TEXTURE COMPRESSION IN MEMORY- AND PERFORMANCE-CONSTRAINED EMBEDDED SYSTEMS

Jens Ogniewski*, Andréas Karlsson†, Ingemar Ragnemalm*
** Information Coding Group, Linköpings University / † Computer Engineering, Linköpings University*
581 83 LINKÖPING, Sweden
{jenso|andreask|ingis}isy.liu.se

ABSTRACT

More embedded systems gain increasing multimedia capabilities, including computer graphics. Although this is mainly due to their increasing computational capability, optimizations of algorithms and data structures are important as well, since these systems have to fulfill a variety of constraints and cannot be geared solely towards performance. In this paper, the two most popular texture compression methods (DXT1 and PVRTC) are compared in both image quality and decoding performance aspects. For this, both have been ported to the ePUMA platform which is used as an example of energy consumption optimized embedded systems. Furthermore, a new DXT1 encoder has been developed which reaches higher image quality than existing encoders.

KEYWORDS

Embedded systems, texture compression

1. INTRODUCTION

Many modern embedded systems were not primarily designed with complex computer graphics in mind, yet are supposed to handle graphics as well, e.g. for user interfaces or simple computer games. Since most of these systems are also constrained by a comparably slow bus, which is furthermore shared between different computing units, an efficient image and texture compression method with low complexity is in dire need.

Already in (Beers et al., 1996) the requirements for such a compression method were pointed out:

- Low complexity of the decoder and high decoding speed
- Ability to access each pixel in the texture randomly (since objects may be obscured and oriented arbitrarily)
- Lossy compression is tolerable, since a screen image is made up of a multitude of different textures, so that a visual loss in one of them is hardly noticeable. Furthermore, lossy compression leads to a higher compression factor.
- A great majority of the used textures are or can be generated beforehand, so the encoding speed is only of a minor concern.

For embedded systems with limited performance and memory, low complexity is by far the most important factor, and vector quantization is a natural choice for this task.

Vector quantization has been used in image compression for several decades. The main idea is that instead of using the whole color space, a few colors will be selected which closely represent the colors contained in the image. Each pixel is then represented by an index into the list of these colors (the so called codebook). An overview of early algorithms can be found in (Nasrabadi and King, 1988). Later examples include (Buhmann et al., 1998) and the more recent (Horng et al., 2011).

Adoptions of vector quantization codecs include block truncation coding (Delp and Mitchell, 1979) and color cell compression (Campbell et al., 1986), on which most texture compression approaches are based upon. Here, the image is divided in non-overlapping blocks of equal size. Each block has an own list of a few colors, and further colors are computed by combination of these (see further section 2). The color of each pixel in the block is then encoded by an index determining which of these colors should be used. The first commercially used texture compression was based on these ideas and was called S3TC at first, but became later the DXT family of texture compression standards. Note that the different DXT codecs only differ in the way how they handle transparency, which will not be considered in this paper. The way they encode the color information is however the same (with the exception of DXT1 if it is used with an alpha channel).

PVRTC (PowerVR Texture Compression), another often used texture compression method, however follows a slightly different, more complicated approach. It was introduced by Fenney (2003) and is aimed mainly towards embedded systems with a comparably high graphical performance, which normally includes a complete (though often relatively small) GPU. It offers one of the highest image qualities as well.

PVRTC includes an upsampling step: instead of encoding color vectors, two color maps are built which are of the size of the original image divided by the blocksize (e.g. if the blocksize is 4x4, the width and the height of the color map will be ¼th of the original width and height). During decoding, the color maps will be upsampled by linear interpolation so that their sizes match the size of the original image. Each decoded pixel can then have the color found at its position in either color map, or a combination of these. This has two drawbacks:

- PVRTC is based on the assumption that the system includes a GPU with a special texture memory that offers hardware support for the upsampling step. However, that is not the case for embedded systems without or with a too simple dedicated GPU, which increases the runtime of the decoder heavily.
- For the decoding of each block 8 different base colors need to be loaded (4 for each color map) instead of only 2, thus decreasing the performance of random access and increasing usage of the bus.

Another texture compression method is ETC (Ericsson Texture Compression, former PACKMAN) which was introduced in (Ström and Akenine-Möller, 2005) and refined in (Ström and Petterson, 2007). Here, instead of calculating colors based on interpolation a color is given, which is modified based on modifiers contained in a codebook. This codebook only contains a luminance component and is fixed, but different ones are available which can be chosen on a block-by-block basis. Furthermore, instead of using blocks of size 4x4, 2x4 blocks are used in ETC.

Other works on texture encoding specializes on different applications, like (Roimela, 2008) on hdr textures (which uses YUV rather than RGB color space) and (Rasmusson et al., 2010) on textures with smooth edges for lightmaps.

Although modern graphic cards in desktop computers contain enough memory to make texture compression unnecessary, it is at least still very important in embedded systems (like e.g. mobile phones, portable multimedia players, settop boxes) due to the memory limit and comparably slow buses, which are often furthermore shared between many different components. An example for these small embedded systems is the energy consumption optimized ePUMA (embedded Parallel DSP with Unique Memory Architecture) platform (Ragnemalm and Liu, 2010), which consists mainly of a master CPU with 8 tightly coupled SIMD (Single Instruction Multiple Data) units which each has access to its own scratchpad memory.

The remainder of this paper is organized as follows: 2. introduces the ePUMA platform in more detail, 3. describes the decoding process for DXT1 and PVRTC, 4. gives an overview of a new encoder for DXT1, 5. shows an evaluation and 6. finishes with a discussion and ideas for future work.

In the following, whenever PVRTC is mentioned, it is referring to its 4bpp mode. The 2bpp mode will not be discussed further in this paper. Furthermore, transparency will not be discussed, and therefore both PVRTC and DXT1 are only considered in their modes without transparency.

2. THE EPUMA PLATFORM

The ePUMA platform is a processor for embedded systems optimized for low energy consumption, which is currently in development and up until now only exists in a simulator. It consists of a master CPU which act as controller for 8 SIMD processors which each has 8 parallel 16-bit datapaths. These can alternatively be used as 4 datapaths of 32-bit width. 8-bit datatypes were not supported yet during the writing of this paper, but are to be included in future implementations. An overview of the platform is given in figure 1. In it, N1 to N8 are the network nodes which handle communication between the different SIMDs, the master CPU and the main memory. PM is the program memory, CM a constant memory and PT are permutation tables designed to assist fast, parallel and conflict-free accesses to the LVMs (local vector memories) even with irregular addressing.

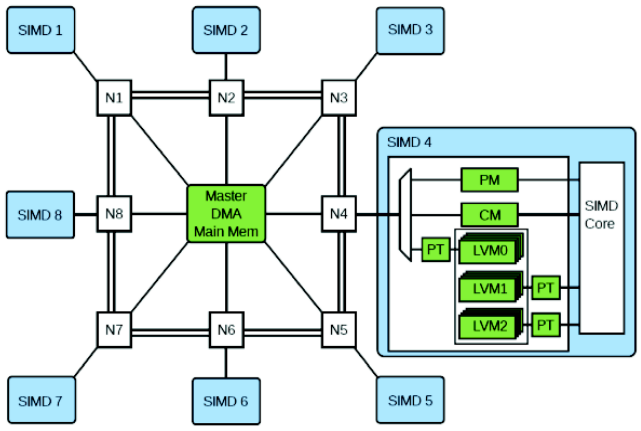

Figure 1. Overview of the ePUMA platform.

The SIMD units build the DSP subsystem of the platform and serve as its primary computing unit. Apart from the main memory, which is globally shared, each SIMD core possesses it own local memory, which can be used as a programmer controlled scratchpad. This memory is organized in three so called local vector memories (LVM), which is used to hide the main memory latency. A possibility would be to use one LVM for input data, one for output data, and load the input data for the next task into the third memory. Another possibility would be to use one LVM as input/output for the current task, one as persistent memory, and one to write back the output of the former task / load the input for the next task.

Since the processor is not primarily designed to serve as a GPU, it lacks dedicated memories for textures or the framebuffer. Thus, these have to be placed in the main and local memories.

Access to the main memory is quite slow compared to the local memory and can therefore become a bottleneck. Furthermore, the size of the local vector memories is limited, 80kB in the current implementation. Considering RGB only and that the pixeldata does not need to be aligned any further, the local memory can only hold 6 64x64 textures or 26 32x32 textures. If transparency is to be used, or mipmapping, the number of textures the local memory can hold will be further reduced. Considering 2 levels of mipmapping (which should be enough for most purposes), this is further reduced to only 5 64x64 textures resp. 21 32x32 textures. Additionally, in most applications the local memory cannot be used solely as a texture memory, thus decreasing the number of textures it can hold even further.

This induces the need for a good texture compression scheme which minimizes the access to the main memory without introducing much runtime for the decoding. Using DXT1 or PVRTC decreases the size of the texture by a factor of 6, thus relaxing memory usage significantly. Ideally, it should be possible to decode the textures “on the fly”, directly before they are used, thus eliminating the need of storing the decoded textures.

More information about the ePUMA platform can be found in (Liu et al., 2010) and (Ragnemalm and Liu, 2010).

3. DECODING

As already mentioned, modern texture codecs are vector quantization based, and use combinations of the encoded color vector to derive additional colors. In most cases 2 colors (c1 and c2) are encoded and 2 other ones (c3 and c4) are calculated through their combination, thus giving 4 different colors in total. Intuitively, evenly spaced combinations come to mind, i.e.:

$$c3 = \tfrac{1}{3} * c1 + \tfrac{2}{3} * c2$$
$$c4 = \tfrac{1}{3} * c2 + \tfrac{2}{3} * c1$$

In fact, these are exactly the weights which are used in many different texture compression standards, like DXT1. PVRTC on the other hand uses weights of ⅜ and ⅝ instead. This is thought to be more advantageous since it more closely resembles the normal distribution:

$$c3 = \tfrac{3}{8} * c1 + \tfrac{5}{8} * c2$$
$$c4 = \tfrac{3}{8} * c2 + \tfrac{5}{8} * c1$$

Both PVRTC and DXT1 use 16 bits to encode one color. In PVRTC, the first bit is used to switch the alpha channel on or off (i.e. switch between RGB and RGBA). The remaining 15 bits encode the color channels. In RGB mode, each of the three channels are described by 5 bits, thus decreasing the resolution by 3 bits per color channel from the more common 8 bit. DXT1 on the other hand uses the whole 16 bits for color information, giving the green channel a higher precision of 6 bits since the human eye has a higher sensitivity for green than for red and blue. The switch between RGB and RGBA is a little more complicated in the DXT coding standards, but since the alpha channel is neglected in this report it is not further explained here.

During decoding, the scaling of the compressed colors to convert them back to the 24-bit RGB colorspace can be combined with the multiplication of the weights as described above.

Since 4 different colors are used, 2 bits are needed for the indexing. The blocksize is set to 4x4, and therefore 32 bits are needed for the indexing of each block. Adding the 32 bits for the 2 color vector gives 64 bits for each block, or 4 bits per pixel (corresponding to a compression factor of 6). Here the disadvantage of PVRTC in terms of random access becomes obvious. Although it uses 4 bits per pixel as well, altogether 160 bits (8 color vector plus the indexes) need to be loaded to decode one block instead of only 64, due to the upsampling step.

4. TEXTURE ENCODING

We have developed and evaluated our own encoder for DXT1 texture compression, called *line matching*. In the following, the encoder will be described, with an evaluation in the next section.

If omitting their spatial position (which is already encoded in the order of the color indices), the pixels of a block can be described as points in a three dimensional space, where each of the axes represents one of the color channels. The 2 encoded color vectors then span a line in the color space, which includes the calculated color vectors as well. A simple (but effective) encoding scheme can then be constructed by first determining the best fitting line (i.e. the line which minimizes the sum of its distance to the different colors), and then searching along the line to find the points on the line which minimizes the distance to the color values of the current block.

For the first step a total least square approach was chosen, which is known to deliver an overall optimal line (up to a certain accuracy), and is used in many other different applications as well. It determines a line of form $l = s*x + a$, with a, l and s a vector, x a scalar. The intercept a is set to the mean color of the block. A matrix M is formed which consists of the different colors, each subtracted by the mean. Then a matrix $N = 1/n * M * M^T$ is built, and decomposed using singular value decomposition (SVD) to the form $U\Sigma V^*$, where U is a 3x3 matrix, and Σ a 3x3 diagonal matrix. The slope s of the line l is set to the column vector of U which corresponds to the largest absolute value in Σ. During this work, the standard SVD decomposition of the GNU Scientific Library was used.

Next, x-values are computed for every color of the block, which lead to the point on the line which is nearest to the current color. Both the total highest and the total lowest x-values are kept, and all possible values between (and including) these two x-values are then used to create possible candidates for c1 and c2. For each candidate pair, the distance between the block after encoding and the original block is computed.

A special case occurs if all values of s are zero, (e.g. when all pixels in the block have the same color). In this case c1 and c2 are set to the average, and the second step is left out.

To increase the performance of the encoding, every possible combination of c1 and c2 should only be checked once. Since c1 and c2 have a fixed resolution, the candidates have to be quantized during the search, and candidates should be rejected immediately, if they are leading to the same value after quantization as another configuration that has already been checked. Furthermore, combinations in which the values for c1 and c2 are simply switched should only be evaluated once to avoid unnecessary computations.

Initially, a third step was included in the encoder, which consisted of a fullsearch around the final values c1 and c2 to find a local optimum. However, this lead to an increase of less than 0.03% in point of the PSNR and less than 0.005% in terms of SSIM, and was therefore omitted in the final version. Intuitively, the optimal candidates (in Terms of PSNR) for c1 and c2 should lay on the best fitting line, but no proof to this effect is known to the authors. The fact that a fullsearch around the final candidates lead to slightly better results is however not a contradiction of this assumption either. They might also be introduced by a low accuracy of the SVD algorithm, or due to inaccurate rounding and quantization during the search along the best fitting line; in the experiments, the encoder proved to be very sensitive to how this was exactly implemented. Furthermore, in the case were all pixels contain the same color, setting c1 and c2 to this color is not an optimal solution either, since the calculated values of c3 and c4 may actually reach a higher resolution than 6 bits, due to their multiplication with non-integer weights.

Table 1. Cycle counts for the decoding of one block with DXT1 and PVRTC

Task	DXT1	PVRTC
Unpacking	3	3
Calculation of color the vectors	3	99
Waiting for pipeline to finish	8	8
total	**14**	**110**

5. EVALUATION

Decoders for both DXT1 and PVRTC have been implemented for the ePUMA platform. The cycles needed to decode one 4x4 block are given in table 1. As suspected, PVRTC needed much longer time for the task, 33 times if only the time needed to compute the color vectors is taken into account, and nearly 8 times as long for the whole decoding. Since the last step (waiting for the pipeline to finish) needs only to be done once, independent on the number of blocks that will be decoded, the performance gain if using DXT1 will probably be more than 8 in practice.

Note that it is not necessary to copy the different colors to each pixel, since this can be quite efficiently combined with the next step in the graphics processing pipeline.

Although decoding speed is the most important criterion for a texture compression scheme used in embedded systems, it is not the only one. To test the image quality of both codecs, several test sequences have been encoded and compared. These sequences consist of all sequences in the MIT vision texture testbench (MIT) which contains color images with a resolution of 512x512 resolution. An additional test sequence was added containing several standard images (simply called images in the following) which are used in image and texture compression, namely the standard Lena picture, the lorikeet picture which is made available by Simon Fenney and the Lichtenstein picture which is published under the wikicommons license. All in all, 170 different images have been evaluated.

For PVRTC, the reference encoder provided by Imagination Technologies (PVRTC) was used. For DXT1, we used both our own encoder, as described in section 3, and an open source implementation called Squish (Squish). Squish is considered to reach high image quality and is close to the implementation used by NVIDIA in their texture encoding solution.

Apart from signal-to-noise ratio, the mean structural integrity (SSIM) (Wang et al., 2004) has also been computed for all images. SSIM is designed as a difference index for luminance, contrast and structure between two images. Its calculated value varies between -1 (completely different) to +1 (perfect match). A standard window size of 8 was chosen for SSIM, and all possible image patches have been tested.

The mean results for the different sequences for both SNR and SSIM are presented in table 2. The standard deviation for each is given in parentheses. As can be seen, while neither algorithm performs best in all sequences, line matching lead to the best overall result. A further subjective evaluation of the image quality is left to the reader. For that, encoded versions of an example image is give in figure 2 along with the difference images in figure 3. The difference images contain the squared difference between the original image and each encoded image.

According to our tests, PVRTC does not give a noticeably better image quality than the DXT1. In some extreme cases it even introduces visible artifacts, as can be seen in the sky of figure 2. Although the used PVRTC encoder could be optimized for better image quality, it is doubtful if the quality will ever compensate for the much higher decoding complexity. Therefore, we conclude that DXT1 is the preferable choice for texture encoding.

Table 2. PSNR and SSIM for the used encoders – PVRTC reference encoder, line matching and squish. Higher values are better, standard deviations are given in parentheses. The respective best results are marked in red.

Testsequence	PSNR			SSIM		
	PVRTC ref.	Line matching	Squish	PVRTC ref.	Line matching	Squish
Bark	27.86 (3.07)	28.04 (3.25)	27.86 (3.07)	0.980 (0.006)	0.981 (0.003)	0.980 (0.003)
Brick	30.73 (2.41)	31.21 (2.38)	30.94 (2.27)	0.978 (0.004)	0.981 (0.002)	0.980 (0.002)
Buildings	29.78 (2.19)	30.59 (2.28)	30.73 (2.01)	0.978 (0.003)	0.983 (0.002)	0.983 (0.002)
Clouds	34.45 (0.02)	35.51 (0.15)	34.60 (0.11)	0.967 (0.002)	0.974 (0.000)	0.971 (0.000)
Fabric	24.82 (2.51)	25.20 (2.41)	25.21 (2.35)	0.974 (0.005)	0.980 (0.004)	0.978 (0.004)
Flowers	29.98 (3.36)	29.86 (2.95)	29.62 (2.86)	0.982 (0.004)	0.983 (0.002)	0.982 (0.002)
Food	27.18 (4.82)	26.61 (4.80)	26.50 (4.48)	0.982 (0.005)	0.981 (0.004)	0.981 (0.004)
Grass	23.62 (4.67)	23.95 (4.48)	23.96 (4.31)	0.980 (0.003)	0.981 (0.000)	0.981 (0.000)
Images	30.47 (0.34)	30.71 (0.81)	30.47 (0.77)	0.976 (0.008)	0.975 (0.005)	0.973 (0.001)
Leaves	26.81 (3.73)	26.70 (3.67)	26.46 (3.55)	0.982 (0.004)	0.983 (0.003)	0.981 (0.003)
Metal	21.90 (1.13)	22.54 (1.00)	22.63 (1.03)	0.972 (0.003)	0.977 (0.002)	0.977 (0.002)
Misc	28.90 (2.97)	29.26 (3.21)	29.05 (3.17)	0.980 (0.003)	0.981 (0.002)	0.980 (0.001)
Paintings	27.70 (2.19)	27.16 (2.23)	26.81 (2.21)	0.977 (0.007)	0.976 (0.007)	0.974 (0.007)
Sand	29.46 (1.65)	29.21 (1.69)	29.07 (1.66)	0.981 (0.004)	0.981 (0.002)	0.979 (0.002)
Stone	28.48 (2.55)	28.64 (2.61)	28.36 (2.50)	0.973 (0.009)	0.978 (0.007)	0.974 (0.007)
Terrain	34.20 (0.72)	34.59 (0.83)	33.98 (0.76)	0.985 (0.001)	0.984 (0.001)	0.981 (0.001)
Tile	30.25 (2.40)	30.66 (2.66)	30.42 (2.55)	0.977 (0.006)	0.981 (0.005)	0.980 (0.004)
Water	31.65 (2.87)	32.23 (3.17)	31.81 (2.98)	0.979 (0.003)	0.982 (0.002)	0.980 (0.003)
WheresWaldo	26.30 (1.15)	25.81 (0.79)	25.59 (0.86)	0.984 (0.004)	0.983 (0.002)	0.981 (0.004)
Wood	31.77 (3.38)	31.77 (4.41)	31.32 (4.26)	0.984 (0.005)	0.984 (0.002)	0.982 (0.002)
average	**28.45 (4.05)**	**28.60 (4.12)**	**28.39 (3.93)**	**0.979 (0.006)**	**0.980 (0.005)**	**0.979 (0.005)**

6. CONCLUSION AND FUTURE WORK

Two different popular texture decoders were implemented for the ePUMA platform, namely DXT1 and PVRTC. Both have been compared in decoding time and image quality. As suspected, the more complicated PVRTC leads to a much higher decoding time, nearly 8 times higher, which might in reality be even higher still since only the decoding of one block at a time was considered, thus rendering an efficient use of the pipeline impossible in the case of DXT1. In fact, the DXT1 decoder is fast enough to enable decoding “on-the-fly”, i.e. to decode the texture directly before it is used and therefore eliminating the need of storing the decoded textures in the local memory.

A comparison of quality aspects of the encoded images shows that PVRTC does not produce an overall better image quality over DXT1. Rather, our new encoder for DXT1, introduced in this paper, leads to the best results. Furthermore, the results were very close and each encoder produced high quality images. Therefore, the much higher decoding complexity of PVRTC is hard to justify. And even an optimized PVRTC encoder with higher image quality is not likely to change that. Therefore, the authors recommend the use of DXT1 as texture codec for embedded systems. Note that the different encoding schemes of the DXT family use the same encoding of the color channel, thus the results of this paper can be directly applied to its other members as well.

The line matching encoder was not much optimized for a fast encoding speed. Although it should be fast enough for all possible non-realtime application (on average the encoding of one image took 21.4s during the experiments), further optimization is possible and desirable. Especially the search along the best fitting line could be improved, which might even heighten the quality slightly. A SVD algorithm which is more optimized to this application (e.g. which omits the calculation of unnecessary values) should lead to better results as well.

Furthermore, although the overall graphical quality is high, blocking artifacts can occur. Note that this problem does not occur if using PVRTC, since neighboring blocks share color values due to the upsampling process. However, another encoding approach for DXT1, which encodes several neighboring blocks at once might solve this problem, although this might come with the price of reduced signal-to-noise ratio.

For simplicity, this paper concentrated on pure color images and did not include transparency. Another analysis should be done with this in mind to find out which of the different DXT1 transparency modes that delivers the best results. This could also be compared to the way PVRTC handles transparency. A comparison with the ETC texture compression could be done as well.

REFERENCES

Beers, A. C. et al., 1996. Rendering from Compressed Textures. *Proc. SIGGRAPH 7.* New York, USA, pp 373-378

Buhmann, J. M. et al., 1998. Dithered Color Quantization. *Proceedings of Computer Graphics Forum 17.* Oxford, UK, pp C219–C231.

Campbell, G. et al., 1986. Two Bit/Pixel Full Color Encoding. *ACM SIGGRAPH Computer Graphics*, Vol. 20, No. 4, pp 215-223.

Delp, E. and Mitchell, O, 1979. Image Compression Using Block Truncation Coding. *In IEEE Transactions on Communications*, Vol. COM-2, No. 9, pp 1335-1342.

Fenney, S., 2003. Texture Compression Using Low-Frequency Signal Modulation. *Proceedings of ACM SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hardware.* Aire-la-Ville, Switzerland, pp 84 - 91.

Horng, M.-H. et al, 2011. Image Vector Quantization Algorithm via Honey Bee Mating Optimization. *In Expert Systems with Applications*, Vol. 38, No. 3, pp 1382-1392.

Liu, D. et al., 2010. Parallel Programming and Its Architectures Based on Data Access Separated Algorithm Kernels. *In International Journal of Embedded and Real-Time Communication Systems*, Vol. 1, No. 1, pp 64-85.

MIT Vision Texture Database, *http://vismod.media.mit.edu/vismod/imagery/VisionTexture/*

Nasrabadi, N. M. and King, R. A., 1988. Image Coding Using Vector Quantization: A Review. *In IEEE Transactions on Communications*, Vol. 36, No. 8, pp 957-971.

PVRTC, *http://www.imgtec.com/powervr/insider/powervr-pvrtexlib.asp*

Ragnemalm, I. and Liu, D., 2010. Towards Using the ePUMA Architecture for Hand-Held Video Games. *Proceedings of IADIS International Conferences on CGVCVIP, VC and WEB3DW.* Freiburg, Germany.

Rasmusson, J. et al., 2010. Texture compression of light maps using smooth profile functions. *Proceedings of the Conference on High Performance Graphics.* Aire-la-Ville, Switzerland, pp 84-91.

Roimela, K. et al., 2008. Efficient High Dynamic Range Texture Compression. *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. New York, USA, pp 207-214.

Ström, J., and Akenine-Möller, T., 2005. iPACKMAN: *High-Quality, Low-Complexity Texture Compression for Mobile Phones*. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, New York, USA, pp 63 - 70.

Ström, J. and Pettersson, M., 2007. ETC2: Texture Compression using Invalid Combinations. *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware.* Aire-la-Ville, Switzerland, pp 143-152.

Squish, *http://code.google.com/p/libsquish/*

Wang, Z. et al., 2004. Image quality assessment: from error visibility to structural similarity. *In IEEE Transactions on Image Processing*, Vol. 13, No. 4.

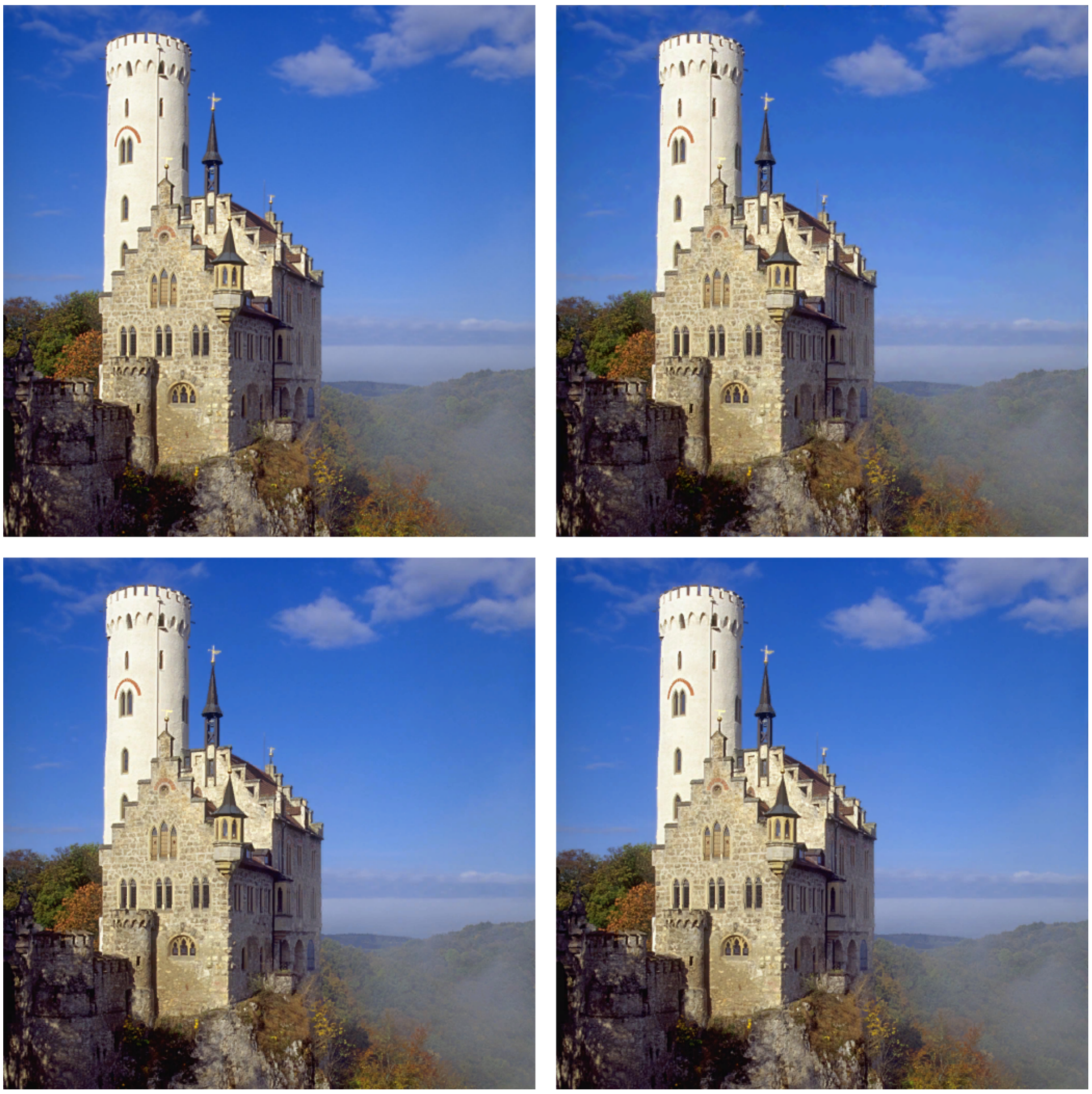
Figure 2. Original (top left), encoded with PVRTC (up right), encoded with squish (bottom left) and with line matching (bottom right).

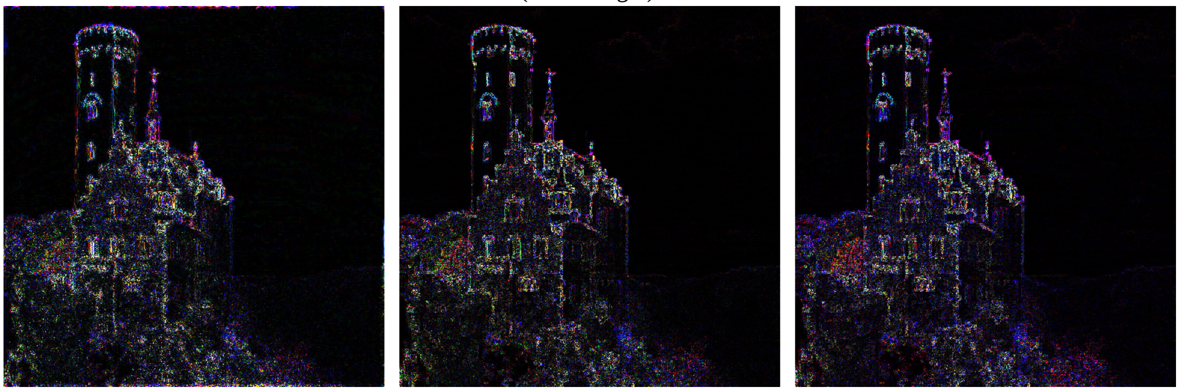
Figure 3. Sum of squared differences between original and encoded image for PVRTC (left), line matching (middle) and squish (right).