A New Approach to Automatic Generation of Tile Mosaic Images for Data Hiding Applications

Shi-Chei Hung[1], Tsung-Yuan Liu[1] and Wen-Hsiang Tsai[1, 2]
[1]National Chiao Tung University
[2]Taichung Healthcare and Management University

Abstract

A new approach to automatic generation of tile mosaic images containing square tiles suitable for the information hiding purpose is proposed. The generated tile mosaic image contains no overlapping tiles, and the orientations of the tiles are deterministic. The properties of the generated image allow simple tile feature detection, and result in efficient data embedding and extraction. The orientations of the tiles are utilized to embed data, and in this work an invisible watermark is embedded for the copyright protection purpose. Experimental results demonstrate that the watermark is still recognizable with malicious damages to the copyrighted image.

1. Introduction

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 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. 2(a). Haeberli [1] used voronoi diagrams to create mosaic-effect images by placing seed sites randomly and filling the voronoi regions, as shown in Fig 2(b). However, Haeberli's method does not attempt to follow the edge features in the image and the tiles all have different shapes. Alejo Hausner [2] created tile mosaic images by the use of centroidal voronoi diagrams. Being an extension of Hoff [3], the method draws voronoi diagrams efficiently, and uses the Lloyd's algorithm [4] to produce centroidal voronoi diagrams by moving each seed to the centroid of its voronoi region. There is also research on other forms of mosaic-effect images, such as *image mosaics* [5, 6]. An image mosaic is a two-dimensional array of rectangular grids, within each of which there is a small image shrunk to the grid size. The main task of image mosaic creation is to search in a large database of small images to find ones that match approximately blocks of pixels in the main image. When viewing the resulting image mosaic from a distance, the grid images (or tiles) combine to yield an impressive integrated painting. Fig. 2(d) shows an image of Lena assembled in such a way.

Researches on data hiding aim to embed information imperceptibly into a given media. Data hiding in images are most commonly 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 [7]. The information embedded in an image can be used to protect the copyright of the image, verify the authenticity of the image, and also to convey a secret message. Lin and Tsai [8] proposed a method to hide information in image mosaics by manipulating the four borders of a tile and the histogram of a tile. To the best knowledge of the authors, data hiding in tile mosaics have not been studied before. The tile mosaic images created by previous works, for example, Alejo Hausner's method, are unsuitable for the purpose of data hiding. As shown in Fig. 2(c), the image created by Alejo Hausner's method results in overlapping tiles, and the position of the tiles are nondeterministic, making tile feature detection work in data extraction complicated and unreliable.

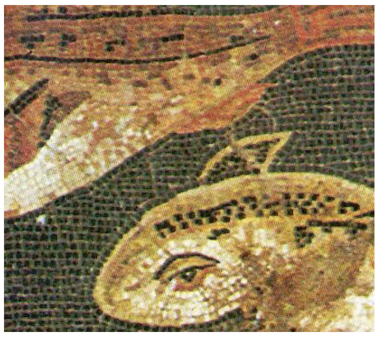


Fig. 1 Detail from "Sea Creatures," exhibited in National Museum of Naples, Italy, 1st century B. C.

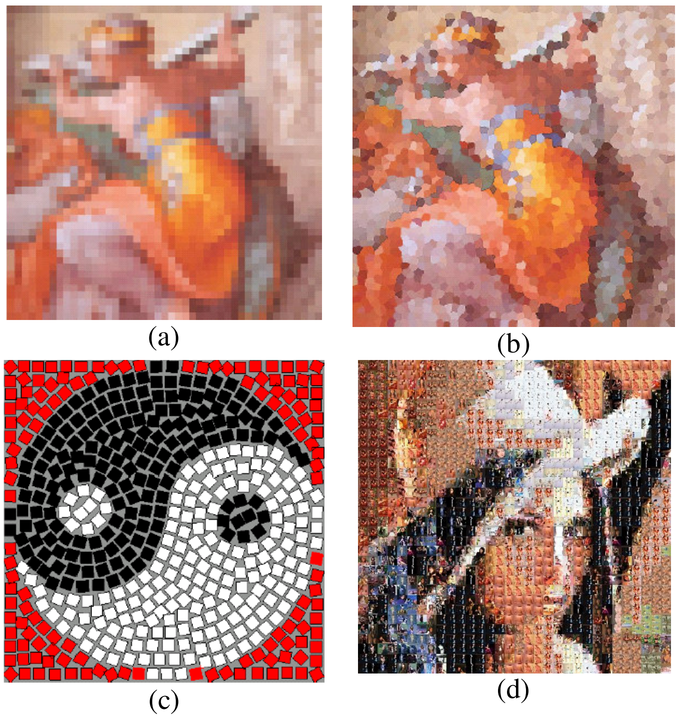


Fig. 2 Several styles of art images. (a) an image mosaic created by Adobe Photoshop; (b) an image mosaic created by Haeberli's method; (c) a tile mosaic image created by Alejo Hausner's method; (d) an image mosaic created by Lin and Tsai's method.

In this study, we propose a new method to create tile mosaic images that are suitable for data hiding. The tiles are squares of identical sizes, and are arranged regularly in the image. The orientations of the tiles are used to embed an invisible watermark, providing copyright protection for the tile mosaic image. The proposed process for tile mosaic image creation is described in Section 2. The proposed technique to detect individual tile orientations in a tile mosaic image is described in Section 3 and the proposed watermarking embedding and extraction processes follow in Section 4. Section 5 concludes this paper with some suggestions for future works.

2. Proposed Tile Mosaic Image Creation Process

The proposed tile mosaic creation process is based on Alejo Hausner's method [2]. 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 1: tile mosaic creation for data hiding.
Input: an image T and an input 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 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. 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. 4) to T and threshold the result into a binary edge-point image.

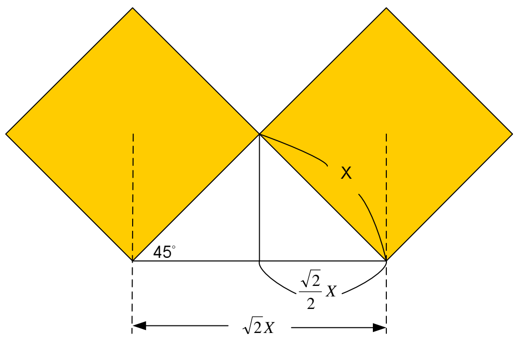


Fig. 3 Illustration of minimum distance between neighboring tiles.

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

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

Fig. 4 3×3 Sobel mask.

After obtaining the edge points of T, we take first in Step 3 an initial neighborhood range of T 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 T.

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.

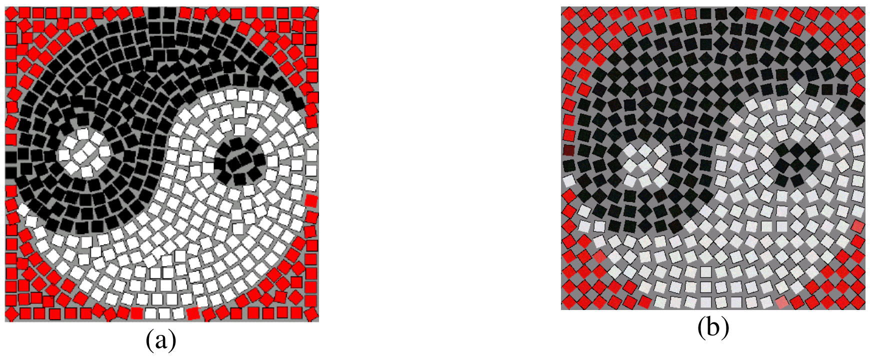


Fig. 5 Tile mosaic image creation. (a) A tile mosaic image by using Alejo Hausner's creation process; (b) a tile mosaic image created by proposed creation process.

As seen from the examples shown in Fig. 5, our creation process creates looser tiles than those created by Alejo Hausner. With looser arrangements of tiles, we can guarantee that there will be no overlapping tile. Furthermore, we notice that the tiles in Fig. 5(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 Alejo Hausner's method, information embedding and extraction become possible and easy.

3. Proposed Tile Orientation Detection Technique

Before introducing the proposed data hiding method, we first describe the proposed tile feature detection method.

Algorithm 2: tile feature detection.
Input: a tile mosaic image *T*.
Output: tile orientations in *T*.
Steps:

Step 1. Derive tile positions in *T* by raster scanning.
Step 2. Derive a tile region map by tile region detection.
Step 3. Derive the four apexes of each tile by tile boundary detection and compute the tile orientation.

3.1 Tile Scanning

In Step 1 of Algorithm 2, 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 *T*, as illustrated in Fig. 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. 6. With accumulation histograms, we can see clearly in the figure the correspondence between the local maximum accumulation values and the grid lines.

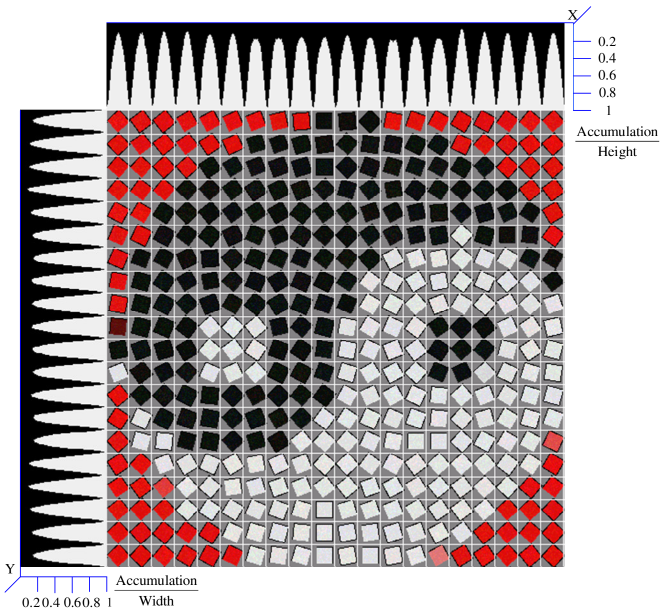


Fig. 6 Finding grid lines with local maximum accumulations.

3.2 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 of Algorithm 2. We use a region growing technique to create a tile region map, 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. 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. 7(b) after we perform *tile boundary detection* in Step 3. 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. 7 (c).

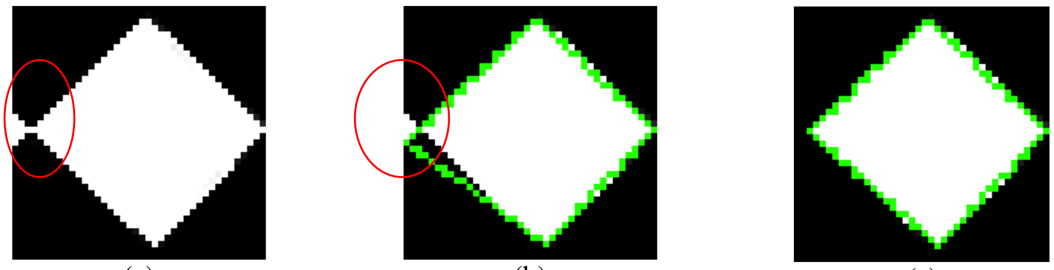

(a) (b) (c)

Fig. 7 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.

3.3 Tile Boundary Detection

The next step of tile orientation detection is the tile boundary detection process, as described in Step 3 of Algorithm 2. 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 presents the proposed tile boundary detection process.

```
Algorithm 3: tile boundary detection.
Input: a tile region map.
Output: four apexes of each tile, $n$, $e$, $s$ and $w$, respectively.
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 tile pixel number, $TPN$, 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
            (A) 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,
            (B) 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 Figs. 8(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 Figs. 8(a) and (b), if the tile is rotated we derive the four apexes by Step 4(B); otherwise, we derive the four apexes by Step 4(A), as shown in Figs. 8 (c) and (d). In Figs. 8(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.

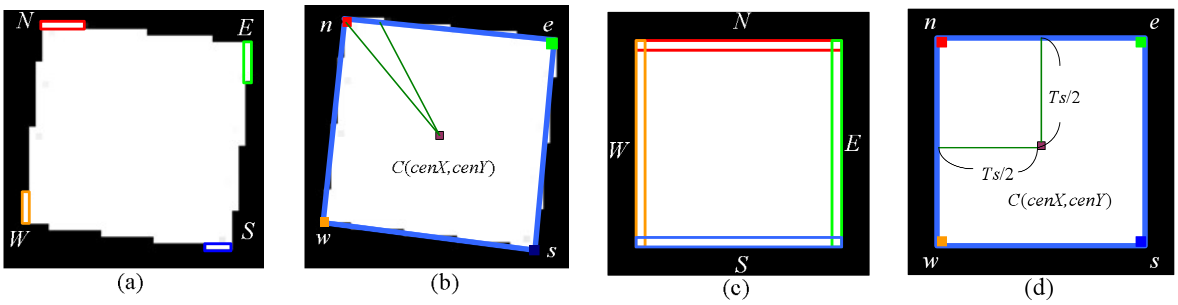
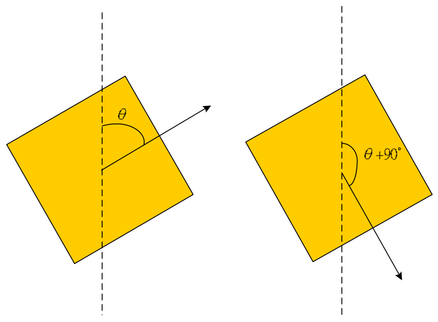


Fig. 8 Boundary detection. (a) Rotated tile with the four detected lists; (b) rotated tile with the detected four apexes and tile boundary; (c) non-rotated tile with the four lists; (d) non-rotated tile with the detected four apexes and tile boundary.

3.4 Tile Orientation Detection

Before we address the tile orientation detection process, we first introduce some properties of tile orientations. As shown in Fig. 9, we can see that a tile rotated with an angle of θ have the same rotation effect as an angle of $\theta+90^{o}$. Thus, the effective range of rotation angles may be limited to be $0 \leqq \theta < 90^{o}$.



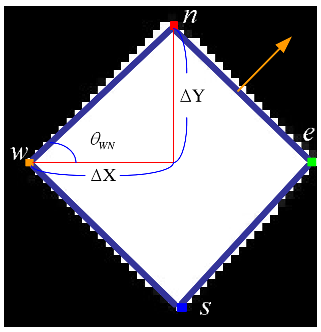


Fig. 9 Properties of tile orientations.

Fig. 10 Illustration of tile orientation detection.

The way we detect the tile orientation is to compute the directions of the four boundaries derived from the tile boundary detection process. As shown in Fig. 10, tile orientation $\theta_{WN} = \arctan(\Delta Y/\Delta X)$, and tile orientations θ_{NE}, θ_{ES}, θ_{SW} are computed similarly. In order to determine the tile orientation with more accuracy, the directions of the four boundaries are averaged, that is, the final tile orientation θ is obtained by the formula: $\theta = (\theta_{NE} + \theta_{ES} + \theta_{SW} + \theta_{WN})/4$.

4. 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 discuss 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, where slight modification of tile orientations is used to achieve our purpose of hiding data in tile mosaic images. Due to the characteristic of tile orientation detection, some errors may occur when we conduction tile orientation detection. The errors are mostly smaller than 3^o, but it may still result in bit errors in embedded 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 embedded 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 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 error points. 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.

4.1 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 Figs. 11(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, presenting 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 11(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. 11(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.

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.

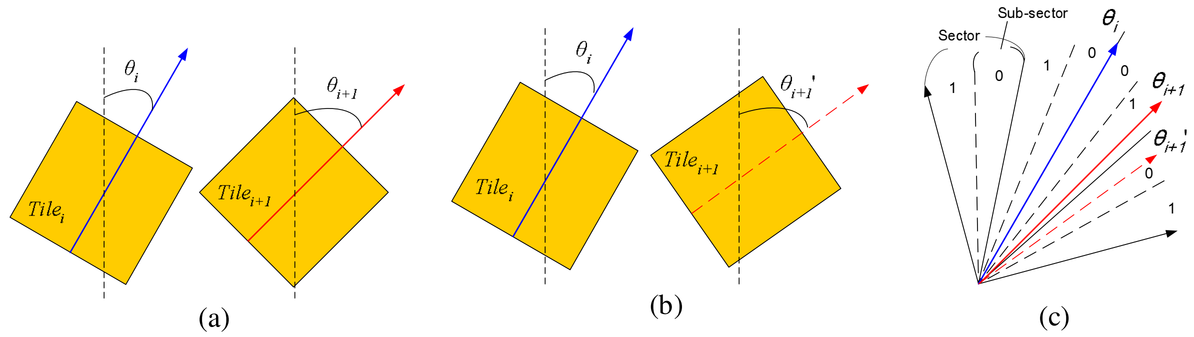


Fig. 11 Data hiding by tile orientation modification. (a) Two adjacent tiles; (b) tiles with bit 0 embedded in $Tile_{i+1}$; (c) data hiding strategy.

4.2 Data Embedding Process

The procedure of embedding a watermark into a tile mosaic image is shown in Fig. 12 and described in Algorithm 4 below.

Algorithm 4: 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 Til_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. 11(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.

4.3 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. 13. First, we derive the tile orientations, θ_0, θ_1, …, θ_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.

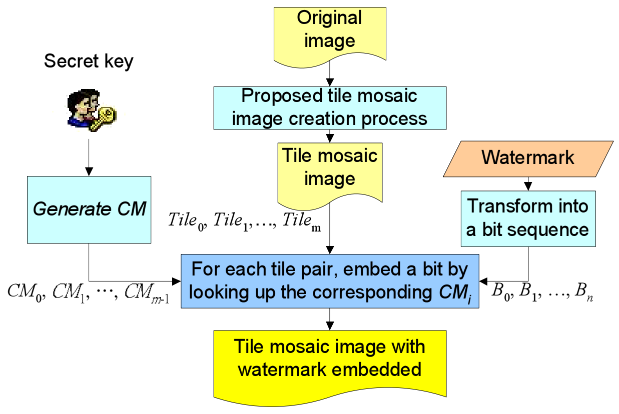

Secret key
Original image
Proposed tile mosaic image creation process
Tile mosaic image
Watermark
Transform into a bit sequence
Generate CM
$Tile_0$, $Tile_1$,…, $Tile_m$
CM_0, CM_1, ⋯, CM_{m-1}
For each tile pair, embed a bit by looking up the corresponding CM_i
B_0, B_1, …, B_n
Tile mosaic image with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

Fig. 12 Watermark embedding process.

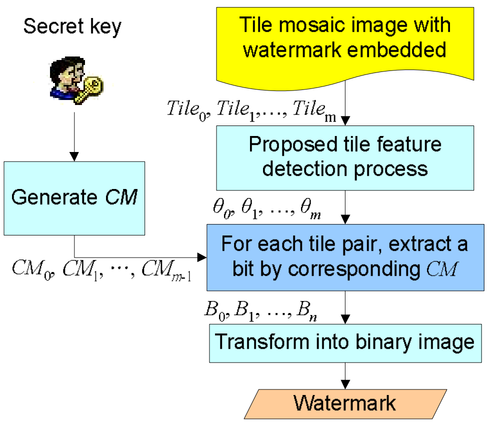

Secret key
Tile mosaic image with watermark embedded
$Tile_0$, $Tile_1$,…, $Tile_m$
Proposed tile feature detection process
Generate CM
θ_0, θ_1, …, θ_m
CM_0, CM_1, ⋯, CM_{m-1}
For each tile pair, extract a bit by corresponding CM
B_0, B_1, …, B_n
Transform into binary image
Watermark

Fig. 13 Watermark extraction process.

4.4 Experimental Results

The proposed method was tested on a series of images. Some experimental results are shown in this section. In Fig. 14, (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. 14(f) is the watermark extracted from (c). Because we resize the watermark before embedding it, Fig. 14(f) is a scaled-down version of Fig. 14(e). Fig. 14(g) is the watermark extracted from Fig. 14(c) with a wrong key. From Fig. 14(g), we can see that the watermark cannot be extracted by a wrong key. Fig. 14(d) is a damaged image of Fig. 14(c) with a mark on it, and Fig. 14(h) is the extracted watermark from Fig. 14(d). From Fig. 14(d) and Fig. 14(h), we can see that the watermark can still be recognized under a certain degree of damaging.

5. Conclusions and Future Works

A tile mosaic creation process and a watermarking method by tile orientation modification have been proposed. The resulting watermark can survive a certain degree of malicious damages, thus useful for protecting copyrights of tile mosaic images. The proposed method may also be applied to certain other types of art images if the images contain features which can be used for data hiding and can be detected in the data extraction process. In the current approach to tile mosaic creation, only square tiles with the same size are used. We may create tiles with different sizes or different shapes to make the tile mosaic image look more appealing. Textures can also be added into tiles to improve the impressions to spectators. Another research topic is moving the tiles away from the edges. This will make the edges in the resulting tile mosaic images much clearer. However, this will also make tile feature detection more difficult. Finally, to embed more data into tile mosaic images, we can use more tile features, like tile size, tile texture, tile border, etc. If we can design the corresponding feature detection process, more data can be embedded.

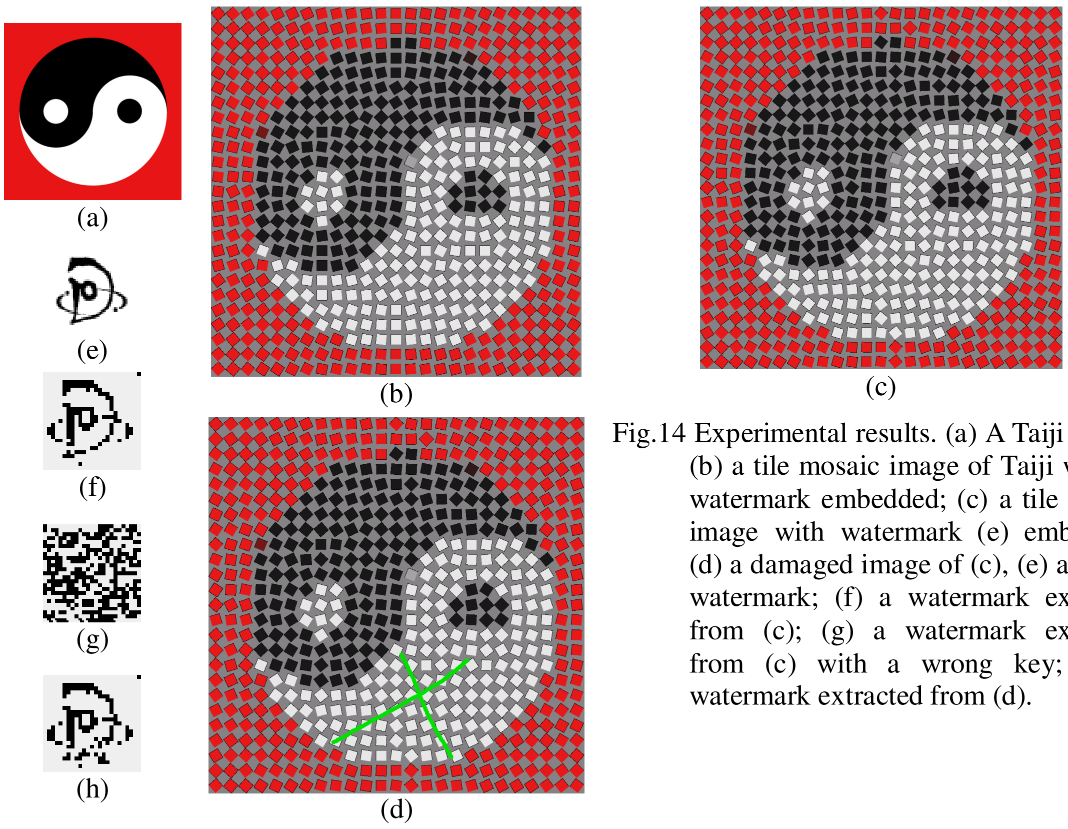


Fig.14 Experimental results. (a) A Taiji image; (b) a tile mosaic image of Taiji without watermark embedded; (c) a tile mosaic image with watermark (e) embedded; (d) a damaged image of (c), (e) an input watermark; (f) a watermark extracted from (c); (g) a watermark extracted from (c) with a wrong key; (h) a watermark extracted from (d).

Acknowledgement

This work was supported by the NSC Project, Advanced Technologies and Applications for Next Generation Information Networks (II), with Project No. NSC93-2752-E-009-006-PAE.

References

[1] Paul E. Haeberli, "Paint by Numbers: Abstract Image Representations," *Proc. of SIGGRAPH*, August 1990, pp. 207-214.

[2] Alejo Hausner, "Simulating Decorative Mosaics," *Proc. of SIGGRAPH*, New York, New York, USA, 2001, pp. 573-580.

[3] Hoff, K., Keyser, J., Lin, M., Manocha, D. and Culver, T. "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware," *Proc. of SIGGRAPH*, Aug. 1999, pp. 277-286.

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

[5] R. Silvers, and M. Hawley, *Photomosaics*, Henry Holt and Co., New York, USA, 1997.

[6] A. Finkelstein and M. Range, "Image Mosaics," *Proc. of EP'98 and RIDT'98*, St. Malo, France, March 1998, pp. 1042-1052.

[7] 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.

[8] 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," *Proc. of International Computer Symposium* 2004, Taipei, Taiwan, Dec. 15-17, 2004.