Complete Quality Preserving Data Hiding in Animated GIF with Reversibility and Scalable Capacity Functionalities

KokSheik Wong[1(✉)], Mohamed N. M. Nazeeb[1], and Jean-Luc Dugelay[2]

[1] Monash University Malaysia, Subang Jaya, Selangor, Malaysia
wong.koksheik@monash.edu, mnmoh48@student.monash.edu
[2] EURECOM, Sophia Antipolis, France
jean-luc.dugelay@eurecom.fr

Abstract. A technique is put forward to hide data into an animated GIF by exploiting the transparent pixels. Specifically, a new frame is crafted based on the data to be embedded. The newly crafted frame is inserted between 2 existing frames, and the delay time of the affected frames are adjusted accordingly to achieve complete imperceptibility. To the best of our knowledge, this is the first attempt to hide data into an animated GIF by exploiting the transparent pixel. Irregardless of the characteristics of the animated GIF image, the proposed method can completely preserve the quality of the image before and after hiding data. The hiding capacity achieved by the proposed method is scalable, where more information can be embedded by introducing more frames into the animated GIF. While file size expansion is inevitable, reverse zero run length is adopted to suppress the expansion. The proposed method is reversible, i.e., the original image can be recovered.

Keywords: Transparent pixel · Animated GIF · Complete quality preservation · Data hiding · Reversible

1 Introduction

Graphic interface format (GIF) is a highly portable and platform-independent image file format designed to show moving pictures through low bandwidth Internet. It was developed by CompuServe in 1987, where further innovations such as *dirty rectangular* and *transparent pixel* took place after the disclosure of the GIF 89a specifications [1]. Although animated GIF contains no sound/voice, the short visual content shows dynamic content, tells story, and captures emotion [4].

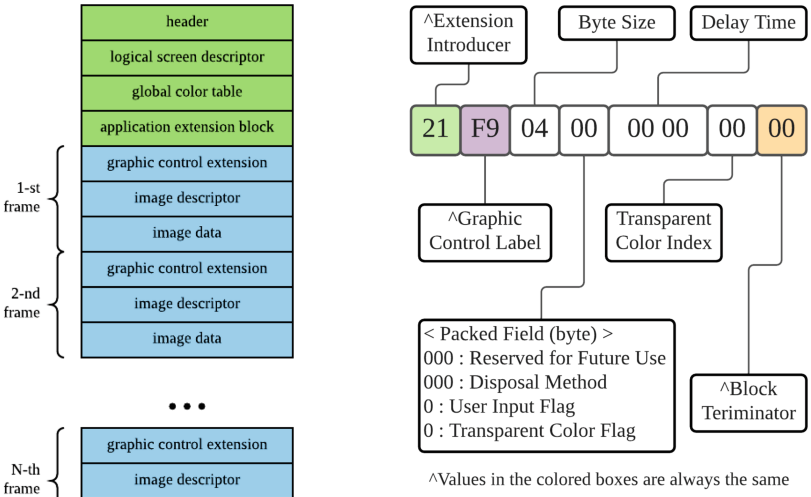
The popularity of animated GIF has been decaying, but recently social networking service platforms and online advertisers are making good use of animated GIFs despite broadband network connectivity. These creative utilizations of animated GIF give new life to the originally dull image, including the transition of different combinations of outfit/shoes on the same model, handbag of a specific model in different colors, to name a few. Furthermore, animated GIFs

can be easily generated nowadays thanks for the availability of freely available encoder in many platforms, including online websites. There are also dedicated websites to blog about, share, search, and create animated GIFs [2,5]. Moreover, users also use animated GIFs in instant messaging platform and online forum to show reactions or emotions.

Due to its popularity and large number in existence, many data hiding methods are designed to better manage GIFs over the years. Traditionally, data is hidden into a digital content such as image to convey secret message [6,10]. One of the earliest techniques designed for GIF is proposed by Kwan, where the color palette is arranged in certain way to convey a secret message [8]. However, the hiding capacity is low. In another technique called EzStego, Machado [9] proposed to analyze the color palette of a GIF image and sort the indices based on luminance. If an index needs to be replaced for hiding data, the nearby indices (post- sorting) are considered. Later, Fridrich et al. [3] proposed to match the parity of the sum of RGB triplet values to the data bit. The nearest RGB triplet with matching sum is selected to represent the message bit. Data can also be hidden without causing any distortion (i.e., complete quality preservation [16]), but the requirement is to start with a GIF with at least 1 un-referenced indice. Kim et al. is able to hide up to 8 bits per pixel without causing distortion when there at least 128 un-referenced indices [7]. Recently, Wang et al. put forward a technique to quantize colors in GIF [14]. Two similar colors C1 and C2 in the color palette are combined by taking their weighted average to generate a new color, where the notion of similarity is defined by some *risk* function. Pixels having the index value of C1 or C2 are manipulated to hide data.

Although there are techniques designed to hide data into animated GIF, they are treating each frame as a static image, where existing techniques such as EzStego [9] and Fridrich et al.'s method [3] are deployed to hide data into the selected frames. In other words, the conventional techniques either modify the pixel index, color table entries, or the combination of both, where distortion is inevitable. In spite the fact that LZW compression is exploited to hide data in GIF [12], other parts of the GIF structure remain unexplored, particularly the parts related to *animation* in GIF. Therefore, in this work, we propose to hide data into an animated GIF file, where new frames are crafted based on the data to be hidden. To the best of our knowledge, our technique is the first of its kind to hide data by inserting new frames and using transparent pixel.

While the conventional techniques surveyed above are mostly designed for steganography, our proposed method can be utilized in the applications of data hiding such as fragile watermark for tamper detection and annotation. In addition, one may envisage a spectacular demo by using the proposed method in animated GIF thanks to its scalable capacity functionality. For example, a binary animation can be hidden in an animated GIF, which is apparently normal.



(a) File structure

(b) Graphic Control Extension. Reproduced from [11].

Fig. 1. File structure of animated GIF and its graphic control extension.

2 Overview of GIF File Structure

Figure 1(a) shows the structure of a GIF file, which consists of protocol blocks (for set-ups) and sub-blocks of graphics. Specifically, an animated GIF A of dimension $M \times N$ pixels consists of an assemble of frames A_f so that $A = \{A_f\}$ for $f = 1, 2, \cdots, F$, where F is the total number of frames. The logical screen descriptor contains information such logical screen width and height, background color index, etc. [1]. On the other hand, the global color table consists of 256 entries of RGB-triplets, and there is a function C that maps index to integer RBG-triplet, i.e., $C : [0, 255] \rightarrow [0, 255] \times [0, 255] \times [0, 255]$. To facilitate the presentation, let $A_f(x, y) \in \{0, 1, \cdots, 255\}$ denote the index at position (x, y) within frame A_f, where $1 \leq x \leq M$ and $1 \leq y \leq N$. Each frame A_f consists of three data blocks, namely: graphic control extension, image descriptor, and image data - see Fig. 1(b) [11].

Next, we focus on *Transparent Color Flag* (TCF) within the *Packed Field* and *Transparent Color Index* (TCI). When TCF is set to TRUE, it enables an index to be utilized as the transparent pixel, where color from a previous frame is rendered instead of the color associated with the index. When TCI $= \tau_f =\leftarrow$ 169 for example, the index '169' is reserved and utilized for transparent pixel. Therefore, if $A_f(x, y) = 169 = A9_{16}$, the color $C(169)$ (i.e., triplet of RBG value) will not be displayed at position (x, y) in frame A_f. Instead, the color from the same location in the previous frame, i.e., $A_{f-1}(x, y)$, will be rendered. The transparent pixel concept is introduced for compression purposes. Although its performance varies depending on the characteristics of the animated GIF, a

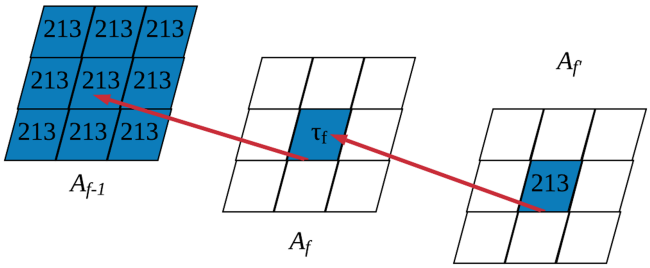


Fig. 2. Illustration of $A_{f'}(x_0, y_0)$ referring to $A_f(x_0, y_0) = \tau_f$, i.e., transparent pixel. The actual color (i.e., index 213) is traced and retrieved from A_{f-1}. Here, (x_0, y_0) refer to the center of the 3×3 image block.

compression ratio of 1.63:1 is reported in [13]. On the other hand, the disposable method informs the decoder what to do with the current frame A_f when the decoder moves on to the next frame A_{f+1}. A value of '0' implies that the image is static and the decoder cannot draw anything on top of it. This value is used for non-animated (i.e., static) GIF. On the other hand, a value of '1' informs the decoder to leave the current image on screen and draw the next image on top of it. There are other modes of operation but we omit the presentation here due to space limitation. The length of display for each frame A_f is controlled by using the value as specified in the *Delay Time* field (denoted by d_f). Basically a frame A_f will stay on the screen for d_f centi-seconds (i.e., 1/100 of a second). For the purpose of this work, we set TCF to TRUE, and use '1' for the disposable method.

3 Proposed Data Hiding Method

In this section, we first propose a pre-processing step to prepare the animated GIF A for data hiding purpose. The actual data hiding and extraction processes are then put forward. Finally, we explain how reverse zerorun length encoding [16] is adopted to overcome the problem of file size increment.

3.1 Pre-processing

A new frame $A_{f'}$ is created and inserted between A_f and A_{f+1} to facilitate data hiding. Each pixel index $A_{f'}(x, y)$ is eventually modified to hide data. Specifically, a new frame $A_{f'}$ is created by copying all indices from A_f, i.e.,

$$A_{f'}(x, y) \leftarrow A_f(x, y). \tag{1}$$

Here, the same indices (hence colors) are copied from A_f to ensure imperceptibility of the newly inserted frame $A_{f'}$. However, $A_{f'}$ requires additional treatment when there is at least one transparent pixel occurring in A_f, or more precisely, $|\{(x, y) : A_f(x, y) = \tau_f\}| > 0$, where $|X|$ refers to the cardinality of the set X.

Specifically, due to the simple duplication process of Eq. (1), an issue arises when $A_f(x_0, y_0) = \tau_f \in [0, 255]$, where the index τ_f is defined as the transparent pixel in frame A_f. In other words, $A_f(x_0, y_0) = \tau_f$ means that an actual color in an earlier frame, i.e., $A_\alpha(x_0, y_0)$, is referred for display, where $\alpha < f$. In the event we set $\tau_{f'} \neq \tau_f$ (i.e., we use different indices to define the transparent pixels in A_f and $A_{f'}$), the actual color of $C(\tau_f)$ will be displayed at $A_{f'}(x_0, y_0)$, instead of the color in an earlier frame, namely, $A_\alpha(x_0, y_0)$.

To overcome the aforementioned issue, the main objective of the pre-processing is to eliminate *all* transparent pixels in the newly created frame $A_{f'}$, where every occurrence of the transparent index τ_f will be substituted by the actual color (i.e., a RGB-triplet referred by an index) in the earlier frame A_α for $\alpha < f$. Figure 2 illustrates a scenario where $A_f(x_0, y_0) = \tau_f$ is a transparent pixel, which refers to the color shown at position $A_{f-1}(x_0, y_0)$. Hence, the actual color $C(213)$ shown at position $A_{f-1}(x_0, y_0)$ is traced and copied, in other words, $A_{f'}(x_0, y_0) = 213$. The process is repeated to eliminate all transparent pixels in $A_{f'}$. Eventually, the newly added frame $A_{f'}$ consists entirely of indices to actual RGB-triplets without any transparent pixels. In other words, $|\{A'_f(x, y) = \tau_f\}| = 0$.

Next, we have 2 scenarios to manage, namely, A_f has a defined transparent pixel, and A_f does not have a defined transparent pixel. For the former scenario, we continue to utilize the same transparent pixel, i.e, $\tau_f \leftarrow \tau_f$, instead of finding another index for such purpose. On the other hand, for the latter situation, we have to choose the transparent pixel $\tau_{f'}$ carefully. Specifically, all indices in frame A_f are scanned and the histogram H_f of the indices is constructed. Let $H_f(i)$ denote the frequency of occurrences for the index i, where $0 \leq i \leq 255$. We select the index i_0 such that $H_f(i_0)$ is the minimum (i.e., occurring the least in A_f), and set $\tau_{f'} \leftarrow i_0$. Note that in practice, a GIF image does not utilize all 256 indices. Therefore, in general, $H_f(i_0) = 0$ holds true and we can hide 1 bit per pixel (bpp). On the other hand, when $H_f(i_0) > 0$, we skip the positions $A_{f'}(x, y)$ (i.e., newly added frame) for data hiding when $A_f(x, y) = i_0$ (i.e., original frame). Here, we loose exactly $H_f(i_0)$ number of pixel locations for data hiding, and the embedding reduces to $1 - H_f(i_0)/M/N$ bpp.

In both cases, data hiding can take place, where defining a new transparent pixel will not confuse the extraction process with the introduction of *usable* and *non-usable* positions in Sect. 3.2.

3.2 Data Hiding

To hide data, the new frame $A_{f'}$ is compared with A_f at each pixel location. Specifically, the position $A_{f'}(x, y)$ is skipped and we call it *non-usable* if the following two conditions are true simultaneously:

$$\tau_f \neq \tau_{f'} \tag{2}$$

$$A_f(x, y) = \tau_{f'}. \tag{3}$$

Note that such a decision is made to avoid ambiguity during data extraction because due to the simple duplication process (i.e., Eq. (1)), we cannot

differentiate whether $A_{f'}(x, y) = \tau_{f'}$ is encoding '0' (see Eq. (4)), which is modified from $A_f(x, y)$, or it is actually the original index for that pixel location. Therefore, we skip these positions.

On the other hand, $A_{f'}(x, y)$ is called *usable* and it is exploited to hide data by using the basic rules below:

$$A_{f'}(x, y) \leftarrow \begin{cases} \tau_{f'} & \text{if } m_k = 0; \\ \text{'No change'} & \text{otherwise.} \end{cases} \tag{4}$$

Here, the payload m is a binary sequence $\{m_k\} \in \{0, 1\}$. The encoding rule basically utilizes the transparent pixel index to encode '0', and utilizes the original index to encode '1'. The process is repeated for each position (x, y) in the frame $A_{f'}$ in the raster scanning order.

In order to maintain the length of the original animated GIF, the delay time for frame A_f and $A_{f'}$ need to be adjusted. Specifically, we set $d_{f'} \leftarrow \lfloor d_f/2 \rfloor$, where $\lfloor z \rfloor$ refers to the largest integer smaller than or equal to z. Next, we update $d_f \leftarrow d_f - d_{f'}$. Essentially, the proposed method splits A_f into 2 frames, both having the exact same pixel values on screen, and the overall display duration (i.e., delay time) remains unchanged. Since the exact same pixel values are displayed for the same duration, the quality is completely preserved. In other words, the pixel values rendered from the original and processed (embedded with data) animated GIF images are exactly the same, and these pixels appear on the screen for exactly the same duration. In fact, the duration d_f and $d_{f'}$ can be further manipulated to hide data, which will be explored as our future work.

By inserting a new frame between every 2 consecutive original frames, we are increasing the number of frames from F to $2F - 1$. In fact, to improve hiding capacity, more new frames can be generated and inserted between any two consecutive frames, including the pairs A_f and $A_{f'}$ as well as $A_{f'}$ and A_{f+1}. This process can be repeated as long as all delay times (i.e., d_f and $d_{f'}$) remain ≥ 0.02s, which is the smallest permissible value allowed by web browser.

3.3 Data Extraction and Reversibility

To extract data from the animated GIF embedded with data A', the inserted frames $A'_{f'}$ are first identified. This process can be achieved by some pre-arrangement, for example, a new frame is always added between 2 original frames (i.e., A_f and A_{f+1}), and hence the odd numbered frames in A' are the newly inserted frames. The status (being *usable* or *unusable*) of each position in $A'_{f'}$ is verified by referring to Eq. (2) and (3). The sequence of embedded bits in $A'_{f'}$ is extracted from the *usable* locations by producing '1' when $A'_{f'}(x, y) = A'_f(x, y)$ or '0' when $A'_{f'}(x, y) = \tau_{f'}$. The process is repeated for all inserted frames $A'_{f'}$.

The proposed method is obviously reversible, where the newly added frames $A_{f'}$ can be removed and the original delay time d_f can be reassigned to recover the original animated GIF image.

3.4 Reducing File Size Increment

When a new frame is inserted, file size is inevitably increased. To reduce file size expansion, the *reverse zerorun length* (RZL) encoding technique [16] is adopted. Note that for each newly created frame $A_{f'}$, prior to any modifications due to data hiding purposes, $A_{f'}$ encodes a sequence of 1's with length $M \times N$ (or slightly lesser depending on $H_f(\tau_{f'})$ in A_f). Instead of using Eq. (4) to hide data directly, the message is first pre-processed. Specifically, for each newly created frame $A_{f'}$, the data to be hidden ϕ_f is divided into segments each of length k-bits, i.e., $\phi_f = [\phi_f^1, \phi_f^2, \cdots, \phi_f^D]$ where $D = |\phi_f|/k$. Here, each segment ϕ_f^k is of length k bits except for ϕ_f^D, which can assume a length $\leq k$ bits.

Next, the decimal equivalent of ϕ_f^i, denoted by d_f^i, is computed and hence $0 \leq d_f^i \leq 2^k - 1$. Subsequently, d_f^i is utilized to generate a new segment μ_f^i for $i = 1, 2, \cdots, D$. The segments μ_f^i are generated as follows:

$$\mu_f^i = \underbrace{00\cdots0}_{d_f^i}1, \tag{5}$$

which is sequence of d_f^i zeros, followed by a '1' that serves as a delimiter. Note that μ_f^i is of variable length. The new representation of the message, i.e., μ_f^i, is then embedded by using Eq. (4). If the newly inserted frame $A_{f'}$ is unable to hide all segments μ_f^i, a new frame $A_{f''}$ can be inserted between $A_{f'}$ and A_{f+1} to create more room for data hiding.

To extract the hidden data segment encoded in the RZL format, the sequence of 0's and 1's are first extracted from all *usable* pixel locations, where $A'_{f'}(x, y) = \tau_{f'}$ outputs a '0', otherwise a '1'. The extracted sequence is then analyzed, where the number of 0's preceeding the value 1 is counted and converted into a binary number with k-bits. For example, the following sequence of 19 bits are extracted from 19 *usable* pixel locations:

$$\underbrace{00000}_{5}1\underbrace{00000000}_{8}1\underbrace{000}_{3}1. \tag{6}$$

The corresponding decimal values 5, 8 and 3, are converted into binary numbers 101, 1000 and 11, respectively. Finally, leading zeros are injected to make up the number of bits (i.e., length) for each segment. Suppose $k = 6$, then the segments become 000101, 001000, and 000011, respectively.

4 Experiments

The proposed data hiding method is implemented in Python. 8 animated GIFs from the world wide web are considered for experiment purpose, where the first frame of each animated GIF is shown in Fig. 3. These GIFs are either generated by using graphic software or merging frames/scenes from video recording.

(a) Mov01

(b) Mov02

(c) Mov03

(d) Mov04

(e) Mov05

(g) Mov06

(j) Draw1

(k) Draw2

Fig. 3. First frame of each animated GIF considered for experiment.

Additional information of these animated GIFs can be found in Table 1. They are also made available online at [15] for reproducibility and future comparison purpose. Google Chrome (version 73.0.3683.103), Safari (version 12.0.2), Firefox (version 66.0.3) and Photos (system viewer for Windows 10) are utilized to display the animated GIFs. The processed animated GIFs can be viewed by using the aforementioned browsers, and this observation also confirms that the processed images are format compliant. It is verified that the hidden data can be extracted by checking the status (i.e., *usable* or *non-usable*) of each pixel locations using Eq. (3). By visual inspection, the GIFs appear to be identical before and after hiding data. Unless specify otherwise, $F-1$ new frames are introduced to an animated GIF with F frames. Although frames of different sizes can be created, for experiment purposes, the dimension of each new frame A'_f is set to be the same as that of the original frame A_f in the respective GIF.

Note that we do not evaluate image quality by using metrics such as MSE or SSIM because the exact same RGB-triplet is rendered at each position, i.e., complete quality preservation. It is also noteworthy that, irregardless of the statitics of the host image, the proposed method can surely embed data without causing any quality degradation, while the conventional methods degrade the image quality because 2 color indices are combined to free up an index [14] or the pixel value is modified by mapping it to different color index [3].

Table 1. Basic information about the animated GIFs considered for experiments.

GIF filename	Image dimensions	Total frames (original)	Bit stream size (KBytes)						
			Original	Basic	RZL				
					$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
Mov01	350 × 196	71	2,000	3,958	3,303	3,022	2,722	2,463	2,283
Mov02	499 × 273	17	472	1,436	1,111	960	805	682	596
Mov03	500 × 254	15	1,276	2,259	1,934	1,776	1,622	1,492	1,404
Mov04	250 × 141	25	505	958	821	750	673	613	568
Mov05	260 × 208	23	485	1,036	875	792	697	624	569
Mov06	480 × 270	17	1,301	2,404	2,054	1,893	1,708	1,558	1,453
Draw1	400 × 426	6	665	1,087	934	865	798	747	714
Draw2	670 × 503	52	2,488	6,754	5,334	4,894	4,262	3,685	3,239

4.1 Hiding Capacity

The number of bits that can be inserted into each GIF (i.e., payload) of each image is recorded in Table 2. All animated GIFs considered in this work consist of transparent pixel in each frame, therefore we could conveniently set $\tau_{f'} \leftarrow \tau_f$. As a result, all pixel locations are *usable*. When using the basic rules (i.e., Eq. (4)) to hide data, the payload is 1 bpp for each added frame. When using RZL, the payload decreases when the parameter k is increased. The payload decreases by a factor of $\sim$3 when k increases from 2 to 6. Although the payload achieved by RZL(6) is slightly less than 1/5 of *Basic*, the suppression of bit stream expansion is significant. On the other hand, the conventional methods are all limited by the number of frames as well as number of pixels in the animated GIF to hide data, i.e., non-scalable, while the proposed method is scalable at the expense of larger file size.

4.2 File Size Expansion

It is expected that the bit stream size will expand since new frames are introduced to hide data. The results are recorded in Table 1, with and without the implementation of RZL. When using the basic rule to hide data, the average expansion of bit stream size is 66.9%, which is reasonable since the number of frames is almost doubled. For completion of discussion, we also record the embedding efficiency η, which is defined as the number of embedded payload bits for every increased bit in the host image. Specifically, we consider the ratio of $\kappa(A, k)$ to $\Delta(A, A')$, where $\kappa(A, k)$ is the embedding capacity for the image A when using the parameter k, and $\Delta(A, A') = FS(A') - FS(A)$ refers to the file size difference between the original image A and the processed image A'. Here, higher value of η implies better performance, and vice versa. The average result $\bar{\eta}$ is recorded in the last row of Table 2 for $k = 1, 2, \cdots, 6$. On average, the host animated GIF image spends $1/0.28 \sim 3.8$ bits for hiding 1 bit of the payload.

On the other hand, when RZL is adopted, it is obvious that the expansion in bit stream size is suppressed, where the effect is more apparent for larger k.

Table 2. Embedding capacity (KBytes) for various k value after applying reserve zero run length encoding [16]

Image	Basic	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$
Mov01	586	334	317	247	165	101
Mov02	266	152	144	112	75	47
Mov03	217	123	117	91	61	38
Mov04	103	59	55	42	29	18
Mov05	145	83	79	61	42	26
Mov06	253	144	138	105	71	44
Draw1	104	59	56	43	29	18
Draw2	2,098	1,198	1,142	885	600	375
$\bar{\eta}$	0.28	0.24	0.29	0.32	0.34	0.35

Specifically, the average bit stream size expansion drops from 44.8% to 16.7% when k increases from 2 to 6. However, as noted in the previous sub-section, payload is reduced when k increases. Interestingly, the embedding efficiency decreases initially when RZL is adopted (i.e., $k = 2$), but the performance improves steadily after for $k > 2$. A potential influence to the performance is the LZW compression process, which is part of the GIF standard. This will also be explored as one of our future work.

In contrast, the conventional mostly maintains the file size, with small variation due data hiding. While the proposed method and conventional methods cited in this paper have their pros and cons, they can be combined to complement one an another. The combined deployment will be further explored as our future work.

5 Conclusions

In this work, transparent pixel in animated GIF is manipulated to hide data. Specifically, a new frame is introduced between 2 original frames, and each pixel location is manipulated to hide data. When a location is assigned the transparent pixel index, color from the previous frame is copied and rendered. Delay time of each frame is adjusted accordingly to ensure the duration of the animated GIF remains unchanged. Complete quality preservation is achieved irregardless of the characteristics of the animated GIF image, and the proposed method is reversible where the original animated GIF can be perfectly restored. Experiments suggest that data can be hidden into and extracted from the animated GIF.

In future work, we want to explore how the delay time parameter in each frame can be utilized to hide data. Furthermore, the joint utilization of the proposed and the conventional data hiding methods will be investigated. The influence of LZW compression in GIF on the file size increment due to data hiding will be also be investigated.

Acknowledgement. This work was supported in part by the Fundamental Research Grant Scheme (FRGS) MoHE Grant under project - Recovery of missing coefficients - fundamentals to applications (FRGS/1/2018/ICT02/MUSM/02/2) and in part by EU Horizon 2020 - Marie Sklodowska-Curie Action through the project entitled Computer Vision Enabled Multimedia Forensics and People Identification (Project No. 690907, Acronym: IDENTITY).

References

1. Graphics Interchange Format: Version 89a (1990). https://www.w3.org/Graphics/GIF/spec-gif89a.txt. Accessed 3 Mar 2019
2. Cooke, J., Chung, A.: Giphy. https://giphy.com/. Accessed 3 Mar 2019
3. Fridrich, J.: A new steganographic method for palette-based images. In: PICS 1999: Proceedings of the Conference on Image Processing, Image Quality and Image Capture Systems (PICS-99), Savannah, Georgia, USA, 25–28 April 1999, pp. 285–289. IS&T - The Society for Imaging Science and Technology (1999)
4. Gygli, M., Soleymani, M.: Analyzing and predicting GIF interestingness. In: Proceedings of the 24th ACM International Conference on Multimedia, MM 2016, pp. 122–126. ACM, New York (2016). https://doi.org/10.1145/2964284.2967195. http://doi.acm.org.ezproxy.lib.monash.edu.au/10.1145/2964284.2967195
5. Karp, D.: Tumblr. https://www.tumblr.com/. Accessed 3 Mar 2019
6. Katzenbeisser, S., Petitcolas, F.A. (eds.): Information Hiding Techniques for Steganography and Digital Watermarking, 1st edn. Artech House Inc., Norwood (2000)
7. Kim, S., Cheng, Z., Yoo, K.: A new steganography scheme based on an index-color image. In: 2009 Sixth International Conference on Information Technology: New Generations, pp. 376–381, April 2009. https://doi.org/10.1109/ITNG.2009.119
8. Kwan, M.: GIF colormap steganography (1998). http://www.darkside.com.au/gifshuffle/
9. Machado, R.: Ezstego (1997). http://www.stego.com/
10. Pan, Z., Wang, L.: Novel reversible data hiding scheme for two-stage vqcompressed images based on search-order coding. J. Vis. Commun. Image Represent. **50**, 186–198 (2018). https://doi.org/10.1016/j.jvcir.2017.11.020. http://www.sciencedirect.com/science/article/pii/S1047320317302286
11. Raymond, E.S.: What's in a GIF - animation and transparency (2012). http://giflib.sourceforge.net/whatsinagif/animation_and_transparency.html
12. Shim, H.J., Jeon, B.: DH-LZW: lossless data hiding in LZW compression. In: 2004 International Conference on Image Processing, ICIP 2004, vol. 4, pp. 2195–2198, October 2004. https://doi.org/10.1109/ICIP.2004.1421532
13. Thyssen, A.: ImageMagick v6 examples - animation basics (2004). https://imagemagick.org/Usage/anim_basics/
14. Wang, X., Yao, T., Li, C.T.: A palette-based image steganographic method using colour quantisation. In: IEEE International Conference on Image Processing 2005, vol. 2, p. II-1090, September 2005. https://doi.org/10.1109/ICIP.2005.1530249
15. Wong, K., Nazeeb, M.N.M., Dugelay, J.L.: Test animated GIFs (2020). http://bit.ly/2IEx26N
16. Wong, K., Tanaka, K., Takagi, K., Nakajima, Y.: Complete video quality-preserving data hiding. IEEE Trans. Circ. Syst. Video Technol. **19**(10), 1499–1512 (2009). https://doi.org/10.1109/TCSVT.2009.2022781