Eurographics/ IEEE-VGTC Symposium on Visualization 2010
G. Melançon, T. Munzner, and D. Weiskopf
(Guest Editors)

Volume 29 (2010), Number 3

Topographic Map Visualization from Adaptively Compressed Textures

C. Andujar[1]

[1]MOVING research group, Universitat Politècnica de Catalunya

Abstract

Raster-based topographic maps are commonly used in geoinformation systems to overlay geographic entities on top of digital terrain models. Using compressed texture formats for encoding topographic maps allows reducing latency times while visualizing large geographic datasets. Topographic maps encompass high-frequency content with large uniform regions, making current compressed texture formats inappropriate for encoding them. In this paper we present a method for locally-adaptive compression of topographic maps. Key elements include a Hilbert scan to maximize spatial coherence, efficient encoding of homogeneous image regions through arbitrarily-sized texel runs, a cumulative run-length encoding supporting fast random-access, and a compression algorithm supporting lossless and lossy compression. Our scheme can be easily implemented on current programmable graphics hardware allowing real-time GPU decompression and rendering of bilinear-filtered topographic maps.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: 3D Graphics and Realism—Texture

1 Introduction

Topographic maps are a valuable tool for representing and conveying geographic entities such as political borders, roads, rivers, buildings and cadastral information. Recent user studies have shown that regular map users often prefer to overlay digital terrain models with topographic maps rather than alternative representations such as orthophotos [PM05]. Although vector-based GIS data provides a more flexible representation of geographic features, raster-based topographic maps such as those published by governments and mapping companies are ubiquitous in current geoinformation systems. These maps are carefully designed by professionals following well-known cartographic principles and aesthetic rules which facilitate object recognition and transfer of contextual meaning, features often lacking in vector data layers selected and composited in real-time.

Storing and accessing large texture maps is still a challenging problem in computer graphics and visualization. Since dedicated texture memory is a limited resource, a common approach is to load texture tiles on-demand, using caching and speculative prefetching techniques [CGG*03] so as to minimize latency times while navigating through the digital model. Using compressed textures might greatly reduce latency times on such systems, either by allowing the full texture set to fit in dedicated memory, or by minimizing cache faults and texture swaps.

Current graphics hardware supports a number of compressed texture formats, but none of them works well with typical topographic maps. On the one hand, topographic maps contain a large amount of high-frequency content (such as text characters) which has to be preserved to ensure proper readability. On the other hand, typical topographic maps make use of a limited number of colors and thus exhibit large uniform areas (Figure 4). Block-based compressed texture formats such as S3TC DXT1, 3Dfx FXT1 and ATI 3Dc use a uniform bitrate across the image, losing information in high-detail regions while over-allocating space in low-detail regions. This lack of adaptivity often results in visible artifacts all over the texture, which are particularly noticeable around sharp image features. The availability of programmable shaders in low-cost graphics hard-

Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

ware provides new solutions for texture compression. Today's programmable graphics hardware allows the integration of texture decoders into the rasterization pipeline. Thus, only compressed data needs to be stored in dedicated texture memory provided that each texture lookup includes a decoding step. Texture compression schemes exhibit special requirements which distinguish them from general image compression systems:

Decoding Speed. In order to render directly from the compressed representation, the scheme must support fast decompression so that the time necessary to access a single texel is not severely impacted. A transform coding scheme such as JPEG is too expensive because extracting the value of a single texel would require an expensive inverse Discrete Cosine Transform computation.

Random Access. Texture compression formats must provide fast random access to texels. Compression schemes based on entropy encoding (e.g. JPEG 2000) and deflate encoding (e.g. PNG) produce variable rate codes requiring decompressing a large portion of the texture to extract a single texel.

There is a large body of work on compressing coherent data, particularly in the context of images. However, most compression schemes involve a sequential traversal of the data for entropy coding, and therefore lack efficient fine-grain random access. Compression techniques that retain random access are more rare. Adaptive hierarchies such as wavelets and quadtrees offer spatial adaptivity, but *compressed* tree schemes generally require a sequential traversal and do not support random access.

Contributions In this paper we present a locally-adaptive texture compression scheme suitable for both lossless and lossy encoding of topographic maps. Novel elements include:

- Use of space-coherent scans such as the Hilbert scan to exploit texel correlation.
- Efficient encoding of homogeneous regions through arbitrarily-sized texel runs which provide a more flexible approach to segment coherent data from fine detail.
- A cumulative run-length encoding of coherent texel runs supporting fast random-access. Cumulative RLE allows for texel recovery using binary search.
- Compression algorithms for lossless and lossy encoding.
- A new approach for computing bilinear-filtered samples from the compressed texture representation.
- A benchmarking with widespread compressed texture formats and quadtree-based approaches.

Limitations Our approach is not intended to compress topographic maps with large color gradients (e.g. those using hypsometric tints or shaded reliefs), as these would suffer from color depth reduction and also limit the detection of homogeneous regions.

2 Previous work

Topographic maps Analytical vector data is one of the main categories managed by geoinformation systems. Methods for the visualization of vector data on a digital terrain model can be broadly divided into two different classes: texture-based and geometry-based techniques. The first group of methods rasterizes the vector data into a texture and projects it onto the terrain geometry by applying texture mapping techniques [KD02, SGK05]. Methods belonging to the second class create geometry from the vector data and render them as separate geometry with an additional offset. Since most terrain representations are based on view-dependent multiresolution meshes, geometry from the vector data has to be adapted to each LoD [WKW*03] or extruded along the nadir direction [SK07]. A few papers address the rendering of antialiased vector graphics encoded as procedural textures (see [NH08] for a review). Recent approaches [QMK08, NH08] support general vector graphics defined as layers of filled and stroked primitives. These methods put the emphasis on high-quality anti-alising when zooming rather than on image compression, and require vector data as input. Unfortunately, many topographic maps are distributed only in raster format. Raster-based topographic maps, such as those published by governments, lack the flexibility and customization potential of their vector-based counterparts, but offer a professional design where selection, symbolization, overlay, and layout of geographic entities (such as text location, flow and size) have been optimized to a particular scale taking into account a number of factors. Digital topographic maps created with the help of cartographic expert systems and manually-edited by cartographers offer superior quality which can be hardly achieved by real-time rendering of GIS vector data.

General image compression The reasons why conventional image compression schemes such as PNG, JPEG and JPEG2000 are not suitable as compressed texture formats have been extensively reported in the literature, see e.g. [BAC96, LH07]. Most compression strategies, including entropy coding, deflate, and run-length encoding, lead to variable-rate encodings which lack efficient fine-grain random access.

Vector quantization and block-based methods Vector quantization (VQ) has been largely adopted for texture compression [NH92, BAC96]. When using VQ, the texture is divided into a set of blocks. VQ attempts to characterize this set of blocks by a small set of representative blocks called a codebook. The image is encoded as a set of indices into this codebook, with one index per block of texels [BAC96]. VQ can also be applied hierarchically [VG88].

Block-based data compression has been a very active area of research [MB98, Fen03, SAM05]. S3TC DXT1 [BA, KKZ99] stores a 4×4 texel block using 64 bits, consisting of two 16-bit RGB 5:6:5 color values and a 4×4 two-bit

lookup table. ETC [SAM05, SP07] also stores a 4×4 texel block using 64 bits, but luminance is allowed to vary per texel. All these fixed-rate schemes are non-adaptive and thus they over-allocate space in low-detail regions while losing quality in detailed parts. Since topographic maps use a limited set of distinct colors, replacing the color of some texels with an interpolation of the color of neighboring texels produces visible artifacts in regions with high-frequency detail.

Adaptive compression Hierarchical structures such as wavelets and quadtrees offer spatial adaptivity and support multi-resolution compression. However, most *compressed* tree structures require sequential traversals and therefore give up random access [LH07]. Kraus and Ertl [KE02] propose a two-level hierarchy to represent a restricted form of adaptive texture maps. The hierarchical representation consists of a coarse, uniform grid where each cell contains the origin and scale of a varying-size texture data block. Inada and McCool [IM06] propose a variable-rate, lossless compression scheme exploiting image sparsity. The texture is divided into tiles of 4×4 pixels which are encoded using a B-tree indexing structure. Internal nodes of the B-tree store key-pointer pairs (Morton codes are used as tile keys), whereas leaf nodes encode a variable number of tiles compressed using a color differencing scheme. Our work also uses space-filling curves and key search, but differs from [IM06] in that our space-filling curve traverses the pixels inside each tile, instead of the tiles of the whole image. As a consequence, our Hilbert curve keys (and the corresponding search process) are local to each tile. This allows cumulative run length encoding to exploit texel correlation along the curve at the finest-grain level. Other adaptive representations include page tables [LKS*06] and random-access quadtrees [LH07]. Our approach differs from prior adaptive schemes in that coherent regions are allowed to have any size (they do not have the power-of-two size restriction) and allows a larger class of shapes (not just squares), thus allowing a better adjustment to the boundary of coherent regions.

Data order and compression Data orders such as Hilbert scans have been extensively explored in the compression literature, especially in the database community. A few papers discuss color image compression using Hilbert scans (see e.g. [Col87, KNB98]), but they lack random-access and therefore cannot be used for texture compression.

3 Locally-adaptive compression

3.1 Overview

The input of our algorithm is a digital topographic map Im containing $w \times h$ color tuples. Our compressed representation operates on a one-dimensional sequence. Therefore our representation is parameterized by a bijective function $f : \mathbb{Z}^2 \rightarrow \mathbb{Z}$ which defines a scan traversal of the pixels in Im. The function f can be defined in a variety of ways, including traversals based on space-filling curves, line-by-line scans, and block-based scans. A desirable property of the scan function f is locality-preservation, i.e. $\left\| f^{-1}(k) - f^{-1}(k+1) \right\|$ should be kept small.

The function f maps two-dimensional texel data into a one-dimensional sequence $L = (c_0, c_1 \ldots c_{wh-1})$ where $c_k = Im(f^{-1}(k))$. A simple example based on a Hilbert scan is shown in Figure 1. Adaptive compression of L can be achieved by grouping neighboring texels with similar color into texel runs, and computing the run-length encoding (RLE) of L. The grouping algorithm might collapse only identical texels for lossless encoding, or approximately-similar texels for lossy encoding. A simple grouping algorithm is described in Section 3.4. The result after grouping and RLE encoding is a collection of pairs (c,r) where c represents a color and r is the run length. The run sequence will be referred to as $R = ((c_0, r_0) \ldots (c_{n-1}, r_{n-1}))$. The reconstruction of the original sequence from R is given simply by $\tilde{R} = (\underbrace{c_0, c_0 \ldots c_0}_{r_0} \ldots \underbrace{c_{n-1}, c_{n-1} \ldots c_{n-1}}_{r_{n-1}})$.

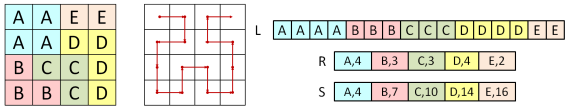


Figure 1: *Sample image and corresponding sequences L, R, S obtained with a Hilbert scan.*

This representation is not suitable as a compressed texture format, as the evaluation of $\tilde{R}(k)$ for any given k takes $O(n)$ time, n being the number of runs. A better option for on-the-fly decompression is to replace run lengths by cumulative run lengths. We denote this encoding as $S = ((c_0, s_0) \ldots (c_{n-1}, s_{n-1}))$ where $s_k = \sum_{i=0}^{k} r_i$. Note that $s_{n-1} = \sum_{i=0}^{n-1} r_i = w \cdot h$, i.e. the number of pixels in the image. The main advantage of S with respect to R is that the computation of $\tilde{S}(k)$ for any given k takes $O(\log_2 n)$ time, as it simply accounts for a binary search of (the interval containing) k in the sorted sequence $(s_0 \ldots s_{n-1})$, see Figure 1.

This new representation has two major limitations, though. First, a fixed-length encoding of each cumulative value s_k requires $\log_2(w \cdot h)$ bits. On a 512×512 input image, each s_k would require $2\log_2 512 = 18$ bits. This would severely limit compression performance since no entropy encoding can be applied without interfering with random access. A second limitation is that worst-case $O(\log_2 n)$ time can still be a limiting speed factor for GPU decoding. For instance, the 8:1 encoding of a 512×512 image results in a search space of $n = 512^2/8 = 32{,}768$ runs. Since $\log_2(32{,}768) = 15$, the decompressor will have to perform in the worst case 15 lookups to S in order to evaluate $\tilde{S}(k)$.

A solution to the above problems is to decide a block size B and uniformly subdivide L into b blocks of size B. This uniform partition of L induces a *non-uniform* partition of S

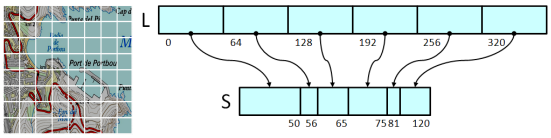


Figure 2: *Partition of L into uniform blocks defining a non-uniform partition on its compressed version S.*

into another b blocks (see Figure 2). A first consequence is that each cumulative run length is now upper-bounded by B, i.e. each s_k value requires only $\log_2 B$ bits. For $B = 64$, this accounts for 6 bits. Furthermore, the above subdivision can be used to reduce the range of the binary search required to decode a texel. The subdivision of L is uniform and hence it is not required to be stored explicitly, provided that B is known. The subdivision induced on S is non-uniform and must be encoded explicitly. A simple option is to encode each block as a pair (o_i, l_i) where o_i is an index to the origin of the i-th block on S and l_i is its length. This results in a collection of b pairs that will be referred to as *index data*, $I = ((o_0, l_0), \ldots (o_{b-1}, l_{b-1}))$. Storing I allows for performing binary search locally on each block in $O(\log_2(B))$ time. For $B = 64$, *at most* 6 (dependent) texture accesses are required to decode a texel inside a block. Indeed, we show that *on average* only about 3.0 texture lookups are needed; on the test images this number was in the range 2.71 to 3.41, including fetching index data (discussed in Section 4.2).

3.2 Compressed representation

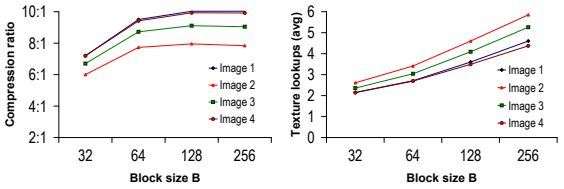
Our compressed representation includes an encoding of the texel runs S plus the index data I. We have control over several parameters that can be used to tradeoff compression rate and decoding speed:

Scan order. The function f defines a scan order to map 2D color data into a one-dimensional sequence. Locality-preservation is critical for exploiting texel correlation. We have tested several scan orders including raster scans and scans defined by space-filling curves such as Hilbert/Moore curves, and Z-order (Morton codes). It turns out that Hilbert/Moore curves clearly outperform other scans, as they provide much better locality preservation [DCOM00]. We have adopted the Hilbert scan for our test implementation.

Block size. Recall that parameter B determines exactly the block size of uniform subdivisions on L, whereas it is only an upper bound of the block size of the corresponding subdivisions on S. The value of B has a varying effect on compression ratio and decompression speed. In particular, choosing a small value for B has the following positive (+) and negative (-) effects:

- (+) Resulting blocks will be smaller, thus reducing the range for the binary searches. This means a *smaller number of accesses for the worst-case scenario*, which occurs when the corresponding block in S has exactly B unit-length runs.
- (+) Less bits ($\log_2 B$) are required to encode s_k values, thus reducing memory space.
- (-) Since B is an upper bound for run length, more runs will be needed to encode large coherent areas. Encoding more runs means consuming more space.
- (-) Smaller block size also means a larger number of indices to encode, as $|I| = b = \frac{w \cdot h}{B}$.

In our prototype implementation we use $B = 64$, which gives a good balance between compression rate and decoding speed (Figure 3). Cumulative run lengths can be encoded using 6 bits/run, binary searches require *at most* 6 textures accesses, and the length of index data is a $\frac{1}{64}$-th of the image size.



(a) (b)

Figure 3: *Impact of the block size on (a) final compression ratio and (b) average number of texture lookups required to decode a texel. Test images are shown in Figure 4.*

Color encoding. In our experiments we tested two color encodings: RGB 8:8:8 for lossless compression, and 9-bit indexed color with a 512 color palette P. Since typical topographic maps make use of a limited number of colors, color depth reduction mainly causes a slight color shift while preserving all important image features. The decision of the palette size was taken after analyzing the minimum PSNR of quantizing four 1:5000 2048×2048 topographic maps with different palette sizes. Resulting PSNR values were 38.1 dB, 45.1 dB, 46.1 dB and 46.8 dB for 6, 7, 8 and 9-bit palettes, respectively. Quantization with 9-bit palettes resulted in visually indistinguishable images, preserving faithfully also colors from antialiased edges.

Cumulative run-length encoding. Since $s_k \in [1, B]$, a natural choice is to simply store $s_k - 1$ using 6 bit integers.

Index data encoding. Index data consists of b pairs (o_i, l_i) where o_i is an index to the origin of the i-th block on S and l_i is its length. Each index o_i can be encoded with $\left\lceil \log_2 \frac{w \cdot h}{c} \right\rceil$ bits, c being the average run length, whereas $l_i$values require $\log_2 B$ bits. Thus encoding them separately requires $\left\lceil \log_2 \frac{w \cdot h}{c} \right\rceil + \log_2 B$ bits. The number of bits required for encoding o_i can be reduced by uniformly subdividing the input image (or equivalently L) into equal-sized clusters. For example, a 512×512 image can be sub-

divided into 64 clusters, each cluster containing 64×64 texels (i.e. 64 blocks). We can store separately the sequence $M = \{m_0, \ldots m_{63}\}$, where each meta-index m_k is defined as $m_k = o_{64k-1}$, i.e. the beginning of (the first block of) the k-th cluster on S. This allows encoding the indices o_i in I as cluster-relative offsets. Assuming an average run length of 4 texels, index data now requires only 16 bits/index (6 bits for l_i plus $\log_2(64^2/4)$=10 bits for o_i).

Data layout. Our complete compressed representation is a tuple (I,S,M,P), where I is index data, S is the cumulative run-length encoding, M are the meta-indices, and P is the color palette (if indexed-color mode is used). Our prototype implementation encodes I and S as a 8:8:8:8 RGBA texture, whereas M and P are shader variables: M is just a uniform array of 64 unsigned integers (assuming 64 clusters per texture) and P is an array of 512 (r,g,b) tuples. All compression rates reported include the whole tuple (I,S,M,P).

3.3 Decompression algorithm

The decompression algorithm takes as input the compressed texture representation $C = (I, S, M, P)$ and integral texel coordinates (i, j), and outputs the color of texel (i, j) through the following steps:

1. Map (i, j) into the one-dimensional structure L using the scan traversal defined by f, i.e. $k \leftarrow f(i, j)$.
2. Compute the block b in S containing texel k, i.e. $b \leftarrow k/B$.
3. Access to I_b to compute the lower bound o and the upper bound e for the binary search.
4. Compute the color c of the run containing texel $(k \bmod B)$ by binary search on the ordered sequence $s_o \ldots s_e$.

Note that the above code makes a single texture lookup to retrieve index data, plus a maximum of 6 texture lookups to retrieve color data. A last optimization consists in interleaving index data (o_i, l_i) with the texel run (c,s) that will retrieved first during binary search. This can be done by encoding (o_i, l_i) in R,G components and (c,s) in B,A components (recall that each of these tuples fit in 16 bits). This allows retrieving the beginning of the block and the first run to search for with a single texture lookup. As a consequence, texels in completely coherent blocks are decoded with a single texture lookup. Decompression speed is discussed in Section 4.

3.4 Compression algorithm

We now present two simple algorithms (for size-driven and error-driven compression, resp.) to convert an input image into a compressed representation $C = (I, S, M, P)$. Both algorithms share the same high-level structure and support true color and indexed-color compression:

1. Create an initial RLE representation R of the input image by scanning the input image in the order defined by function f (actually a Hilbert scan) and creating a unit-length run $(c, 1)$ for each color c encountered in the image.
2. Compress R by iteratively grouping neighboring runs until the compression goal is satisfied. For error-bounded compression, grouping stops when no more run pairs can be collapsed within a user-provided tolerance. For fixed-rate encoding, grouping stops when a user-defined compression ratio is reached. Runs from different blocks are not grouped together, as discussed in Section 3.2.
3. Create a cumulative run-length encoding S by replacing each run $(c_k, r_k) \in R$ by (c_k, s_k) with $s_k = \sum_{i=o}^{k} r_i$.
4. Create the index data I by traversing S and adding an index k-1 for each pair (c_k, s_k) with $s_k = B$. These indices correspond to the position of the last run in each block.
5. Compute meta-index data M with $m_k = o_{B \cdot k-1}$

Compression in indexed-color mode is similar, but a preprocessing is applied to the input image to reduce the color depth to 512 colors, replacing colors by indices to a color palette P. In our experiments color depth reduction was implemented using the Median Cut algorithm [Hec82]. Actual compression is performed in step 2, whose implementation differs depending on whether size-driven or error-driven compression is required.

Size-driven compression For size-driven simplification, the user provides a target compression ratio and run grouping stops when this compression ratio is reached. Our outer optimization strategy of the grouping algorithm is similar to that of greedy iterative simplification algorithms. The grouping algorithm (step 2) can be summarized as follows:

1. For each pair of neighboring runs (c_i, r_i), (c_{i+1}, r_{i+1}) in R, compute the cost E_i of collapsing that pair.
2. Insert all the pairs in a heap keyed on cost with the minimum-cost pair at the top.
3. Iteratively extract the pair (c_i, r_i), (c_{i+1}, r_{i+1}) of least cost from the heap, group this pair in R, and update the cost of the adjacent pairs.

During grouping, each run is represented as a triple (C_i, C_i^2, r_i) where C_i (resp. C_i^2) is the sum of the (resp. squared) colors of the r_i texels in the run. Merging two runs simply involves a component-wise addition of these triples. The cost produced by joining two runs (C_i, C_i^2, r_i) and $(C_{i+1}, C_{i+1}^2, r_{i+1})$ can be computed in the L_2 error sense as $E = C_i^2 + C_{i+1}^2 - (C_i + C_{i+1})^2/(r_i + r_{i+1})$. The average run color is simply $c_i = C_i/r_i$. We used colors in RGB space. In color-index mode, the color index resulting from joining two runs is the most used index in the run texels.

Error-driven compression For error-driven simplification, the user provides a maximum color error. In this case, compression is much faster as no heap is used: runs are visited in sequence and each run is grouped with his adjacent runs as long as their collapse does not exceed the user-defined error. This algorithm runs in $O(n)$ time, n being the number of texels in the image. In the experiments discussed in next section, we benchmarked the error-driven compression in lossless mode (each color being represented as RGB 8:8:8),

Image	Lossless	Quasi-lossless		Lossy	
	CR	CR	PSNR	CR	PSNR
Image 1	4.5	9.5	47.6	11	39.1
Image 2	3.7	7.9	47.1	8.3	39.1
Image 3	4.1	8.7	47.6	10	38.3
Image 4	4.3	9.4	47.6	10.7	38.7

Table 1: *Compression ratio (CR) and PSNR with our lossless, quasi-lossless and lossy compression.*

in quasi-lossless mode (converting the image into indexed-color in a preprocess, and allowing no color index change during compression), and lossy mode (also indexed-color but with varying error tolerances).

4 Results and discussion

We have implemented the compression and decompression algorithms described above and tested them on four different 1:5000 topographic maps covering urban, mountainous and shoreline regions (Figure 4). Unless otherwise stated, all compressed textures were created using the Hilbert scan, RGB 8:8:8 color (lossless) or 9-bit indexed color (quasi-lossless), block size $B = 64$, and cluster size 64.

4.1 Image quality and compression ratio

Figure 4 shows the test images compressed using our scheme in lossless, quasi-lossless and lossy modes. Note the high visual quality of resulting images, which preserve highly-detailed features. Compression rates and PSNR values are shown in Table 1. Quasi-lossless and lossy modes provide much higher compression rates than lossless compression because indexed-color is used. Note that standard 8-bit palettized compressed formats only provide 3:1 compression, whereas our compression rates are 2-3 times higher, depending on image coherence. Resulting PSNR values for quasi-lossless compression were all above 47 dB (Table 1). From now on we only show results in quasi-lossless mode.

Figure 5: *Close-up views showing the ability of our scheme to perform adaptive compression. Input image (left), detail (middle), and color-coded runs (right).*

Figure 5 illustrates the ability of our scheme to segment coherent regions from detailed parts. The presence of features in otherwise coherent areas does not produce oversegmentation. Note that the input images had antialiased lines. Relative frequencies of run-lengths for a sample image are shown on the left of Figure 6. Short-length runs clearly predominate (about 70% of the runs are unit-length), allowing the preservation of detail on high-frequency regions, followed by runs with the maximum allowed length $B = 64$, with relative frequency varying from 8% to 10%, depending on spatial coherence in the image. Figure 6-right shows the number of input texels represented by each run, grouped by run length. Since longer runs represent a higher number of texels, about 60% of the input texels are encoded in maximum-length runs. This has an important consequence for decoding speed, as texels belonging to such runs can be decoded with a single texture lookup (the corresponding block contains a single run), thus significantly reducing average decoding times.

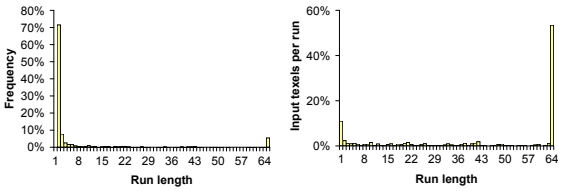


Figure 6: *Relative frequencies of run-lengths (left) and percentage of input texels per run (right) for image 1.*

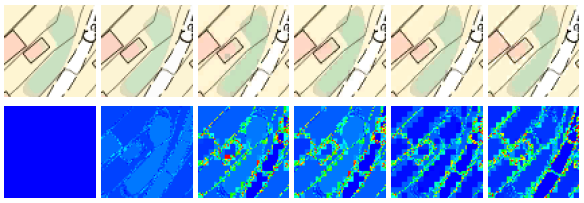
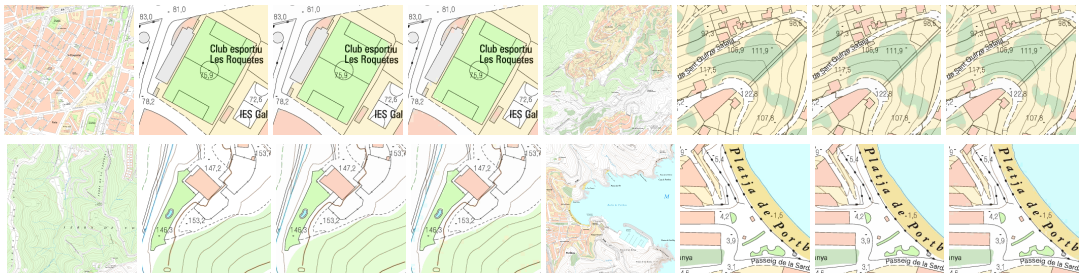


Figure 7: *Comparison with alternative compressed formats: (a) our lossless compression, i.e. identical to the input image; (b) our quasi-lossless compression; (c) DTX1 fast compression; (d) DXT1 high quality; (e) DXT1 highest quality; (f) ETC. Images differences have been amplified* 10×.

We also compared our scheme with the readily-available compressed texture formats DXT1 and ETC. These fixed-rate formats target general color images and do not take advantage of the reduced number of colors and spatial coherence of typical topographic maps. Figure 7 compares our approach (in quasi-lossless mode) with these formats. Compressed textures in these formats were generated with ATI Compressonator 1.50. DXT1 textures were generated with the three quality settings available in the ATI tool (highest quality, normal, and fast compression). Since only four different colors are possible for each 16-texel block, DXT1 and ETC produce visible artifacts around sharp image features, damaging the uniformity of some coherent regions. In some regions the 4×4 blocks used by DXT1 and ETC are clearly



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 4: *Compressed textures with increasing tolerance values: input image (a,e), and results with lossless compression (b,f), quasi-lossless compression (c,g), and lossy compression (d,h). Compression rates are listed in Table 1.*

	DXT1 HQ	DXT1	DXT1 Fast	ETC	Ours
CR	6:1	6:1	6:1	6:1	8:1
PSNR	33.22	33.27	34.72	33.02	47.1
Max error	15.3%	12.1%	11.1%	16.4%	1.6%

Table 2: *Comparison of DXT1 and ETC compressed formats with our quasi-lossless approach for test Image 2.*

Filtering	Lookups (avg)	1024^2	512^2
Nearest-neighbor	3.0	376 fps	1325 fps
Bilinear	12.0	108 fps	363 fps
Pseudo-bilinear	4.8	221 fps	716 fps

Table 3: *Decompression performance.*

distinguishable. Compression rates and errors are reported in Table 2. Note that our approach achieves significantly higher compression rates with better PSNR and lower error. Maximum error values also reveal the poor behavior of DXT1 in images with blocks with high variance.

4.2 Performance

Table 3 shows rendering times of our prototype shader running on NVidia GTX 280 hardware with different viewport sizes. Note that for a typical 8:1 compression, only about 3.0 texture fetches are required, on average (Table 5). Although our decompression rates are about 10× slower than natively supported formats such as DXT1 and ETC, they still allow high-speed, real-time rendering.

Our error-driven compression algorithm is able to compress large textures in a few milliseconds. Table 4 shows compression times for varying texture sizes. Lossy compression slightly increases compression times as more run pairs need to be collapsed. All times were measured on an Intel Core2 Quad Q9550 at 2.83GHz. For textures larger than 512×512 we increased the number of clusters to force 16-bit indices in I.

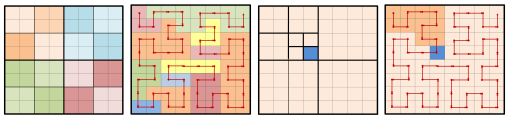
Texture size	# texels	Compression time (ms)		
		Lossless	Quasi-lossless	Lossy
256×256	65,536	12	10	12
512×512	262,144	55	52	61
1024×1024	1,048,576	218	200	225
2048×2048	4,194,304	830	816	841

Table 4: *Compression performance*

4.3 Comparison with tree-based methods

Our approach provides a number of advantages with respect to tree-based methods. Our framework is not tied to a particular grid and thus provides a more flexible approach to segment coherent data from fine detail. In particular, our approach can encode regions with a variety of shapes and sizes. Tree-based methods can only segment coherent data in square regions whose size and boundaries are given by the tree subdivision. Consider for example a 8×8 texel block. A quadtree-based approach (Figure 8-a) can only detect coherent data on regions corresponding to quadtree nodes: one 8×8 root, four 4×4 nodes and sixteen 2×2 nodes. This accounts for up to 21 potentially coherent regions with only three possible sizes, $\{4, 16, 64\}$ texels. Our run-based approach is able to extract coherent regions with any integer size in $\{2, 3, \ldots 64\}$. We support e.g. 3-texel, L-shaped runs. Since for a run with length l we have $64 - l + 1$ choices for its start position, this accounts for $\sum_{l=1}^{63} i = 2016$ potentially-coherent regions in a 8×8 block, i.e. two orders of magnitude more regions than quadtree-based approaches. This results in much more flexibility in adapting the subdivision to the boundary of coherent data (see Figure 8). A further consequence is that our approach is less sensitive to outlier texels. A single outlier texel in an otherwise coherent region can force several subdivisions of the quadtree (Figure 8-c), whereas in our approach an outlier texel would split a single run covering m texels into three runs covering $(m-n)$, 1 and $n-1$ texels, resp. (Figure 8-d).

We benchmarked our compression scheme with a quadtree subdivision. We built a quadtree stopping re-



(a) (b) (c) (d)

Figure 8: *Our approach vs quadtree-based methods: (a) Potentially-coherent regions supported by a quadtree on an 8×8 block; (b) Example of potentially-coherent regions with a Hilbert scan; (c) A single outlier pixel causes the quadtree to include 13 nodes (including internal nodes); (d) In the same situation, our approach produces only 3 runs.*

	Quadtree-based			Our approach		
	# nodes	pixels	TL	# runs	pixels	TL
Image 1	2.18M	2.56	10.1	701K	6.51	2.71
Image 2	2.30M	2.43	10.6	871K	5.13	3.41
Image 3	1.41M	3.96	9.9	768K	5.89	3.04
Image 4	1.14M	4.91	8.7	707K	6.44	2.67

Table 5: *Quadtree subdivision vs our approach. Column pixels shows the average number of pixels per leaf node (quadtrees) or texel run (our approach). TL is the average number of texture lookups required to decode a single texel.*

finement at perfectly homogeneous nodes (again 9-bit indexed color was used), and compared it with our compression scheme in quasi-lossless mode (both compressed formats reproduce exactly the same image). All maps were 2048×2048. Table 5 compares the average number of texels per leaf node in the quadtree against the average texel length, and the average number of texel lookups required to decode a texel. Our approach is able to detect larger regions (with 5-6 texels/run) than the quadtree subdivision, resulting is fewer items to encode (about 800K runs vs. more than one million tree nodes). Even with state-of-the-art encodings of the tree structure [LH07], at least 8 bits/node are required, resulting in lower compression rates. Our approach also wins in the average number of texture lookups. It must be acknowledged though that quadtree-based methods would compress better than our approach on very sparse images. An extreme case would be a solid blue map corresponding to a sea region, which would require a single quadtree node vs. $(wh)/B$ runs in our case. However, uniformity in general topographic maps occurs more frequently at small-scale (e.g. empty space between contour lines) rather than at large-scale.

5 Bilinear filtering

A naive approach to implement bilinear filtering is to execute four times the decompression algorithm proposed in Section 3.3. This would multiply by 4 the number of dependent texture lookups, resulting in 12 texture fetches, on average (rendering performance though does not decrease by this amount, as most memory reads to index data and the first visited runs should be intercepted by memory caches). An alternative approach is to apply deferred filtering [FAM*05], i.e. decoding a subset of the texture at its native resolution in a first rendering pass, and rendering the decoded data using the native GPU filtering in a second pass. Deferred filtering has proven to be helpful for high-quality rendering of compressed volume data, as it guarantees that the decoding cost is incurred only once per texel. Instead, we propose a single-pass approach which exploits data coherence. A first straightforward optimization is to store the index and the runs visited during the decoding of the first texel, and try to re-use these values on the three remaining texels. The probability of a random sample (s,t) requiring a group of 2×2 texels in the same $n\times n$ block can be shown to be the area ratio between a half-pixel offset of the block and the block itself, i.e. $(n-1)^2/n^2$. For 8×8 blocks as in our case, this means that index data can be re-used for about 76% of samples. This simple optimization decreases the average number of lookups for bilinear filtering from 12 to 9.7 . A more drastic optimization, at the expense of a minimal visual quality loss, is based on the following observation: since topographic maps contain many uniform areas, many samples fall in a 2×2 group of texels with identical color (Figure 9). Since our decompression algorithm decodes a texel by retrieving information about a whole run containing the texel, it should be possible to identify whether the texel belongs to an homogeneous region or not, and apply bilinear filtering or not accordingly. The decompression algorithm in Section 3.3 obtains the run (c,s) containing the current texel. Cumulative run length does not give, by itself, enough information to know the length of the run (unless the preceding run is accessed, but we do not want to add more texture lookups). So our approach is to use a bit to indicate whether the length of the run is above some threshold L. Since we use 9-bit indexed-color and cumulative run lengths need 6 bits, each run still fits in 2 bytes after adding this extra flag. All compression rates reported so far already included this *coherence flag*. Therefore, the idea is to decode the first texel as in nearest-neighbor sampling, and act accordingly to the coherence flag.

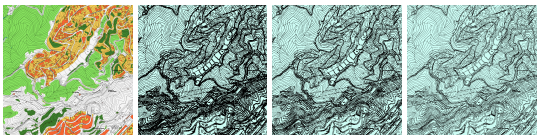


Figure 9: *We apply bilinear filtering only to highly-detailed regions (shown in black). Homogeneous regions (with at least L=16, 8 and 4 texels, respectively) can be reconstructed without additional texture accesses.*

In order to reproduce standard bilinear interpolation, we should also identify whether the texel belongs to the interior of an homogeneous region (thus requiring no additional texture lookups) or to its border (thus potentially requiring

bilinear interpolation). Due to the large range of shapes produced by arbitrary cuts of a Hilbert curve (Figure 5), knowing whether a texel is in the interior or on the border of a run is expensive to compute. Therefore, we followed a slightly different approach which does not reproduce exactly standard bilinear interpolation, but provides smooth output. Given a (s,t) sample, we decode the texel containing point (s,t) as in nearest-neighbor sampling. If the coherence flag indicates that the texel belongs to a coherent region, then the color returned for the sample is that of the texel (no additional texture accesses are required). Otherwise, we apply bilinear interpolation as described below. Note however that all samples inside coherent texels are reconstructed with a uniform color, regardless of whether they are on the border or not. To provide a smooth transition around border texels, we slightly modify the way colors are interpolated for those texels without the coherence flag. Figure 10 shows the six different cases that might appear (under rotations and symmetries) on a 2×2 texel group, depending on the coherent/non-coherent property of each texel. Each image in the first row shows a 2×2 group of texels; the bottom-left texel being the one containing the (s,t) sample. Coherent texels are drawn in red and with a thicker border. All samples inside these red texels are reconstructed with no interpolation. We indicate, for one of the quadrants of the nearest-neighbor texel, which four colors must be interpolated so as to ensure a smooth transition between non-coherent and coherent regions. Figure 11 compares standard bilinear filtering with our modified approach. Texels in coherent regions (runs longer than L texels) need exactly the same number of texture lookups than with nearest-neighbor filtering (3.0 on average). Texels in non-coherent regions require bilinear interpolation, which requires 9.7 texture lookups, on average. However, since topographic maps are very coherent, most of the pixels need no extra texture lookups. The map shown in Figure 9, for example, contains 27.5% of non-coherent texels and 72.5% of coherent texels, considering $L = 4$. These percentages are quite representative of typical topographic maps. In summary, our modified bilinear interpolation algorithm requires, on average, about 4.8 texture lookups.

Mipmapping can be trivially supported by compressing each LOD image independently; this would incur approximately a 33% memory overhead with respect to the non-mipmapped version. Since a straightforward implementation of trilinear filtering doubles the average number of texture lookups, a more reasonable approach is to run-length encode only the higher resolution levels (e.g. levels 0 and 1), and to transition to uncompressed images for the coarser levels. As the first two LOD levels represent $\frac{1+1/4}{1+1/3} = 93.75\%$ of the texels, this would achieve a good trade-off between compression rate and decoding speed and opens the possibility of native anisotropic filtering at the coarser levels.

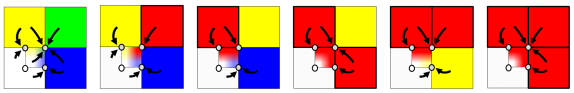


Figure 10: *Bilinear interpolation inside a texel quadrant.*

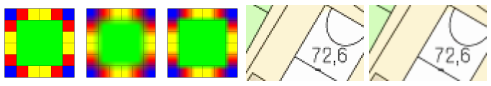


Figure 11: *Interpolation using nearest-neighbor (a,d), bilinear filtering (b), and our modified bilinear filtering (c,e).*

6 Conclusions

In this paper we have presented a locally-adaptive texture compression scheme specifically tailored for encoding topographic maps. Our approach achieves a good balance between compression of coherent regions and preservation of high-frequency details. When compared with alternative approaches such as DXT1 and 8-bit palettized textures, our compressed representation in quasi-lossless mode clearly wins in compression ratio and visual quality but loses in decoding speed. Its practical use thus depends on the amount of texture data the application has to handle in realtime. If uncompressed topographic maps fit in available GPU memory, our compression scheme is worthless as it would increase the GPU workload without any clear advantage. However, if topographic maps do not fit in GPU memory and have to be loaded on-the-fly, typically using prefetching and caching strategies, then our compressed representation allows a significant reduction of latency times and memory bandwidth, minimizing texture swaps and providing a virtual cache about eight times larger than the GPU texture memory.

Our compression scheme can be adapted to encode other types of graphics data encompassing spatial coherence and highly-detailed regions. In particular, we plan to extend our cumulative RLE approach to compress volume datasets for pre-shaded direct volume rendering [MHB*00] by grouping voxels according to pre-classified opacities, as proposed in [LL94]. Other interesting avenues for further research are to adapt our scheme for multi-resolution compression and to implement anisotropic filtering directly from the compressed representation by exploiting data coherence in homogeneous regions.

Acknowledgments The author would like to thank the anonymous reviewers for their helpful comments. This work has been funded by the Spanish Ministry of Science and Technology under TIN2007-67982-C02. Test images courtesy of Institut Cartografic de Catalunya.

References

[BA] BROWN P., AGOPIAN M.: EXT texture compression DXT1. opengl extension registry. http://opengl.org/registry/specs/EXT/texture_compression_dxt1.txt.

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *Proceedings of ACM SIGGRAPH '96* (1996), pp. 373–378.

[CGG*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Planet-sized batched dynamic adaptive meshes (P-BDAM). In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), IEEE Computer Society, pp. 147–154.

[Col87] COLE A. J.: Compaction techniques for raster scan graphics using space-filling curves. *The Computer Journal 30*, 1 (1987), 87–92.

[DCOM00] DAFNER R., COHEN-OR D., MATIAS Y.: Context-based space filling curves. *Computer Graphics Forum 19*, 3 (2000), 209–218.

[FAM*05] FOUT N., AKIBA H., MA K., LEFOHN A., KNISS J.: High-quality rendering of compressed volume data formats. In *Proceedings of the 2005 Eurographics-IEEE Symposium on Visualization* (2005), pp. 77–84.

[Fen03] FENNEY S.: Texture compression using low-frequency signal modulation. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 84–91.

[Hec82] HECKBERT P.: Color image quantization for frame buffer display. *SIGGRAPH Computer Graphics 16*, 3 (1982), 297–307.

[IM06] INADA T., MCCOOL M. D.: Compressed lossless texture representation and caching. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2006), pp. 111–120.

[KD02] KERSTING O., DÖLLNER J.: Interactive 3d visualization of vector data in gis. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems* (2002), pp. 107–112.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), Eurographics Association, pp. 7–15.

[KKZ99] KONSTANTINE I., KRISHNA N., ZHOU H.: Fixed-rate block-based image compression with inferred pixel values, 1999. US Patent no. 5956431.

[KNB98] KAMATA S., NISHI N., BANDOH Y.: Color image compression using a hilbert scan. *International Conference on Pattern Recognition 2* (1998), 1575–1578.

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. In *Proceedings of the Eurographics Symposium on Rendering* (2007), Eurographics, pp. 339–349.

[LKS*06] LEFOHN A., KNISS J., STRZODKA R., SENGUPTA S., OWENS J.: Glift: Generic, efficient, random-access GPU data structures. *ACM TOG 25*, 1 (2006), 60–99.

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of SIGGRAPH'94* (1994), ACM, pp. 451–458.

[MB98] MCCABE D., BROTHERS J.: Directx 6 texture map compression. *Game Developer Magazine 5*, 8 (1998), 42–46.

[MHB*00] MEISSNER M., HUANG J., BARTZ D., MUELLER K., CRAWFIS R.: A practical evaluation of popular volume rendering algorithms. In *Proceedings of the 2000 IEEE symposium on Volume visualization* (2000), pp. 81–90.

[NH92] NING P., HESSELINK L.: Vector quantization for volume rendering. In *VVS'92: Workshop on Volume visualization* (1992), pp. 69–74.

[NH08] NEHAB D., HOPPE H.: Random-access rendering of general vector graphics. *ACM Transactions on Graphics (TOG) 27*, 5 (2008), Article 135.

[PM05] PETROVIC D., MASERA P.: Analysis of user's response on 3d cartographic presentations. In *Proc. of the 22nd International Cartographic Conference* (A Coruña, Spain, 2005).

[QMK08] QIN Z., MCCOOL M., KAPLAN C.: Precise vector textures for real-time 3D rendering. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games* (2008), pp. 199–206.

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 63–70.

[SGK05] SCHNEIDER M., GUTHE M., KLEIN R.: Real-time rendering of complex vector data on 3d terrain models. In *In Proceedings of the 11th International Conference on Virtual Systems and Multimedia* (2005), pp. 573–582.

[SK07] SCHNEIDER M., KLEIN R.: Efficient and accurate rendering of vector data on virtual landscapes. *Journal of WSCG 15* (2007), 59–64.

[SP07] STRÖM J., PETTERSSON M.: ETC2: texture compression using invalid combinations. In *GH '07: Proceedings of symposium on Graphics hardware* (2007), Eurographics Association, pp. 49–54.

[VG88] VAISEY J., GERSHO A.: *Signal Processing IV: Theories and Applications*. 1988, ch. Variable rate image coding using quadtrees and vector quantization, pp. 1133–1136.

[WKW*03] WARTELL Z., KANG E., WASILEWSKI T., RIBARSKY W., FAUST N.: Rendering vector data over global, multi-resolution 3d terrain. In *VISSYM '03: Joint Eurographics - IEEE TCVG Symposium on Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 213–222.