Real-time Rendering with Wavelet-Compressed Multi-Dimensional Datasets on the GPU

Stephen DiVerdi Nicola Candussi Tobias Höllerer
University of California, Santa Barbara, CA

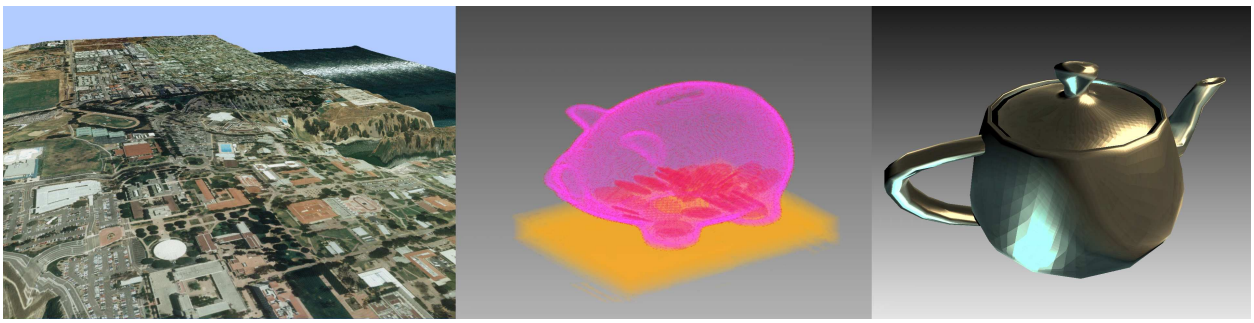

Figure 1: Large datasets, wavelet-compressed and rendered on graphics hardware. Left to right: (a) A 16384^2 *sample aerial photograph texture mapped on exaggerated elevation mesh. (b) A* 512^3 *sample volume dataset, rendered with view-facing orthographic slices. (c) A teapot rendered with a* 64^4 *sample metallic BRDF material.*

Abstract

We present a method for using large, high dimension and high dynamic range datasets on modern graphics hardware. Datasets are preprocessed with a discrete wavelet transform, insignificant coefficients are removed, and the resulting compressed data is stored in standard 2D texture memory. A set of drop-in shader functions allows any shader program to sample the wavelet-encoded textures without additional programming. We demonstrate our technique in three applications – a terrain renderer with a 16384^2 sample RGB texture map, a volume renderer with a 512^3 sample 3D dataset, and a complex material shader using an unapproximated BRDF dataset, sampled at 64^4 in RGB.

Key words: texture compression, wavelets, programmable shaders, large textures, BRDF, volume rendering, multi-dimensional functions

1 Introduction

Dedicated 3D graphics hardware has experienced a revolution in capabilities in recent years, reaching a level of complexity and programmability comparable to general purpose computing hardware. One considerable obstacle to unleashing the full power of these advancements is the limited support and capacity for general purpose data storage on the graphics card. The main form of storage is texture memory, of which modern consumer graphics cards have up to 512MB, though much more commonly 256MB or even 128MB. All modern graphics cards include support for 1D, 2D, and 3D textures, with 8-bit fixed precision or more recently as 32-bit floats, but are limited to preset maximum sizes. Higher dimensional textures are not commonly supported.

We present a technique to encode multi-dimensional datasets in texture memory, with drop-in pixel shader functions for ease of sampling, greatly surpassing the storage limitations of current graphics hardware. The encoding is based on wavelet techniques, providing lossy compression and transparent support for multi-dimensional and high dynamic range data. We demonstrate the real-time rendering of wavelet-compressed data in three applications: a 2D texture in a landscape renderer, a 3D texture used in a simple volume renderer, and a full 4D BRDF material renderer. Each application uses very large datasets compressed with our technique.

Wavelet encoding is popular for processing multi-dimensional datasets, often applied to image compression, as in JPEG2000 [26]. The discrete wavelet transform is easily implemented in software, and has even been adapted to graphics hardware for improved performance [15]. However, existing GPU implementations have been limited to 2D data sets, and are not capable of randomly accessing pixels from the transformed data set, making them inappropriate for general texture data compression. Additionally, these algorithms traditionally have been too complex to be applied in real-time applica-

tions. Our technique addresses these issues by adapting an existing technique for storing and randomly accessing wavelet-encoded data in texture memory using the latest capabilities in programmable shaders to achieve interactive framerates.

The advantages of storing texture data with a wavelet encoding are numerous. First, wavelet encoding lends itself to a straightforward lossy compression scheme, by disregarding coefficients below a certain threshold. This means that textures will require less memory for storage, and also that the texture-size limitations of the hardware can be exceeded, allowing huge textures without tiling. Wavelet encoding is also easily extended into n dimensions with the same decoding algorithm. This includes BRDF and BTF data, which previously required approximations [21] to be used in hardware. Finally, using floating point values, the wavelet encoding can transparently support high-dynamic range data.

All programs and performance measurements are from our test system, an Intel Xeon 3GHz with an NVIDIA GeForce FX 6800 GT 256MB, running Windows XP with DirectX 9, and Linux 2.4.20 with OpenGL 1.5, NVIDIA driver 1.0-6629 for XFree86 4.3.0, with NVIDIA Cg 1.3.

2 Related Work

Wavelets have long been used for data compression, in particular of 2D images [11]. The JPEG2000 standard [26] uses a Daubechies 9/7 wavelet basis which arguably allows for better compression than is possible using the discrete cosine transform [22]. Work has been done using graphics hardware to accelerate wavelet decoding [15, 14], but this work has focused on decoding an image in its entirety, not on a per-pixel basis for random

Hardware-accelerated 2D texture compression is employed by most off-the-shelf graphics hardware. S3 introduced five simple lossy block-decomposition–based compression schemes with compression rates of 4:1 and 8:1 [1] that were widely adopted as part of DirectX 6.0. Wavelet-based compression techniques allow more flexibility and higher compression rates while yielding better quality, at the cost of higher decompression complexity. However, as we demonstrate in this paper, decompression speed can be improved by efficient use of modern hardware pixel shaders to yield interactive frame rates even for challenging applications with considerable amounts of texture sampling.

Many other techniques have been developed for 2D texture compression that are amenable to graphics hardware. Beers et al [3] introduced a vector-quantization–based technique that uses a precomputed codebook and stores a smaller texture of indices into this codebook. The size of the codebook determines the level of compression. More recently, Fenney [12] developed a way to store a compressed texture so that decompression only requires a single lookup per sample. These techniques work well in their domain, but require design and implementation effort to effectively handle multi-dimensional and high-dynamic range datasets, as they lack the general applicability of our technique.

There is a considerable body of work on methods for texture management and caching, including hardware-supported techniques [8]. Clipmapping [25] is a high-end hardware/software approach to render with textures that are much bigger than will fit into texture memory using SGI Performer and special hardware features on the Infinite Reality platform. MIP-levels were paged in efficiently over a high-speed bus. Tiling, progressive loading, texture roaming and multi-resolution rendering are all important techniques for terrain visualization applications [6][7] and volume rendering of arbitrarily large datasets [4]. These techniques are useful alternatives and complements to texture compression in certain domains, but in 3D volume rendering, BRDF shading, or general random access of high-dimensional datasets when all texture data must be accessible immediately, they are insufficient.

The immense storage requirements for 3D textures has motivated more advanced texture compression technique research. Bajaj et al [2] have proposed a technique based on wavelet compression and vector quantization that achieves compression ratios of over 50 to 1, but their implementation for interactive rendering is limited to low-fillrate applications such as 3D texture mapping of polygonal surfaces, making it inappropriate for, for example, volume rendering.

No commodity hardware currently supports textures with dimensions greater than three. Several research projects are concerned with hardware-accelerated decompression of encoded higher-dimensional functions, in particular for Light-Field Rendering [13]. Kraus and Ertl [17] present a vector-quantization–based ”adaptive texture map” approach for 3D volume data and 4D light fields. However, the primary usefulness of adaptive texture maps is limited to sparse datasets with large uniform areas. Our results using the more expensive wavelet encoding for dense 3D or 4D datasets yield about an order of magnitude higher compression rates, while experiencing less pronounced compression artifacts. We achieve this at the cost of rendering speed, which, however, is only a noticeable factor in texture-evaluation intensive applications such as volume rendering with contributions from every voxel (see Section 4.2).

Lalonde and Fournier proposed an algorithm based on nonstandard wavelet transform and zero-tree encoding to efficiently store and sample BRDF datasets and light fields [18, 19]. Their solution can be easily extended to

multi-dimensional datasets, but it is not amenable to hardware implementation.

Different approximation techniques have been used to reduce the dimensions and size of BRDF data, such as separable approximation and homomorphic factorization [16, 21]. The 4D BRDF is represented as a product of two 2D functions and rendered in hardware using two cubemaps. While this method is very efficient in terms of compression and rendering time, the quality of the approximation is largely dependent on the type of material the BRDF data represents and the chosen parametrization. Additional work has been done to extend this method to include importance sampling and suggest its use for higher-dimensional functions such as BTFs (bidirectional texturing functions) [20]. Our compression scheme is much more general and flexible in weighing quality against space than these approximation approaches, again at the cost of rendering performance - but we still achieve real-time frame rates in our BRDF renderings.

3 Compression Technique

Our compression technique is based on the discrete wavelet transform with a Haar basis. The wavelet data is stored in texture memory for fast random access by our shader. We explain our shader, review performance data, and discuss methods of improving performance.

3.1 Wavelet Transform

The mathematics of the discrete wavelet transform are well discussed in the literature [9, 11, 10, 24] – we defer to their expositions. What is important for data compression is that wavelet transform produces the same number of coefficients as the original data, but many of them are close to zero. By using a sparse representation of the coefficients, the original data can be stored more compactly. Lossy compression can be applied as well, decreasing data size at the cost of accuracy by removing coefficients below a certain threshold and quantizing the remaining coefficients. This has the effect of removing localized frequencies with little impact on the final image, resulting in a small change in visual quality and low error when compared to the original data.

The choice of the wavelet basis is critical for wavelet compression, especially for the hardware implementation. The two important characteristics of a basis are the width of support and the compression it can provide. Wider bases provide greater compression but are also more expensive computationally. We chose the Haar basis because it is the one with the most compact support, maximizing decoding performance.

Using wavelet compression for textures in hardware requires a data structure that allows a sparse representation of the wavelet coefficients in texture memory, and at the same time allows efficient random access. Lalonde introduced a solution to this problem, the *Wavelet Coefficient Tree* (WCT) [18], as a general way to store and selectively reconstruct wavelet-compressed 4D BRDF data for software rendering. Due to the limited functionalities of the GPU with respect to a general purpose CPU, this solution cannot be directly implemented with current hardware however, so we present an adapted version of the WCT that is amenable to current graphics hardware. Afterwards, we discuss optimizations to improve the performance of our adapted WCT.

3.2 Hardware Implementation

For simplicity of discussion, we describe here the solution for the case of a 2D dataset.

We use two textures to store the WCT – the coefficient texture and the index texture. The coefficient texture stores the wavelet coefficients for each node of the tree, while the index texture stores children pointers for each node. The tree is laid out so that the children of a node are consecutive, allowing each node to store a single pointer to the first child, and use an offset to access the siblings. A simple one-to-one mapping is used between the two textures, for easy access – each node's data is stored in the pixels at the same location in the separate textures, thereby requiring a single texture coordinate to access all a node's data.

We chose this division of coefficient and index textures to allow the use of different precision for the representation of the wavelet coefficients from the child pointers. This way, we can take advantage of the fact that wavelet coefficients usually need fewer bits of precision than the values stored in the index texture, especially if quantization is applied. Since for a 2D wavelet transform there are three coefficients per node, we use an RGB 2D texture for the coefficient texture. The index texture is only an RG texture, because it stores the 2D texture coordinate of the child pointer.

Figure 2 shows an example of a WCT that could result from an 8x8 image, assuming that the children of nodes N_2, N_3, and N_4 have all coefficients equal to zero and so are not stored. The coefficients for node N_n are the set $C_n = \{c_{n,0}, c_{n,1}, c_{n,2}\}$. To store the example coefficient tree using our scheme, we use two textures of size 4x4, which can also be seen in Figure 2. Node N_0 is stored at position $(0,0)$ in the textures. Its four children are stored starting at $(0,1)$, so that is the value of N_0's child pointer. The children of N_1 start at $(0,2)$. All the child pointers of the leaves of the tree point to the special *zero nodes*, which start at position $(0,3)$. The zero nodes have coefficients $C_z = \{0,0,0\}$ and all their child pointers point back to the first zero node at $(0,3)$.

The pseudocode for accessing a point at coordinate (u,v) is the following:

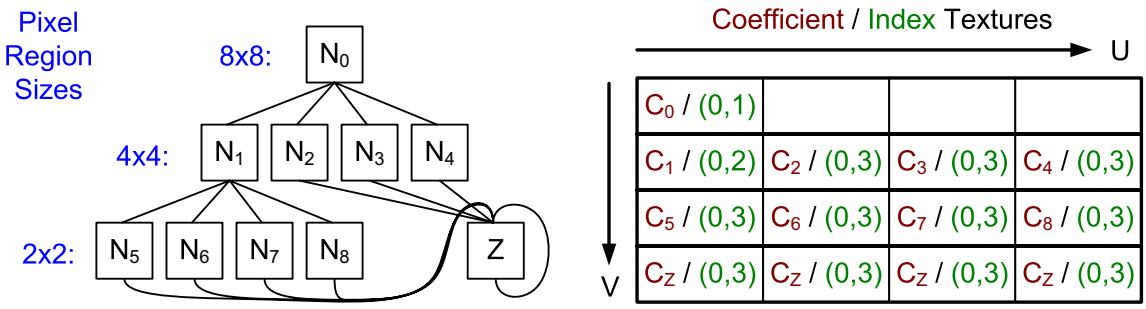

Figure 2: Left: Example wavelet coefficient tree for an 8x8 image. Right: Resulting coefficient and index textures, overlaid to show one-to-one correspondence. Each node's coefficients and child pointer (in the form of a texture coordinate) are stored in a single pixel in each texture.

	1024^2 nearest	2048^2 nearest	1024^2 linear	2048^2 linear
All Levels	14.8	12.4	-	-
Last 4	61.1	61.0	6.91	6.92
Last 3	72.2	72.2	7.86	7.84

Table 1: Decompression performance and different resolutions, different sampling, and different partial evaluations, in millions of texels per second.

```
float waveletSample (float2 uv)
{
  child = tree root node
  average = 0
  for each level of the tree
  {
    coeffs = sample coefficient texture at child
    child = sample index texture at child

    quadrant = 0
    quadrant += (uv.x < half level size) ? 0 : 1
    quadrant += (uv.y < half level size) ? 0 : 2

    average = dot(coeffs,basis[quadrant])
    child += quadrant
  }
  return average;
}
```

The zero nodes are necessary because the shader must iterate over every level of the tree's height, even though some of the data may have been removed because it was insignificant. Current shader programming languages do not support early termination of loops, even though this feature is in some specifications. Therefore, when a pruned node is reached, the shader must continue to evaluate it as if the tree had not been pruned. The zero nodes act as a replacement for all the pruned nodes – parents of pruned nodes point at the zero nodes and the zero nodes point back to themselves and all their coefficients are zero, so the final decompressed value will not change once it has been reached.

3.3 $n > 2$ Dimensions

Extending wavelet encoding to multiple-dimensional datasets is straightforward as a product of one-dimensional decompositions, as discussed in [18]. The modification to our WCT involves increasing the number of coefficients per node – for an n-dimensional dataset, there are $2^n - 1$ coefficients per node of the WCT. For 3D and 4D textures, the 7 and 15 coefficients respectively can be spread across multiple 32-bit coefficient textures, or packed into the channels of a single 64-bit or 128-bit coefficient texture, depending on the capabilities of the graphics card. The sampling algorithm must also be modified slightly, so the `quadrant` variable takes contributions from each axis of the nD grid to select among 2^n basis functions.

3.4 Discussion

The compression ratio obtained with our technique is mainly dependent on two factors: the number of coefficients that are kept from the full WCT and the number of bits of precision present in the index and coefficient textures. More coefficients give a better quality image but also increase the texture size. The number of coefficients also determines the precision necessary for the index texture because the index texture can potentially address each coefficient in the tree. The precision of the coefficient texture determines the amount of quantization of the coefficients themselves. The wavelet compression uses 32-bit float values, which can then be stored in texture memory as 32-bit or 16-bit floats, or 8-bit (or even fewer!) fixed. Generally, 8-bits or fewer are sufficient. Figure 3 shows the quality of 2D textures obtained with different compression ratios and with their respective RMS errors. Compression takes place as an offline process, ranging in time from seconds to tens of minutes, depending on the size of the dataset.

Decompression speed is dependent on both the number of texture samples and the number of operations in

Figure 3: Comparison of 2D image compression, zoomed in to show detail. Clockwise from Top-Left: (a) Original 16384^2 *image, uncompressed at 768MB. (b) Downsampled to* 4096^2*, largest size supported by our graphics card, at 50MB, RMS error 0.65. Detail is uniformly lost everywhere. (c) Wavelet compressed to 138MB (ratio of 5.6), RMS error 0.027. High-frequency information remains. (d) Wavelet compressed to 42MB (ratio of 18.3), RMS error 0.046.*

the shader. Each level of the tree requires one sample of the index texture and one sample per coefficient texture - in 3D and 4D each node has 7 and 15 coefficients respectively, which may be stored across multiple textures. To reduce the necessary texture sampling, we packed all the coefficients into a single texture. The number of operations depends on the height of the tree as well as the number of dimensions. For each dimension, there are more instructions to determine which basis function to multiply the coefficients by, and for larger images there are more levels of the tree to process before the final result is computed. Table 1 shows different fillrate performances with different 2D texture sizes.

The main limitation of our technique is that it only offers point sampling. Linear filtering comes at a serious cost, requiring 2^{dim} samples, plus instructions for interpolation. Figure 4 shows a comparison of rendering with and without filtering, and Table 1 compares their performance. It is worth noting that our technique transparently supports mipmapping at no additional cost. Each successive evaluation of the WCT computes a finer-resolution sample of the original dataset. Therefore, by halting the tree traversal early, a sample equivalent to a higher level of the mipmap hierarchy will be generated.

While we tested our implementation on an NVIDIA GeForce FX 6800, it is compatible with any card that supports pixel shader 2.0 and above. Due to the number of instructions required for large images, especially in the 2D and 3D cases, the instruction limit can easily be reached. However, for newer NVIDIA cards and drivers, loops are supported without unrolling, so arbitrary height trees can be accommodated.

3.5 Improvements

Partial Evaluation. The height of the WCT is equal to $h = log_2(s)$ where s is the size of the data along one dimension. For example, a 2D texture of size $16384x16384$ would produce a WCT with height 14. Only evaluating the first $n < h$ levels of the WCT results in a lower resolution image. One way we improve performance of our algorithm is to pre-evaluate the first n levels of the WCT and store the resulting low-resolution image into a texture, storing the wavelet representation for only the $h - n$ remaining levels. The decompression algorithm then samples the lower resolution image and adds details by evaluating only the last levels of the WCT. In the case of a $16384x16384$ image with the first 10 levels pre-evaluated, the $1024x1024$ image only requires 0.39% additional storage space, while saving 71% of the computational cost of decompression. The storage and sampling of this lower resolution data is trivial in the 2D and 3D cases because of the native 2D and 3D texture capabilities of graphics hardware. For higher dimensions, a separable approximation technique [21] could be used, which is adequate to store the necessary low-frequency information. Table 1 shows a performance comparison between full and partial evaluation. Since the height of the tree depends on the size of the dataset, the performance of the full evaluation decreases as the dataset grows, while the performance of the partial evaluation remains constant.

Early Bailout. Removal of near-zero coefficients from the WCT not only reduces size, but should also increase performance of decompression by reducing the average height of the WCT. However, our current implementation must always evaluate the full height of the tree even if data has been discarded, because current shader programming languages lack support for the ability to break out of a loop early. Once such support exists, it will be a simple modification to our code to take advantage of the speed-up. In the meantime, we performed a comparison of our software decompressor with and without early bailout, and found a performance increase of between 20% and 40%. We can expect a similar speed-up over the performances reported in Table 1 when this support is available in hardware.

4 Applications

To demonstrate the capabilities of our technique, we have implemented three example applications that use wavelet-encoded multi-dimensional datasets in various

Figure 4: The terrain renderer zoomed in, showing the difference between nearest and linear sampling. Relative performances are shown in Table 1. Our terrain renderer runs at 70fps at 640x480.

capacities.

4.1 Terrain Rendering

Terrain rendering requires high resolution 2D textures to adequately represent all the surface details. The maximum texture size allowed by most modern graphic cards is 4096*x*4096 and the amount of available memory is a rigid constraint on the total size of the terrain that can be held in local memory. We implemented a simple terrain renderer that surpasses these limitations by displaying a landscape textured with a single image of size 16384*x*16384 (768MB uncompressed). In this case, the amount of memory used in the video card for storing the compressed texture is 144MB. With point sampling, the fillrate is around 72 million texels per second, resulting in a framerate of 70fps at 640*x*480. With bilinear filtering, the fillrate drops by approximately a factor of five, which is because of the fact that filtering is implemented by interpolating the nearest four point samples. Figure 4 compares the quality obtained with and without bilinear filtering at a very close-up view of the textured terrain.

Another technique related to terrain rendering that could take advantage of our 2D decoder is displacement mapping together with adaptive tessellation. The decompression algorithm can be easily implemented in the vertex shader, allowing high resolution displacement maps to be stored and decompressed on the fly. Hardware adaptive tessellation would add details to the mesh only where it is needed, without impeding vertex throughput.

4.2 Volume Rendering

The second example application is a volume renderer of large 3D datasets. We chose volume rendering as a worst-case test-scenario for our technique, because of the large fillrate requirements for volume rendering applications.

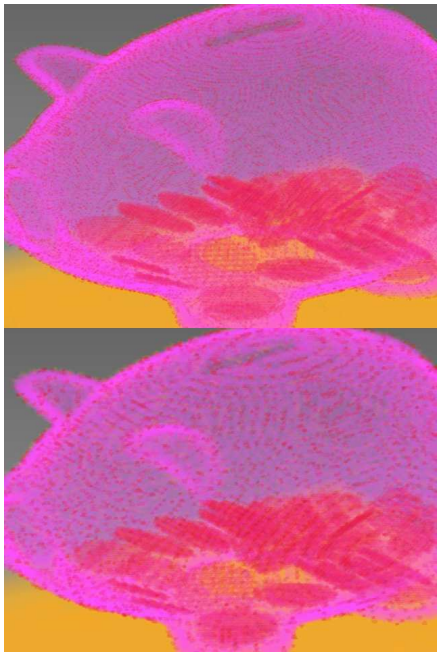

Figure 5: Closeup view of piggy bank volume dataset rendered in 3D. Top: 512^3 *compressed to 5.5MB (ratio of 23.4). Bottom:* 128^3 *uncompressed. Note the grainy appearance of the lower resolution dataset. Our volume renderer runs at 2fps with 128 slices at* 640*x*480*.*

With approximately 2fps at a 640*x*480 viewport, our renderer is still just about interactive when using wavelet-compressed 3D textures, showing that our technique can handle high-fillrate applications. Applications using 3D textures for low-fillrate techniques such as 3D texture mapping of polygonal surfaces, would achieve performance between one and two orders of magnitude faster.

We use OpenGL to render 2D slices of a volume back-to-front with alpha blending [5], to create an image of a dataset stored in a wavelet-encoded 3D texture. In general, modern graphics cards natively support 3D textures of sizes up to 512^3, but the amount of data contained in such a texture, 128Mb times the number of channels, would eclipse the storage found on these graphics cards. This makes 3D textures good candidates for compression, which the builtin hardware compression on many graphics cards is not available for. Additionally, many 3D datasets come from medical measurements such as MRI scans, at depths greater than 8-bits. Current graphics cards support 16- and 32-bit float textures, but 3D float textures are restricted to a small subset of available formats, so our technique is useful in that it will always allow an application to use the full resolution of arbitrary source data.

For our example dataset, we used a scan of a piggy bank containing some coins supported by a piece of wood

from [23], at a resolution of 512^3 with one 8-bit channel. In Figure 5, we show the dataset rendered at a resolution of 512^3 compressed to 5.5Mb for a compression rate of 23:1, compared to the same dataset at 128^3 uncompressed, both without filtering. A simple transfer function is applied to highlight the three objects in the image. The lower resolution dataset clearly suffers from graininess artifacts from aliasing, while the high resolution image is much smoother with an RMS error of 0.013 compared to the uncompressed dataset. The same compression applied to a 1024^3 image would results in a dataset around 125MB, and a slightly lossier encoding would easily be achievable in hardware.

4.3 BRDF Shading

The third example application is a simple renderer that shades objects with complex materials represented by full BRDFs. Support for 4D textures does not exist on current commodity graphics cards, which is why real-time rendering with BRDF materials has been limited to lower-dimensional representations created using a separable approximation algorithm [21]. However, separable approximations depend heavily on choosing a good parameterization of the BRDF, which can be difficult to do well, and even then, high-frequency features are often muted if not lost entirely.

The main advantage of our technique is the ability to render with a full, unapproximated BRDF texture on the graphics card. These datasets are very large however, weighing in at 48Mb for a 64^4 RGB BRDF. Thankfully, BRDFs are very amenable to wavelet compression as they generally contain mostly low frequency information, with very localized high frequency highlights. By decomposing the data into spatial and frequency components, the highlights can be maintained while more heavily compressing the diffuse regions. Finally, both measured and analytic BRDF datasets can contain high dynamic range information that is clamped at 1.0 because of the lack of support for float cube map textures. Our wavelet encoding is able to retain the full dynamic range of the original BRDFs.

Figure 6 shows a comparison of our wavelet compression with results using separable approximation, on an approximated analytic Schlick BRDF simulating a complex metal from [27]. While separable approximation yields a very compact representation of a BRDF, it is fundamentally limited in how accurately it can reproduce the fine detail of a complex material. As can be seen in Figure 6, even with heavy compression, our wavelet encoded BRDF has a lower error than the separable approximation and clearly reproduces the high frequency specular highlights much more accurately. Unfortunately, because we do not support filtering yet, the more lossy wavelet compression suffers from quantization of low frequencies, resulting in color banding. Linear filtering, either through interpolation or a higher-order wavelet basis would remove this artifact.

5 Conclusion

In this paper, we presented a new technique for storing and sampling multi-dimensional datasets on commodity graphics hardware. The main advantage of our technique is its general purpose nature, coupled with good compression rates and high visual quality – it enables graphics cards to store extremely large datasets, through variable rate lossy wavelet compression. Graphics cards with support for only 1-, 2-, or 3D textures can use our technique to store higher dimensional datasets, such as BRDFs and BTFs. We achieve these results by encoding data as a wavelet tree stored in texture memory, and sampling it with a simple to use drop-in shader function. The generic nature of our technique is demonstrated in three different applications – a terrain renderer, a volume renderer and a BRDF material shader.

The main improvement we would like to see in our technique is in decoding performance. We achieved real-time framerates for most applications, but fillrates remain lower than existing hardware texture compression schemes. However, we presented a method of increasing the performance of our technique with partial evaluation, and improved support for general purpose programming in next generation GPUs will provide other significant performance improvements. The performance of linear-filtered sampling could also be addressed by investigating different wavelet bases, such as linear or debauchies. These wider bases should provide filtering support without the need for multiple samples of the compressed texture.

Overall, we are very enthusiastic about the compression results that our method achieves at interactive rendering speeds. We believe that this method opens the door to hardware-accelerated rendering of datasets that, because of their dimensionality or sheer size, were previously excluded from interactive examination.

References

[1] S3TC DirecteX 6.0 Standard Texture Compression. S3 Inc, 1998.

[2] Chandrajit Bajaj, Insung Ihm, and Sanghun Park. 3d rgb image compression for interactive applications. *ACM Trans. Graph.*, 20(1):10–38, 2001.

[3] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 373–378. ACM Press, 1996.

[4] Praveen Bhaniramka and Yves Demange. OpenGL volumizer: a toolkit for high quality volume rendering of large

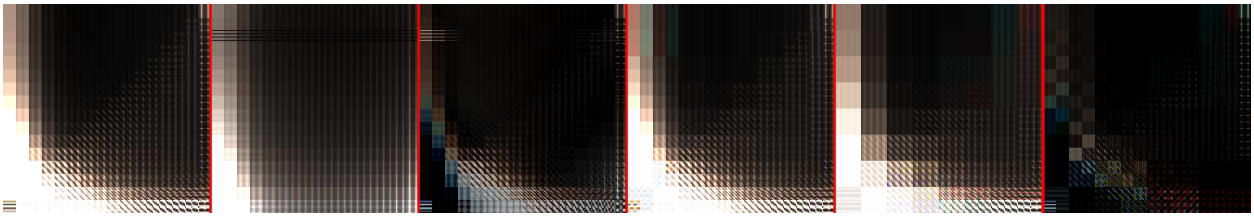

Figure 6: A comparison of BRDF compression results. Left to right: (a) The original BRDF at 16^4*. (b) A separable approximation, RMS error 0.31. (c) Difference between (a) and (b), clearly showing the missing high-frequency highlights. (d) Wavelet compressed to 37KB (ratio of 5.2), RMS error 0.055. No difference image, because the error is so low. (e) Wavelet compressed to 31KB (ratio of 6.2), RMS error 0.14. (f) Difference between (a) and (e). The high frequencies are better represented, but low frequencies suffer from quantization. Our shader runs at 23fps at* $640x480$*.*

data sets. In Stephen N. Spencer, editor, *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics (VOLVIS-02)*, pages 45–54, Piscataway, NJ, October 28–29 2002. IEEE.

[5] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume renering and tomographic reconstruction using texture mapping hardware. In *Proceedings of 1994 Symposium on Volume Visualization*, pages 91–97, 1994.

[6] David Cline and Parris K. Egbert. Interactive display of very large textures. In *Proceedings IEEE Visualization'98*, pages 343–350. IEEE, 1998.

[7] Daniel Cohen-Or, Eran Rich, Uri Lerner, and Victor Shenkar. A Real-Time Photo-Realistic Visual Flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):255–264, September 1996.

[8] Michael Cox, Narendra Bhandari, and Michael Shantz. Multi-level texture caching for 3d graphics hardware. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 86–97. IEEE Computer Society, 1998.

[9] I. Daubechies. Orthonormal basis of compactly supported wavelets, 1988.

[10] Ingrid Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics, 1992.

[11] R. A. Devore, B. Jawerth, and B. J. Lucier. Image compression through wavelet transform coding. *IEEE Transactions on Information Theory*, 38, March 1992.

[12] Simon Fenney. Texture compression using low-frequency signal modulation. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 84–91. Eurographics Association, 2003.

[13] Wolfgang Heidrich, Hendrik Lensch, Cohen Cohen, and Hans-Peter Seidel. Light field techniques for reflexions and refractions. In Dani Lischinski and Greg Ward Larson, editors, *Rendering Techniques '99*, Eurographics, pages 187–196. Springer-Verlag Wien New York, 1999.

[14] M. Hopf and T. Ertl. Hardware accelerated wavelet transformations. In *Proceedings of EG/IEEE Symposium on Visualization*, pages 93–103, May 2000.

[15] Pheng-Ann Heng Jianqing Wang, Tien-Tsin Wong and Chi-Sing Leung. Discrete wavelet transform on gpu. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, August 2004.

[16] Jan Kautz and Michael D. McCool. Interactive rendering with arbitrary brdfs using separable approximations. In *Proceedings of ACM SIGGRAPH 1999*, page 253, August 1999.

[17] Martin Kraus and Thomas Ertl. Adaptive texture maps. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–15. Eurographics Association, 2002.

[18] Paul Lalonde and Alain Fournier. A wavelet representation of reflectance functions. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):329–336, 1997.

[19] Paul Lalonde and Alain Fournier. Interactive rendering of wavelet projected light fields. In *Proceedings of the 1999 conference on Graphics interface '99*, pages 107–114. Morgan Kaufmann Publishers Inc., 1999.

[20] Jason Lawrence, Szymon Rusinkiewicz, and Ravi Ramamoorthi. Efficient BRDF importance sampling using a factored representation. *ACM Transactions on Graphics*, 23(3):496–505, 2004.

[21] Michael D. McCool, Jason Ang, and Anis Ahmad. Homomorphic factorization of brdfs for high-performance rendering. In *Proceedings of ACM SIGGRAPH 2001*, pages 171–178, 2001.

[22] K. R. Rao and P. Yip. *Discrete cosine transform: algorithms, advantages, applications*. Academic Press Professional, Inc., 1990.

[23] Stefan Roettger. Volume library. http://www9.cs.fau.de/Persons/Roettger/library/.

[24] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin. Wavelets for computer graphics: A primer, part 1. *IEEE Computer Graphics and Applications*, 15(3):76–84, 1995.

[25] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In *Proceedings of SIGGRAPH 98*, pages 151–158, 1998.

[26] David S. Taubman and Michael W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, 2001.

[27] Chris Wynn. BRDF-based lighting. http://developer.nvidia.com/object/BRDFbased_Lighting.html.