Rendering from Compressed High Dynamic Range Textures on Programmable Graphics Hardware

Lvdi Wang Xi Wang Peter-Pike Sloan* Li-Yi Wei Xin Tong Baining Guo

Microsoft Research Asia *Microsoft Corporation

Abstract

High dynamic range (HDR) images are increasingly employed in games and interactive applications for accurate rendering and illumination. One disadvantage of HDR images is their large data size; unfortunately, even though solutions have been proposed for future hardware, commodity graphics hardware today does not provide any native compression for HDR textures.

In this paper, we perform extensive study of possible methods for supporting compressed HDR textures on commodity graphics hardware. A desirable solution must be implementable on DX9 generation hardware, as well as meet the following requirements. First, the data size should be small and the reconstruction quality must be good. Second, the decompression must be efficient; in particular, bilinear/trilinear/anisotropic texture filtering ought to be performed via native texture hardware instead of custom pixel shader filtering.

We present a solution that optimally meets these requirements. Our basic idea is to convert a HDR texture to a custom LUVW space followed by an encoding into a pair of 8-bit DXT textures. Since DXT format is supported on modern commodity graphics hardware, our approach has wide applicability. Our compression ratio is 3:1 for FP16 inputs, allowing applications to store 3 times the number of HDR texels in the same memory footprint. Our decompressor is efficient and can be implemented as a short pixel program. We leverage existing texturing hardware for fast decompression and native texture filtering, allowing HDR textures to be utilized just like traditional 8-bit DXT textures. Our reduced data size has a further advantage: it is even faster than rendering from uncompressed HDR textures due to our reduced texture memory access. Given the quality and efficiency, we believe our approach suitable for games and interactive applications.

Keywords: high dynamic range image, texture compression, games & GPUs, game programming, graphics hardware, texturing techniques

1 Introduction

HDR images are gaining popularity in games and interactive applications, as both sources of illumination and intermediate results in multi-pass rendering. However, due to their range, HDR images need to be stored as 16-bit or 32-bit floating point textures, taking 2 to 4 times the amount of storage with respect to traditional 8-bit textures. This not only causes storage issues on commodity graphics hardware often equipped with limited texture memory, but also incurs significant performance penalties caused by poorer texture cache coherence and more memory bandwidth consumption.

I3D 2007, Seattle, Washington, April 30 – May 02, 2007.
© 2007 ACM 978-1-59593-628-8/07/0004 $5.00

Unfortunately, current graphics hardware only supports compression of 8-bit textures, and to our knowledge there exists no suitable compression technique for floating-point HDR textures on current generation graphics hardware. This makes the size disparity even bigger between compressed 8-bit and uncompressed HDR textures. Recently, various extensions have been proposed for supporting compressed HDR textures on graphics hardware [Munkberg et al. 2006; Roimela et al. 2006]. However, these solutions are not immediately available, as the proposed extensions are not part of current (DX9) or even future (DX10) graphics APIs or hardware.

The goal of our paper is to provide a solution that can be implemented on current generation GPUs as well game consoles. For games and other interactive applications, the solution must satisfy the following requirements. First, the data size should be small and the reconstruction quality must be good. Second, the decompression must be efficient; in particular, bilinear/trilinear/anisotropic texture filtering ought to be performed via native texture hardware instead of custom pixel shader filtering.

One possibility is to encode a HDR image into a shared exponent format, RGBE [Ward 1994], via RGBA8 format on current graphics hardware. However, since it is incorrect to linearly interpolate exponent values in the A channel, we will have to perform texture filtering via custom shader code. A similar format employed in the game developer community is RGBS [Green and McTaggart 2006] where the A channel is interpreted as a scaling (hence the acronym S) factor of the RGB channels. Similar to RGBE, this format does not support direct filtering even though it is used as a render target only.

In this paper, we perform extensive study and comparison of potential methods for supporting compressed HDR textures on programmable graphics hardware. Our solution represents an input HDR image with a pair of DXT5 textures. Since DXT is widely supported on commodity graphics hardware, our decompressor can be implemented as a pixel program without any hardware modification. For HDR RGB textures in 16-bit floating point formats, we obtain a 3:1 compression ratio, meaning that applications utilizing our technique can store 3 times the amount of HDR textures with the same amount of memory. This is achieved without major perceptual quality degradation, as rendering results using our technique visually match those rendered from original sources. We present a novel encoding scheme to allow native DXT texture hardware filtering without the need to simulate trilinear or anisotropic filtering by fetching multiple samples in a fragment program. Even more importantly, our technique runs faster than rendering from uncompressed HDR textures due to reduced texture memory traffic.

Since both desktop and mobile GPUs as well as modern consoles (e.g. PS3 and XBOX360) all support DXT but not HDR compression, we believe our technique provides significant benefit for many graphics applications.

2 Previous Work

HDR Tone Mapping Significant research has been conducted in converting a HDR image into a LDR one so that it can be displayed on LDR devices [Li et al. 2005; Goodnight et al. 2003; Reinhard

et al. 2005; Ward and Simmons 2004]. However, these techniques are mainly designed for dynamic range reduction (i.e. tone mapping), a non-reversible process, not for data size reduction while attempting to preserve original dynamic range.

HDR Encoding and Compression OpenEXR [Industrial Light & Magic 2003] is a popular format for encoding HDR images in the film industry. However, it focuses on quality but not efficient decoding, which is crucial for real-time applications. In addition to tone mapping, [Li et al. 2005] can also be utilized as a compression technique; however, its complexity makes GPU implementation infeasible. Utilizing compression standards, [Mantiuk et al. 2004] compresses HDR videos via MPEG4 while [Xu et al. 2005] compresses HDR images via JPEG 2000; due to the variable-rate nature of MPEG and JPEG, these techniques cannot be easily implemented on graphics hardware.

GPU Texture Compression Unlike traditional image compression whose main concerns are quality and data size, techniques for GPU texture compression also needs to take into account speed, random-accessibility, and implementation feasibility [Beers et al. 1996]. To achieve these goals, a variety of techniques have been proposed [Strom and Akenine-Moller 2005; Fenney 2003; Beers et al. 1996; Iourcha et al. 1997]; however, to our knowledge they are all designed for 8-bit LDR images, and therefore cannot be directly applied to compress HDR textures.

DXT Compression Since our algorithm builds upon the DXT (aka S3TC) standard [Iourcha et al. 1997], we provide a brief summary here for completeness. There exist several variations of DXT, but in our implementation we only utilize DXT1 (the one offering highest compression ratio) and DXT5 (twice the data size of DXT1 but with higher quality in alpha channel), so below we only describe these two.

DXT1 operates on individual 4×4 pixel blocks independently. Each 4×4 block is represented by two 16-bit $R_5G_6B_5$ colors, and 2 bits per pixel to decide how to interpolate from these two colors. (In other words, all colors in the block will lie on a line in the color cube.) If the block is totally opaque, then the 2 bits provide 4 possible linear interpolations of the two extreme colors. If the block has some transparent (punch through) pixels, then the encoding is slightly different. But in our algorithm, we only utilize the opaque mode. The total DXT1 data size is 64 bits per 4×4 pixel block.

DXT5 has an identical color encoding as the opaque mode in DXT1, plus 64 more bits for encoding the alpha channel. In a similar linear interpolation mindset, the alpha channels per 4×4 block are represented by two 8-bit representative values, and 3 bits per pixel to decide how to interpolate individual alpha channels. The total DXT5 data size is 128 bits per 4×4 pixel block.

3 HDR Encoding and Decoding

Here, we describe how we utilize currently available texture formats and programmable pixel shaders to support compressed HDR textures. In this section, we concentrate on what we have done, not why. The whole process of our algorithm is illustrated in Figure 1. Our design decisions may seem arbitrary in many places; for discussions and justifications on our detailed design process, please refer to Section 4.

Our encoding is conducted offline on a CPU, and is performed only once. Given a HDR input, we first convert it from RGB into our custom LUVW color space where all the high dynamic range luminance information is concentrated in the L channel only. We then quantize the LUVW channels so that they can be encoded into fixed-point DXT channels. Since L is the only channel that contains HDR information, we allocate more bits for L than UVW. Finally, we properly map the LUVW channels into the DXT channels to allow native hardware filtering and efficient decoding. Our decoding process is simple and can be implemented on a GPU via pixel program.

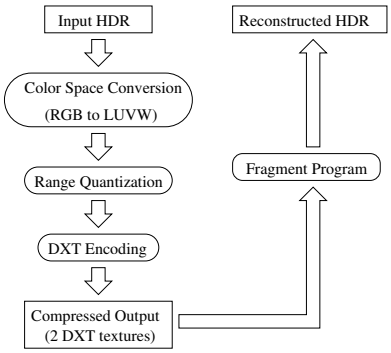


Figure 1: Overview of our algorithm pipeline. Down arrows indicate encoding, while up arrows indicate decoding.

We describe the detailed process below.

3.1 Color Space Conversion

The goal of this step is to concentrate all the HDR information into one color channel instead of three. Given a HDR image in floating point format, we first convert it into our LUVW color space. The formula is:

$$\begin{aligned} L &= \sqrt{R^2 + G^2 + B^2} \\ U &= R/L \\ V &= G/L \\ W &= B/L \end{aligned} \tag{1}$$

Note that division-by-L concentrates all HDR information into the L channel. This allows us to spend more bits for L than UVW in the subsequent process. However, since division-by-L is a non-linear operation, it is theoretically incorrect to perform native filtering for UVW via texture hardware. Fortunately, we have not found this to be an issue in practice since most natural HDR images have smooth-varying L values. See Section 4.1 for detailed discussion.

3.2 Range Quantization

After the color conversion, we proceed to convert the floating-point LUVW values into a form suitable for DXT compression. Since DXT supports only fixed point values, we need to find a good method to represent the original high dynamic range floating-point as fixed-point ones.

We quantize the channels as follows. First, note that UVW has low-dynamic range, i.e. their values lie in roughly the same exponent range. As a result, we only need to preserve their mantissa parts, essentially reducing the conversion process into a uniform quantization. Second, for the L channel, since it still has high dynamic range, it is usually not enough to use only one uniform-quantization interval. As detailed in Section 4.3, we have concluded that two uniform quantization intervals strike the best quality/storage balance.

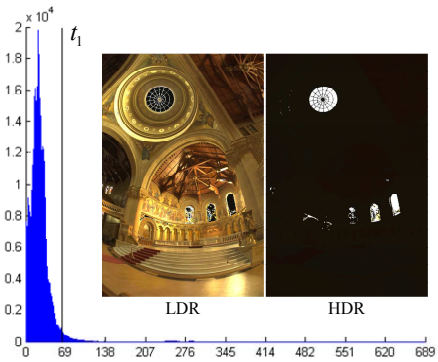


Figure 2: Example histogram of the luminance channel for the memorial scene. The HDR/LDR images indicate L in the range $[t_1, t_{max}]$/$[t_{min}, t_1]$.

Given that we have two uniform quantization intervals, we now describe how we actually quantize L. We divide the range of luminance into two zones, $[t_{min}, t_1]$ and $[t_1, t_{max}]$, and quantize them separately, as shown in Figure 2. We choose t_1 so that the total quantization error of the two zones is minimized. Specifically, t_1 is computed by minimizing the following error function:

$$E(t_1) = \frac{n_l\,(t_1) \times (t_1 - t_{min})}{2^{b_l}} + \frac{n_h\,(t_1) \times (t_{max} - t_1)}{2^{b_r}} \quad (2)$$

where n_l and n_h are the number of luminance pixels falling into the $[t_{min}, t_1]$ and $[t_1, t_{max}]$ regions, and b_l and b_r are number of bits used to quantize $[t_{min}, t_1]$ and $[t_1, t_{max}]$. (Note that both n_l and n_h are functions of t_1.) This formula is only an approximation for the true quantization error, but it is much easier to optimize and we have found it sufficient for our data set. In our implementation, we use a simple linear search of t_1 for optimization.

Notice that we have not described exactly how many bits we use quantizing LUVW; these will be described below for detailed DXT encoding.

3.3 DXT Encoding

We now describe how to map the quantized LUVW channels into the channels of two DXT textures. Our encoding is via DXT5 which offers a 3:1 compression ratio for FP16 RGB inputs. Other alternatives (such as two DXT1 or DXT1+ DXT5) are discussed in Section 4.

Below is our encoding; tex0 and tex1 refers to the two DXT5 textures.

$$\begin{aligned}
\text{tex0.a} &= \begin{cases} \text{L} & \text{L} \in [t_1, t_{max}] \\ 0.0 & \text{L} \in [t_{min}, t_1] \end{cases} \\
\text{tex1.a} &= \begin{cases} 1.0 & \text{L} \in [t_1, t_{max}] \\ \text{L} & \text{L} \in [t_{min}, t_1] \end{cases} \\
\text{tex0.r} &= \text{U} \\
\text{tex0.g} &= \text{V} \\
\text{tex0.b} &= \text{W}
\end{aligned} \quad (3)$$

Here are some explanations for this encoding. First, since DXT5 has 5/6/5 bits for rgb channels and 8 bit for alpha channel, we put L into alpha channel for higher accuracy. Second, if pixels in both $[t_{min}, t_1]$ and $[t_1, t_{max}]$ are mixed in the same 4×4 block, which could happen in transition region from low to high intensity zones, then direct filtering L across the same DXT5 channel would produce incorrect results. The conditional assignments for tex0.a and tex1.a avoids this problem by putting L in different zones into different textures.

Claim 3.1 *Our encoding shown in Equation 3 allows us to reconstruct L via the following formula, enabling native hardware filtering with no conditional shader code.*

$$L = \textit{tex0.a} \times (t_{max} - t_1) + \textit{tex1.a} \times (t_1 - t_{min}) + t_{min} \quad (4)$$

Proof See Appendix A. ∎

Luminance Residual Note that our encoding in Equation 3 has unused channels in tex1.rgb. We utilize these channels to further improve reconstruction quality via luminance residual S, defined as the difference between the original luminance L and the reconstructed value L' via Equation 4:

$$\text{S} = \text{L} - \text{L'} \quad (5)$$

The basic idea is to encode the residuals into tex1.rgb and utilize them during reconstruction. Note that similar to L, S is also a HDR quantity. Furthermore, it is a signed value. Similar to L encoding where we use two quanitization ranges, for S we use three ranges since we have three available channels in tex1.rgb. We compute the three zones $[S_{min}, S_1]$, $[S_1, S_2]$, and $[S_2, S_{max}]$ via an optimization procedural similar to Equation 2. From this, we encode S as follows: (analogous to dividing L into two zones and encode via Equation 3)

$$\begin{aligned}
\text{tex1.r} &= \begin{cases} \text{S} & \text{S} \in [S_{min}, S_1] \\ 1.0 & \text{otherwise} \end{cases} \\
\text{tex1.g} &= \begin{cases} 0.0 & \text{S} \in [S_{min}, S_1] \\ \text{S} & \text{S} \in [S_1, S_2] \\ 1.0 & \text{S} \in [S_2, S_{max}] \end{cases} \\
\text{tex1.b} &= \begin{cases} \text{S} & \text{S} \in [S_2, S_{max}] \\ 0.0 & \text{otherwise} \end{cases}
\end{aligned} \quad (6)$$

The encoded residual values allow us to obtain higher quality reconstruction than Equation 4 via the following formula:

$$\begin{aligned}
\text{L} &= \text{tex0.a} \times (t_{max} - t_1) + \text{tex1.a} \times (t_1 - t_{min}) + t_{min} \\
&+ \text{tex1.r} \times (S_1 - S_{min}) + \text{tex1.g} \times (S_2 - S_1) \\
&+ \text{tex1.b} \times (S_{max} - S_2) + S_{min}
\end{aligned} \quad (7)$$

(Note the first line is the same as Equation 4.) Via a similar proof to Claim 3.1, we could claim that this formula allows native filtering without conditional code as well. Residual provides superior reconstruction quality as demonstrated in Figure 3.

3.4 Decoding in Pixel Shader

Rendering from our compressed HDR images is as follows. For each texel request, we first fetch the corresponding values from the two DXT textures; note that the DXT decompression and filtering is performed by the hardware automatically. Because we read

Figure 3: Reconstruction quality comparison with and without residual. Left: without residual via Equation 4. Right: with residual via Equation 7. Notice the severe banding artifacts on the left image.

from two textures for each input request, we implement our decoding as a simple pixel shader subroutine. Specifically, we compute L via Equation 7, and UVW from tex0.rgb (see Equation 3). The LUVW image is then converted into the final RGB image by inverting Equation 1:

$$\begin{aligned} R &= U \times L \\ G &= V \times L \\ B &= W \times L \end{aligned} \tag{8}$$

Note that Equation 8 can be performed very efficiently via a float3 multiplication in our pixel shader, as shown in Table 1.

```
sampler2D tex0; sampler2D tex1;
const float g_LMinPlusRMin;
const float2 g_LRangeLowHigh;
const float3 g_RRangeLowMidHigh;

float4 Decode_BPP16_LUVW( float2 texCoord : TEXCOORD0 ) : COLOR
{
    float4 t0 = tex2D( tex0, texCoord );
    float4 t1 = tex2D( tex1, texCoord );
    float L = t0.a * g_LRangeLowHigh.y + t1.a * g_LRangeLowHigh.x
            + g_LMinPlusRMin + dot(t1.rgb, g_RRangeLowMidHigh);
    return float4(t0.rgb * L, 1.0f);
}
```

Table 1: Our pixel shader in HLSL for decompression. Blue indicates keywords while magenta indicates constants. This shader takes 6 virtual machine cycles to execute.

3.5 Codec

General DXT encoders like NvDXT or ATI-Compressonator are not suitable for us since our channels contain information other than RGB as in traditional images. In particular, numerical optimizations in existing codecs cannot guarantee the best quality when dealing with L and luminance residuals S. To remedy this issue, we provide a customized codec as detailed below.

L and S have unique attributes compared to general RGB values. First, both of them are in high dynamic range, incurring different perceptual errors than ordinary LDR color channels. Second, even though S is split into three channels in our encoding (tex1.rgb as in Equation 6) it is inherently a 1D quantity rather than 3D. We customize our codec to address these two specific issues.

High dynamic range Psychophysical literature has long demonstrated that human-eyes are more sensitive to relative difference in the log domain for high dynamic range quantities. General DXT encoders do not capture this property as they usually use the linear difference between the original L/S and the encoded values L'/S'. To take into account this logarithmic effect, we instead use the following error metrics E_L and E_S for L and S in our codec:

$$E_L = \frac{|\text{L - L'}|}{\text{L} + c} \qquad E_S = \frac{|\text{S - S'}|}{\text{L} + c} \tag{9}$$

where c is a positive constant in case L is zero.

1D quantity split in three channels Since the luminance residual is split into tex1.rgb, general DXT encoders will naturally treat them as 3D values, leading to misjudgment because they are actually 1D values. For example, a general encoder may think (0.5, 0, 0) and (0, 0, 0.5) are far from each other in 3D. However, after converting back to 1D values via Equation 7, the two values can be quite close to each other if (S_1-S_{min}) $\approx$ (S_{max}-S_2).) To avoid this problem, we perform iterative optimization for finding proper palette colors in the original 1D space instead of 3D as in other codecs, and only split up the 1D value into 565 RGB channels in the final step.

4 Design Process

Supporting compressed HDR textures on currently available graphics hardware is essentially a constrained design and optimization problem, and can be stated as follows. Given an input HDR image, represent it as a combination of textures and pixel shaders on current generation (DX9) hardware. The representation must have small data size, and must satisfy the following three requirements: (1) high reconstruction quality (measured on filtered results in addition to original pixel samples), (2) fast decoding via shader program, and (3) native filtering in texture hardware.

In the following, we describe the reasoning and experimental process that lead to our final design. We believe it is important to expose this process rather than simply presenting our final decision in order to justify that it is not a random choice, from both a theoretical and experimental point of view. Readers who would just like to implement our algorithm without going through this laborious discussion could skip this section.

4.1 Color Space Conversion

Color space conversion is a common choice for compression algorithms. In particular, both [Munkberg et al. 2006; Roimela et al. 2006] concentrate HDR information into one luminance channel for easy processing.

The choices we have are (×/√ indicates bad/good options):

× no conversion
× standard YCbCr (and other similar color spaces)
× LUV as in [Roimela et al. 2006]

$$\begin{aligned} L &= R + 2G + B \\ U &= R/L \\ V &= 2G/L \end{aligned} \tag{10}$$

√ our LUVW as in Equation 1

Since it is much harder to compress three HDR channels than one, we eliminate the no conversion option. Standard YCbCr (and other color spaces) require a matrix multiplication which is too computationally expensive for our decompressor shader code, so we eliminate that as well.

Among the remaining two options, LUVW may seem unusual at first glance, but it has the following advantages. First, the inverse conversion can be performed via a simple float3 multiplication on GPU as via Equation 8, so it is more efficient than LUV. Second, LUV would leave tex1.b channel unused which is utilized by the W channel in LUVW. Consequently, LUVW provides better reconstruction quality than LUV both visually and statistically (Table 2).

Division-by-L issue Due to division-by-L as shown in Equation 1, it is theoretically incorrect to perform native filtering for the UVW channels via texture hardware. However, despite this theoretical error, our scheme still produces much better quality than simply encoding UVW in the original HDR range without division-by-L due to limited quantization bits. Furthermore, we have found out that this non-linear filtering error unnoticeable for the data sets we have tried, which include real-world light probes and shipping game assets from Half-Life 2. This could be mainly attributed to the fact that L is smooth for natural images; see Appendix B for a detailed analysis.

4.2 Texture Format

For HDR encoding, we have to choose an available texture format. A variety of possibilities exist, such as putting L in one FP16 texture followed by downsampling UVW into an 8-bit texture (20 bpp). The decision primarily depends on properties of the input we use. We have observed that, similar to their 8-bit LDR counterparts, HDR images also possess a local linear property that fits the underlying assumption of DXT compression. In addition, DXT provides the highest data reduction among currently available formats. As a consequence, we choose DXT as our base representation.

Note that there are multiple variations of DXT (DXT1-5); below we enumerate possible options and discuss how we choose among them.

× two DXT1 (8 bpp)
× one DXT5 (8 bpp)
× one DXT1 + one DXT5 (12 bpp)
× four DXT1 (16 bpp)
√ two DXT5 (16 bpp)

The above list is constructed as follows. Since a FP16 RGB texture has 48bpp, we would like to use at most two DXT5 textures (16 bpp) to provide enough data reduction. Since we do not use pre-multiplied alpha, using DXT2 is equivalent to DXT3 (as well DXT4/5). Between DXT3 and DXT5, we have found that the indexed alpha mode (8-bit palette and 3-bit index) in DXT5 provides higher quality than the direct alpha mode in DXT3 (4-bit direct).

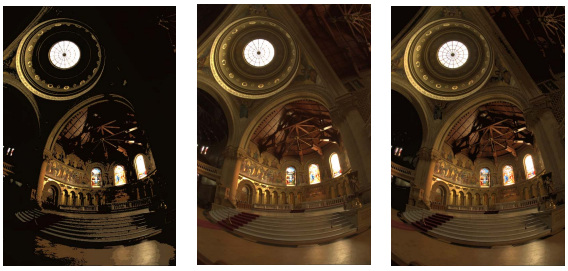
Also, as discussed in Section 4.3, we need two 8-bit alpha channels. This leaves us only the option of two DXT5.

4.3 Luminance Range Quantization

Essentially, we are using multiple uniform-quantization ranges to represent a HDR floating point value. The options we have are:

× 1 range
√ 2 ranges
× 3+ ranges

Since most HDR images have a luminance histogram similar to the one in Figure 2, one uniform-quantization range is certainly not enough, as it would under-sample small values which constitute the bulk of the histogram. So 1-range option is out.



1×12-bit Original 2×6-bit

Figure 4: Uniform quantization effects. Notice that even with the same number of bits, 2×6-bit quantization is preserves the original dynamic range better than 1×12-bit quantization.

Using three or more ranges would be infeasible for DXT channel allocation; since two DXT5 can provide at most two alpha channels, we will have to encode one range in one of the RGB channels. This will cause two problems. First, each of the individual RGB channels has fewer bits (5/6/5) than an alpha channel (8 bit), resulting in more quantization error. Even though this could be overcome by combining multiple RGB channels, the quality is still hampered by a second issue where DXT only provides 2 interpolation bits for RGB versus 3 bits for alpha. In our experiments, we have found that using a single 8-bit alpha actually provides higher quality than combining multiple RGB channels.

So the only option left is two quantization ranges. Since many natural HDR images have a histogram similar to Figure 2, two ranges work really well as the LDR hump and HDR flat region are covered in different zones.

Figure 4 illustrates the effect of number of quantization ranges. Notice that 2-range produces much better result than 1-range, even though both use the same total number of bits.

4.4 Residual Computation

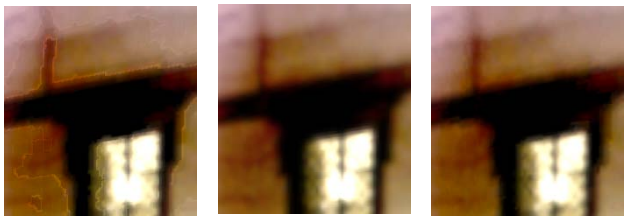
In addition to residual S defined in Equation 5, we have also experimented with an alternative measurement termed residual ratio T, which is a normalized version of S:

$$T = S/L' \tag{11}$$

√ residual
× residual ratio

In theory, residual-ratio T is in LDR and therefore could be coded more efficiently than residual S (in a reason similar to why we choose UVW over RGB). In our experiments we have confirmed this by observing that T indeed outperforms S at integer pixel locations (although only slightly). However, at non-integer pixel locations whose values are computed by linear interpolation (i.e. bilinear or trilinear texture filtering) we have found that T introduces excessive artifacts due to its non-linear nature. Specifically, it may appear self-contradictory that this artifact is only problematic for T but tolerable for UVW, which are computed from RGB via dividing-by-L (Equation 1). However, S and RGB possess very different statistical properties. As detailed in Appendix B, RGB color channels in general go in the same direction as L while the residual S, being a difference between L and L', often does not satisfy this condition. (For the data set used in this paper, the satisfaction rates are 97.5% for RGB pixel pairs but only 76.1% for S.)

For these reasons, we select residual over residual ratio as our final choice for implementation. A comparison is shown in Figure 5.



residual ratio original residual

Figure 5: Comparison of filtering effects for residual and residual-ratio. Notice the severe filtering artifact for residual ratio.

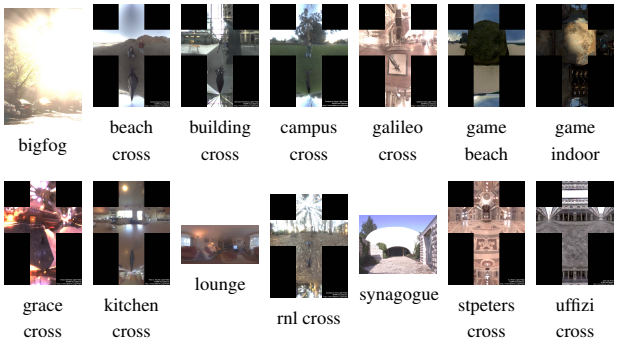


Figure 6: HDR images used in our experiments.

5 Results and Discussion

We have applied our technique to a variety of HDR inputs; several representative cases are shown in Figure 8, with the corresponding statistics in Table 2. In addition to ground truth, we have also compared our technique with two alternative techniques commonly employed by game developers:

FP16×4 Each RGBA channel is encoded in 16-bit IEEE floating point. The advantage of this format is that hardware filtering is supported, plus it can be used not only for input texture but also for render target. The disadvantage is that it consumes more storage and bandwidth than our encoding.

RGBE This has the same storage size as RGBA8, but the alpha channel is used as a shared exponent for all the RGB channels. The advantage of this format is that it is more compact than FP16×4 (but still 2 times the size of our encoding). In addition, it requires custom filtering on current GPU since the shared exponent cannot be linearly interpolated. (A variation of this RGBE format will be supported on DX10 [Blythe 2006].)

For real-time GPU applications, we are mostly concerned with three criteria: size (bits-per-pixel), quality (PSNR), and speed (frame-per-second).

The data size can be easily measured by bits-per-pixel, and we measure quality via PSNR via the metric in [Li et al. 2005]; however, unlike [Li et al. 2005] where the formula is only applied on the L channel, we measured errors across all color channels so both luminance and chrominance errors are considered. We measure performance speed via a teapot scene in Figure 7 which renders a single HDR source twice: once for environment map and another once for the teapot reflection map. We choose this simple teapot scene so that the performance bottleneck would come from shader/texture instead of geometry/rasterization.

	LUV	LUVW	RGBE	FP16×4
Bits per pixel	16	16	32	64
Speed (fps)	476	482	297	366
Scene	PSNR	PSNR	PSNR	PSNR
beach cross	64	69	68	inf
bigfog	64	66	69	95
building cross	54	58	68	123
campus cross	71	74	70	127
galileo cross	69	70	74	132
game beach	54	57	67	96
game indoor	51	54	63	93
grace cross	74	75	64	133
kitchen cross	67	68	74	127
lounge	57	61	66	inf
memorial	64	65	77	inf
rnl cross	68	70	72	129
stpeters cross	74	76	77	133
synagogue	44	48	57	84
uffizi cross	56	63	70	108
average PSNR	62	65	69	115
average PSNR / bits per pixel	3.9	4.1	2.2	1.8

Table 2: Comparison of various algorithms. The top two rows list the data size (in bits-per-pixel) and rendering speed (in trilinear filtering mode for the scene in Figure 7); note that for RGBE8 the trilinear filtering is simulated in shader. The bottom rows demonstrate PSNR (peak-signal-to-noise ratio) across various inputs. The PSNR = inf cases indicate perfect reconstruction (which are ignored during average PSNR computation). All source image thumb nails can be found in Figure 6.

FP16 Our LUVW

Figure 7: Rendering from HDR environment map. All renderings are performed with trilinear filtering on an NVIDIA Geforce 6800 card.

As shown in Table 2, our technique consumes the minimum storage and runs fastest; indicating that the GPU is memory bound under this circumstance. Note that even though RGBE has smaller data size than FP16, it is actually slower due to the need of shader simulation for trilinear filtering.

Figure 7 serves an additional purpose: since users usually see only the final rendering, not the source texture, it is more important to assess the perceptual quality on rendered images [Beers et al. 1996]; we utilize the smooth but curved teapot surface to achieve this goal. As shown, our result is visually indistinguishable from the result generated by FP16. In addition, our result achieves the best compression ratio, measured as quality/data-size (last row in Table 2).

6 Conclusions and Future Work

Despite proposals for future hardware design [Munkberg et al. 2006; Roimela et al. 2006], current generation GPUs do not provide adequate support for compressed HDR textures. We provide a solution to address this issue. Our algorithm can be implemented on

DX9 graphics hardware, has high reconstruction quality, and runs efficiently due to a simple and fast pixel shader and native hardware texture filtering. Our compression ratio is 3:1; even though lower than that offered by novel hardware [Munkberg et al. 2006; Roimela et al. 2006], this still offers a significant boost for games and interactive applications which are often texture intensive. Finally, supporting HDR on DX9 GPUs can be considered a constrained design and optimization problem; we expose our decision process to demonstrate how we have come up with our final design.

For future work, we plan to extend our algorithm to alternative texture formats available in DX10 and XBOX360. In particular, DX10 and XBOX360 both support textures that are two DXT5 alpha blocks (8bpp) and DX10 has a format that is a single channel DXT5 alpha block (4bpp). (Such as the ATI2N and ATI1N formats already shipped by ATI.) For the former format, we could store L via two DXT5 alpha blocks and then down-sample UVW into quarter resolution and store them via a single LDR DXT1 texture (9bpp in total). This single channel format could be used to store tex1.a as in Equation 3; this encoding does not have a residual, but saves 4bpp.

Acknowledgement We would like to thank Meng Qiang and Xianyou Hou for help on early experiments and the anonymous reviewers for their comments.

A Proof for Claim 3.1

Proof Let the L channel have an original value of t, then due to uniform quantization we have:

$$\text{tex0.a} = \frac{t - t_{min}}{t_1 - t_{min}}, \quad \text{tex1.a} = 0, \quad \text{if } t \in [t_{min}, t_1] \quad (12)$$

$$\text{tex1.a} = \frac{t - t_1}{t_{max} - t_1}, \quad \text{tex0.a} = 1, \quad \text{if } t \in [t_1, t_{max}] \quad (13)$$

From Equation 12 and 13, we can derive the reconstruction formula as follows:

$$t = \text{tex0.a} \times (t_1 - t_{min}) + t_{min}, \quad \text{if } t \in [t_{min}, t_1] \quad (14)$$

$$t = \text{tex1.a} \times (t_{max} - t_1) + t_1, \quad \text{if } t \in [t_1, t_{max}] \quad (15)$$

We now show that our encoding in Equation 3 would allow us perfect reconstruction via this equation, without any conditional code. If t lies in $[t_{min}, t_1]$, we have tex1.a = 0, so $t = \text{tex0.a} \times (t_1 - t_{min}) + t_{min}$, which is the same as the Equation 14. Otherwise, t lies in $[t_1, t_{max}]$, and we have tex0.a = 1, so $t = \text{tex1.a} \times (t_{max} - t_1) + t_1$, which is the same as Equation 15. Thus our encoding satisfies requirement 4 for no conditional code.

Furthermore, due to the commutability of linear operations, it is easy to show that (1) performing reconstruction via Equation 4 followed by trilinear interpolation is equivalent to (2) performing trilinear interpolation of tex0.a/tex1.a values followed by reconstruction via Equation 4. Since in (2) the trilinear interpolation of tex0.a/tex1.a can be accomplished in texturing hardware, our encoding satisfies the native filtering requirement. ∎

B Interpolation error for division-by-L

Let two adjacent pixels have scalar HDR values P_1 and P_2, and their LDR values (after dividing by L) be $Q_1 = \frac{P_1}{L_1}$ and $Q_2 = \frac{P_2}{L_2}$. For example, in RGB to LUVW color conversion (Equation 1) we have P = RGB and Q = UVW, and in residual to redisual-ratio conversion (Equation 11) we have $P = S$ and $Q = T$.

For any pixel P in between P_1 and P_2 we can compute its value via linear interpolation

$$P = \alpha P_1 + (1 - \alpha) P_2 \quad (16)$$

However, since in our hardware decoding scheme we interpolate Q and L instead of P, we have

$$\begin{aligned} Q' &= \alpha Q_1 + (1 - \alpha) Q_2 \\ L' &= \alpha L_1 + (1 - \alpha) L_2 \\ P' &= Q' L' \end{aligned} \quad (17)$$

Taking the squared differences of P and P', we have

$$\left(P - P'\right)^2 = \left(\alpha(1-\alpha)\left[P_1(\frac{L_2}{L_1} - 1) + P_2(\frac{L_1}{L_2} - 1)\right]\right)^2 \quad (18)$$

It can be easily proven that this interpolation error is small when (1) $L_1 \approx L_2$ or (2) P and L are "going in the same direction" (i.e. both increasing/decreasing). In our experiments, we have found that both conditions are true for RGB color space so division-by-L does not introduce noticeable filtering artifacts (Equation 1). However, for residual S, since it is the difference between L and L', it in general does not satisfy condition 2. As a result, we have found that residual ratio T (Equation 11) has more severe interpolation artifacts than UVW.

References

BEERS, A. C., AGRAWALA, M., AND CHADDHA, N. 1996. Rendering from compressed textures. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 373–378.

BLYTHE, D. 2006. The Direct3D 10 system. *ACM Trans. Graph. 25*, 3, 724–734.

FENNEY, S. 2003. Texture compression using low-frequency signal modulation. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 84–91.

GOODNIGHT, N., WANG, R., WOOLLEY, C., AND HUMPHREYS, G. 2003. Interactive time-dependent tone mapping using programmable graphics hardware. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, 26–37.

GREEN, C., AND MCTAGGART, G., 2006. High performance HDR rendering on DX9-class hardware, Mar. Poster presented at the ACM Symposium on Interactive 3D Graphics and Games.

INDUSTRIAL LIGHT & MAGIC, 2003. OpenEXR. `http://www.openexr.com/`.

IOURCHA, K., NAYAK, K., AND HONG, Z., 1997. System and method for fixed-rate block-based image compression with inferred pixel values". US Patent 5,956,431. See also `http://en.wikipedia.org/wiki/S3TC/`.

LI, Y., SHARAN, L., AND ADELSON, E. H. 2005. Compressing and companding high dynamic range images with subband architectures. *ACM Trans. Graph. 24*, 3, 836–844.

MANTIUK, R., KRAWCZYK, G., MYSZKOWSKI, K., AND SEIDEL, H.-P. 2004. Perception-motivated high dynamic range video encoding. *ACM Trans. Graph. 23*, 3, 733–741.

MUNKBERG, J., CLARBERG, P., HASSELGREN, J., AND AKENINE-MOLLER, T. 2006. High dynamic range texture compression for graphics hardware. *ACM Trans. Graph. 25*, 3, 698–706.

REINHARD, E., WARD, G., PATTANAIK, S., AND DEBEVEC, P. 2005. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann Publishers.

ROIMELA, K., AARNIO, T., AND ITARANTA, J. 2006. High dynamic range texture compression. *ACM Trans. Graph. 25*, 3, 707–712.

STROM, J., AND AKENINE-MOLLER, T. 2005. iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 63–70.

WARD, G., AND SIMMONS, M. 2004. Subband encoding of high dynamic range imagery. In *APGV '04: Proceedings of the 1st Symposium on Applied perception in graphics and visualization*, 83–90.

WARD, G. J. 1994. The radiance lighting simulation and rendering system. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 459–472.

XU, R., PATTANAIK, S. N., AND HUGHES, C. E. 2005. High-dynamic-range still-image encoding in JPEG 2000. *IEEE Comput. Graph. Appl. 25*, 6, 57–64.

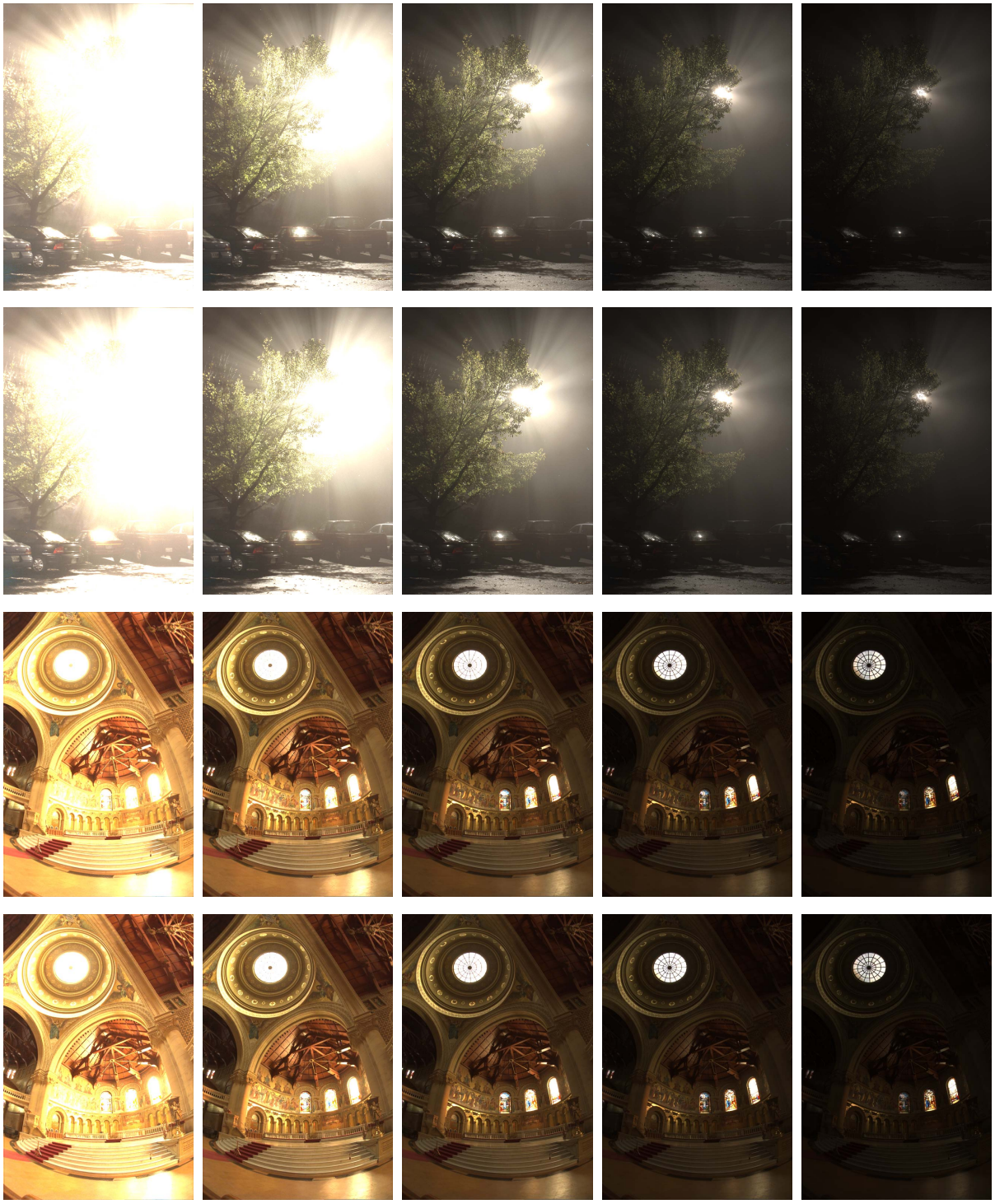


Figure 8: HDR compression results. For each group of images, the top row shows the original while the bottom row shows our reconstruction. We show different exposures of the same HDR image for comparison.