High-quality compression of mipmapped textures

C. Andujar

MOVING research group, Universitat Politècnica de Catalunya, Barcelona, Spain

ABSTRACT

High-quality texture minification techniques, including trilinear and anisotropic filtering, require texture data to be arranged into a collection of prefiltered texture maps called mipmaps. In this paper we present a compression scheme for mipmapped textures which achieves much higher quality than current native schemes by exploiting image coherence across mipmap levels. The basic idea is to use a high-quality native compressed format for the upper levels of the mipmap pyramid (to retain efficient minification filtering) together with a novel compact representation of the detail provided by the highest-resolution mipmap. Key elements of our approach include delta-encoding of the luminance signal, efficient encoding of coherent regions through texel runs following a Hilbert scan, a scheme for run encoding supporting fast random-access, and a predictive approach for encoding indices of variable-length blocks. We show that our scheme clearly outperforms native 6:1 compressed texture formats in terms of image quality while still providing real-time rendering of trilinearly-filtered textures.

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

1. Introduction

Storing and accessing large texture datasets is still a challenging problem in computer graphics. The benefits of using compressed texture formats [BAC96] are manifold: (a) memory space savings, thus minimizing texture paging, (b) faster texture uploads, due to the reduced size, and (c) higher rendering speed, due to cache benefits and reduced bandwidth requirements. Note that (a,b) and (c) are often opposite goals: some sophisticated schemes achieve higher compression rates at the expense of higher decoding times.

Compressed texture formats natively supported by current hardware provide unsurpassed decoding performance but at the expense of a poor quality/space ratio. DXT1 and ETC formats, for example, provide an acceptable 6:1 compression ratio but at a substantial loss in image quality. Furthermore, since these formats use a uniform bitrate across the image, they loose 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. Furthermore, since each mipmap level is encoded independently, native methods do not exploit data redundancy across mipmap levels. These facts have prevented the widespread use of compressed textures in those applications aiming at providing the highest image quality. Native formats for RGBA textures such as DXT5 can be trivially extended to encode RGB textures with much higher quality [vWC07] but this results in a less attractive 4:1 compression ratio.

Multiresolution texture formats achieve better compression but fail to guarantee efficient random access to individual texels without assistance from specialized hardware. These approaches add a significant performance overhead because decoding a single texel often involves multiple (possibly dependent) lookups to the compressed texture data. This fact makes texture filtering a major issue.

In this paper we present a high-quality texture compression scheme for mipmapped RGB textures. We aim at exploiting image coherence across mipmap levels by encoding inter-level residuals instead of absolute color values. Ideally, we could use inter-level residuals across all levels of the mipmap pyramid, but this would require extensive data traversals and result in a prohibitive number of texture fetches during rendering. Alternative multiresolution approaches, such as those formats based on the wavelet transform [Per99, SR06, STC09] typically work with 4×4 texel blocks (thus encoding explicitly only two additional LOD levels) and suffer from decoding performance issues. Instead we focus on the encoding of the highest-resolution level, which represents about 75% of the samples in the whole pyramid. The image pyramid is encoded as follows. The upper levels of image pyramid (levels L_1 to L_n) are compressed with a native texture format. A native mipmapped texture is the most efficient way for computing trilinearly filtered samples. Since levels L_1 to L_n represent only one

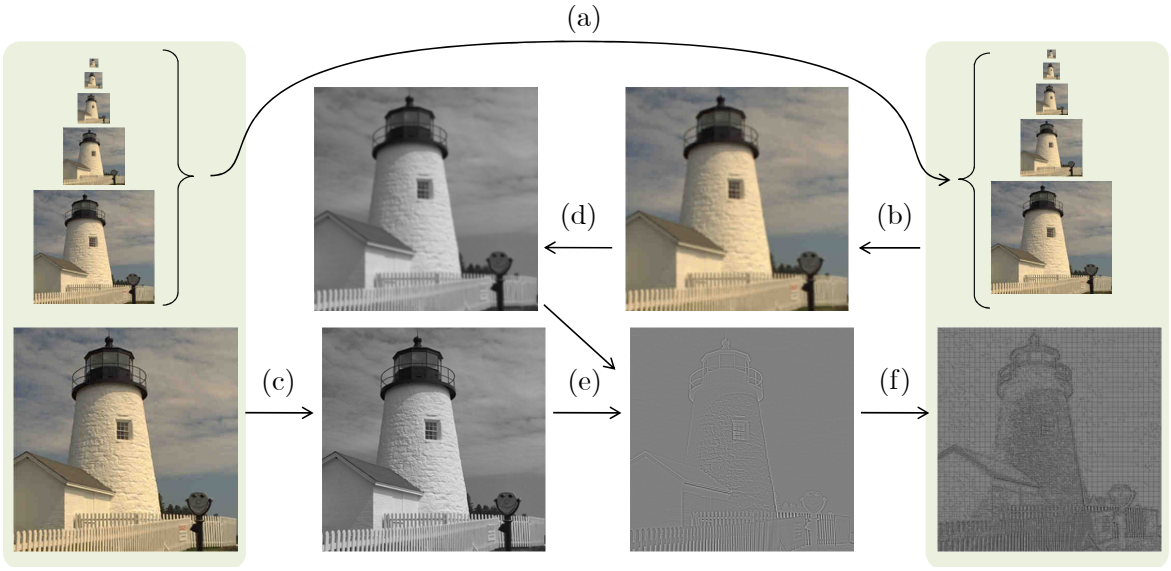

Figure 1: *Overview of our compression scheme. The input of the encoder is a mipmap pyramid with levels $L_0 \ldots L_n$. The upper levels $L_1 \ldots L_n$ of the pyramid are compressed using a high-quality native texture format (a). Level L_1 is decompressed and upsampled 2× (b), thus producing a low-pass filtered version of the highest-resolution mipmap L_0. The luminance signal is extracted by performing an RGB to YCoCg conversion on both the original (c) and the reconstructed image (d). Residuals are computed by subtracting (e) the reconstructed luminance signal from the original one. Sparsity in the residuals is exploited by traversing the pixels in Hilbert order and grouping pixels with similar residuals into runs (f). The compressed representation consists of the compressed mipmap pyramid plus a random-access encoding of the residual runs.*

fourth of the samples of the whole pyramid, we can adopt a high-quality but low-compression rate format without significantly increasing the overall compression rate.

In contrast, the highest-resolution level L_0 is encoded incrementally from its downsampled version L_1 (Fig. 1). It is well-known that a low-pass filtered version of an image is often a good approximation of it. Furthermore, subtracting a low-pass filtered copy of the image from the image itself produces an image with much lower variance and entropy. This fact has been used extensively in the compression literature [BA83]. In our approach, the low-pass filtered version is just a bilinear reconstruction from a downsampled and compressed version of the image.

Contributions The main contribution of this paper is a new scheme for high-quality texture compression. The major benefits of our approach are (a) high-quality compression at 6:1 with no noticeable loss in image quality; our approach is particularly good at preserving well defined edges in highly-detailed areas, a feature often lacking in competing compressed texture formats; (b) fast GPU-based decoding of individual texels, with trilinear filtering support.

Key elements of our approach include:

- Delta-encoding of the luminance signal Y required to reconstruct an image from a downsampled version of it. This way we exploit coherence across the image pyramid.
- Locally-adaptive encoding of coherent image parts through texel runs following a Hilbert scan. This allows us to exploit residual coherence and sparsity across (larger parts of) the image.
- A block-based encoding of texel runs allowing fast random-access to individual texels. We use a bitmask encoding a variable number of cuts along the Hilbert curve. Decoding a residual requires only three lookups to compressed texture data.

2. Previous work

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, lack efficient fine-grain random access. Entropy encoders, for example, use a few bits to encode the most commonly occurring symbols. Entropy coding does not allow random access as the compressed data preceding any given symbol must be fetched and decompressed to decode the symbol.

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 [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. The two bits of the lookup table are used to select one color out of four possible combinations computed by linear interpolation of the two 16-bit color values. 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. YCoCg-DXT5 [vWC07] extends DXT5 (originally conceived to encode RGBA textures) to provide high-quality compression of RGB data, at the expense of a modest 4:1 compression rate. We adopt this format for the lowest-resolution part of the image pyramid, and provide a novel compression scheme for the highest-resolution mipmap.

Hierarchical compression Hierarchical structures such as wavelets and quadtrees offer spatial adaptivity and support multi-resolution compression, but these approaches often require extensive data traversals and therefore lack efficient random access to individual texels. Only a few hierarchical schemes have been proposed for texture compression. Kraus and Ertl [KE02] propose a two-level hierarchy to represent a simple form of adaptive texture maps. Their representation consists of a coarse, uniform grid where each cell contains the origin and size of a varying-length texture data block. Pereberin [Per99] presents a fixed-rate compression scheme using wavelet decomposition on 4x4 blocks. Luminance values (extracted from YCbCr space) are transformed by applying two levels of a 2D Haar wavelet decomposition, and the eight most significant coefficients (plus the mean value) are quantized and stored. Chrominance values are encoded in a similar way but at a lower resolution. This allows encoding three mipmap levels with 12 bytes, thus providing 4:1 compression. Pereberin's approach has been recently extended to RGBA textures by Sun *et al.* [STC09], which exploit the correlation between luminance and alpha values. Stachera and Rokita [SR06] also adopt a block-based hierarchical scheme. Each 4x4 block is subject to local fractal compression [SS06] and the resulting fractal codes are further compressed using either a Laplacian pyramid or a wavelet decomposition. All the hierarchical methods discussed above are fixed-rate and thus cannot guarantee detail preservation in regions with high luminance variance. Another issue is that these methods cannot address the decompression step without a hardware implementation: the inverse wavelet transform should be computed only once per block, but unless this is implemented in hardware, a fragment shader would have to compute it once per texel. In this situation, assuming 4x4 blocks, bilinear sampling near the block's corner would require four inverse wavelet transforms involving $16\times4 = 64$ coefficients.

A completely different approach is taken by Fenney [Fen03]. As in our approach, he exploits the fact that low-pass filtered signals are often a good approximation to the original signal. Two low-resolution images and a full-resolution but low-precision modulation signal are used to encode the image. Modulation data is arranged in 4x4 texels. Again, efficient decompression of 2x2 neighboring texels requires specific hardware.

Adaptive representations suitable for shader-based decoding include page tables [LKS*06] and random-access quadtrees [LH07]. Our approach differs from quadtree-based schemes in that coherent regions are defined from luminance difference between the original image and a downsampled version of it, and that these regions are allowed to have any size (they do not have the power-of-two restriction) and allows a larger class of shapes (not just squares), thus allowing a better fit 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. 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 whereas leaf nodes encode a variable number of tiles compressed using a color differencing scheme. Our work also uses space-filling curves 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 run encoding can exploit texel correlation along the curve at the finest-grain level. Andujar [And10] encodes images with low color depth by grouping texels with similar colors. Our approach also groups texels but coherence is established in terms of inter-level residuals instead of absolute color values (thus supporting compression of true color images), and run lengths are encoded implicitly through a bitmask instead of explicitly.

3. Locally-adaptive compression

3.1. Overview

The encoding of a mipmapped texture with levels $L_0 \ldots L_n$ involves the following steps, illustrated in Fig. 1:

1. Compress levels L_1 to L_n of the mipmap pyramid. The mipmap pyramid $L_1, \ldots L_n$ is compressed using a native compressed format (to retain efficient minification). Since these levels represent only one fourth of the samples, we can adopt a high-quality but low-compression rate format without significantly increasing the overall compression rate. In our experiments we used the YCoCg-DXT5 format [vWC07] because it provides excellent image quality (generally better than 4:2:0 JPEG at the highest quality setting) at a fixed 4:1 compression rate.

2. Uncompress and upsample level L_1. This results in a low-pass filtered reconstruction $\tilde{L}_0$ of the highest-resolution level L_0. Each upsampled value in $\tilde{L}_0$ is computed by taking one bilinear sample from L_1 right at the center of the corresponding L_0 texel.

3. Extract luminance from L_0 and $\tilde{L}_0$. Since humans are more sensitive to changes in luminance than in chrominance, chrominance values can be substantially downsampled without adding noticeable artifacts. Chroma subsampling is adopted in JPEG and in many compressed texture formats [Per99, SR06]. Our approach also uses chroma subsampling, thus encoding at full resolution only the residuals of the luminance signal. We extract luminance using the YCoCg color space [MS03a] because it provides better decorrelation than competing color spaces [MS03b] and it is the simplest transform in terms of decoder operations. At encoding time, Y is extracted as $Y = \frac{1}{4}(R + 2G + B)$. The reverse conversion, required at decoding time, only requires four additions [MS03a]: $G = Y + Cg$; $m = Y - Cg$; $R = m + Co$; $B = m - Co$.

4. Compute residuals. Luminance residuals y are computed at full resolution by simply subtracting the low-pass

filtered luminance values $\tilde{Y}$ from the luminance values Y of the original image, i.e. $y = Y - \tilde{Y}$. This differencing approach is more redundant than e.g. two levels of a Haar wavelet transform. For a 4x4 block, we end up with 16 residuals instead of just 15 detail coefficients. However, we can decode luminance using just one addition, instead of computing the more expensive inverse wavelet transform.

5. Group texels with similar residuals. This step aims at exploiting sparsity in the residuals: parts of the image with no high-frequency features lead to regions with similar (nearly null) residuals. Grouping is performed block-wise (we used 8×8 blocks), i.e. only texels in the same block can be grouped together. This step ends up with a sequence of runs $\{(y_0, r_0), \ldots (y_m, r_m)\}$, where r_i is the length of the i-th run, and y_i its representative residual. The algorithms we propose for this step are discussed in Sect. 3.2.

6. Encode texel runs. The sequence $\{(y_0, r_0), \ldots (y_m, r_m)\}$ might be acceptable for sequential decompression, but not for random-access, as decoding an individual texel would require a linear search. Our contribution in this respect is a new run encoding resulting in a much more efficient decoding (discussed in Sect. 3.3).

7. Build index data. Since compressed blocks are variable-length, we need to store indices to the beginning of each block. Indices are stored as offsets relative to a predicted index (Sect. 3.4).

3.2. Grouping texels with similar residuals

Once residuals have been computed, texels inside a block are traversed in Hilbert order, and those with identical (loss-less) or similar (lossy) residuals are grouped into runs. This step ends up with a run sequence $\{(y_0, r_0), \ldots (y_m, r_m)\}$, where r_i is the length of the i-th run. Here we describe a simple algorithm whose outer optimization strategy is similar to that of greedy mesh simplification algorithms. The grouping algorithm can be summarized as follows:

1. Traverse texels in Hilbert order and create a unit-length run $(y, 1)$ for each texel.
2. For each pair of neighboring runs (y_i, r_i), (y_{i+1}, r_{i+1}), compute the cost E_i of collapsing that pair (see below).
3. Insert all the pairs in a heap keyed on cost with the minimum-cost pair at the top.
4. Iteratively extract the pair (y_i, r_i), (y_{i+1}, r_{i+1}) of least cost from the heap, group this pair, and update the cost of the adjacent pairs.

During grouping, each run is represented as a triple (S_i, S_i^2, r_i) where S_i (resp. S_i^2) is the sum of the (resp. squared) residuals of the r_i texels in the run. Merging two runs simply involves a component-wise addition of these triples. The cost E_i produced by joining two runs (S_i, S_i^2, r_i) and $(S_{i+1}, S_{i+1}^2, r_{i+1})$ can be computed in the L_2 error sense as $E_i = S_i^2 + S_{i+1}^2 - (S_i + S_{i+1})^2/(r_i + r_{i+1})$. The average run residual is simply $y_i = S_i/r_i$.

3.3. Encoding texel runs

Run encoding is critical for both compression rate and decoding performance. Representing each block with the sequence $\{(y_0, r_0), \ldots (y_m, r_m)\}$ might be acceptable for sequential decompression but not for random-access, as decoding an individual texel would require a linear search. Replacing length values by cumulative lengths $s_i = \sum_{j=0}^{i} r_i$, as proposed in [And10], allows decoding through a binary search on $\{(y_0, s_0), \ldots (y_m, s_m)\}$. Unfortunately, binary search on an 8x8 block still requires up to $\log_2(8 \cdot 8) = 6$ search steps. Our run encoding consists of a B-bit bitmask M and a sequence of detail values $\{y_0, \ldots y_m\}$. The bitmask M is defined as follows: the i-th bit of M is 1 iff $\exists j | s_j = i$, and 0 otherwise. In other words, the ones of M correspond to the cuts of the Hilbert curve, i.e. to points along the curve where a new run starts (Fig. 2). The major benefit of this approach is that extracting the residual of a pixel can be accomplished with very little work. Finding the run containing the k-th pixel is equivalent to counting the number of bits set to 1 in the submask $M[0,k]$. The resulting value is the offset needed to recover the pixel residual in the sequence $\{y_0, \ldots y_m\}$. Since current GPUs are SIMD, we first set to 0 all bits to the right of the k-th bit using M = M & $(1 << (k+1)) - 1$, and then count the number of ones in the resulting mask. Counting ones in a mask can be efficiently accomplished with this simple code,

```
n= T[M&0xFF]; M>>=8; n+=T[M&0xFF]; M>>=8;
n+=T[M&0xFF]; M>>=8; n+=T[M&0xFF];
```

where T[] is a lookup table storing 256 integers, with T[i] storing the number of ones in the binary representation of i.

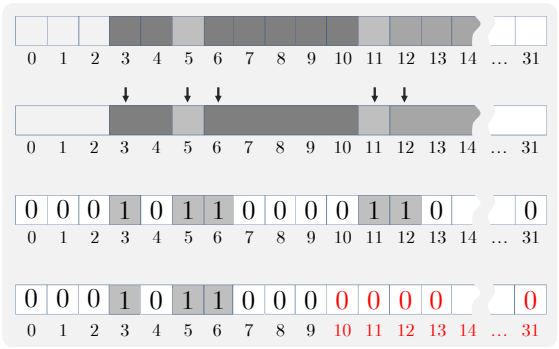

Figure 2: *Run encoding. From top to bottom: (a) sequence of residuals in Hilbert scan order, (b) resulting runs, (c) bitmask encoding the start of each run, (d) mask used for searching the run containing the pixel at position 9. The three bits set to 1 indicate that the residual for pixel 9 has offset three in the sequence of run residuals.*

3.4. Building index data

Our compressed blocks have variable length, so we need to store indices to the beginning of each block (Fig. 3). We encode block indices through a prediction-correction method. Let s_j be the size of the j-th compressed block, and let I_j be its start position in the sequence of compressed blocks, i.e. $I_j = \sum_{i=0}^{j-1} s_i$. We use a very simple linear predictor $\tilde{I}_j = \lfloor s_{avg} \cdot j + 0.5 \rfloor$, where s_{avg} is the average size of a compressed block. We just store the corrections $i_j = I_j - \tilde{I}_j$.

3.5. Compressed representation

Our compressed mipmapped texture thus consists of (a) an image pyramid with $L_1, \ldots L_n$ levels compressed in a native format, and (b) the compressed residuals to recover L_0

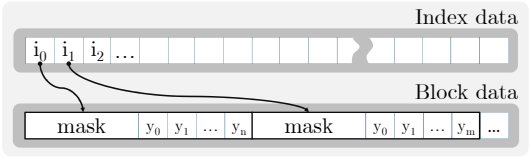

Figure 3: *Representation of the highest-resolution level*

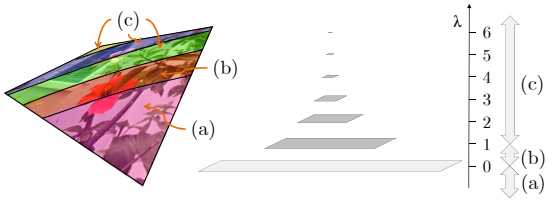

Figure 4: *Three different filtering situations depending on the level-of-detail parameter* λ

(Fig. 1). We now give some details on the particular representation of these two pieces used for the experiments. The image pyramid can be stored as a conventional mipmapped texture, in any native (compressed or uncompressed) format. In our experiments we used YCoCg-DXT5 format, which provides high-quality compression at a fixed 4:1 compression rate. The residuals for recovering L_0 are stored in a separate texture unit which encodes both index data and block data (Fig. 3).

Index data Index data contains one index record per image block. Each index record is encoded with 1+15 bits. The first bit (*single-run flag*) indicates whether the corresponding block has one or more runs. Blocks with a single run correspond to image regions needing identical (often null) residual correction. The single residual value of such blocks can be stored in the remaining 15 bits, although we did not implement this option as for these blocks the residual is very low or zero. Blocks with more than one run have a corresponding block record. In this case, the start of the compressed block (measured from the end of the index data) is encoded in the remaining 15 bits using the prediction-correction strategy explained in Sect. 3.4.

Block data Each block with more than one run has a corresponding block record consisting of a B bit bitmask (B being the block size) and a collection of residuals (Fig. 3). In our implementation we used 8x8 (B=64) blocks, and (signed) residuals were quantized using 8 bits.

3.6. Decompression algorithm

Our compressed representation supports efficient decoding of trilinear-filtered samples through a fragment shader. According to the OpenGL specification, the mipmap levels needed for sampling a texture at a given fragment (x,y) are decided by computing a level-of-detail parameter $\lambda(x,y)$, which basically depends on $\frac{\partial s}{\partial x}$, $\frac{\partial s}{\partial y}$, $\frac{\partial t}{\partial x}$, $\frac{\partial t}{\partial y}$. We can distinguish three situations, depending on the value of λ (refer to Fig. 4). Situation (a) corresponds to $\lambda < 0$, thus a magnification filter is applied. On an uncompressed texture, only L_0 would need to be accessed. In our case, we compute the output color by accessing L_1 (mipmapped texture) and fixing the luminance using the detail image, as discussed below. Situation (b) corresponds to $0 \leq \lambda < 1$, thus minification requires accessing LOD levels 0 and 1. In our case, this simply accounts for an additional linear interpolation between the level 0 and level 1 colors. Situation (c) corresponds to $\lambda \leq 1$, thus minification does not involve the detail image. In this case we transition to native (potentially anisotropic) filtering.

We now detail how to get filtered samples in the situations (a) and (b) described above, which involve accessing the compressed data. We sample the mipmap texture directly at (s,t) using native bilinear filtering, and perform bilinear interpolation of the luminance residuals just after decoding them:

1. Get color $\tilde{c}$ by taking a bilinear sample from L_1 at (s,t).
2. Extract luminance $\tilde{Y}$ from $\tilde{c}$ (Sect. 1).
3. Fetch residuals $y_0, \ldots y_3$ and compute their bilinear interpolation y at (s,t).
4. Recover level 0 luminance values Y by adding the residual, i.e. $Y = \tilde{Y} + y$.
5. Recover level 0 color c by setting the luminance of $\tilde{c}$ to Y.
6. Output c (if $\lambda \leq 0$) or $\lambda\tilde{c} + (1-\lambda)c$ otherwise.

All the steps above are straightforward, except for the residual fetch in step 3, which is detailed below.

Residual fetch Let (i,j) be the integral coordinates of the residual to be fetched. We use the following steps (assuming 8×8 blocks):

1. Fetch the index data of the block containing pixel (i,j).
2. If the single-run flag is set, output residual 0 and finish.
3. Otherwise, retrieve the mask M of the block.
4. Get the local Hilbert key k of (i,j) using a lookup table indexed by $(i\%8, j\%8)$.
5. Identify the run r containing pixel k by counting the number of one's in the mask (Sect. 3.3).
6. Return the residual value y_r at offset r from the mask.

Note that only three texture lookups are needed to decode each residual (Steps 1, 3 and 6 above). Indeed, these texture accesses are very coherent among neighbor 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, this means that index data and the block bitmask M can be cached for about 76% of samples.

4. Results

We have tested our compression scheme with the well known Kodak image suite plus two test images from [RAI06]. All mipmap levels were created through bilinear interpolation, the value of a downsampled pixel being just the average of the corresponding 2x2 group of texels.

Performance Regarding compression, we are able to compress large textures in a few milliseconds. Table 1 shows compression times for varying texture sizes on an Intel Core2 Q9550 at 2.83GHz. Table 2 shows rendering times of our prototype shader running on NVidia GTX 285 hardware

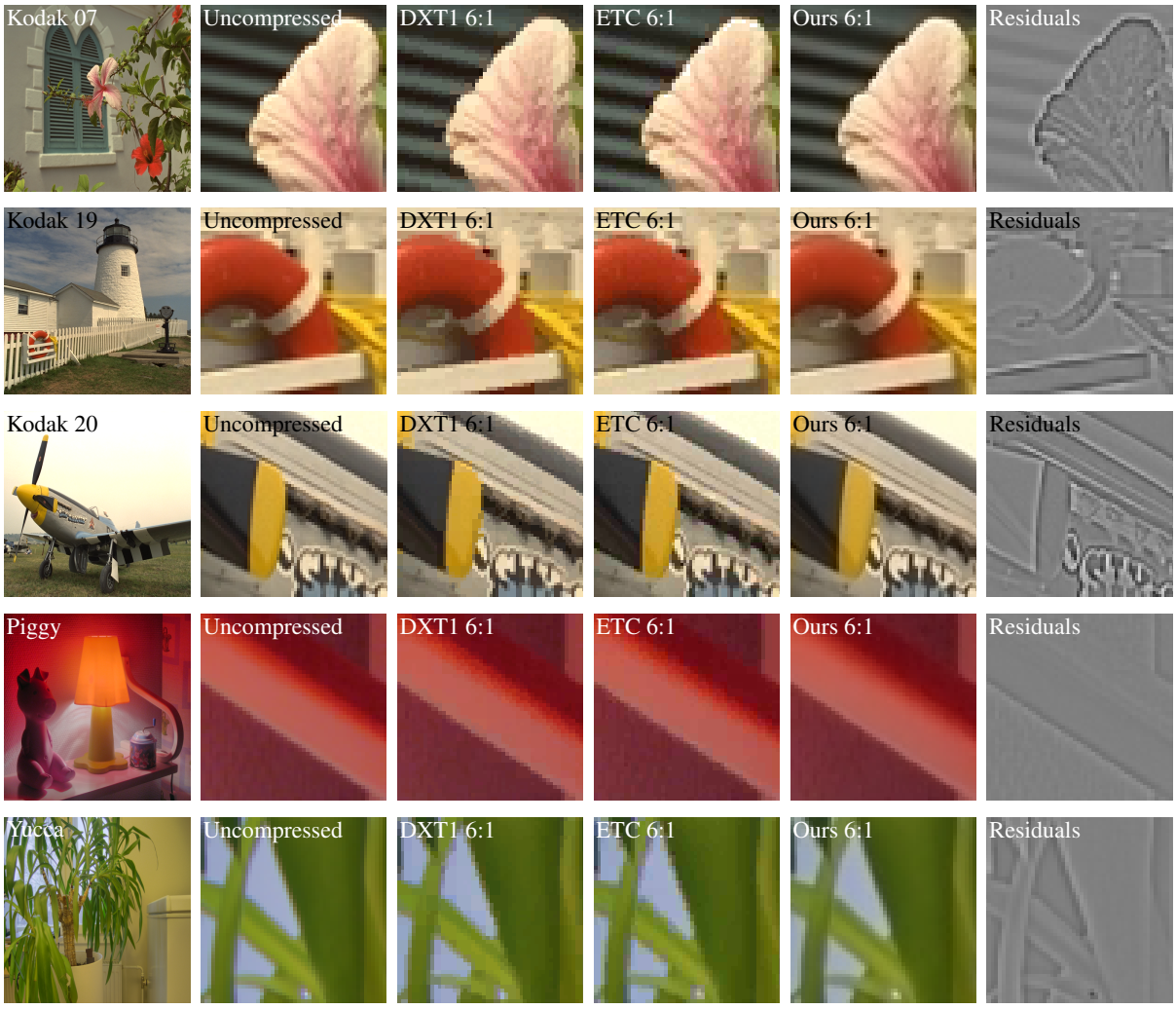

Figure 5: *Results with the 512×512 test images. See text for explanations.*

Texture size	# texels	Compression time (ms)	
		6:1	10:1
512×512	262,144	83	75
1024×1024	1,048,576	251	245
2048×2048	4,194,304	985	971

Table 1: *Compression performance*

Filtering	Texture lookup calls	fps
$\lambda \leq 0$ (bilinear)	13	340 fps
$0 < \lambda \leq 1$ (trilinear)	13	340 fps
$\lambda > 1$ (native trilinear)	1	1210 fps

Table 2: *Decompression performance.*

with a 1024×1024 viewport. Note that the number of texture lookup calls is the same for bilinear ($\lambda \leq 0$) and trilinear ($0<\lambda \leq 1$) interpolation, as in both cases we have to retrieve the $\tilde{L}_0$ color (one call) and the four residuals (3×4 calls), as explained in Sect. 3.6. Although our decompression rates are sometimes 10× slower than natively supported formats such as DXT1 and ETC, they still allow real-time rendering without assistance from specialized hardware.

Image quality and compression ratio We compared our scheme with DXT1 and ETC texture formats using ATI Compressonator 1.50 with the default (highest quality, slowest compression) settings. These formats provide an overall 6:1 compression ratio (CR) with respect to the uncompressed mipmap pyramid. When using our approach, we

	DXT1 (6:1)		ETC (6:1)		Ours @ 6:1	
Image	PSNR	Max	PSNR	Max	PSNR	Max
Kodak 07	39.40	0.11	40.14	0.17	46.71	0.09
Kodak 19	37.91	0.13	39.08	0.16	44.09	0.08
Kodak 20	40.40	0.16	40.84	0.16	48.26	0.14
Piggy	43.75	0.07	44.04	0.13	47.21	0.07
Yucca	38.58	0.16	40.50	0.16	43.45	0.15
Averages	**40.00**	**0.13**	**40.92**	**0.16**	**45.90**	**0.11**

Table 3: *Image quality results (PSNR, maximum error) with the test dataset at an overall 6:1 compression ratio.*

used a 7.2:1 CR for mipmap L_0 and a fixed 4:1 CR for the rest of the image pyramid, which resulted in an overall 6:1 CR (all CR values reported refer to the whole mipmap pyramid). The results are shown in Table 3. Both DXT1 and ETC provided rather poor quality at their fixed 6:1 CR, with an average PSNR of 40.00 dB and 40.92 dB, respectively. Our approach at 6:1 provided much higher PSNR for all the images: 45.90 dB on average, i.e. an average coding gain of 5 dB over DXT1. Regarding maximum RGB errors, our approach also produced lower errors, 0.11 on average against 0.13 (DXT1) and 0.16 (ETC). Images with large coherent regions (Kodak 07 and Kodak 20) compress much better. Note that *coherent* here means *not deviating much from the downsampled version of the image*. On the other extreme, images with well-defined edges everywhere (Yucca) require more residuals to achieve this quality.

Fig. 5 shows close-up views of the resulting images. Our method is particularly good at preserving well defined edges even in highly contrasted areas (compare our 6:1 reconstruction with that of DXT1/ETC for the yucca leaves). Our scheme also succeeds in reproducing color gradients with no blocking artifacts (see window blind in Kodak 07 and the detail of the piggy image). Blocking artifacts in DXT1/ETC are quite noticeable in some rather smooth areas (life saving in Kodak 19) or object's outlines (airplane silhouettes in Kodak 20). All the results above were quite expected, as DXT1 and ETC formats favor decoding speed instead of exploiting coherence across mipmap levels.

We also benchmarked our compression scheme with a quadtree-based compression of the residuals. In both cases we considered loss-less compression of residuals previously quantized to 6 bits, on the Kodak 19 image. The quadtree subdivision resulted in 239 313 nodes (159 356 of them were leaves), whereas our approach produced only 88 805 runs. Even disregarding the space required by the quadtree subdivision, our approach required storing less than one half of residuals.

5. Conclusions

In this paper we have presented a new scheme for high-quality texture compression. Our approach deviates from most previous approaches in that, instead of attempting to quantize or discard detail coefficients, we aim at exploiting sparsity by grouping together inter-level residuals with identical or similar values. When configured to yield an overall 6:1 compression rate, our approach provides much better quality than 6:1 DXT1/ETC formats, being also more compact than YCoCg-DXT5 alone. Our approach is not a competitor nor a replacement for fixed-rate compressed texture formats such as DXT1 or ETC. These formats give up coding efficiency to favor decoding speed, and thus exhibit much faster decompression. We aim at filling the gap between high-performance but low quality formats, and hierarchical schemes with better compression but much slower performance. Unlike other hierarchical approaches, we support GPU-based decoding of trilinearly filtered samples with no assistance from specific hardware. Since we trade off decoding performance for compression efficiency, our approach is mostly useful for applications pursuing the highest image quality whose frame budget can accommodate the extra decoding overhead.

An interesting avenue for further research is to evaluate the perceptual response of the HVS to the particular features of our compression scheme. Our hypothesis is that, at a fixed PSNR, the perceived sharpness of most images is higher with our approach than with competing approaches. We base this hypothesis on the fact that our approach favors those aspects which influence our perception of resolution and acutance (well-defined edges) and tends to remove low-contrast image noise (or grain) which is often detrimental to an image. This might facilitate computing proper CR values for a given image with no user assistance.

Acknowledgments The piggy and yucca images are courtesy of Roimela et al. [RAI06]. This work has been partially funded by the Spanish Ministry of Science and Innovation under grant TIN2010-20590-C02-01.

References

[And10] ANDUJAR C.: Topographic map visualization from adaptively compressed textures. *Computer Graphics Forum 29* (June 2010), 1083–1092. 3, 4

[BA83] BURT P., ADELSON E.: The laplacian pyramid as a compact image code. *IEEE Transactions on Communications 31*, 4 (1983), 532–540. 2

[BAC96] BEERS A. C., AGRAWALA M., CHADDHA N.: Rendering from compressed textures. In *Proceedings of ACM SIGGRAPH '96* (1996), pp. 373–378. 1, 2

[Col87] COLE A. J.: Compaction techniques for raster scan graphics using space-filling curves. *The Computer Journal 30*, 1 (1987), 87–92. 3

[Fen03] FENNEY S.: Texture compression using low-frequency signal modulation. In *HWWS '03: Proceedings of the Graphics Hardware* (Aire-la-Ville, Switzerland, 2003), pp. 84–91. 2, 3

[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. 3

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *HWWS '02: Proc. of the ACM conference on Graphics hardware* (2002), pp. 7–15. 3

[KKZ99] KONSTANTINE I., KRISHNA N., ZHOU H.: Fixed-rate block-based image compression with inferred pixel values, 1999. US Patent no. 5956431. 2

[KNB98] KAMATA S., NISHI N., BANDOH Y.: Color image compression using a hilbert scan. *International Conference on Pattern Recognition 2* (1998), 1575–1578. 3

[LH07] LEFEBVRE S., HOPPE H.: Compressed random-access trees for spatially coherent data. In *Proceedings of the Eurographics Symposium on Rendering* (2007), pp. 339–349. 2, 3

[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. 3

[MB98] MCCABE D., BROTHERS J.: Directx 6 texture map compression. *Game Developer Magazine 5*, 8 (1998), 42–46. 2

[MS03a] MALVAR H., SULLIVAN G.: Transform, scaling & color space impact of professional extensions, 2003. ISO/IEC JTC1/SC29/WG11 & ITU-T SG16 Q.6 Document JVT-H031. 3

[MS03b] MALVAR H., SULLIVAN G.: Tycocg-r: A color space with rgb reversibility and low dynamic range, 2003. ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6 Document JVT-I014. 3

[NH92] NING P., HESSELINK L.: Vector quantization for volume rendering. In *VVS'92: Workshop on Volume visualization* (1992), pp. 69–74. 2

[Per99] PEREBERIN A. V.: Hierarchical approach for texture compression. In *Proceedings of GraphiCon'99* (August 1999). 1, 3

[RAI06] ROIMELA K., AARNIO T., ITÄRANTA J.: High dynamic range texture compression. *ACM Transactions on Graphics (TOG) 25*, 3 (2006), 712. 5, 7

[SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *HWWS '05 : Proc. of the ACM conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 63–70. 2

[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. 2

[SR06] STACHERA J., ROKITA P.: GPU-based hierarchical texture decompression. In *Eurographics'06* (2006). 1, 3

[SS06] STACHERA J., SLAWOMIR N.: Large texture storage using fractal image compression. In *Computer Vision and Graphics*, vol. 32 of *Computational Imaging and Vision*. Springer Netherlands, 2006, pp. 758–767. 3

[STC09] SUN C.-H., TSAO Y.-M., CHIEN S.-Y.: High-quality mipmapping texture compression with alpha maps for graphics processing units. *IEEE Transactions on Multimedia 11*, 4 (2009), 589–599. 1, 3

[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. 2

[vWC07] VAN WAVEREN J., CASTAÑO I.: Real-time YCoCg-DXT compression, 2007. NVidia Developer Zone. http://developer.nvidia.com/object/real-time-ycocg-dxt-compression.html. 1, 3