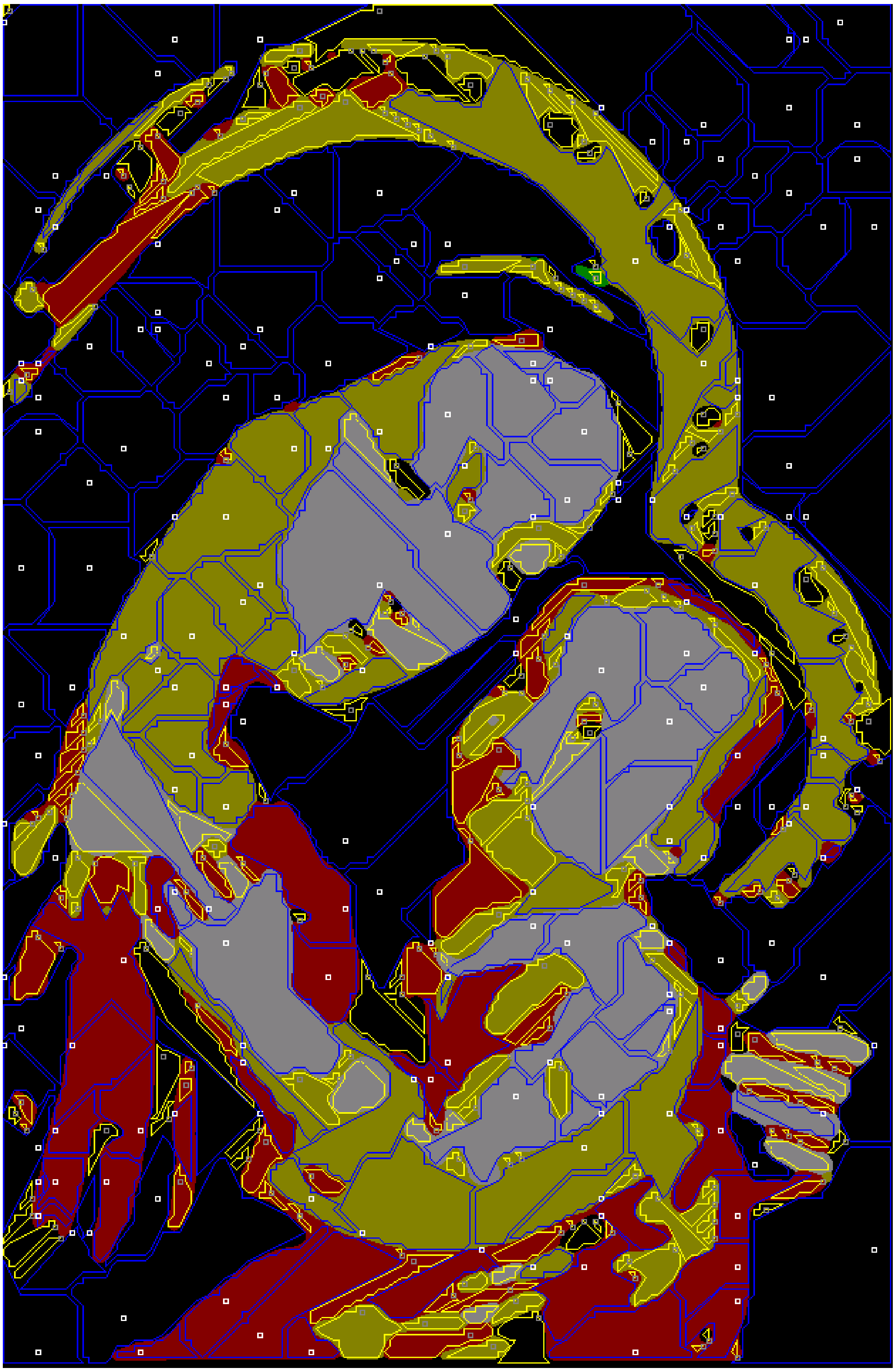
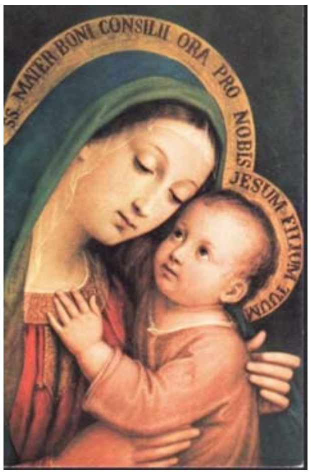


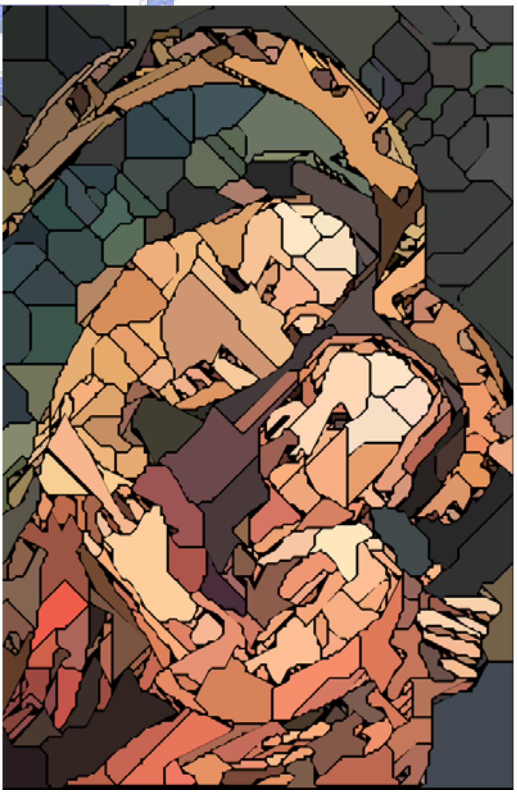
Fig. 4.6 Illustration of gap filling process.

4.4 Experimental Results and Discussions

Fig. 4.7 is an experimental result of applying the proposed stained glass creation process to an original image. Fig. 4.7(a) is the original image and Fig. 4.7(b) is the final stained glass image. We can find that the glasses in Fig. 4.7(b) do not overlap each other. This characteristic is good for data embedding and extraction process. Some other experimental results are shown in Fig. 4.8 Some experimental results. In Fig. 4.8, the images in the first column are original images, the images in the middle column are the stained glass images created by Mould's method [13], and the images in the last column are the stained glass images created by the proposed methods. In Chapter 5, we will utilize a key for seed sprinkling, and the data hiding technique and some applications will also be discussed.



(a)



(b)

Fig. 4.7 Experimental results. (a) An original image. (b) A stained glass image of (a).

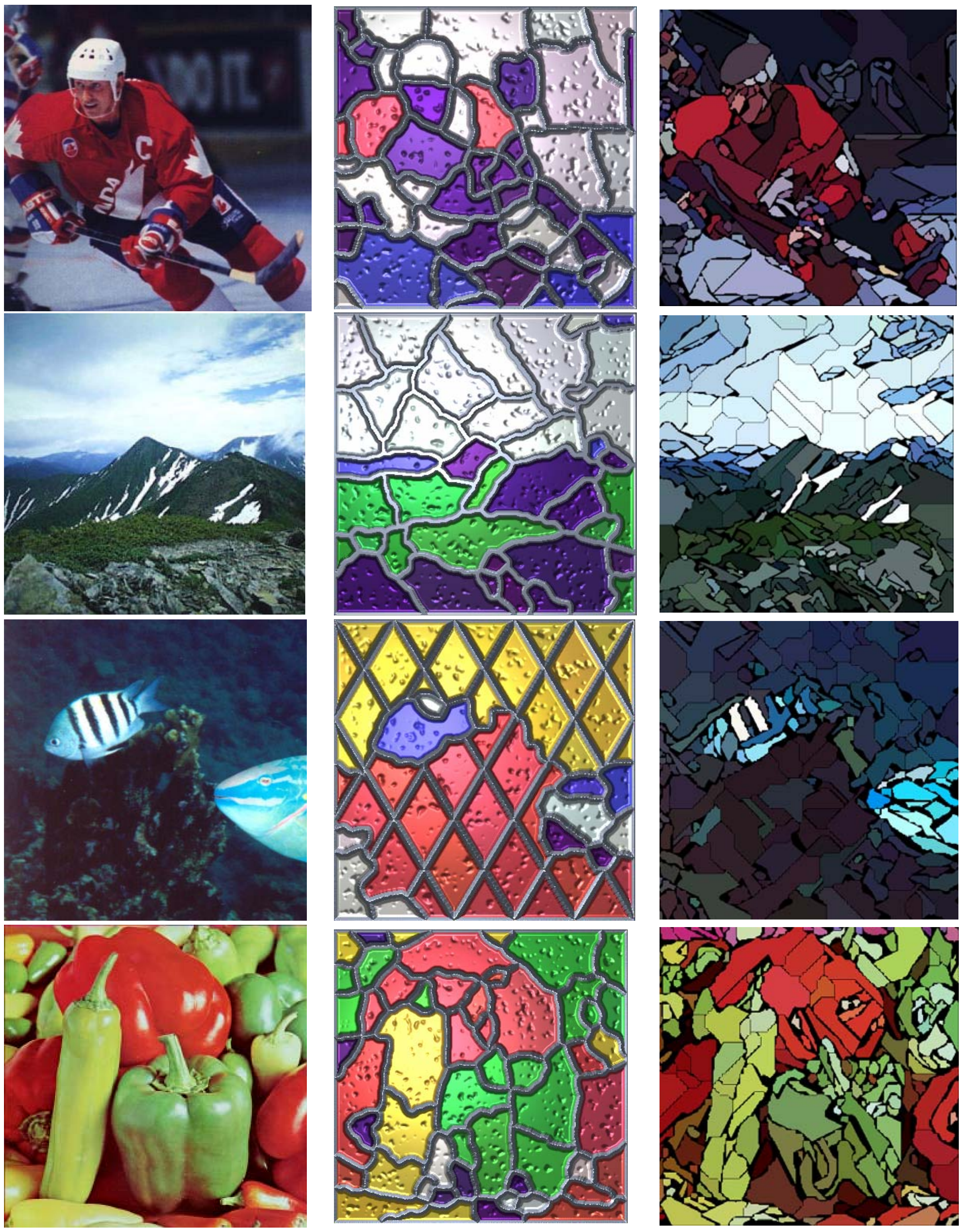


Fig. 4.8 Some experimental results.

Chapter 5
Data Hiding in Stained Glass Images and Applications

5.1 Introduction

In this chapter, we will introduce a data hiding technique by utilizing the number of tree nodes in a glass region. We will also discuss the three applications which are introduced in Chapter 3. The feature detection process for stained glass images will also be described.

5.1.1 Concepts behind Proposed Technique

The feature we utilize for data hiding in stained glass images is the number of tree nodes in a tree of a glass region. By removing the deepest nodes we can hide data in a stained glass image. However, it will result in cracks at the edges and corners of a glass region, though still acceptable. As shown in Fig. 5.1(a), we remove the two nodes which are the deepest nodes in tree *Nt* of this glass region. Fig. 5.1(b) is the result of removing the two nodes. We can see the resulting cracks in the glass region where the nodes are removed. There are four trees contained in a glass, but not all the four trees can be used for data hiding. The number of nodes of a tree must be large enough so that the data can be embedded by removing tree nodes. We name the trees that can be used for data hiding *effective trees*. More details will be discussed in the following sections and so will the glass feature detection process.

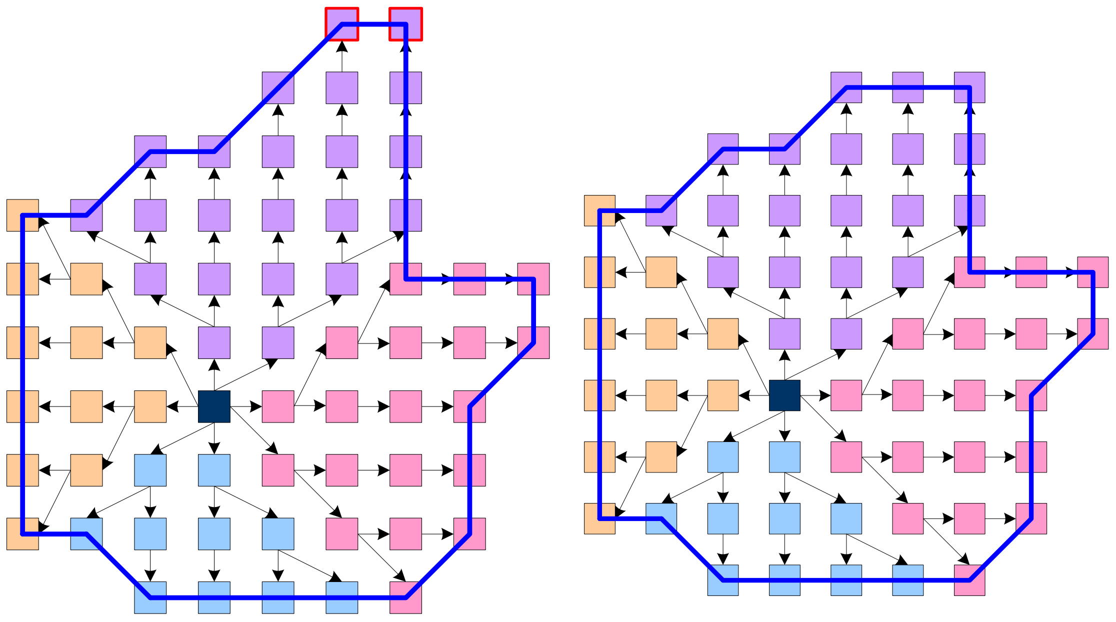

(a) (b)

Fig. 5.1 An example of removing nodes for data hiding. (a) A glass region copied from Fig. 4.4(b). (b) A glass region with data embedded.

5.1.2 Concepts behind Proposed Data Embedding and Extraction Techniques

The overall concept of embedding data into a stained glass image is illustrated in Fig. 5.2. Since only one glass feature is found for data hiding, only one of the three applications can be embedded at a time. The three applications are watermarking, secret communication, and image authentication. No matter what kind of data we want to embed is, we first transform the input data into a bit sequence before the data embedding process.

Unlike what we do in a tile mosaic image, the secret key used for protecting embedded data is now utilized in the image creation process. In the creation process, initially, we use a secret key as a seed of a random number generator, and then generate a sequence of seeds randomly. Each of the generated seeds will be the root of the four trees (i.e. *Nt, Et, St* and *Wt*) of a glass region. After the creation process, we embed the bit sequence of the data by removing the deepest nodes of every single tree. As mentioned before, the target tree must have an sufficiently large number of nodes, so that the bits can be embedded. More details will be discussed in Section 5.3.1.

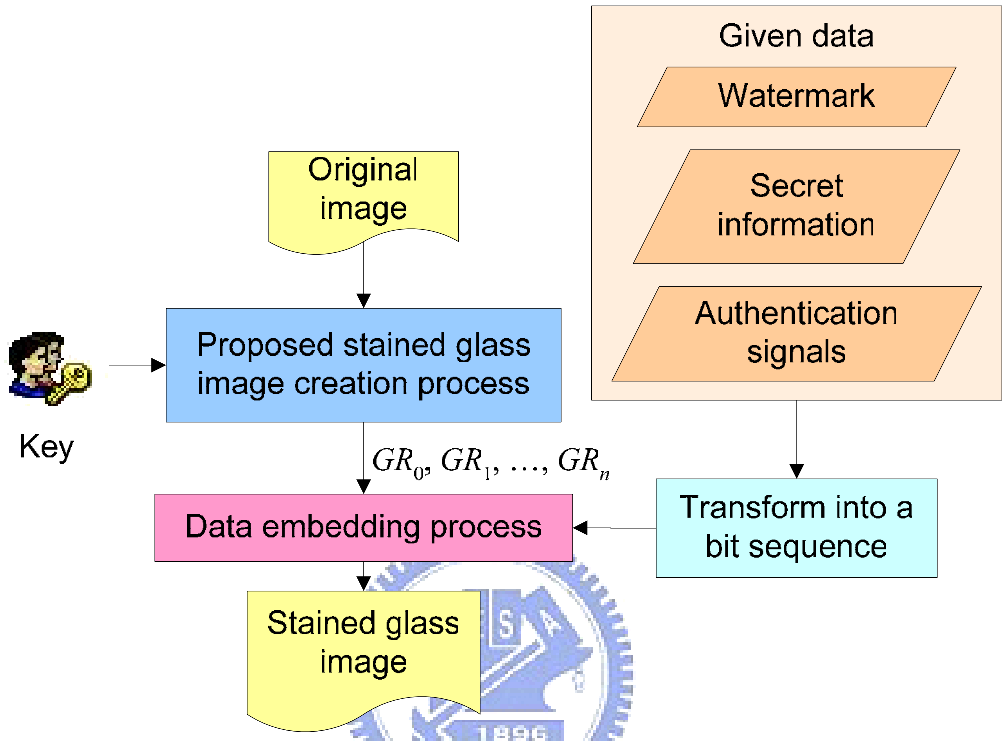


Fig. 5.2 Proposed data embedding process.

A flowchart of data extraction process in stained glass images is shown in Fig. 5.3. The secret key is applied in the glass feature detection process. In the glass feature detection process, as what we do in the creation process, the secret key will be used for seed generation. By using the same key as a seed of a random number generator for seed generation, we can derive the same seed sequence as the one derived in the creation process. We then apply the region growing process, which is proposed in Section 4.3.3, on the seed sequence for deriving a sequence of glass regions. Therefore, the two sequences of glass regions derived in the creation and detection processes will be identical. By counting the number of nodes in each single tree of the detected sequence of glass regions, the embedded bit sequence can be extracted. So the embedded data will be derived by transforming the bit sequence into a watermark image, a secret text, or an authentication image. The glass feature detection process will be discussed with more details in Section 5.2. And the core concepts of the data embedding and extraction process will be described in Section 5.3. The three applications will also be discussed is Sections 5.4, 5.5, and 5.6.

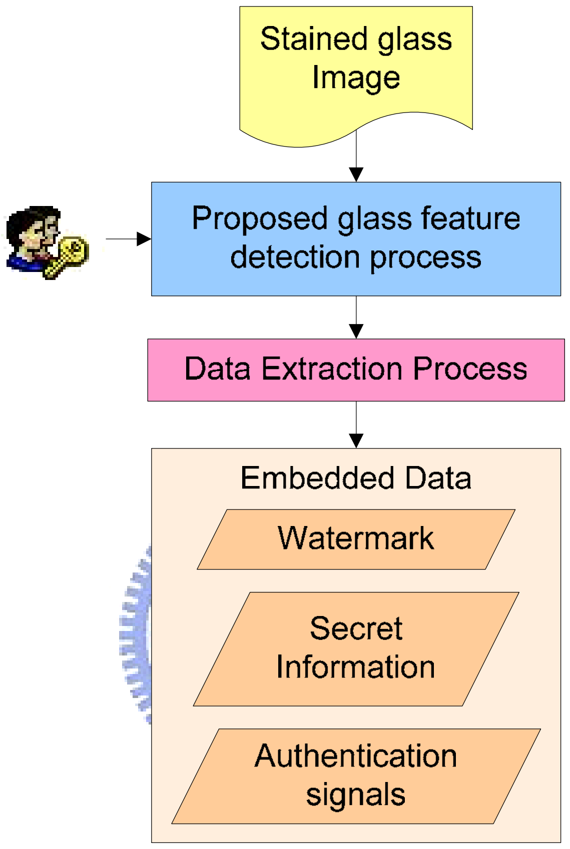


Fig. 5.3 Proposed data extraction process.

5.2 Glass Feature Detection Process

Before the data embedding and extraction process, we have to derive a sequence of trees from a sequence of glass regions. Each glass region will have at most four trees to be added into the tree sequence. In the data embedding process, we derive the tree sequence from the glass regions which are created by the stained glass creation process. On the other hand, if we want to extract data from a stained glass image, glass regions are derived by the glass feature detection process. So it is clear that the detection process is mainly to extract a sequence of glass regions, so that the corresponding sequence of trees with data embedded can be derived. Finally, a watermark, a secret message, or an authentication image can be extracted.

The entire glass feature detection process is shown in Fig. 5.4. As what we do in the creation process, we first sprinkle seeds on the stained glass image (instead of a color region image) by a secret key as the root of glass regions. Then we create a sequence of glass regions by applying the region growing technique proposed in Section 4.3.3. A sequence of glass regions will thus be derived. Finally, we pick the trees which we have embedded data into. The method to judge whether data have been embedded in a tree or not, will be discussed in Section 5.3.

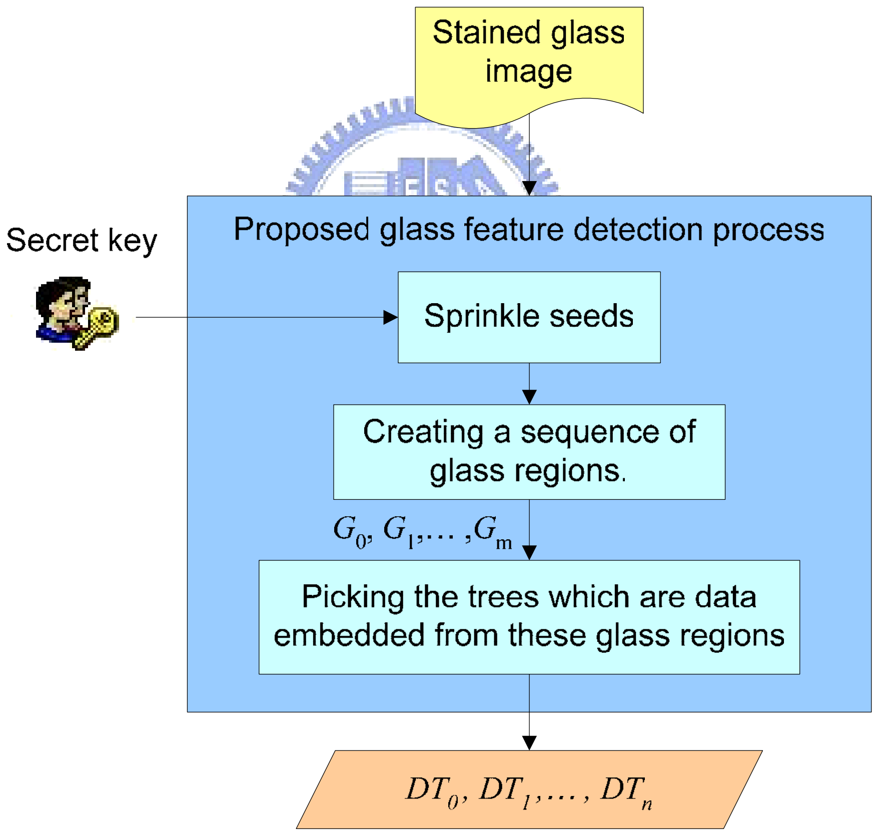

Stained glass image
Secret key
Proposed glass feature detection process
Sprinkle seeds
Creating a sequence of glass regions.
$G_0, G_1, \ldots, G_m$
Picking the trees which are data embedded from these glass regions
$DT_0, DT_1, \ldots, DT_n$

Fig. 5.4 Proposed glass feature detection process.

Fig. 5.5 shows the detection result of a stained glass image. The white squares are the seeds generated by the secret key which is the same as the one used in the image creation process. Since the seed positions and the applied region growing method are identical to the ones used in the creation process, an identical sequence of glass regions is derived. The blue polygons in Fig. 5.5 are the detected boundaries of the glass regions. We can see that they fit the glass regions very well. Thus, we can extract the embedded data by picking up the trees, which we have embedded data into, from these glass regions into a tree sequence. Although the detected boundaries fit the glass regions well, some detection errors do come out. Take Fig. 5.5 as an example. There are 400 glass regions in the stained glass image and we have embedded data in 1173 trees among these glass regions. Assume the tree sequence picked in the data embedding process (i.e., the effective trees of data embedding) to be ET_0, ET_1, …, ET_n and the tree sequence picked in the data extraction process (i.e., the effective trees of data extraction) to be DT_0, DT_1, …, DT_n. We compute the detection error, *diff*, by the formula:

$$diff_i = \left| \text{number of tree nodes in } ET_i - \text{number of tree nodes in } DT_i \right| .$$

Table 5.1 is the error statistics of Fig. 5.5. As shown in Table 5.1, the values of *diff* are mostly 0 and all the values are smaller than 1. So there are detection errors but they are still acceptable. We can introduce error tolerance into the proposed data hiding technique for eliminating the detection errors.

Table 5.1 error statistic of Fig. 5.5

Number of glass regions: 200, Number of effective trees: 481

diff	Number of tree pairs	Rate
0	478	0.9938
1	3	0.0062
2	0	0.0
3	0	0.0
total	481	1.0

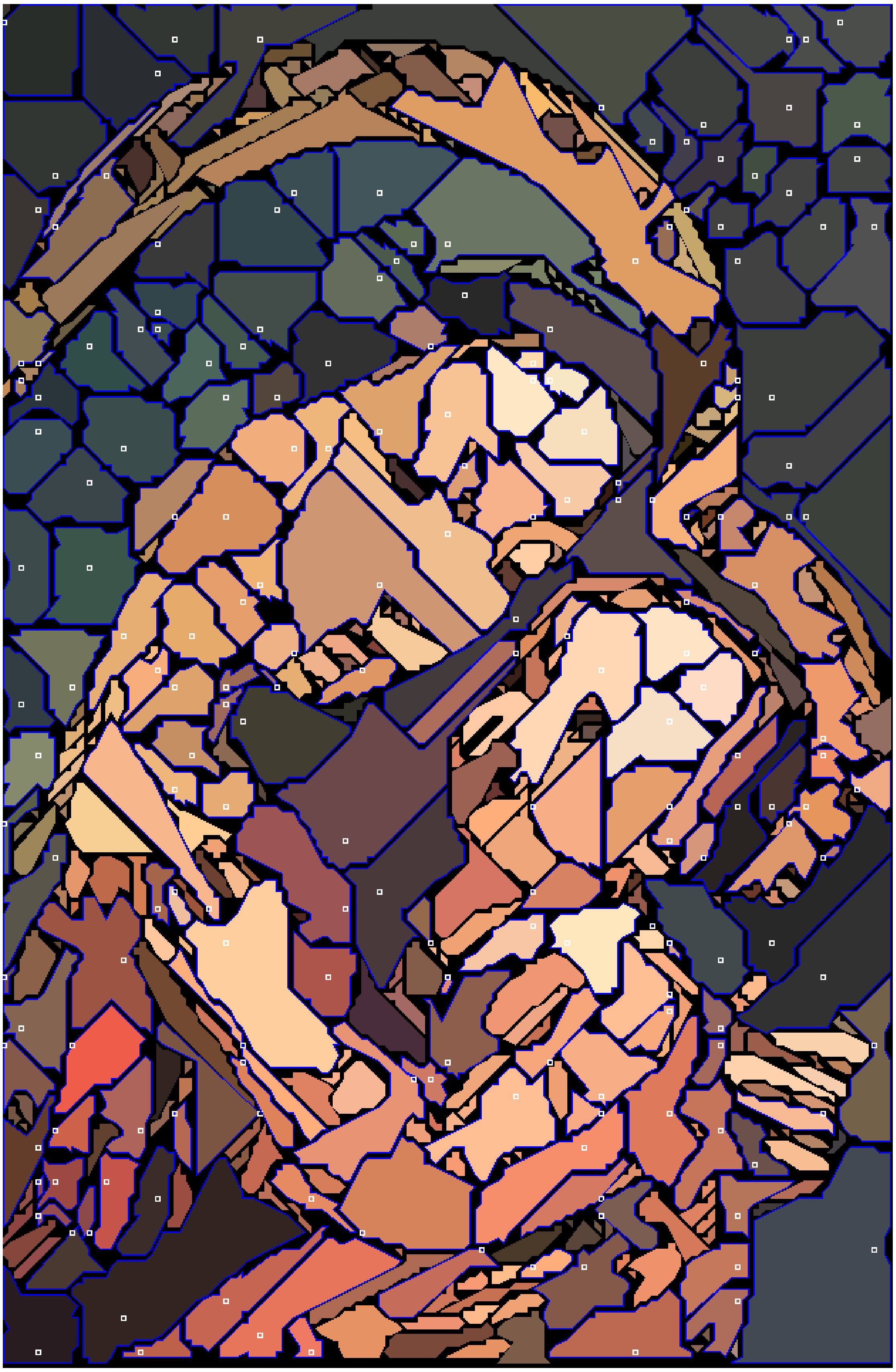


Fig. 5.5 Detection result of a stained glass image.

5.3 Data Hiding by Glass Boundary Cracking

The core concept of data hiding by tree node number modification is illustrated in Fig. 5.6. Initially, we compute the remainder, *REM*, of dividing the tree node number, *TNN*, by a divisor, *DIV*. The value of *REM* will thus range between 0 and *DIV*-1 as shown in the following formula:

$$REM = TNN \bmod DIV,\ 0 \leqq REM < DIV-1.$$

Then, we divide the range of *REM* into several sub-ranges. Each range will represent a specific bit code. In the case of Fig. 5.6, the bit code contains two bits, so we divide the range into four sub-ranges which represent 00, 01, 10 and 11, respectively. In other words, the number of sub-ranges, *NSR*, is computed by the following formula:

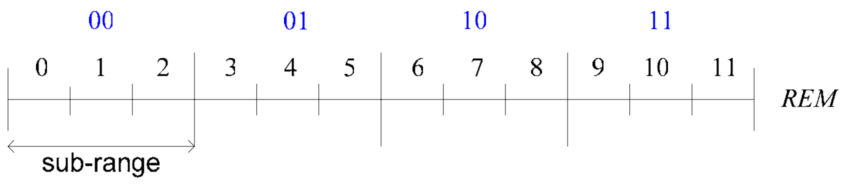
$$NSR = 2^{bitN},$$

where *bitN* is the number of bits we want to embed in an effective tree. Furthermore, because each sub-range includes several *REM* values, tolerance of detection errors are achieved. Take Fig. 5.6 as an example. Each sub-range includes three *REM* values. No matter what the value of *REM* is (zero, one, or two), it means that we have embedded bit code 00 in the associated effective tree. The three rest sub-ranges may be deduced by analogy. Finally, the value of *DIV* can be computed as follows:

$$DIV = SSR*NSR,$$

where *SSR* is the number of *REM* values spanned by a sub-range.

So, the value of *DIV* in Fig. 5.6 is $3 \times 4 = 12$.



$bitN$=2, SSR=3, $NSR=2^{bitN}=2^2=4$, $DIV= SSR*NSR=3*4=12$.

Fig. 5.6 Core concept of data hiding in stained glass images.

After deciding the values of *bitN* and *SSR*, the data embedding and extraction methods can be described as follows.

A. Data Embedding Process

If we want to embed a bit code into an effective tree, we adjust the value of *REM* to be the center of the sub-range of *REM* which represents the bit code we want to embed. Take Fig. 5.6 as an example, assume that the value of *REM* is nine and the bit code we want to embed is 01. The sub-range representing bit code 01 is the second one which centers on the *REM* value of four. So the number of nodes to remove from the effective tree, *NNR*, for embedding bit code 01 is computed as follows:

$NNR= | REM -$ central value of target sub-range $| = | 9 - 4 | = 5$.

By removing the deepest five nodes from the target effective tree, bit code 01 can be embedded.

B. Data Extraction Process

The process of extracting data from an effective tree is simply to compute the value of *REM* and find the sub-range where it falls. The corresponding bit code is the one we want to extract.

C. Acquiring Effective Trees for Data Hiding

Before data embedding and extraction from a stained glass image, for each tree of a glass region, we have to specify if we can embed data in it or if we can extract data from it. In other words, we have to specify whether the trees are *effective trees* or not.

In data embedding, an effective tree is one of sufficient tree nodes to be removed for data hiding (i.e., the value of *TNN* of the tree must be large enough). In the proposed data embedding process, we remove at most *DIV*-1 nodes from a target tree and each tree in a glass region contains at least three nodes in the proposed creation process. So an effect tree must contain at least *DIV*+3 nodes, so that a bit code can be embedded in all cases. We name the minimum number of nodes in an effective tree for data embedding, *minimum tree size*, and denote it by *minTSE*. To prevent glass regions from being broken into tiny pieces, we let the value of *minTSE* be *DIV**2+3 instead of *DIV*+3. So the way we acquire effective trees among glass regions is to pick the trees whose number of nodes is larger than the specified value of *minTSE*.

In data extraction, because the value of *minTSE* is *DIV**2+3 and we remove at most *DIV*-1 nodes from a target tree for data embedding. The minimum number of nodes in an effective tree for data extraction, which is denoted by *minTSD*, will thus be *DIV*+3. In other words, if the value of *TNN* of a tree is larger than *minTSD*, we can claim that there are data embedded in it. However, there is one more condition which needs to take care before making such a claim. It is that, if there is a tree with its number of node smaller than *minTSE* and larger than *minTSD*, as shown in the blue area of Fig. 5.7, we will embed no data in the embedding process and will extract some meaningless bit codes when doing data extraction. To prevent such a condition, we reduce the value of *TNN* to *minTSD* by removing the deepest nodes of the target tree while acquiring effective trees for data embedding. So, there will be no tree of *TNN* values between *minTSD* and *minTSE* (i.e., the blue area in Fig. 5.7), after the acquiring process for data embedding.

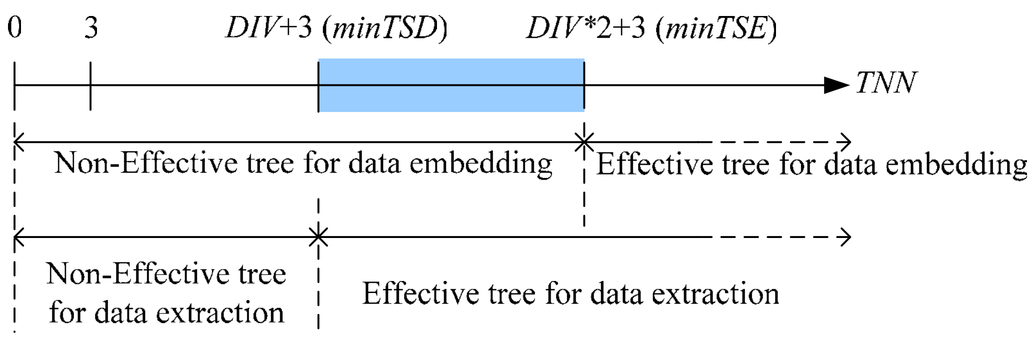

0
3
DIV+3 (minTSD)
DIV*2+3 (minTSE)
TNN
Non-Effective tree for data embedding
Effective tree for data embedding
Non-Effective tree for data extraction
Effective tree for data extraction

Fig. 5.7 Illustration of acquiring effective trees in data embedding and extraction process.

5.4 Application to Secret Hiding

5.4.1 Data Embedding Process

The algorithm of embedding a secret message into a stained glass image is shown in Algorithm 5.1. After the proposed creation process, in Step 2, we retrieve the four trees (i.e. *Nt, Et, St* and *Wt*) from each of the generated glass regions into a sequence *RT*. In Step 3, we acquire the effective trees by checking the *TNN* value. We then embed the bit codes, which are derived in Step 1, into the effective trees until the effective trees are exhausted. Finally, a stained glass image with secret information embedded is created.

Algorithm 5.1: Embedding a secret message in a stained glass image.

Input: a sequence of stained glass region $GR = GR_0, GR_1, \ldots, GR_n$, and a secret message M.

Output: a stained glass image with secret message embedded.

Steps:

Step 1 Append an ending pattern to M. Transform M into a sequence of bit codes $B = B_0, B_1, \ldots, B_m$. The number of bits contained in a bit code is *bitN*.

Step 2 Retrieve the four trees from each GR_i and concatenate them into a sequence, $RT = RT_0, RT_1, \ldots, RT_{4n}$.

Step 3 Acquire a sequence of effect trees, $ET = ET_0, ET_1, \ldots, ET_k$, where $k \leqq 4n$, from RT in the following way:

for each RT_i in RT, if the value of TNN_i is larger than the value of $minTSE$, then

3.1 add the associated RT_i into ET; otherwise, if the value of TNN_i is larger than $minTSD$ and smaller than $minTSE$, then

3.2 remove the deepest nodes of RT_i to make TNN_i equal to the value of $minTSD$.

Step 4 For each pair of B_i and ET_i, compute the corresponding value of NNR_i by the formula, $NNR_i = \mid REM_i -$ central value of target sub-range $\mid$, and remove the deepest nodes from ET_i with the amount of NNR_i.

5.4.2 Data Extraction Process

The secret message extraction process is simply an inverse version of the embedding process. First, we derive the glass regions, $GR_0, GR_1, \ldots, GR_n$, from the stained glass detection process with a secret key as proposed in Section 5.2. And we acquire next the effective trees, $DT_0, DT_1, \ldots, DT_k$, by checking whether TNN_i is larger than $minTSE$ or not. We then find the corresponding bit code B_i by checking the value of REM_i. Finally, we transform the sequence of the extracted bit codes into text format, and thus complete the extraction of the embedded message.

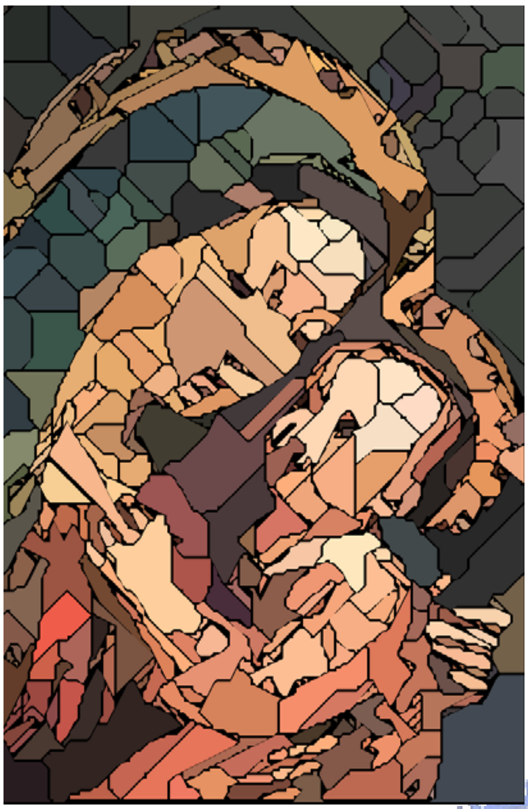
5.4.3 Experimental Results and Summary

Fig. 5.8 shows some experimental results of secret hiding in a stained glass image. Fig. 5.8(a) and (b) are stained glass images without and with data embedded, respectively. Fig. 5.8(c) and (d) are the details at the upper left corner of (a) and (c), respectively. By comparing (c) and (d), we can find that the glass regions of (d) are cracked slightly. Fig. 5.8(e) is a secret message extracted from (b). The message is identical to the one we embedded. Fig. 5.8 (f) is the secret message extracted from (b) with a wrong key. We can see that the text shown in Fig. 5.8 (f) is disordered and meaningless. It proves that the key we applied in the proposed creation process really works.

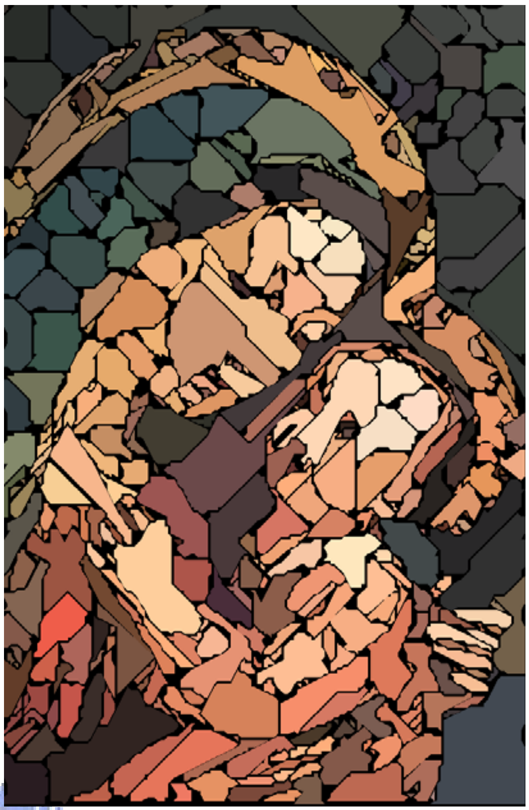
5.5 Application to Watermarking for Copyright Protection

5.5.1 Data Embedding Process

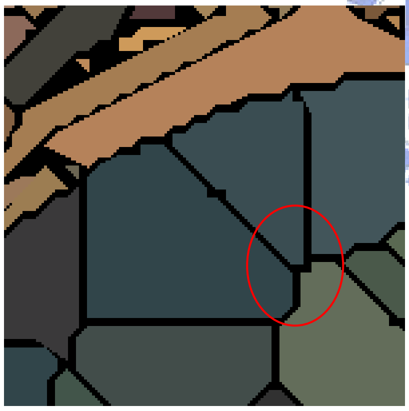
As shown in Algorithm 5.2, the proposed method for embedding a watermark is similar to the one of embedding a secret message. The difference is that we transform a watermark W into a bit sequence $B = B_0, B_1, \ldots, B_m$, instead of transforming a secret message. Furthermore, the number of glass regions created by the proposed creation process equals the number of bit codes of B, namely, m. We denote the sequence of glass regions by GR_0, GR_1, …, GR_m. After acquiring the effective trees, $ET = ET_0$, ET_1, …, ET_k, we embed B_i into the effective trees which comes from GR_i. Because we pick from each glass region, GR_i, at most four trees into ET, each bit code B_i will thus be embedded at most four times into a glass region, GR_i. The repetition of embedding a bit code will result in additional robustness of the embedded watermark, W.



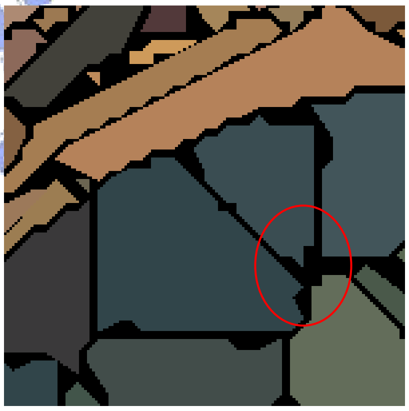
(a)



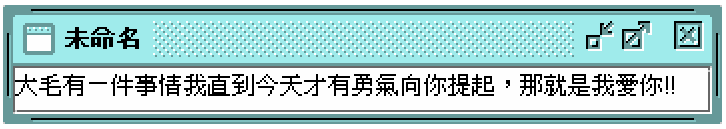
(b)



(c)

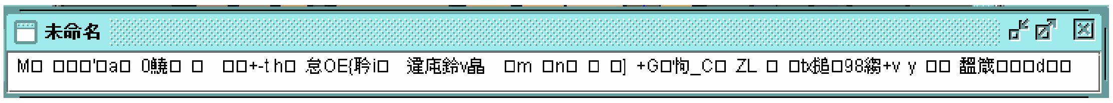


(d)



大毛有一件事情我直到今天才有勇氣向你提起，那就是我要你!!

(e)



MD DDD'DaD 0鱙D D DD+-thD 怠OE{聆iD 違庵鈴v晶 Dm DnD D D] +GD恂_CD ZL D Dt槌D98綹+v y DD 罏箴DDDdDD

(f)

Fig. 5.8 Experimental results of secret hiding in stained glass image. (a) A stained glass image without hidden data. (b) A stained glass image with secret message embedded. (c) Details of (a). (d) Details of (b). (e) The secret message extracted from (b). (f) The extraction result of (b) with a wrong key.

Algorithm 5.2: Embedding a watermark in a stained glass image.

Input: a sequence of stained glass region $GR = GR_0, GR_1, \ldots, GR_m$, and a watermark W.

Output: a stained glass image with watermark embedded.

Steps:

Step 1 Transform W into a sequence of bit codes $B = B_0, B_1, \ldots, B_m$. The number of bits contained in a bit code is *bitN*.

Step 2 Retrieve the four trees from each GR_i and put them into a sequence, $RT = RT_0, RT_1, \ldots, RT_{4n}$.

Step 3 Acquire a sequence of effect trees, $ET = ET_0, ET_1, \ldots, ET_k$, where $k \leqq 4n$, from RT in the following way:

for each RT_i in RT, if the value of TNN_i is larger than the value of *minTSE*, then

3.1 add the associated RT_i into ET; otherwise, if the value of TNN_i is larger than *minTSD* and smaller than *minTSE*, then

3.2 remove the deepest nodes of RT_i to make TNN_i equal to the value of *minTSD*.

Step 4 Embed B_i into the effective trees which come from GR_i by computing the corresponding value of *NNR* and removing the deepest nodes with the amount of *NNR*.

5.5.2 Data Extraction Process

The extraction process is also similar to the one of extracting a secret message. The difference is that we apply the voting strategy on the effective trees coming from the same glass regions which are derived from the detection process. By the voting strategy, we can derive the bit code B_i embedded in a glass region, GR_i. Finally, we derive the embedded watermark, W, by transforming the extracted bit codes.

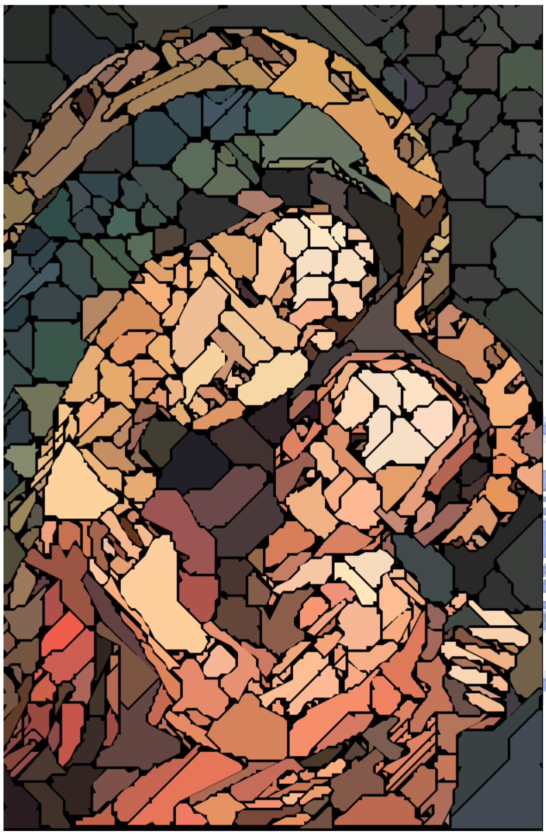
5.5.3 Experimental Results and Summary

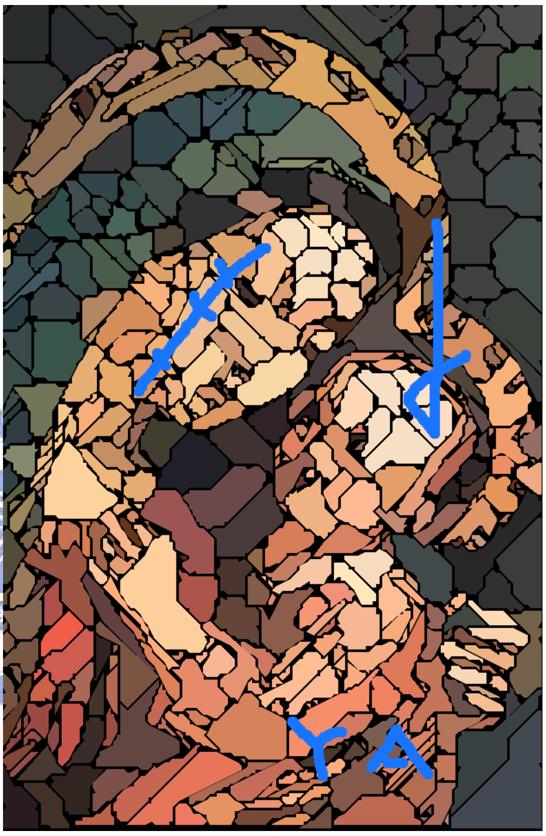
Some experimental results are shown in this section. In Fig. 5.9, (c) is an input watermark image, (a) is the resulting stained glass image with watermark (c) embedded, and (b) is a damaged image of (a). Fig. 5.9(d) and (f) are the watermark images extracted from Fig. 5.9(a) and (b), respectively. We can find that there are some salt-and pepper noise in (f), but the watermark is still recognizable. From Fig. 5.9(f), we can see that the watermark can still be recognized under a certain degree of damaging. Fig. 5.9(e) is a watermark extracted from (b) with a wrong key. From Fig. 5.9(e), we can see that the embedded watermark is protected properly by the secret key.

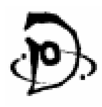
5.6 Application to Authentication of Images

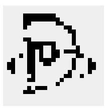
5.6.1 Authentication Signal Embedding Process

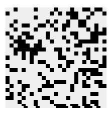
The proposed method for embedding authentication signals is also similar to the one of embedding a secret message. In order to verify the entire stained glass image, we have to embed authentication signals in the glass regions created in the gap filling process. Assume the sequence of glass regions to be $GR = GR_0, GR_1, \ldots, GR_n, GR_{n+1}, \ldots, GR_f$, where the glass regions with serial numbers coming from n+1 to f denote the glass regions created in the gap filling process. As shown in **Algorithm 5.3**, we use a secret key as a seed of a random number generator in the creation process. We denote the random number generator by a function $Ran(x)$, and the range of the generated number is $0 \leqq Ran(x) < x$. In Step 2, we can generate a sequence of authentication signals, $S = S_0, S_1, \ldots, S_n, S_{n+1}, \ldots, S_f$, by $Ran(x)$. Finally, as we do in watermarking, we embed S_i into the effective trees which come from GR_i. Like the process of watermarking, each S_i will be embedded at most four times in a glass region RG_i.


(a)


(b)


(c)


(d)


(e)

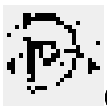

(f)

Fig. 5.9 Experimental results of watermarking a stained glass image. (a) A stained glass image with watermark, (c), embedded. (b) A damaged image of (a). (c) An input watermark. (d) A watermark extracted from (a). (e) A watermark extracted from (a) with a wrong key. (f) A watermark extracted from (b).

Algorithm 5.3: Embedding authentication signals in a stained glass image.

Input: a sequence of stained glass region $GR = GR_0, GR_1, \ldots, GR_n, GR_{n+1}, \ldots, GR_f$, a secret key K.

Output: a stained glass image with authentication signal embedded.

Steps:

```
Step 1  Use $K$ as the seed of a random number generator, $Ran(x)$.
Step 2  Compute a sequence of authentication signals, $S = S_0, S_1, \ldots, S_n, S_{n+1}, \ldots, S_f$ by the formula:
        $RGBhash = (R \times G \times B) \bmod 1013.$
        $S_i = (RGBhash + Ran(K + RGBhash)) \bmod bitN.$
        where $R$, $G$ and $B$ are the RGB color channels of $GR_i$, $bitN$ is the bit number of $S_i$, and 1013 is a chosen prime number.
Step 3  Retrieve the four trees from each $GR_i$ into a sequence, $RT = RT_0, RT_1, \ldots, RT_{4f}$.
Step 4  Acquire a sequence of effect trees, $ET = ET_0, ET_1, \ldots, ET_k$ from $RT$, where $k \leqq 4n$, in the following way:
        for each $RT_i$ in $RT$, if the value of $TNN_i$ is larger than the value of $minTSE$, then
        4.1  add the associated $RT_i$ into $ET$; otherwise, if the value of $TNN_i$ is larger than $minTSD$ and smaller than $minTSE$, then
        4.2  remove the deepest nodes of $RT_i$ to make $TNN_i$ equal to the value of $minTSD$.
Step 5  Embed $S_i$ into the effective trees which come from $GR_i$ by computing the corresponding value of $NNR$ and removing the deepest nodes with the amount of $NNR$.
```

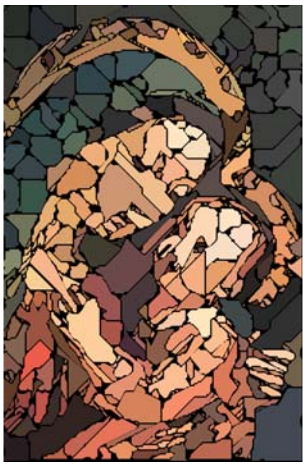
5.6.2 Authentication Signal Extraction Process

To verify a stained glass image, we first compute a sequence of authentication signals $S = S_0, S_1, \ldots, S_n, S_{n+1}, \ldots, S_f$ of each glass region by utilizing the formula described in Step 2 of Algorithm 5.3. To compute S_j where $n < j \leqq f$, we have to apply the gap filling process after glass feature detection for deriving the glass region GR_j. Assume the authentication signals extracted from the effective trees to be $P = P_0, P_1, \ldots, P_k$, where $k \leqq 4f$. If there is any signal p of P, which is extracted from an effective tree of the glass region RG_i, different from S_i, we claim that the glass region GR_i is tampered with.

5.6.3 Experimental Results and Summary

Some experimental results are shown is this section. Fig. 5.10 (a) is a stained glass image with authentication signals embedded. Fig. 5.10 (b) is the verification result of (a). The blue polygons are the detected results which indicate that the bounded glass regions are not tampered with. The polygons of darker blue are detected in the detection process. The polygons of lighter blue are detected in the additional gap filling process after the detection process. The white rectangles within the polygons of darker blue are the seeds generated by a secret key. The white rectangles within the polygons of lighter blue are the seeds generated in the gap filling process.

Fig. 5.11 is a damaged image of Fig. 5.10 (a). We bound the glass regions which are tampered with by a red rectangle. In the rectangle, the region color of the glass region is modified. And some strokes are added in Fig. 5.11. Fig. 5.12 is the verification result of Fig. 5.11. The red areas are found regions which have been tampered with. By Fig. 5.11, we can see that the verification result can indicate the changed areas properly.


(a)

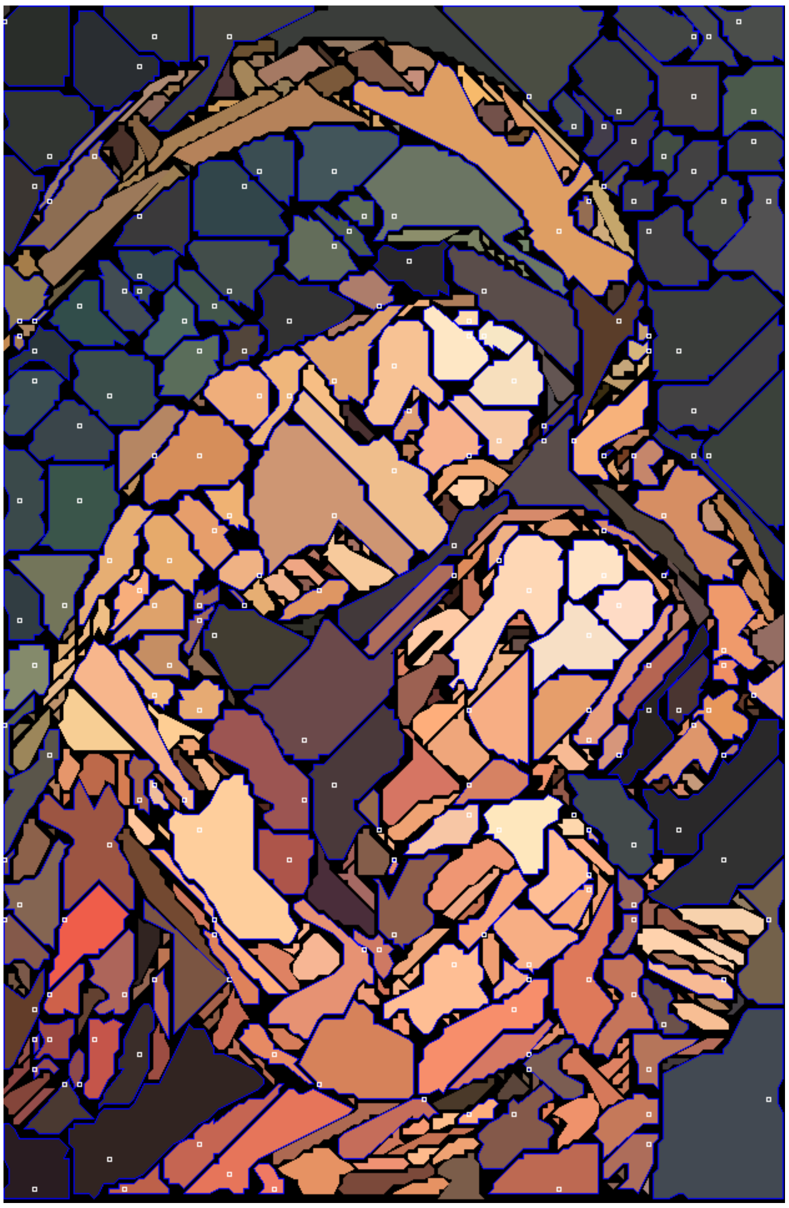

(b)

Fig. 5.10 Experimental results of authentication. (a) A stained glass image with authentication signals embedded. (b) An authentication image of (a).

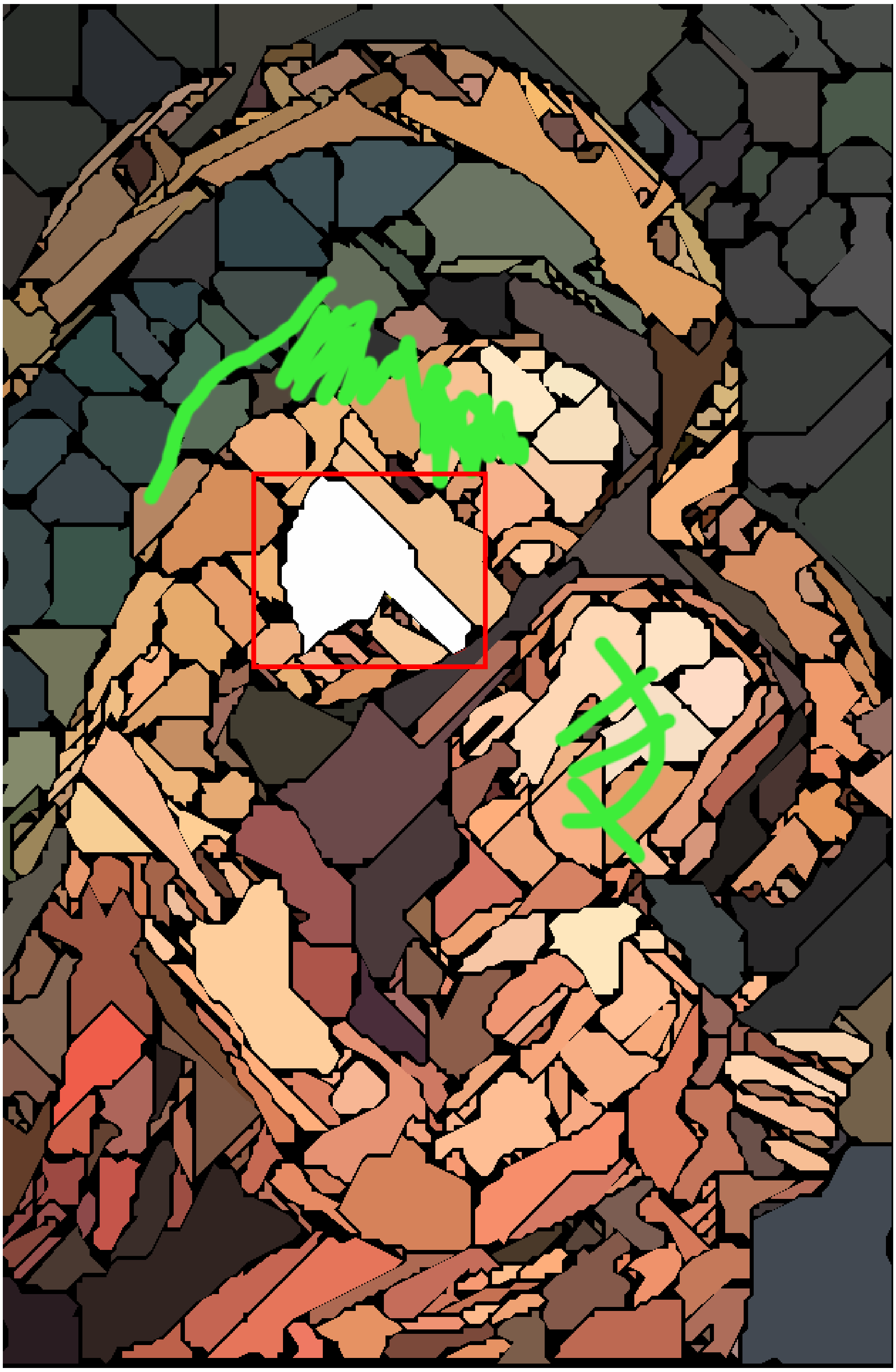


Fig. 5.11 A damaged image of Fig. 5.10 (a).

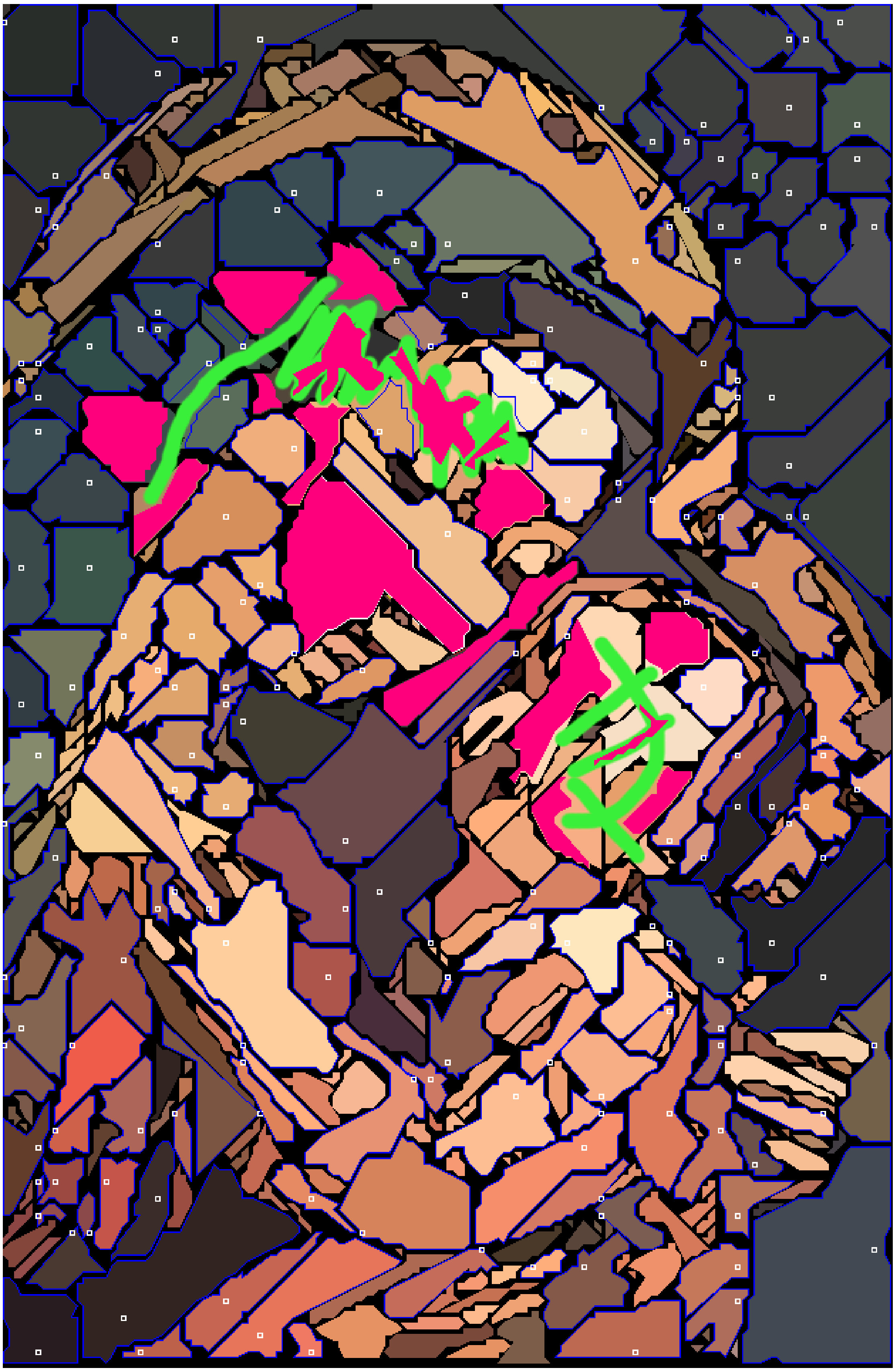


Fig. 5.12 An authentication image of Fig. 5.11.

Chapter 6
Conclusions and Suggestions for Future Works

6.1 Conclusions

In this study, we have proposed methods for art image creation and image data hiding. These two topics are integrated into one which is then solved by a single approach in the proposed methods, so that a common user can easily generate art images and embed data in them. The embedded data may be a watermark, a secret message, or some authentication signals. By embedding the given data, a user can achieve the purposes of copyright protection, covert communication, and image authentication, respectively or integrally. Unlike traditional image hiding techniques, we hide data in the individual features of art images.

In this study, we investigated two different types of art images. They are tile mosaic image and stained glass image, both of them being mosaic-effect images.

In tile mosaic images, we found three different types of image features into which data can be embedded, namely, tile orientation, tile size, and tile texture. In the case of utilizing tile orientations, some erroneous bits may be produced when conducting the associated detection process. However, a watermark can still be recognized visually under the condition of two or three erroneous pixels. In the case of utilizing tile sizes, no bit error results in the detection process according to our experimental results. Due to the high detection accuracy of using the tile size as a feature, a secret message, which can tolerate no erroneous bit while being extracted during the detection process, will be embedded into tile sizes. In the case of utilizing tile textures, the method has the highest data embedding capacity among the uses of the three tile features. So authentication signals, which need high embedding capacity to lower the probability of missing to find out an attacked tile, will be embedded into tile textures. In the proposed system, we can sequentially embed data into the three features as needed.

In stained glass images, we use only one feature for data hiding, namely, the number of tree nodes. So, only one kind of data can be embedded at a time, i.e., only one of a watermark, a secret message, and a set of authentication signals can be embedded. By removing the nodes of a tree contained in a glass region, we can achieve the purpose of data hiding. However, the proposed method will yield slight cracks at the edge of glass regions, but the result is still acceptable. Although only one glass feature is utilized for data hiding, the data embedding process need only be changed slightly for each of the three types of embedded data. In the case of watermarking, since there may be at most four effective trees in a glass region, we embed one bit code into a glass region. Because we embed bit codes repeatedly, additional robustness can be achieved. In the case of secret communication, we embed one bit code into one effective tree, so that the maximum hiding capacity can be achieved. In the case of image authentication, authentication signals are embedded in each glass region repeatedly, so the authentication results are more sensitive than tile mosaic images.

According to our research, we can claim that if there is an image feature of an art image that can be modified and detected, there will be a corresponding data hiding technique. And the three applications of data hiding, copyright protection, covert communication, and image authentication, can thus all be achieved.

6.2 Suggestions for Future Works

In this study, we have proposed some methods for creation of tile mosaic images and stain glass images, as well as data hiding techniques for these two types of images. The three applications of copyright protection, secret communication, and image authentication are also be carried out by utilizing the proposed data hiding techniques. However, there are still some interesting topics which are worth further study.

For tile mosaic images:

1. Creating tiles with different sizes or different shapes to make the tile mosaic image look more appealing.
2. Moving the tiles away from the edges in the creation process and finding methods to detect them.
3. Using more tile features such as tile position, tile border, etc. for embedding more data into tile mosaic images.
4. Applying tile mosaic images on words or characters and geometric totems.

For stained glass images:

1. Creating smoother glass regions and leadings.
2. Revealing the edges by more complicated image analysis of an original image.
3. Adding visual effects, such as burnish, texture, etc, onto the glass regions to make the stained glass image look more appealing.
4. Using more glass features such as glass texture and glass color, for embedding more data into tile mosaic images.
5. Enlarge the embedding capacity by applying more than 4 trees in a glass region.

For all art images:

1. Extending the proposed techniques to other types of mosaic-effect images, such as geometrics totems and wall papers.
2. Creating other types of art images and searching for the specific image features for data hiding.
3. Data hiding in art images with overlapping strokes.
4. Keeping the image quality good after data hiding.

References

[1] A. Hertzmann, “A Survey of Stroke-Based Rendering,” *IEEE Computer Graphics and Applications*, vol. 23, no. 4, issue 4, July-Aug. 2003, pp. 70-81.

[2] A. Hertzmann, “Painterly Rendering with Curved Brush Strokes of Multiple Sizes,” *Proceedings of SIGGRAPH 98*, Orlando, Florida. July 1998, pp. 453-460.

[3] A. Hertzmann, “Fast Paint Texture,” *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering*, Annecy, France, June 3-5, 2002. pp. 91-96, 161.

[4] A. Hertzmann, “Paint by Relaxation,” *Proceedings of Computer Graphics International 2001*, Hong Kong, July 3-6 2001, pp. 47-54.

[5] P. E. Haeberli, “Paint by Numbers: Abstract Image Representations,” *Proceedings of SIGGRAPH 90*, August 1990, pp. 207-214.

[6] A. Secord, “Non-Photorealistic Rendering with Small Primitives,” *M. S. Thesis*, Department of Computer Science, Universary of British Columbia, Oct. 2002.

[7] A. Hertzmann and D. Zorin, “Illustrating Smooth Surfaces,” *Proceddings of Siggraph 2000*, ACM Press, 2000, pp. 517-526.

[8] A. Hausner, “Simulating Decorative Mosaics,” *Proceedings of SIGGRAPH* 2001 New York, New York, USA, 2001, pp. 573-580.

[9] K. Hoff, J. Keyser, M. Lin, D. Manocha and T. Culver, “Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware,” *Proceedings of SIGGRAPH* 99, August 1999, pp. 277-286.

[10] S. P. Lloyd, “Least Square Quantization in PCM,” *IEEE Transactions on Information Theory*, vol. IT-28, no. 2, March 1982, pp. 129-137.

[11] E. L. Armitage, “Stained Glass: History, Technology, and Practice,” *Newton*, Charles T. Branford Company, 1959.

[12] J. Osborne, *Stained Glass in England.* Alan Sutton Publishing, Phoenix Mill, 1997.

[13] David Mould, “A Stained Glass Image Filter,” *Proceedings of the 14th Eurographics workshop on Rendering*, Leuven, Belgium, 2003, pp. 20-25.

[14] C. Christoudias, B. Georgescu and P. Meer, “Synergism in low-level vision,” *International Conference on Pattern Recognition 4*, 16 Aug. 2002, pp. 150-155.

[15] D. Comanicu and P. Meer, “Mean Shift: A Robust Approach Toward Feature Space Analysis,” *IEEE Trans. Pattern Anal. Machine Intell. 24*, 4 May 2002, pp. 603-619.

[16] P. Meer and B. Georgescu, “Edge Detection with Embedded Confidence,” *IEEE Trans. Pattern Anal. Machine Intell. 23*, 12 Dec. 2001, pp. 1351-1365.

[17] R. Arthur and J. Weeks, “Fundamentals of Electronic Image Processing,” *SPIE Optical Engineering Press*, Bellingham,1996.

[18] L. Shapiro and G. Stockman, *Computer Vision.* Prentice-Hall, Upper Saddle River, 2001.

[19] D. C. Wu and W. H. Tsai, “A Steganographic Method for Images by Pixel-Value Differencing,” *Pattern Recognition Letters,* Vol. 24, No. 9-10, 2003, pp. 1623-1636.

[20] Y. C. Chiu “A study on digital watermarking and authentication of images for copyright protection and tampering detection,” *M. S. Thesis*, Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China, June 2004..

[21] W. L. Ling “Data hiding in image mosaics,” *M. S. Thesis*, Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, Republic of China, June 2004.

[22] W. L. Lin and W. H. Tsai, “Data Hiding in Image Mosaics by Visible Boundary

Regions and Its Copyright Protection Application against Print-And-Scan Attacks," *Proceedings of International Computer Symposium* 2004, Taipei, Taiwan, Dec. 15-17, 2004.

[23] http://christianity.about.com/library/weekly/aa032002e.htm, the Illuminated Easter.