CEIG'09, San Sebastián, Sept. 9-11 (2009)
C. Andújar and J. LLuch (Editors)

# Locally-Adaptive Texture Compression

C. Andujar[1] and J. Martinez[1]

[1]MOVING research group, Universitat Politècnica de Catalunya

**Abstract**

*Current schemes for texture compression fail to exploit spatial coherence in an adaptive manner due to the strict efficiency constraints imposed by GPU-based, fragment-level decompression. In this paper we present a texture compression framework for quasi-lossless, locally-adaptive compression of graphics data. 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 suitable for fixed-rate and variable-rate encoding. Our scheme can be easily integrated into the rasterization pipeline of current programmable graphics hardware allowing real-time GPU decompression. We show that our scheme clearly outperforms competing approaches such as S3TC DXT1 on a large class of images with some degree of spatial coherence. Unlike other proprietary formats, our scheme is suitable for compression of any graphics data including color maps, shadow maps and relief maps. We have observed compression rates of up to 12:1, with minimal or no loss in visual quality and a small impact on rendering time.*

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 maps with fine detail is still a challenging problem in computer graphics. In hardware systems supporting real–time texture mapping, textures are generally placed in dedicated memory that can be accessed quickly as fragments are generated. Because dedicated texture memory is a limited resource, using less memory for textures may yield caching benefits, especially in cases where the textures do not fit in main memory and cause the system to swap. Texture compression lowers both memory and bandwidth requirements and thus can help to achieve higher graphics quality with a given memory budget, or reduce memory and bandwidth consumption without substantially degrading quality.

The availability of programmable shaders in low-cost graphics hardware provides new, attractive 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 decoding in programmable hardware is considerably faster than software decoding, and also reduces bandwidth requirements as compressed texture data may reside permanently in texture memory.

There are many techniques for image compression, most of which are geared towards compression for storage or transmission. Texture compression systems for GPU on-the-fly decoding exhibit special requirements which distinguish them from general image compression systems. In choosing a texture compression scheme there are several issues to consider:

**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 requir-





DOI: 10.2312/LocalChapterEvents/CEIG/CEIG09/105-114

ing decompressing a large portion of the texture to extract a single texel.

**Cache-friendly.** Texture caches are used in graphics hardware to speed up texture accesses. A texture compression system should work well with existing caches, thus it is important to preserve the locality of references.

A number of fixed-rate, block-based compressed texture formats such as S3TC DXT1, 3Dfx FXT1 and ATI 3Dc are natively supported by low-cost hardware. However, block-based approaches present two major limitations. A first limitation is the lack of flexibility of proprietary formats, which difficults the compression of non-color data such as light fields, parallax occlusion maps and relief maps. A second limitation is that most block-based approaches use a uniform bitrate across the image, and thus lose 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 and areas with high color variance.

Spatial data in computer graphics is often very coherent. There is a large body of work on compressing coherent data, particularly in the context of images. However, most compression schemes such as JPEG2000 involve 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 [LH07].

### Contributions

In this paper we present a locally-adaptive texture compression scheme suitable for both fixed-rate and variable-rate encoding. Novel elements include:

- Use of space-coherent scans such as the Hilbert scan to exploit texel correlation.
- Efficient encoding of homogeneous image regions through arbitrarily-sized texel runs which provide a more flexible approach to segment coherent data from fine detail. Tree-based methods can only segment coherent data in square regions whose size and boundaries are given by the tree subdivision. Our approach is able to detect correlated areas in several orders of magnitude more regions than quadtree-based approaches.
- A cumulative run-length encoding of coherent texel runs supporting fast random-access. Cumulative RLE allows for texel recovery using binary search. We present an $O(1)$ decoding algorithm requiring less than 3.5 texture lookups on average.
- A compression algorithm suitable for fixed-rate and variable-rate encoding. The compression algorithm is driven by an error-sorted priority-queue which decides the order in which texel runs are collapsed.

Our scheme can be easily integrated into current programmable graphics hardware allowing real-time GPU decompression.

## 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 has 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. 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. Thus the most common approach for texture compression is fixed-rate encoding of small image blocks, as fixed-rate compression eases address computations and facilitates random access.

### Vector quantization and block-based methods

Vector quantization (VQ) has been largely adopted for texture compression [NH92, BAC96]. When using VQ, texture is regarded as a set of texel blocks. VQ attempts to characterize this set of blocks by a smaller set of representative blocks called a codebook. A lossy compressed version of the original image is represented as a set of indices into this codebook, with one index per block of texels [BAC96]. Indexed color can be seen as vector quantization using 1x1 blocks. Color quantization algorithms such as the Median Cut Algorithm use this same indexed-lookup technique for representing 24 bit images with 8 bpp. The most critical part of VQ encoding is the construction of the codebook. The Generalized Lloyd Algorithm yields a locally optimal codebook for a given set of pixel blocks. VQ can also be applied hierarchically. Vaisey and Gersho [VG88] adaptively subdivide image blocks and use different VQ codebooks for different-sized blocks.

Block-based data compression has been a very active area of research [MB98, SAM05]. S3TC formats [MB98] are a de facto standard in today's programmable graphics hardware. S3TC DXT1 [BA] stores a 4x4 texel block using 64 bits, consisting of two 16-bit RGB 5:6:5 color values and a 4x4 two-bit lookup table. The two bits of the look-up table are used to select one color out of four possible combinations computed by linear interpolation of the two 16-bit color values. A variant called VTC has been proposed by Nvidia to extend S3TC to allow textures of width and height other than multiples of 4. Unfortunately, the accuracy of the DXT1 texture compression is often insufficient and the scheme is hardly customizable. The low number of colors (4) used to encode each 16-texel block, and the RGB

| | |
|---|---|
| $f$ | Scan order to map 2D data into 1D data |
| $L = (c_0, \dots c_{wh-1})$ | Linear version of the input image |
| $R = ((c_0, r_0) \dots (c_{n-1}, r_{n-1}))$ | RLE version of $L$. $r_i$ is run length |
| $S = ((c_0, s_0) \dots (c_{n-1}, s_{n-1})$ | Cumulative RLE of $R$. $s_k$ is accumulated length |
| $B$,$b$ | Block size and no. of blocks resp. |
| $I = ((o_0, l_0), \dots (o_{b-1}, l_{b-1}))$. | Index data. $o, l$ are origin and length of blocks in $S$ |

**Tabla 1:** *Notation used in the paper*

5:6:5 color quantization result in visible artifacts all over the texture. ETC [SAM05] also stores a 4x4 texel block using 64 bits, but luminance is allowed to vary per texel while the chrominance can be changed for multiple texels at a time only, reflecting the fact that humans are more sensitive to changes in luminance than in chrominance. ETC is good at areas of uniform chromacity where only luminance varies, but it cannot represent smooth chrominance changes or hard edges between colors of the same luminance. ETC2 [SP07] extends ETC by introducing three new modes.

All fixed-rate schemes discussed above are non-adaptive as a uniform bit-rate is used across the image, and thus they over-allocate space in low-detail regions while loosing quality in detailed parts.

### Adaptive compression

Hierarchical structures such as wavelets and quadtrees offer spatial adaptivity. 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. This allows representing different blocks of the image at varying resolutions, using only one level of indirection. The main challenge though is to provide continuous interpolation across data block's boundaries and to avoid visible artifacts at these boundaries. Other adaptive representations include page tables [LKS*06] and random-access quadtrees [LH07] for compressing spatially coherent graphics data using an adaptive multiresolucion hierarchy encoded as a randomly-accessible tree structure.

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.

## 3. Locally-adaptive compression

### 3.1. Overview

We first focus on 2D textures encoding RGB color data. The input of our algorithm is an image $Im$ containing $w \times h$ color tuples $(r,g,b)$. 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 $Im$. Function $f$ can be defined in a variety of ways, including traversals based on space-filling curves, line-by-line scans, column-by-column 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.

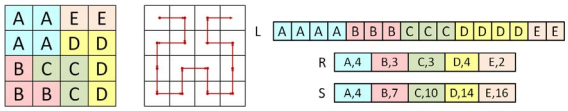



**Figura 1:** *Sample image and corresponding sequences L, R, S obtained with the Hilbert scan shown on the right.*

Function $f$ maps two-dimensional texel data into a one-dimensional sequence $L = (c_0, c_1 \dots c_{wh-1})$ 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 (LRE) of $L$. The grouping algorithm might collapse only identical texels for loss-less encoding, or approximately-similar texels for lossy encoding. A simple grouping algorithm is described in Section 3.4. The result after grouping and LRE encoding is a collection of pairs $(c,r)$ where $c$ is an $(r,g,b)$ tuple and $r$ is the run length. The run sequence will be referred to as $R = ((c_0, r_0) \dots (c_{n-1}, r_{n-1}))$. The reconstruction of the original sequence from $R$ is given simply by $\tilde{R} = (\underbrace{c_0, c_0 \dots c_0}_{r_0} \dots \underbrace{c_{n-1}, c_{n-1} \dots c_{n-1}}_{r_{n-1}})$.

This representation is not suitable as a compressed texture format, as the evaluation of $\widetilde{R}(k)$ 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 $S = ((c_0, s_0) \dots (c_{n-1}, s_{n-1}))$ 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 $\widetilde{S}(k)$ 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 $(s_0 \dots s_{n-1})$, see Figure 1.

This new represention 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 $512 \times 512$ 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 be still a limiting speed factor for GPU decoding. For instance, the 8:1 encoding of a $512 \times 512$ image results in a search space of $n = 512 \cdot 512/8 = 32,768$ runs. Since $\log_2(32,768) = 15$, the decompressor will have to perform in the worst case 15 random accesses to $S$ in order to evaluate $\widetilde{S}(k)$.

Our 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 $S$ into another $b$ blocks (see Figure 2). This constraint can be easily enforced during compression by simply compressing each block independently. 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.

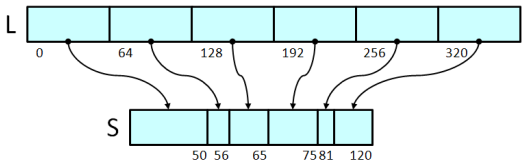



**Figura 2:** *Partition of L into uniform blocks defining a non-uniform partition on its compressed version S.*

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 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,* $I = ((o_0, l_0), \dots (o_{b-1}, l_{b-1}))$. Storing $I$ allows for performing binary search locally on each block in $O(\log_2(B))$ time. For $B = 64$, *at most* 6 texture accesses are required to decode a given texel. Indeed, we have found that the *average number* of texture accesses varies from 2.57 to 3.51 (including fetching index data), depending on the compression ratio and the run-length distribution (discussed in Section 4.2).

### 3.2. Compressed representation

Our compressed representation consists of an encoding of the cumulative run-length encoding $S$ plus the index data $I$. We have control over several parameters that can be used to tradeoff compression rate for quality and decoding speed:

**Scan order.** 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 (line-by-line or column-by-column), block-based 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 and Z-order clearly outperform raster scans. On the one hand, space-filling curves provide much better locality preservation. On the other hand, raster scans produce runs which correspond to thin regions of the image (line segments), whereas space-filling curves lead to thicker runs. Thick runs are often more efficient at encoding coherent regions. Thick runs also minimize the ratio between run's boundary and run's area, which is often desirable as aggressive compression might produce artificial edges along run boundaries. Finally, space-filling curves yield a more evenly distribution of the error, whereas raster scans tend to produce coherent error. Since humans are more sensitive to coherent error, space-filling curves provide perceptually better compression. 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 $S$. Meaningful values for the block size parameter $B$ vary from 2 to $\frac{w \cdot h}{2}$. The value of $B$ have 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. 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.

**Color encoding.** Programmable hardware allows for a number of options to encode the color associated to each run. Encoding can be performed in any color space (RGB

and YUV color spaces are two reasonable options), with or without color quantization (e.g. DXT1 uses RGB 5:6:5 quantization, ETC uses RGB 5:5:5 and OpenGL supports RGB 3:3:2) and with optional indexed color (palettized texture). In our experiments we tested two color encodings: RGB 8:8:8 for loss-less compression and RGB 6:6:6 for lossy compression. This last choice allows encoding color data and cumulative run-lengths in 24 bit textures.

**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 compression ratio, 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. A more concise option is to just encode the sum $e_i = o_i + l_i - 1$, (i.e. the end of the $i$-th block) which requires $\left\lceil \log_2 \frac{w \cdot h}{c} \right\rceil$ bits. This is possible because we know that $o_0 = 0$, $o_i = e_{i-1} + 1$, and obviously $l_i = e_i - o_i + 1$. Using 24 bits for each $e_i$ value is enough for compressing up to 8192×8192 textures, assuming a minimum compression ratio of 4:1.

**Data layout.** We represent the sequence $C = (I,S)$ as a 1D virtual array stored as a 2D texture.

### 3.3. Decompression algorithm

The decompression algorithm takes as input the compressed texture representation $C(I,S)$ and integral texel coordinates $(i,j)$, and outputs the color of texel $(i,j)$. The decompression algorithm proceeds 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 $k$ using integer division, i.e. $b \leftarrow k/B$.
3. Compute the lower bound for the binary search with $o \leftarrow I_{b-1} + 1$ if $b > 0$, $o \leftarrow 0$ otherwise.
4. Compute the upper bound for the binary search, $e \leftarrow I_b$.
5. Compute the color of the run containing texel $k \bmod B$ by binary search on the ordered sequence $s_o \ldots s_e$.

### 3.4. A simple compression algorithm

We now present a simple algorithm to convert an input image into a compressed representation $C = (I,S)$. The algorithm proceeds through the following steps:

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 starting at a location multiple of $B$ are not allowed to be grouped with preceding runs, to ensure that 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. Quantize color data using RGB 6:6:6 quantization (optional)
5. 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.
6. Output $C = (I,S)$ as a 1D virtual array.

Actual compression is performed in step 2. Our outer optimization strategy of the grouping algorithm is similar to that of greedy iterative simplification algorithms. The grouping algorithm (step 2) can be quickly 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.

The essential aspects of the grouping algorithm are:

- How to compute the error produced by joining two runs $(c_i, r_i)$, $(c_{i+1}, r_{i+1})$.
- How to compute the color after grouping two runs.

Experimentally we have found the following simple choices to yield very good results. The color $c'$ resulting from joining two runs $(c_i, r_i)$ and $(c_{i+1}, r_{i+1})$ is computed as the weighted average $c' = tc_i + (1-t)c_{i+1}$ with $t = r_i/(r_i + r_{i+1})$. Conversely, the cost produced by joining two runs is defined as $E = r_i \cdot ||c_i - c'|| + r_{i+1} \cdot ||c_{i+1} - c'||$, where $c'$ is computed as above, and $||\cdot||$ denotes color difference. Our prototype implementation uses color difference in RGB space to facilitate the comparison with competing compressed formats (see Section 4.1), although other color spaces such as CIE Luv might be more appropriate for measuring perceptual color distance.

## 4. Results and discussion

We have implemented the compression and decompression algorithms described above and tested them on a large class of RGB textures. Unless otherwise stated, all compressed textures were created using the Hilbert scan, RGB 6:6:6 quantization and block size $B = 64$.

### 4.1. Image quality and compression ratio

Figure 3 shows several textures compressed using our scheme with rates varying from 6:1 up to 24:1. RMS and

**Figura 3:** *Compressed textures with rates varying from 6:1 to 24:1. From left to right: input image, reconstruction from our compressed representation, close-up views, and image difference amplified* $10\times$.

PSNR values are shown in Table 2. Note the high visual quality of resulting images, which preserve highly-detailed features. The last column shows the image difference in RGB space amplified $10\times$, computed as $10 \cdot (|r-r'|+|g-g'|+|b-b'|)$ and coded in a color temperature scale. The error appears to be distributed evenly across the image, with the only exception of Maggie, which exhibits highly spatial coherence and hence the errors are restricted to detailed parts. Resulting PSNR values were all above 31.5 dB (Table 2).

| Image | Size | Compression | RMS | PSNR(dB) |
|---|---|---|---|---|
| Quadrics | $1024^2$ | 6:1 | 0.0164 | 35.71 |
| Buck Bunny | $512^2$ | 6:1 | 0.0256 | 31.83 |
| Watch | $1024^2$ | 12:1 | 0.0231 | 32.71 |
| Grape | $512^2$ | 12:1 | 0.0262 | 31.64 |
| Maggie | $512^2$ | 24:1 | 0.0229 | 32.81 |

**Tabla 2:** *Compression results on several test images*

Figure 4 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 (the white background of the input image shown in Figure 4 is not completely uniform, exhibiting a slight gradient and soft shadows). Relative frequencies of run-lengths for two sample images are shown on the left of Figure 5. In all cases the average run length matches the compression ratio, but the distribution of relative frequencies varies. Short-

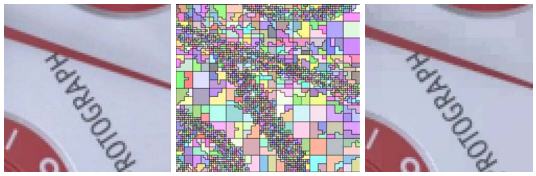
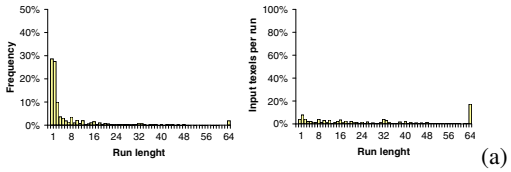


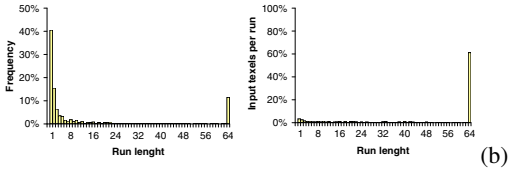
**Figura 4:** *Close-up views showing the ability of our scheme to perform adaptive compression. Input image (left), color-coded runs (middle), and reconstruction from the compressed texture (right).*

length runs clearly predominate (around 30-40% 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 2% to 30%, depending on the compression ratio, which in turn reflects the degree of spatial coherence of the image. Figure 5-right shows the number of input texels represented by each run, grouped by run length. Since longer runs represent a higher number of texels, around 20%-60% of the input texels, depending on the compression factor, 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 reducing significantly average decoding times.




(a)

Quadrics (6:1)

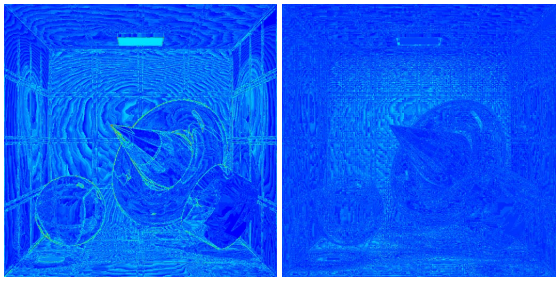



(b)

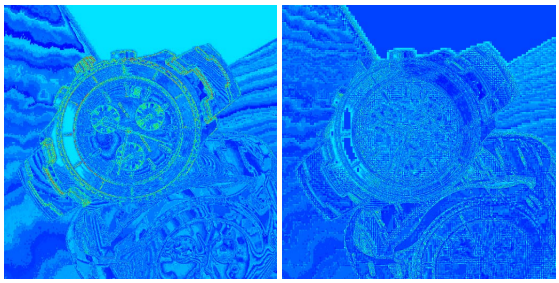
Watch (12:1)

**Figura 5:** *Relative frequencies of run-lengths (left) and percentage of input texels per run (right).*

We have extensively compared our scheme with readily-available compressed texture formats. Figure 8 compares our approach with three alternative approaches: 4:1 image subsampling (using bilinear filtering), 256-indexed color (computed using the Median-Cut algorithm) and DXT1 compression (using a high-quality software coder). Images appear roughly sorted by increasing degree of spatial coherence. Image subsampling (third column) has been included only as a reference, as it blurs the image and causes detail loss everywhere. Indexed color (fourth column) is good at preserving detail on images with a limited number of colors, but nothing else. In particular, color gradients are poorly supported by indexed color schemes (see e.g. the molecule image in last row of Figure 8). Here we focus on the comparison of DXT1 with our scheme configured to yield the same compression ratio (last two columns of Figure 8). Resulting RMS, PSNR and maximum error values are reported in Table 3. DXT1 produces higher RMS errors and poorer PSNR values in all the test images. Visually, DXT1 produced a quality loss everywhere, which is particularly noticeable around sharp edges and regions with a high chrominance and/or luminance variance (recall that DXT1 decodes sixteen input pixels using only four colors). See for example the artifacts around sharp edges in the DXT1 molecule image and those around the text in the topographic map. In some regions the $4\times4$ blocks used by DXT1 are clearly distinguishable. Maximum error values also reveal the poor behavior of DXT1 in images with blocks with high variance (Table 3). Our approach produced more visually pleasant images with better PSNR values and much lower maximum error. We would like to remark that the test images in Figure 8 contain a number of distinct colors much higher than most people would guess. For instance, the original molecule image contains 525,023 distinct colors (see Table 3). It is also important to notice that the input topographic map was stored in JPEG format and already contained some color artifacts, which in some extent did not disappear after compression.



(a) DXT1 (b) Ours



(c) DXT1 (d) Ours

**Figura 6:** *Image difference amplified* $10\times$ *for DXT1 and our approach for the quadrics (a-b) and watch (c-d) images.*

| Image | Size | Colors | S3TC DXT1 (6:1) | | | Our approach (6:1) | | |
|---|---|---|---|---|---|---|---|---|
| | | | RMS | PSNR | Max | RMS | PSNR | Max |
| Castle | $512^2$ | 102,869 | 0.046 | 26.5 | 0.281 | 0.036 | 28.7 | 0.254 |
| Still life | $512^2$ | 93,338 | 0.042 | 27.3 | **0.737** | 0.027 | 31.3 | 0.174 |
| Clock | $512^2$ | 107,352 | 0.047 | 26.4 | **0.562** | 0.034 | 29.6 | 0.285 |
| Quadrics | $1024^2$ | 192,164 | 0.024 | 32.2 | **0.445** | 0.016 | 35.7 | 0.083 |
| Topography | $2048^2$ | 238 | 0.047 | 26.3 | 0.317 | 0.027 | 31.3 | 0.147 |
| Molecule | $2048^2$ | 525,023 | 0.055 | 25.1 | **0.881** | 0.017 | 35.2 | 0.073 |

**Tabla 3:** *RMS, PSNR and maximum error for DXT1 and our approach with 6:1 compression.*

We also analyzed our approach in terms of the error distribution. Figure 6 compares the error distribution of DXT1 compression with ours. Note that DXT1 errors are accumulated in high-variance regions around hard edges, whereas our approach produces a more evenly distribution of the error, which at the end is the ultimate goal of adaptive compression. It can be observed, though, that our approach slightly tends to produce locally coherent error due to large coherent parts represented with a single color. However, resulting artifacts are hard to notice at moderate compression rates (see Figure 7).

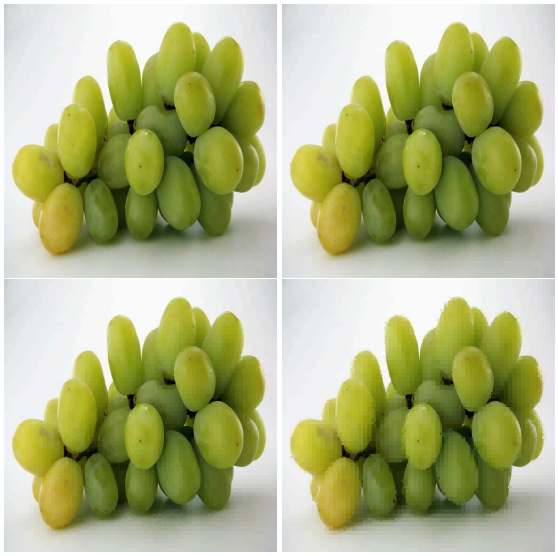


**Figura 7:** *Compressed textures with 6:1, 12:1, 24:1 and 48:1 compression rates.*

### 4.2. Decompression speed

Our scheme can be easily integrated into the rasterization pipeline of current programmable graphics hardware allowing GPU decompression. Table 4 shows rendering times of our prototype shader running on NVidia GTX 280 hardware with OpenGL 2.1. Higher compression rates result in less runs per block and thus less texture fetches during the binary search step. Note that for typical 6:1 compression, only about 3.51 texture fetches are required, on average. This number of texture lookups is quite reasonable compared with the number of fetches required by popular techniques such as parallax occlusion mapping and relief mapping. Our decompression rates are about one order of magnitude slower than DXT1. But of course, DXT1 benefits from specialized hardware in the GPU, and texture caching and filtering strategies have been optimized for their use. Even without assistance from specialized hardware, our scheme allows real-time rendering at high rates.

| Ratio | Lookups(avg) | Mtexels/s | Fps | |
|---|---|---|---|---|
| | | | $1024^2$ | $512^2$ |
| 6:1 | 3.51 | 392 | 374 | 1320 |
| 12:1 | 3.05 | 438 | 418 | 1518 |
| 18:1 | 2.76 | 507 | 484 | 1716 |
| 24:1 | 2.57 | 553 | 528 | 1870 |

**Tabla 4:** *Decompression performance*

### 4.3. Compression speed

Texture compression is somewhat asymmetric since decoding speed is essential while encoding speed is useful but not necessary. Nevertheless, our compression algorithm is able to compress large textures in a few seconds. Table 5 shows compression times for varying texture sizes and compression goals. Higher compression rates slightly increase compression times as more run pairs need to be collapsed. All times were measured on an Intel Core2 Quad Q9550 at 2.83GHz. Compression times are dominated by heap updates. Therefore compression times can be further reduced by using a lazy queuing algorithm for ordering grouping operations. Moreover, since each of the $B$ runs of each block are grouped independently, the compressor is amenable to GPU implementation, although we did not try this option.

| Texture size | # texels | Compression time (s) | | | |
|---|---|---|---|---|---|
| | | 6:1 | 12:1 | 18:1 | 24:1 |
| 256x256 | 65,536 | 0.4 | 0.4 | 0.4 | 0.4 |
| 512x512 | 262,144 | 0.6 | 0.6 | 0.6 | 0.7 |
| 1024x1024 | 1,048,576 | 3.5 | 3.8 | 4.0 | 4.1 |
| 2048x2048 | 4,194,304 | 16.2 | 16.6 | 16.8 | 17 |

**Tabla 5:** *Compression performance*

### 4.4. Filtering and caching

One concern with our approach is that it may increase memory bandwidth since each query involves several texture lookups. Although we did not analyze this topic in detail, access patterns created by a Hilbert scan are very coherent, so most memory reads should be intercepted by memory caches. The same applies for the binary search, as the initial search bounds are the same for all texels belonging to a given block.

## 5. Conclusions and future work

In this paper we have presented a locally-adaptive texture compression scheme suitable for GPU decoding. Our approach achieves a good balance between compression of coherent regions and preservation of high-frequency details. Novel elements include the use of locality-exploiting scans, efficient encoding of homogeneous regions through arbitrarily-sized runs, and cumulative run-length encoding allowing fast random-access. Texel decoding relies on a binary search requiring less than 3.5 texture lookups on average. Several rendering techniques such as relief mapping and parallax occlusion mapping significantly exceed this number of texture fetches while retaining real-time speed. We have also presented a simple compression algorithm driven by a heap keyed on a data different metric. The algorithm is suitable for both loss-less and lossy with bounded error. Our scheme outperforms the visual quality of competing approaches such as DXT1 on a large class of images. Best results are obtained with images exhibiting some degree of spatial coherence. Unlike other proprietary formats, our scheme is suitable for compression of any graphics data and can be easily extended to support multidimensional textures. Decoding speed is slower than hardware-supported compressed formats, but still provides excellent frame rates on current consumer hardware.

There are several directions in which this work may be extended. The compression algorithm can be extended to find the optimal block size; images with large coherent regions would benefit from higher block sizes, whereas images with high-frequency detail everywhere will be more efficiently encoded with smaller block sizes. Our current implementation uses 6 bits to encode run lengths which vary from 1 to 64. It might be useful to use a non-uniform quantization of the run lengths, allowing e.g. run lengths from 1 to 59, and 64, 128, 256, 512, 1024. An in-depth analysis of cache efficiency is required. In particular, it would be useful to optimize trilinear interpolation taking into account the locality-preserving features of the decoding algorithm. Finally, we plan to analyze the application of our scheme for compressing volume data; volume data often exhibits high voxel correlation and can potentially benefit from our adaptive encoding.

### Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Technology under grant TIN2007-67982-C02-01.

### 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.

[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.

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

[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).

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

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

[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.

[SP07] STRÖM J., PETTERSSON M.: ETC2: texture compression using invalid combinations. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 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.

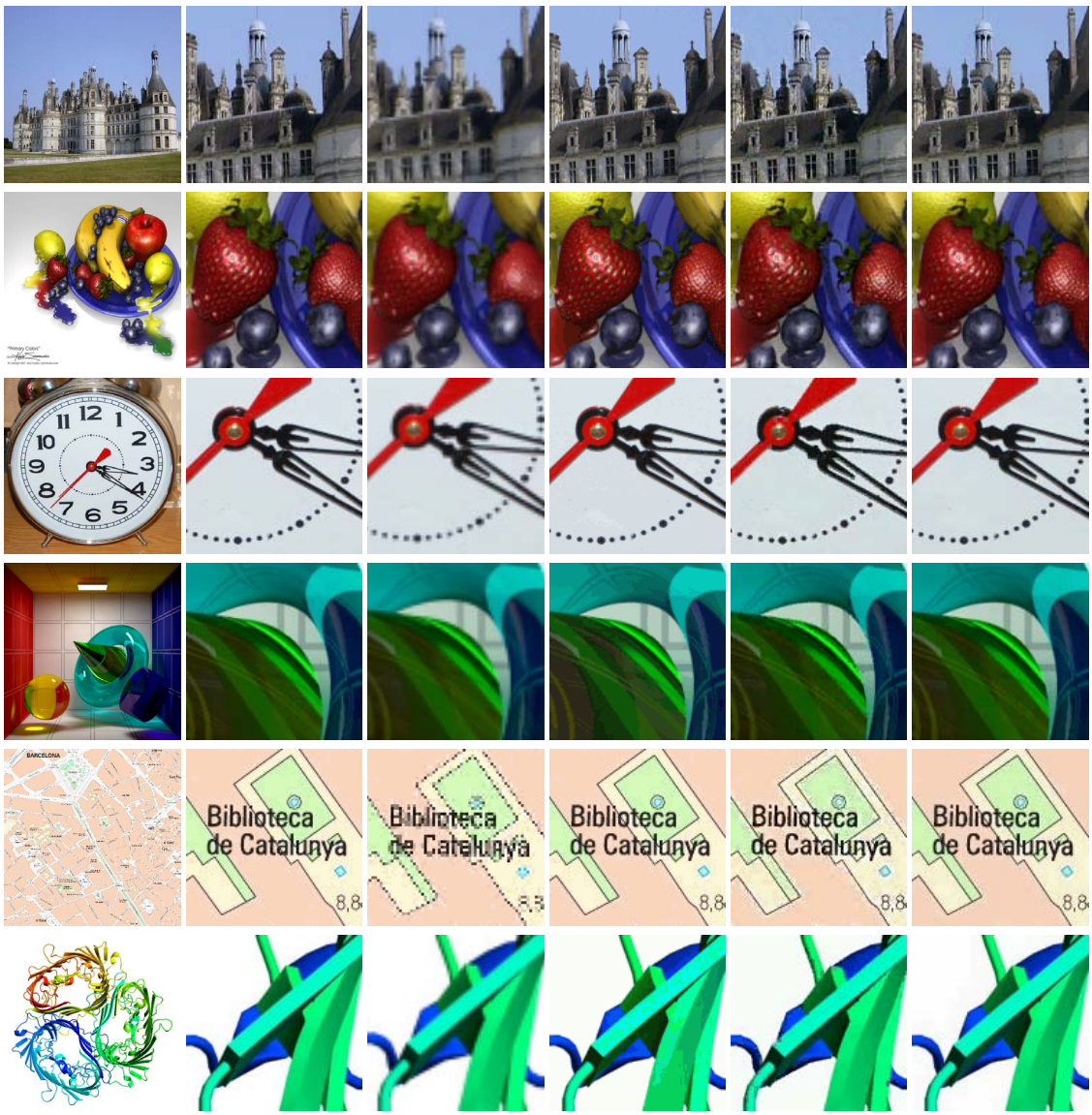



**Figura 8:** *Comparison of our approach with several compressed alternatives.*