CGI2012 manuscript No.
(will be inserted by the editor)

Compressing Repeated Content within Large-scale Remote Sensing Images

Wei Hua · Rui Wang* · Xusheng Zeng · Ying Tang · Huamin Wang · Hujun Bao

Abstract Large-scale remote sensing images, including both satellite and aerial photographs, are widely used to render terrain scenes in real-time geographic visualization systems. Such systems often require large memories in order to store fine terrain details and fast network speeds to transfer image data, if they are built as web applications. In this paper, we propose a progressive texture compression framework to reduce the memory and bandwidth cost by compressing repeated content within and among large-scale remote sensing images. Different from existing image factorization methods, our algorithm incrementally find similar regions in new images so that large-scale images can be more efficiently compressed over time. We further propose a descriptor, the Gray Split Rotate (GSR) descriptor, to accelerate the similarity search. The reconstruction quality is finally improved by compressing residual error maps using customized S3TC-like compression. Our experiment shows that even with the error maps, our system still has higher compression rate and higher compression quality than using S3TC alone, which is a typical compression solution in most existing visualization systems.

*Corresponding author: Rui Wang, rwang@cad.zju.edu.cn.
This work was supported in part by NSFC (No. 60903037) and the 973 program of China (No.2009CB320803).

Wei Hua · Rui Wang · Xusheng Zeng · Hujun Bao
State Key Lab of CAD&CG, Zhejiang University
Tel.: +86-571-88206681
Fax: +86-571-88206680

Ying Tang
Department of Computer Science, Zhejiang University of Technology

Huamin Wang
Department of Computer Science and Engineering,The Ohio State University

Keywords Texture compression, Image epitomes and Large-scale remote sensing image

1 Introduction

Real-time geographic visualization system uses a considerable number of large-scale remote sensing images to reconstruct 3D terrain scenes. Such a system often requires large memory to store fine terrain details and fast network speed to transfer image data, if it is implemented as a web-based application. For example, Google Earth requires 256MB GPU memory and a 768Kbits/sec network connection for fluent rendering quality; Bing Maps 3D recommends 1GB system memory, 256MB GPU memory and a high-speed or broadband internet connection. Including more user-specific data and local street scenes will further increase the system requirement. We believe this issue cannot simply be solved by having more affordable large memory and faster network speed in the near future.

Our goal is to efficiently compress and decompress multiple large-scale remote sensing images for visualization systems, so that memory and network bandwidth can be saved for other purposes. Motivated by the method proposed in [19], our basic idea is to find similarities among images and simplify image representation by removing duplicate regions. Compared with traditional image compression algorithms, such as JPEG 2000 [18], our method removes redundancy globally and supports real-time GPU decompression. But different from [19], we focus on dealing with a great number of large-scale images, which are difficult to handle by their method due to a large computational cost.

To this end, we propose a progressive texture compression system by incrementally decomposing new images into codebooks and transformation maps. Our specific contributions in this system are:

- **Codebooks for Specific Purposes:** We classify remote sensing images into different categories based on their terrain contents, and we construct codebooks separately for each category. In each category, we use a public codebook to store common features among images, and a private codebook to store distinctive contents in each individual image. Compared with using a single codebook for all images, this structure is more efficient to build for large-scale images with varying contents.
- **Gray Split Rotate (GSR) descriptor and space:** We propose a GSR descriptor to better depict similarity of image blocks and use the metric defined in GSR space to speed up the similarity search. Our compact descriptor encodes the appearance, orientation and light intensity for one pixel and its neighborhood. By using the GSR descriptor as a similarity metric and searching in the GSR space, we efficiently distinguish pairs of regions that are not similar to each other.
- **Error Map Compression:** To further improve the compression quality, we develop a S3TC-like compression algorithm to compress the error map, which is defined as the difference between the original image and the decompressed image. The error map greatly improves the result quality and provides high compression rate than using S3TC alone.

2 Previous Work

Our work is mainly related to texture compression techniques, most of which have been developed for image compression, texture synthesis and texture rendering purposes.

Comparing to general image compression techniques, for example the JPEG 2000 [18], texture compression owns several issues to consider differently [2], such as decoding speed, pixel random access, compression rate and visual quality, encoding speed and etc. The early texture compression method proposed by Beers and his collaborators [2] used Vector Quantization (VQ) algorithm to produce a codebook and an index map. This method was later adopted to compress a collection of light-field images in [14]. By using larger image blocks and more flexible transformations, Wang and his colleagues [19] proposed a texture factorization method that automatically removes duplicate contents from images and assembles representative contents into a compact codebook, or called the *epitome*. An important advantage of these methods is that the decompression process contains only simple transformations and it can be easily accelerated by graphics hardware, so they are suitable for real-time rendering applications. While previous methods are focused on dealing with a single image or a small set of images, we are more interested in compressing a great number of large-scale images, which would require a huge computational cost if the method has not been properly optimized.

Fractal image compression technique proposed by Fisher [5] utilizes the self-similarity implied in images to build a set of contractive maps for image compression. Its decompression process is done iteratively by applying a series of affine transformations so it is not suitable for real-time rendering.

Developed by Iourcha and his collaborators [7], S3 Texture Compression (or called DXTC in Microsoft DirectX 3D) is a lossy texture compression algorithm commonly used in commercial graphics hardware. Based on the original S3TC idea, Pereberin [17] later proposed a hierarchical representation and a block-wise scheme to support S3TC mip-mapping. Levkovich-Maslyuk and his colleagues [13] improved color variance within each block by classifying pixels into different groups and creating a sub-palette separately for each group. Ivanov and Kuzmin [8] enriched the number of color choices by allowing colors shared among multiple blocks. Fenney [4] proposed an efficient representation to avoid block artifacts by blending multiple low frequency signals with a low-precision and high-frequency modulation signal. In general, S3TC and its extensions have a fixed compression ratio independent of the actual image content.

Given a small texture input, texture synthesis techniques [3, 9, 22] automatically generate seamless textures over a large surface. In recent years, researchers have designed the GPU texture synthesis algorithms to render the large synthesized results in real-time with the small input sample stored in the texture memory [11, 12, 20]. They may also be formulated as real-time texture decompression algorithms, if the goal is to get the surface textured rather than to recover an exact image. One question is how to summarize large textures into a small texture image as an inverse texture synthesis problem. A possible solution is the optimization-based inverse synthesis framework presented by Wei and his collaborators [21].

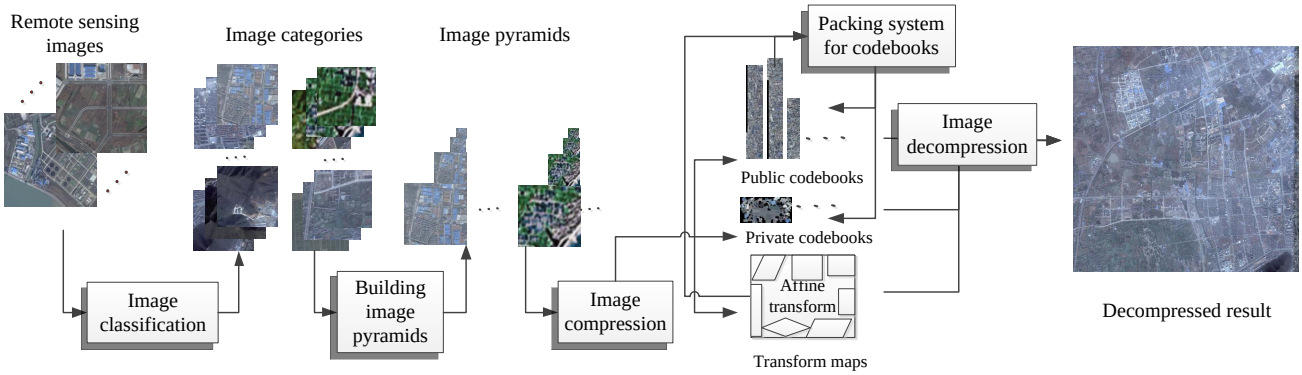


Fig. 1: *System pipeline.*

Our GSR descriptor is also related to local descriptors proposed in the computer vision community, such as SIFT [15] and SURF [1]. These descriptors are typically developed for image classification and object recognition purposes, so they are robust against image noises and varying illumination conditions. Since our final goal is to faithfully replace each original image region by its similar counterpart in the codebook, these descriptors are not sensitive enough to discriminate distinctive regions, even though they share certain common features.

3 System Overview

A geographic visualization system often requires a huge data set of large-scale remote sensing images. While most existing texture compression algorithms are focused on dealing with a single image or a small collection of images, our system can efficiently compress multiple remote sensing images in a progressive way, and decompress them on the fly using random-access texture fetches.

Following the same representation used in [19], we separate images into two components: a *codebook* and a transformation map. The codebook contains representative small samples from original images; the transformation map indicates how an image can be recovered from the codebook. Here we use the name *codebook* instead of *epitome* in order to emphasize its general use among multiple images. The basic idea behind this system is to incrementally update the contents in codebooks, if and only if its content is not sufficient to recover a new image. This dynamic feature is crucial to reducing the computational cost when dealing with a varying data set of multiple images. A simple solution here is to directly insert new contents into a single codebook when the system receives a new image. However, it is likely to produce a huge codebook and slow down the compression process. So we propose to use two codebooks instead of one. We use a public codebook to contain common contents among images and it will be used to recover multiple images during the decompression process. Meanwhile, we use a private codebook to store and recover distinctive contents in each image. This data structure allows us to effectively reduce the codebook size and makes the similarity search more efficient, by only considering the public codebook in the similarity search. To further reduce the public codebook size, we classify remote sensing images into different categories and construct public codebooks separately for each category. Since images in each category share more common features and the number of categories for remote sensing images is limited, the codebook for each category can also be more efficiently built.

Figure. 1 shows the pipeline of our system. It contains two components, compression and decompression. The compression process first classifies a set of initial images into different categories using the K-mean clustering algorithm [10]. We choose the color histogram as a simple classifier. The initial value of each cluster center is assigned by user input. After classification, we construct image pyramids using Gaussian filters and they are then compressed using our progressive compression algorithm, as will be discussed in Section 4. The compression process produces a codebook bank that contains both public and private codebooks for images in different categories, and transformation maps that instruct the renderer how to recover images using random texture access. We will explain the decompression algorithm in Section 5.

4 Texture Compression

Our texture compression approach takes a three-step algorithm. The first step is to find similar regions within and among images by our new descriptor. Then, these similar regions are progressively factorized into codebooks and transform maps. After these steps, to improve the compression quality, residual errors are computed and quantized into error maps. Since two former steps of our algorithm bear some similarities to that in [19], we highlight two distinctive adoptions of our algorithm for compressing large-scale images. First, compared with the color and orientation histograms used in [19], a new descriptor, the GSR descriptor, is proposed in this paper. With extra local rotation-variant information, our descriptor brings more accurate depiction of similarities among image regions. Second, instead of comparing every pair of image blocks, a quadratic number of comparisons to the number of blocks [19], we utilize the high-dimension feature space of the GSR descriptor and employ K-nearest neighbor search to accelerate the pruning process. In this section, we first introduce the GSR descriptor, then describe the construction of codebook and finally give out details of the error maps. The validation of these adoptions comparing to [19] is given in section 6.

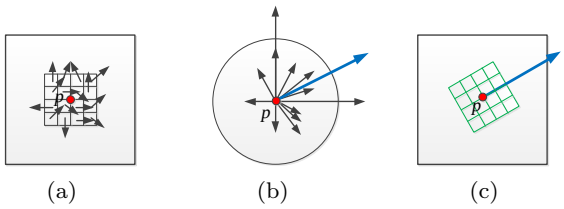


Fig. 2: *Gradient vectors (a) in each block are used to find the main direction (b). The overall direction allows us to create a grid for GSR formulation as (c) shows.*

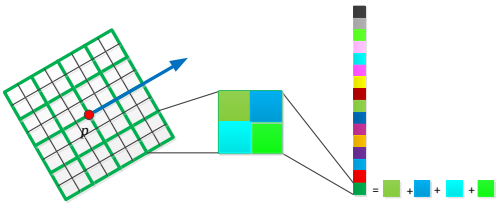


Fig. 3: *We use interpolated grayscale values within the grid to formulate the descriptor in N^2 dimensions. This figure is only an illustration that an 8×8-pixel block is formulated in a 16-dimensional descriptor. The color blocks are for illustration, in our method, we only use the grayscale values of each block.*

4.1 Similarity Search by GSR Descriptor

Our Gray Split Rotate (GSR) descriptor combines intensity/color statistics with the rotation-variant descriptor used in SURF. Although the descriptor can be formulated using all three RGB channels of the original image, we prefer to use only the grayscale intensity so that the descriptor can be more efficiently calculated. We first apply a Gaussian blur filter to remove high-frequency image noises. We then segment the image into square blocks and calculate intensity gradient for each pixel in every block (Figure 2a). These gradients on pixels are grouped to obtain the main direction (Figure 2b). Once we find the main direction, we create a N-by-N square grid that is aligned with the main direction as Figure 2c shows. Given this grid, we simply use interpolated grayscale values in the grid to formulate the descriptor in N^2 dimensions. N is typically chosen from 4 to 8 (Figure. 3).

The GSR descriptor presented above allows us to quickly compare two image blocks and terminate further computations if they appear too different in the descriptor space. Wang and his colleagues [19] compared every pair of image blocks so the number of comparisons is quadratic to the number of blocks. While doing this is acceptable for a single image, it is no longer affordable when we deal with multiple large-scale images. So instead, we build a N^2-dimensional tree for all GSR descriptors and only compare a pair of blocks that are neighbors in the GSR space using [16]. The neighborhood can either be defined within a fixed radius or by a fixed number of neighbors. Since it is not straightforward to automatically adjust radius parameters for different types of images, we prefer to set a fixed number of neighbors to define the similarity neighborhood in the descriptor space. After a pair of image regions passed this pruning process, they will be further tested under the computationally expensive KLT metric and compute the affine transform. We recommend readers to check [19] for more details about using the KLT feature tracker and the computation of affine transform.

When images are represented in multi-resolution, we process the similarity search separately at each image pyramid level. Although we can incorporate similarity search across different pyramid levels as that in [19], we find from our experiment that it is unnecessary because features in remote sensing images are at similar scales.

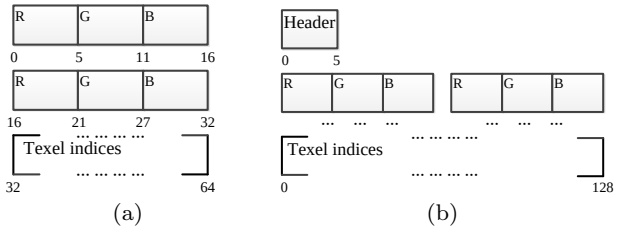


Fig. 4: *Bits formations. (a) is the bits formations used in DXT1 for a* 4×4*-pixel block and (b) shows our bits formation for a* 8×8*-pixel block to compress error map.*

4.2 Codebook Construction

Given the GSR descriptor proposed in Section 4.1, our next goal is to progressively factorize images into codebooks and transformation maps.Without losing generality, given a new image and an existing public codebook, we take following steps to update the public codebook and construct a private codebook.

We first separate the image into a set of blocks, each of which contains 16×16 pixels. We then run similarity search between each image block and the public codebook. If a match exists, we compute and store the appropriate transformation, with which the block can be directly recovered from the existing codebook. For these remaining blocks that cannot be represented by the public codebook, we build a similarity match list for each block as in [19] and count how representative a repeated content is. If the reused times are more than a threshold, the block will be added into the public codebook bank. Otherwise, we think it is not representative enough and it will be assembled into the private codebook instead. Details on the assembling process can be found in [19].

4.3 Compression with Error Maps

The combination of codebooks and transformation maps allows us to reconstruct comparable results to original images. However, this compression algorithm can be highly lossy if original images do not have sufficient similarity regions. Here we use an error map to further improve the compression quality. The error map $\mathbf{E}$ is defined as the difference between the reconstruction result and the original image. Ideally, using a lossless compression algorithm [6] to compress this error map, we are able to fully recover the original image without any residual error. However, the compression ratio would be greatly affected in this way. So we prefer to use a lossy compression algorithm instead.

We modify the standard DXT1 in S3TC to compress this error map. In DXT1, a 64-bit word is used to represent a 4×4-pixel block. The first half of the word is used to store two 16-bit RGB565 colors and the second half is used to store a 2-bit control code per pixel. Since an error map has relatively smaller intensity values, we use a 8×8-pixel block and store two colors with dynamic bit lengths (12 bits in maximum if using 4 bits per channel, or 0 bits in minimum) and a 6-bit prefix header to indicate the actual bit length. The bit format is illustrated in Figure 4. In total, our method requires 156 bits in maximum for a 8×8 pixel block, compared with 256 bits used by DXT1. Compression results with and without error maps are shown in Section 6.

5 Texture Decompression

In this section, we introduce our decompression algorithm for real-time rendering. Before the rendering process, codebooks, transformation maps and error maps are loaded into the GPU memory. Each of them is represented as an image texture. They are used to recover images by fragment shaders in GPU.

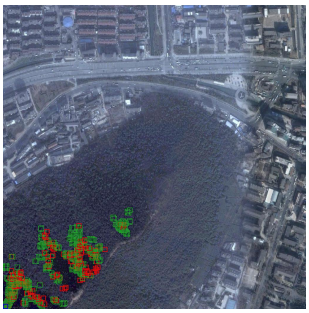
The pseudo code of the actual decompression process is shown in Algorithm 1. For each pixel, we first determine its texture coordinate in the transformation map (line 6). We also compute the coordinate difference, which gives the relative position of the current texel within a block (line 7). We then fetch transformation coefficients from two transformation textures, including 6 affine coefficients and a codebook type flag (line 8-9). According to these coefficients, we prepare the codebook texture coordinate (line 11-12) and then read the texture color from public or private codebook based on the flag taffine1.w (line 13-15). If the user chooses not to use the error map for better rendering quality, the pixel value will be directly sent to the display buffer. Otherwise, we use a S3TC-like decompression algorithm (line 18-19) to compensate the residual error.

6 Results

We implemented and tested our system on an Intel Core^{TM}2 Quad 2.83GHz workstation with 3 GB of RAM and an nVIDIA GeForce 280 GTX graphics card.

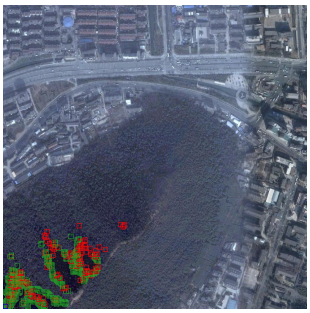
Examples (1K×1K size)	Algorithms	Search Time (s)	Codebook Size	RMS Error	Acceleration Ratio
Hill, 	[19]	658.00	704×288	8.97	1.00
	Acceleration 1	15.39	480×448	9.15	42.76
	Acceleration 2	29.91	448×384	9.05	22.00
Building, 	[19]	246.50	960×272	7.52	1.00
	Acceleration 1	15.25	576×512	8.08	16.16
	Acceleration 2	29.90	512×448	7.81	8.24
Field, 	[19]	508.55	448×384	9.42	1.00
	Acceleration 1	15.18	504×320	9.83	33.50
	Acceleration 2	30.73	448×336	9.62	16.55

Table 1: *Comparison of our method versus the method proposed in [19]. Acceleration 1 is our method that uses 250 nearest neighbors as similarity candidates for each block, while acceleration 2 uses 500 neighbors. The grid size N is chosen to be 4 for both cases and the descriptor is a 16-dimensional vector.*



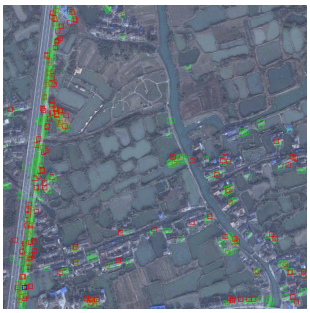
IR = 86.4%,
Avg. IR = 85.9%

(a) Our gray split rotate descriptor



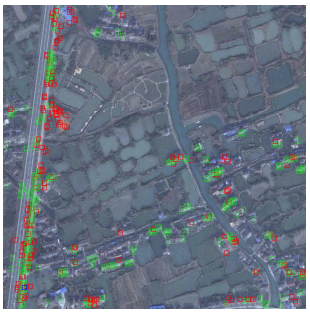
IR = 58.2%,
Avg. IR = 58.1%

(b) The color and orientation histogram descriptor [19]



IR = 90.8%,
Avg. IR = 90.9%

(c) Our gray split rotate descriptor



IR = 51.2%,
Avg. IR = 51.0%

(d) The color and orientation histogram descriptor [19]

Fig. 5: *Descriptor performance comparison between the GSR descriptor proposed in this paper with the color and orientation histogram descriptor proposed in [19] on different types of images. Given a sample block, shown in blue box, red boxes shown in images are similar regions computed by KLT feature tracker and green boxes are computed by specific feature descriptor. The more overlaps of red and green boxes indicate larger IR and higher accuracy. The "IR" ratio depicts the accuracy of a descriptor for the given sample block. The "Avg. IR" ratio depicts the accuracy of a descriptor for the entire image, which is computed by averaging "IR" ratio of blocks of the sample image.*

6.1 Descriptor Comparison

6.1.1 Speed Comparison

Table 1 compares the compression time, the reconstruction quality and the compression ratio of our method versus the method proposed in [19]. Acceleration 1 is our method that uses 250 nearest neighbors as similarity candidates for each block, while acceleration 2 uses 500 neighbors. The grid size N is chosen to be 4 for both cases and the descriptor is a 16-dimensional vector. This comparison shows that under almost the same compression ratio and reconstruction quality, our method is approximately ten times faster than [19]. While applying different acceleration techniques, more searching neighbors bring less reconstruction errors but have more acceleration ratio. In our following results, Unless mentioned otherwise, we use acceleration 2 in the compression.

6.1.2 Accuracy Comparison

Let $\mathbf{D}$ be the set of blocks that are accepted under the descriptor metric, and $\mathbf{M}$ be the set of blocks that are accepted under the KLT metric, we define the accuracy factors as follows:

$$IR = \frac{|\mathbf{M} \bigcap \mathbf{D}|}{|\mathbf{M}|}. \quad (1)$$

Intuitively, IR ratio measures the accuracy of a descriptor, since a larger IR value means that the descriptor is consistent with KLT and there will be less false-positives.

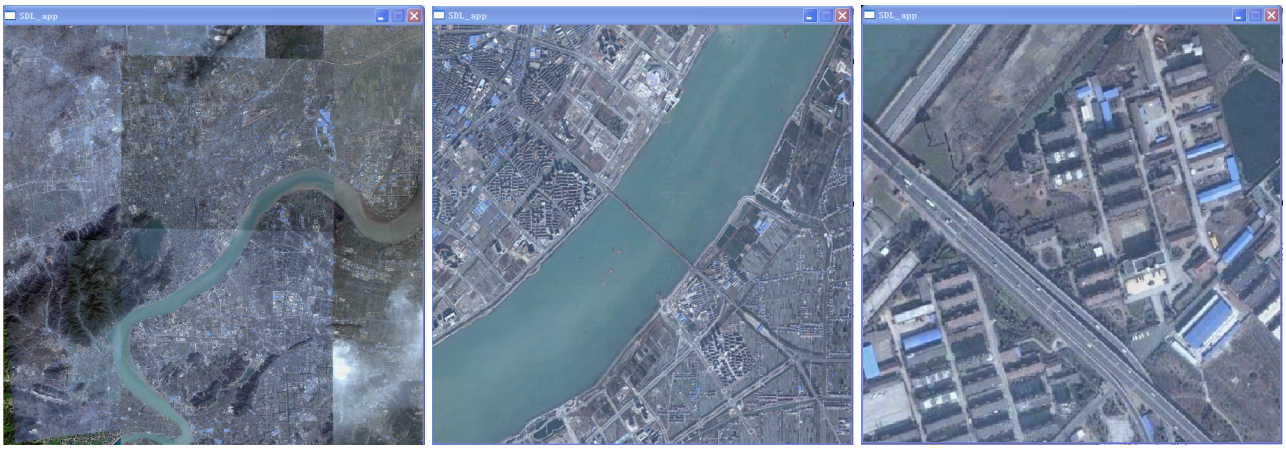


Fig. 6: *Screenshots of our visualization system.*

Algorithm 1 Decompression code in shader

```
int blockNum, blockLength;
sampler2D publicEptm, privateEptm, eMap;
sampler2D tfmAffine0, tfmAffine1;
procedure main(in t, out outColor, const use_emap)
begin
    vec2 tblock = (floor(t.st)+vec2(0.5,0.5))/blockNum;
    vec2 diff = (frac(t.st)-vec2(0.5,0.5))*blockLength;
    vec4 taffine0 = tex2D(tfmAffine0,tblock);
    vec4 taffine1 = tex2D(tfmAffine1,tblock);
    vec2 tEpitome;
    tEpitome.x = taffine0.xy*diff.xy+taffine1.x;
    tEpitome.y = taffine0.zw*diff.xy+taffine1.y;
    vec2 tidx; tidx.x = taffine1.w; tidx.y = 1-tidx.x;
    vec4 eColor = tidx.x * tex2D(publicEptm,tEpitome)
                + tidx.y * tex2D(privateEptm,tEpitome);
    outColor = eColor;
    if use_emap then
       vec4 errorColor = decode_emap(eMap, t);
       outColor = eColor + errorColor;
```

Using this ratio, Figure 5 compares descriptors proposed in this paper and that used in [19] on image examples. The gray split rotate descriptor (Figure 5a,c) here uses a 4-by-4 grid. The color and orientation histogram descriptor (Figure 5b,d) uses 16 buckets. Given a sample block, for each descriptor, 500 nearest neighbors in feature space are identified and used to compute corresponding IR. This example shows that our GSR descriptor performs better in accuracy than that used in [19].

6.2 Large-scale Images Compression

We compress multiple large-scale remote sensing images with an overall resolution of $76K \times 76K$. The total uncompressed size of such images is about 16.92 GB and if including mipmaps built in video memory, it is 22.56 GB. At compression, we divide these images into $1K \times 1K$ sub-images and process them progressively. Six categories are initially set to classify these sub-images and then 6-level image pyramids are built for further processing. For each similarity search task, four threads are parallelized for acceleration. It takes a total of 16 hours to complete the whole compression process. After compression, the public codebook bank size is 150.96 MB and the private codebook bank size is 2661.2 MB. We then compress them by DXT1 and obtain final public codebook bank with 25.16 MB and private codebook bank with 443.5 MB for rendering. The error maps occupy 2205.6 MB. The final compression ratio for these multiple large-scale remote sensing images is 2.08% (without error maps) and 11.19% (with error maps). Please check the supplemental material for a captured real-time video demo that shows the quality and performance of our visualization system. Three screenshots are shown in Figure 6.

6.2.1 Compression Quality

Figure 7 compares the residual error of our compression results with the results compressed by the DXT1 method. This example shows that without using the error map, our result is comparable to DXT1. After using the error map, our compression quality becomes better than DXT1.

6.3 Rendering Frame Rate and Memory Consumption

We compare rendering speeds using different decompression methods through the FPS curves sampled at d-

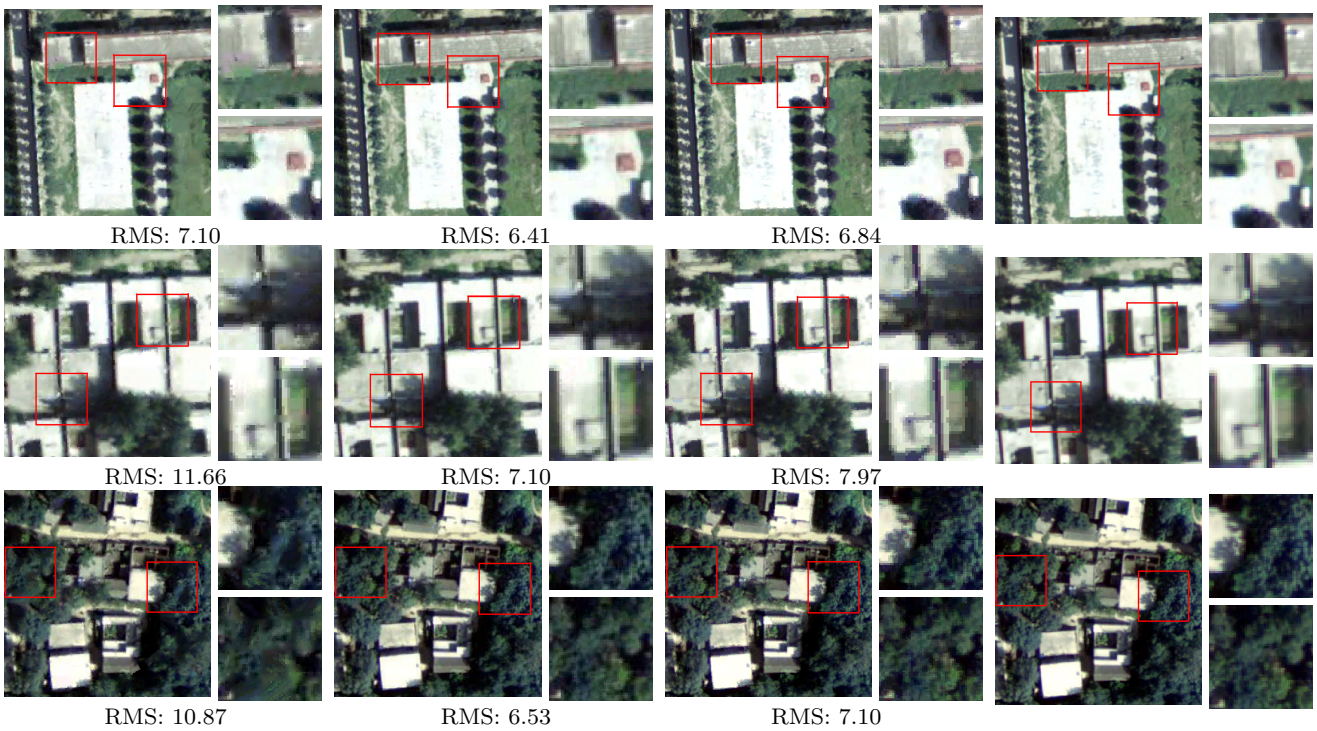


Fig. 7: *Quality comparison among a) our compression results only using codebooks, b) our compression results using codebooks and error maps, c) results compressed by DXT1 and d) uncompressed original image. For each image, we enlarge two regions for details comparison.*

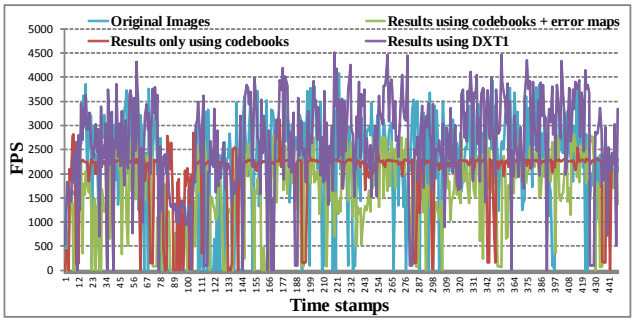


Fig. 8: *FPS curves sampled at different time stamps.*

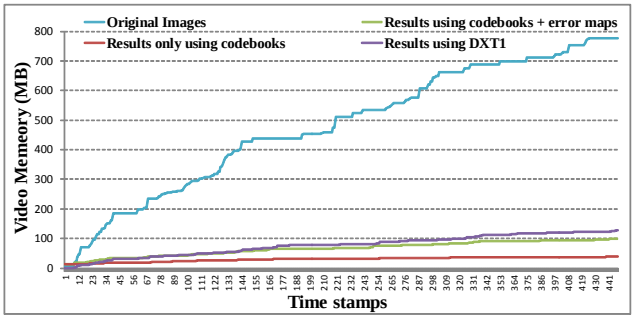


Fig. 9: *Occupied memory curves sampled at different time stamps.*

ifferent time stamps in Figure 8. The FPS curve demonstrates how FPS fluctuates over time, and it shows that our method performs more smoothly compared with pure DXT1 compression or directly rendering the original image. In particular, loading the whole original image into the GPU memory for rendering will cause frequent memory bandwidth bottlenecks and they are shown as sudden jumps in its FPS curve.

The plot in Figure 9 explores how the size of occupied video memory varies as a function of time for different methods. It shows that directly rendering the original image requires a large GPU memory as expected. Both DXT1 and our method use much less memory than the original image, and even using the error map, our method consumes less memory than that of using DXT1 compression.

7 Conclusions and Future Work

How to efficiently compress and decompress multiple large-scale remote sensing images for geographic visu-

alization applications is an interesting problem, and we present a systematic solution to this problem under the image factorization framework. The system uses a set of codebooks for different purposes, so that each of them can built in a compact and efficient way. We also propose a GSR descriptor to accelerate the similarity search and we demonstrate that similarity matches can be efficiently found in the GSR space. Finally we improve the compression quality by compressing the error map using a customized S3TC-like compression algorithm.

Looking into the future, we would like to further improve the compression speed by implementing compression algorithms on GPU. We are also interested in applying different importance weights to local image regions, so that we adaptively control the local compression quality. How to assemble codebooks into a compact form is also an interesting topic to study in the future.

Acknowledgements We would like to thank the reviewers for their thoughtful comments. We also would like to thank student Hong Yu for her efforts on the demos.

References

1. Bay, H., Tuytelaars, T., Gool, L.V.: Surf: Speeded up robust features. In: In ECCV, pp. 404–417 (2006)
2. Beers, A.C., Agrawala, M., Chaddha, N.: Rendering from compressed textures. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, SIGGRAPH '96, pp. 373–378. ACM, New York, NY, USA (1996)
3. Efros, A.A., Freeman, W.T.: Image quilting for texture synthesis and transfer. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, SIGGRAPH '01, pp. 341–346. ACM, New York, NY, USA (2001)
4. Fenney, S.: Texture compression using low-frequency signal modulation. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, HWWS '03, pp. 84–91. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2003)
5. Fisher, Y.: Fractal Image Compression, Theory and Application. Springer-Verlag (1995)
6. Inada, T., McCool, M.D.: Compressed lossless texture representation and caching. In: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, pp. 111–120. ACM, New York, NY, USA (2006)
7. Iourcha, K., Nayak, K., Hong, Z.: System and method for fixed-rate block image compression with inferred pixels values. US Patent 5,956,431 (1999)
8. Ivanov, D.V., Kuzmin, Y.P.: Color distribution - a new approach to texture compression. Computer Graphics Forum **19**(3), 283–290 (2000)
9. Kwatra, V., Essa, I., Bobick, A., Kwatra, N.: Texture optimization for example-based synthesis. ACM Trans. Graph. **24**, 795–802 (2005)
10. Lai, J.Z.C., Huang, T.J., Liaw, Y.C.: A fast k-means clustering algorithm using cluster center displacement. Pattern Recogn. **42**, 2551–2556 (2009)
11. Lefebvre, S., Hoppe, H.: Parallel controllable texture synthesis. ACM Trans. Graph. **24**, 777–786 (2005)
12. Lefebvre, S., Hoppe, H.: Appearance-space texture synthesis. ACM Trans. Graph. **25**, 541–548 (2006)
13. Levkovich-Maslyuk, L., Kalyuzhny, P., Zhirkov, A.: Texture compression with adaptive block partitions (poster session). In: Proceedings of the eighth ACM international conference on Multimedia, MULTIMEDIA '00, pp. 401–403. ACM, New York, NY, USA (2000)
14. Levoy, M., Hanrahan, P.: Light field rendering. In: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, SIGGRAPH '96, pp. 31–42. ACM, New York, NY, USA (1996)
15. Lowe, D.G.: Object recognition from local scale-invariant features. In: Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2, ICCV '99, pp. 1150–. IEEE Computer Society, Washington, DC, USA (1999)
16. Mount, D.M., Arya, S.: Ann: A library for approximate nearest neighbor searching (2010). URL `http://www.cs.umd.edu/~mount/ANN/`
17. Pereberin, A.: Hierarchical approach for texture compression. In: Proceedings of GraphiCon, pp. 195–199 (1999)
18. Skodras, A.N., Christopoulos, C.A., Ebrahimi, T., Ebrahimi, T.: JPEG2000: The upcoming still image compression standard. pp. 1337–1345 (2001)
19. Wang, H., Wexler, Y., Ofek, E., Hoppe, H.: Factoring repeated content within and among images. ACM Transactions on Graphics (SIGGRAPH 2008) **27**(3), 14:1–14:10 (2008)
20. Wei, L.Y.: Tile-based texture mapping on graphics hardware. In: ACM SIGGRAPH 2004 Sketches, SIGGRAPH '04, pp. 67–. ACM, New York, NY, USA (2004)
21. Wei, L.Y., Han, J., Zhou, K., Bao, H., Guo, B., Shum, H.Y.: Inverse texture synthesis. ACM Trans. Graph. **27**, 52:1–52:9 (2008)
22. Wei, L.Y., Levoy, M.: Fast texture synthesis using tree-structured vector quantization. In: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, SIGGRAPH '00, pp. 479–488. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)