Chapter 4

Using Wavelets with Current and Future Hardware

Mike Boulton[8]
Rare/MGS

Figure 1. *Example of using wavelets to compress images*

[8] mboulton@microsoft.com

4.1 Introduction

Much of the data we wish to encode over a surface (such as lighting data) is not homogeneous in complexity[9], and is becoming less homogeneous as we pursue higher graphical fidelity.

This data is typically stored using fixed compression methods available in hardware (such as DXT, for example). The main advantages with this approach are that decompression can be performed quickly by the hardware, and also that the size of the data, when compressed, is predictable.

However, this approach also represents a growing inefficiency, since areas requiring a high density of data are under-represented, and vice-versa. In addition, current fixed-compression formats are generally limited to 8 bits per channel.

For applications such as lighting, the problem can be addressed to a certain extent by modifying the injective function which maps the data onto the surface (typically referred to as the *UV Mapping Function*) to map a greater density of texels onto areas of high data complexity, and to also map texels more sparsely over areas of low complexity. Traditionally, this is achieved by either warping the mapping function, or splitting the data up into discrete complexity bands, and uniformly mapping across those bands at a pre-set band density.[10]

There are problems with this approach, however – in particular, when warping the mapping function it is often hard to avoid unpleasant mapping characteristics on the scale of an individual triangle.

Ideally, one would like a variable compression scheme which can focus more on the areas of importance, and less on the areas where not much is happening. This is where wavelets can help!

4.2 An Introduction to Wavelets

Wavelets are mathematical functions formed from scaled and translated copies of a single waveform (referred to as the *mother wavelet*). They allow a function to be decomposed into a superposition of different frequency components, which can be operated on individually (this is referred to as *multi-resolution analysis*).

[9] In other words, some areas require us to store considerably more data than others for consistent quality.
[10] See [Hu08] for example.

A function can be put into wavelet form by the use of a *Wavelet Transform*, and can be translated back into the original function via the inverse transform (analogous to a Fourier transform).

Wavelets as basis functions have several significant advantages over the standard Fourier representation (and its analogue on the sphere, spherical harmonics). They are much better at representing functions with discontinuities or sharp changes, functions that are non-periodic, and in many cases have local support, which can permit efficient windowed modifications of the data set. They are a system of hierarchical refinement, and as such can sparsely represent localized areas of low contrast within the data.[11] At the same time, they can be orthogonal.

A wavelet transform can be either continuous or discrete, and can represent data of any dimensionality. In general, we will be interested in discrete two-dimensional wavelets, in particular *2D non-standard Haar*.[12]

See [SDS95] for a simple introduction to wavelets, and how the non-standard 2D Haar basis is constructed.[13] For a more advanced discourse, refer to [DAUBECHIES92].

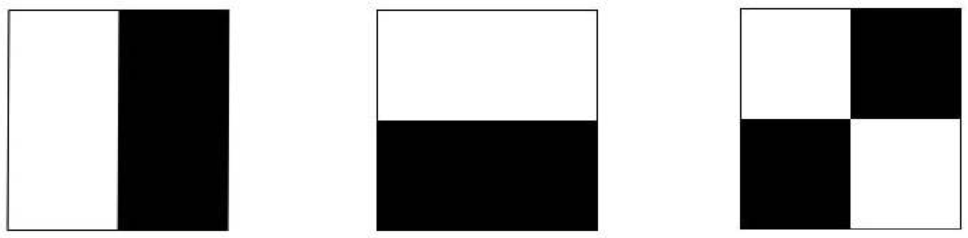


Figure 2. *The vertical, horizontal and diagonal wavelets of non-standard 2D Haar.*

Figure 2 shows a depiction of the three different wavelet basis types for non-standard 2D Haar. The white areas represent +1, and the black areas -1. The scaling function is simply a value of +1 defined over the whole domain.

[11] Since in areas of low contrast, there is very little that actually requires refinement. This is what facilitates compression.
[12] Haar is the simplest wavelet set, and can exhibit blocky artefacts when higher compression is required. This is in general less of a problem for operations such as integration over a hemisphere, except in the case where you have a BRDF whose specular cover is comparable to the cover of the finest wavelet.
[13] Note that we prefer the *non-standard* 2D Haar basis over the *standard* basis because each non-standard basis function has square support which in general produces a sparser representation for the kind of data we wish to compress.

As an example to aid intuition, consider a grid of data with resolution $2^N \times 2^N$, where each cell within the block contains a single real number. We represent the data with this wavelet basis as a tree (commonly referred to as a *wavelet tree*), which in this case has a structure very similar to a quad-tree.

How many wavelets does this grid require in order to be losslessly represented? There is one of each wavelet from figure 2 at the root level, covering the whole grid. These three wavelets tell you how to *refine* the scaling function down to a 2×2 block (where each element has dimensions $2^{N-1} \times 2^{N-1}$). The coefficients are stored in the root via a real triple $\{c_v, c_h, c_d\}$.[14] At this stage, each element of the newly-generated 2×2 block contains the average of all cells contained within it.

The root has four children, one for each $2^{N-1} \times 2^{N-1}$ block. These children each have their own real triple $\{c_v, c_h, c_d\}$, which is used to refine each $2^{N-1} \times 2^{N-1}$ block down to four blocks each of resolution $2^{N-2} \times 2^{N-2}$ (see figure 3).

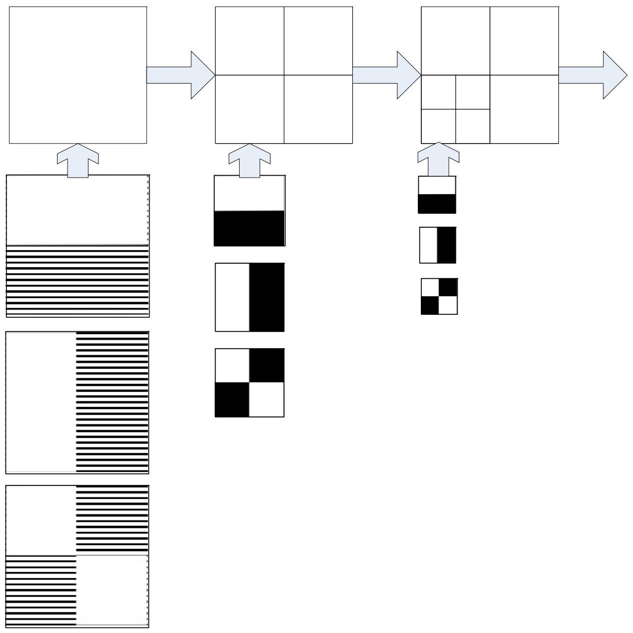


Figure 3. *The first few levels of subdivision.*

This is continued recursively until the whole grid is represented on the level of an individual cell. Note that this is conceptually very similar to traversing a mip-map pyramid, where each level has been generated using a box filter on the preceding level.

[14] Following the convention in [DAUBECHIES92] – c_v is the coefficient for the vertical wavelet, c_h the horizontal, and c_d the diagonal.

So the number of nodes (starting from the root) is $1 + 4 + 16 + \ldots + 2^{N-1} \times 2^{N-1}$, and each node contains three wavelets. Adding this up, and including the single value used to modulate the scaling function, you have exactly $2^N \times 2^N$ terms, which are used to scale $2^N \times 2^N - 1$ wavelets, and a single scaling function.

See listing 1 for pseudo-code to calculate the triple $\{c_v, c_h, c_d\}$ for a node, and listing 2 for pseudo-code to retrieve the refinement value for a supplied coordinate.

```
void CalculateNodeCoefficients( int xMin, int yMin, int xMax, int yMax )
{
   int  xHalf = (xMin + xMax)>>1;
   int  yHalf = (yMin + yMax)>>1;

   // calculate grid average over node cover
   float       nodeAv = GetAverageWithinGridRegion( xMin, yMin, xMax, yMax );

   // calculate grid average over cover of each child
   //  note "UL" stands for "Upper Left", etc.
   float       quadrantAvUL = GetAverageWithinGridRegion( xMin, yMin, xHalf, yHalf );
   float       quadrantAvUR = GetAverageWithinGridRegion( xHalf, yMin, xMax, yHalf );
   float       quadrantAvLL = GetAverageWithinGridRegion( xMin, yHalf, xHalf, yMax );
   float       quadrantAvLR = GetAverageWithinGridRegion( xHalf, yHalf, xMax, yMax );

   // calculate wavelet coefficients for this node
   float       cv = nodeAv - 0.5f*(quadrantAvUL + quadrantAvLL);
   float       ch = nodeAv - 0.5f*(quadrantAvUL + quadrantAvUR);
   float       cd = nodeAv - 0.5f*(quadrantAvUL + quadrantAvLR);
}
```

Listing 1. *Pseudo-code to generate the wavelet coefficients for a node.*

```
// Get refinement for an (x, y) within a particular node with bounds
//  xMin, yMin to xMax, yMax and wavelet coefficients cv, ch and cd.
// Note that this assumes that (x, y) is contained within the bounds.
void GetNodeRefinement( int x, int y )
{
   float       refinement = 0.0f;

   int  xHalf = (xMin + xMax)>>1;
   int  yHalf = (yMin + yMax)>>1;

   // vertical contribution
   refinement += (x < xHalf) ? -cv : cv;

   // horizontal contribution
   refinement += (y < yHalf) ? -ch : ch;

   // diagonal contribution
   refinement += ((x < xHalf) ^ (y < yHalf)) ? cd : -cd;
}
```

Listing 2. *Pseudo-code to calculate a refinement value given an enclosed (x, y).*

Note that the refinement value needs to be scaled according to the area of the corresponding wavelet. So if the root has scale 1, the children of the root have scale ¼ since they cover a quarter of the area (similarly, the grandchildren of the root would have scale one sixteenth).

Compression occurs at this stage by applying non-linear optimization to the wavelet terms. There are several ways to do this – the simplest being to discard any wavelet with a coefficient whose magnitude is below some threshold (referred to as *unweighted selection*). This can be shown to minimize the squared error, but that is not always what you want to do.[15]

4.3 Wavelet Applications

Wavelets have many potential applications for real-time rendering. Below is a (far from exhaustive) list of potential applications.

- **Real-time shader texture decompression.** A texture representing arbitrary scalar data (an image or spherical harmonic coefficients for instance) can be compressed lossily into a wavelet tree, which is then compactly represented using a line texture. Given a (u, v) in the pixel or vertex shader, the value of the original texture at that location can be recovered in real-time using unrolled traversal within the shader.[16] In addition, arbitrary filter operations can be performed hierarchically between the image and a filter kernel (note this includes the bilinear filter kernel). This will be discussed in greater detail in section 4.4.

- **Real-time double and triple product integration for lighting.** Following from [NRH03] and [NRH04], wavelets can be used to encode and compress the elements of the lighting integral. Choosing a wavelet set that is orthogonal (such as non-standard 2D Haar) allows a double-product integral to be decomposed into a sparse list of dot product operations. As described in [NRH04], we can extend this to triple product integration, by describing how to derive the tripling coefficients via a simple (and small) set of rules. In section 4.5 we will discuss how to implement these operations on current hardware, and also discuss an approximation which allows the BRDF to be represented analytically, and its contribution to the integral to be approximated in real-time.

- **Static shadow maps.** Shadow maps that are either entirely static, or mainly static, can be represented in wavelets, potentially with a high degree of compression. Depth-values are queried in the same way as the texture compression described above. Occasional, local changes to the shadow map could be incorporated by only considering those wavelets whose cover intersects

[15] See [NRH03] for a discussion of three methods for non-linear lighting approximation.

[16] As will be discussed in part 4.4, a 1024^2 grey-scale image with sub-blocks of size 16^2 can be decompressed on the Xbox 360™ at a resolution of 1280×720 at over 500Hz.

the area of change. This demonstrates the value of having a basis set with local support.

- **Displacement map compression.** Similarly, displacement maps can be wavelet compressed. This can be used for e.g. morph target (or *blend-shape*) compression – in the case of many morph targets applied to a single mesh, typically only a modest subset of the vertices are significantly displaced. Wavelet compression would heavily compress areas of low change, and represent areas of high change to an arbitrary level of precision. Additionally, wavelet compression is a useful method for compressing very large displacement maps for e.g. terrain representation – here (similar to morph targets) you often have smallish areas of high frequency change scattered around, with smooth low-frequency data in-between. With reasonable compression ratios, this would allow a single texture (here storing a wavelet tree rather than the displacement values exactly) to span areas much larger than the maximum texture resolution that current commodity graphics hardware would allow, and might allow for easier streaming and LOD methods.

- **Easier static and dynamic texture packing.** Automatic generation of a UV mapping function over an arbitrary mesh is a tricky problem, made more so when additional mapping characteristics are required, such as distortion minimization, and trying to match texels up across atlas boundaries.[17] In addition to these requirements, we also want to maximize total texture usage in order to efficiently use available memory. There are several programs available which can generate good UV mappings,[18] but in general there are still cases where mappings with poor texture usage are generated. Wavelet image compression will heavily compress the unused gaps between polygons, and can produce good results even for near-lossless compression.[19] For dynamic packing, one could adopt a scheme where new texture space was allocated in large blocks with no effort to effectively tile with existing atlases. When filled, this block would be wavelet compressed.

- **Geometry representations.** A deformable object with associated UV mapping can have the deformations represented by wavelets over the surface. One possible approach would be to allow a fixed upper limit on the number of wavelets to be used (to control memory usage, perhaps). Space could be made for new deformations by removing the oldest existing high-frequency wavelets – so the broad shape of older deformations could be maintained for a longer time, at the expense of the finer details. Additionally, the multi-resolution

[17] See [HLS07] for a good reference on mesh parameterization.

[18] *UVAtlas* in the DirectX® SDK June 2008 for instance.

[19] Because the unused gaps will generate lots of zero-valued wavelet coefficients, and they can be freely pruned without compromising reconstruction fidelity.

representation could be used to directly update the centre of mass and moment of inertia of the object, potentially at a different level of accuracy. This suggests that data represented in a single multi-resolution form is easier to use across different systems with varying precision requirements.

4.4 Real-Time GPU Image Decompression

In this section, a method for real-time wavelet decompression of an 8-bit monochrome texture by a vertex or pixel shader is described.[20] Note this can easily be extended to color textures by storing each channel as a separate wavelet tree,[21] and also can be extended to handle any data type (float for example).

In section 4.2, the generation of a wavelet tree from a block of data was discussed. Given a texture, we firstly partition it into fixed-size sub-blocks. For each sub-block, we generate a wavelet tree. This tree is pruned recursively as follows:

1. Flag every node which is a leaf and where the absolute value of each of the three wavelet coefficients falls below a user-defined threshold.
2. For every node which is not a leaf, and which has all four children flagged, prune all four children.
3. Repeat procedure until no further pruning can be performed.

Note that a node is only pruned if all three associated siblings are also pruned. This is a concession to allow easier tree indexing on the GPU, and does not add a significant overhead to total storage requirements. The scheme can be extended to support more fine-grain pruning at the cost of a more complicated traversal process.

The method used to prune a tree can be entirely user-defined, and can easily be extended to incorporate an *importance mask* – for example, there could be areas of the data which, although lower in contrast, the user still wishes to preserve at a greater fidelity, perhaps losslessly (and vice-versa). The importance mask value can be used to modify the pruning function on a per-texel level.

Each tree is stored linearly and breadth-first, with each node packed into a single ARGB8 texel,[22] where {r, g, b} stores the quantized and windowed wavelet coefficients {c_v, c_h, c_d} and {a} stores the linear offset to the first child of this node, if a child exists.

[20] See [DCH05] for a description of a similar approach.

[21] Although you might consider an HSV (or YCbCr) encoding rather than RGB, and store *hue* and *saturation* at a coarser level than *value*. See http://en.wikipedia.org/wiki/Jpeg2000.

[22] The actual texture format used is `D3DFMT_LIN_Q8W8V8U8` since this automatically maps the wavelet coefficients onto [-1,1] which saves a few shader instructions.

All of the trees are then consecutively packed into a single line texture.

We choose to firstly partition the image into sub-blocks (in this case, sub-blocks of resolution 16 × 16 texels) for two reasons. Firstly, it allows each sub-block wavelet tree to lie within at most two texture cache lines,[23] and secondly it allows a single unsigned 8-bit integer to be used per node to encode the tree structure.

In addition, we require a texture at the resolution of one sub-block per texel to encode the scaling function coefficient, and record the offset from the start of the line texture where the associated sub-block wavelet tree begins. See Figure 4 for more details.

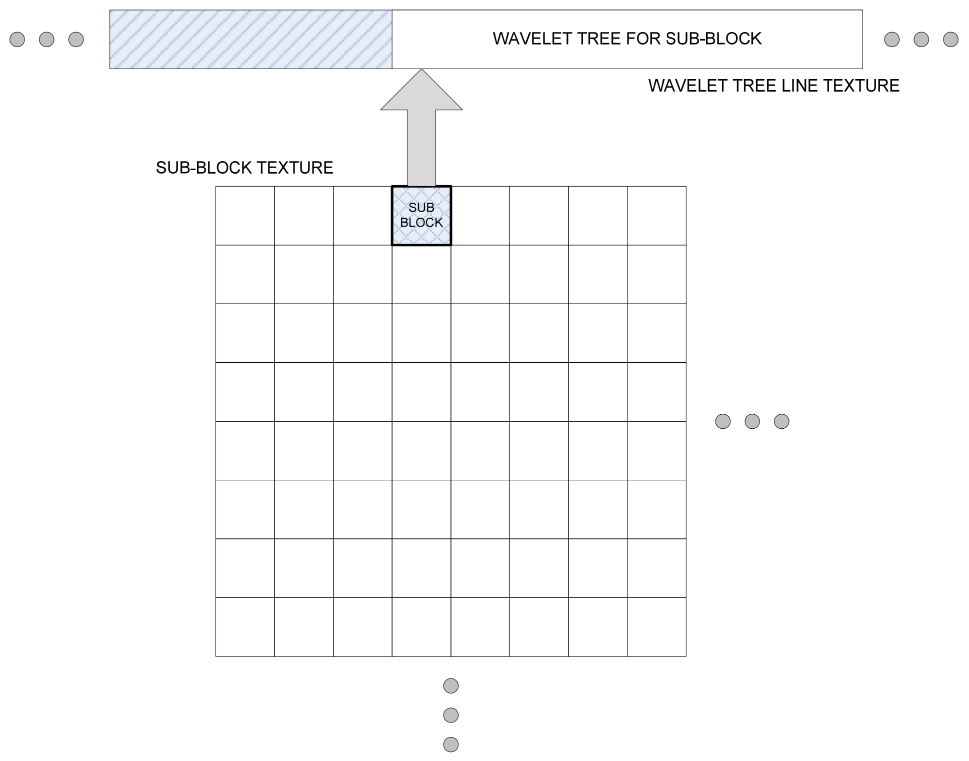


Figure 4. *Each texel in the sub-block texture contains an offset to the wavelet tree.*

So given a (*u*, *v*) within the unit square, we reconstruct the value as follows:

1. Fetch the sub-block texel containing the given (*u*, *v*).
2. Use the wavelet tree offset to point to the start of the sub-block wavelet tree in the wavelet tree line texture.

[23] This is important for performance since we are traversing a tree on the GPU, which in general is not a good fit for fetch coherency and texture cache usage.

3. Perform depth-first traversal down to the leaf containing the given (*u*, *v*) accumulating the contribution made by each wavelet node along the way. As soon as traversal completes, attempt to jump to the end of the shader.[24]
4. Add the result to the scaling term contained within the sub-block texel.

See Listing A1 in the appendix for an optimized Microsoft Xbox 360™ microcode shader which performs these steps.

Figure 5 shows the results for three different cases on a monochrome 1024^2 texture.

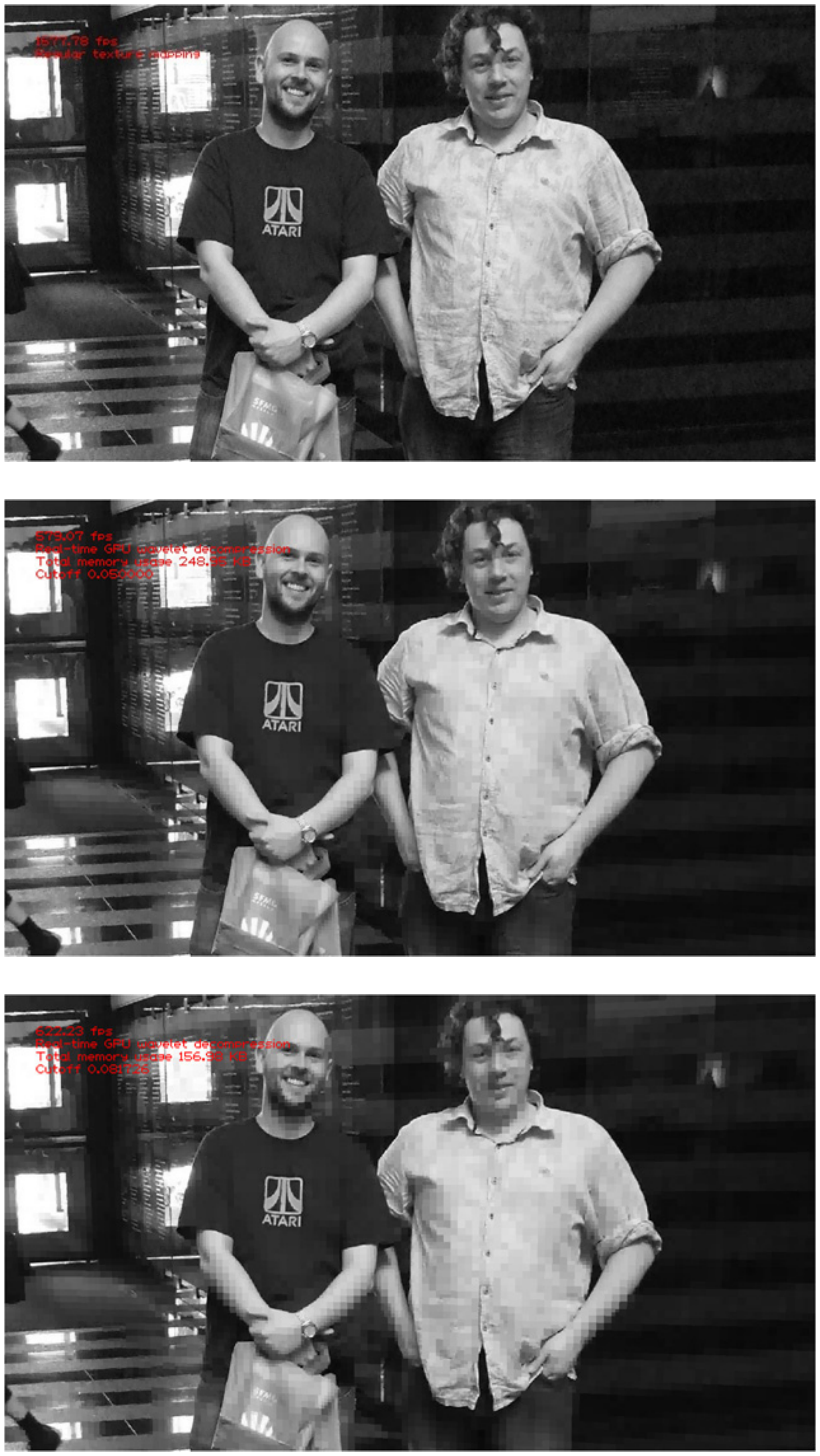
1. Reference case. Here we have a standard uncompressed texture rendering via a single texture fetch. Memory size (with mip-maps) is 2MB.
2. Wavelet compressed with cut-off value of 0.05. The total memory requirement for the wavelet tree line texture, and sub-block texture is 249KB (representing around an 8:1 compression ratio). It is running full-screen at 1280 × 720 with a speed of 579fps.[25]
3. Wavelet compressed with cut-off value of 0.082. The total memory requirement is 157KB (13:1), and it runs with a speed of 622fps.

Notice that as the cut-off is increased, the frame rate also increases. This is because the shader attempts to perform a dynamic branch to the end of the program whenever it encounters a leaf prematurely. For higher compression, more premature leaves will be encountered and so this will yield a greater benefit.[26]

Because of the refinement-based nature of wavelets, a separate mip-map chain is not required, since by simply terminating traversal prematurely the box-filtered mip-map value at that level is obtained.

See figure 6 for a zoomed-in comparison. Note that at higher compression ratios, the blocky artifacts discussed previously are evident in areas of lower contrast. Notice however that even at these high compression ratios, fine high-contrast detail is still represented to a high degree of accuracy (note particularly the collar and buttons).

[24] We would ideally also want to terminate shader traversal if screen-space minification was encountered.
[25] Note that this frame rate also includes a small amount of unrelated system overhead, such as clearing and resolving screen buffers, etc.
[26] On the Xbox 360™, the GPU operates on vectors of 64 pixels wide. For a dynamic branch to be effectively executed (and the performance benefits gained), every element of the vector must follow the same branch path.



```
1577.78 fps
Regular texture mapping
```

```
579.07 fps
Real-time GPU wavelet decompression
Total memory usage 248.95 KB
Cutoff 0.050000
```

```
622.23 fps
Real-time GPU wavelet decompression
Total memory usage 156.98 KB
Cutoff 0.081726
```

Figure 5. *Comparison between reference, cut-off 0.05 and cut-off 0.082.*

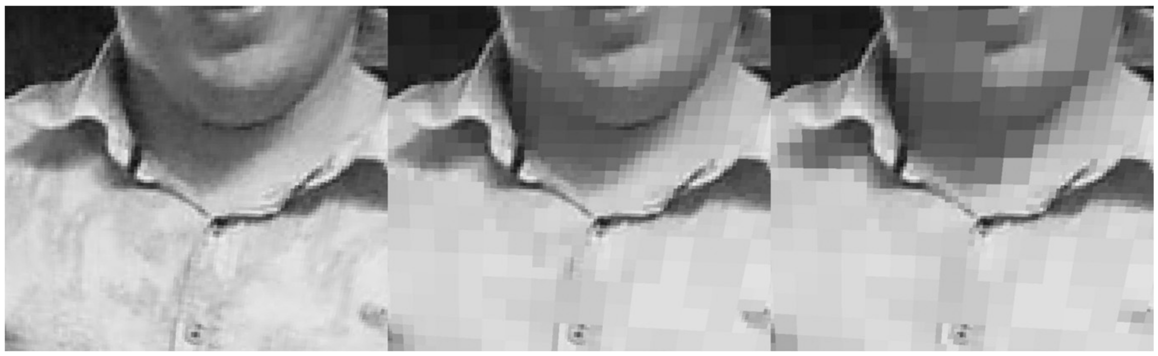


Figure 6. *Zoomed-in comparison for all three cases.*

An associated user-generated importance mask could improve the overall quality of the most compressed case, whilst maintaining the same compression ratio. In particular, if areas around the arms and face of both figures were increased in relative importance a more agreeable result might be obtained.
However, for general images this technique does not often produce results significantly better than DXT5A, for example (which has a fixed compression ratio of 8:1), and for many image applications the blocky artifacts produced by the 2D Haar basis would not be acceptable. Image compression does serve as a good method for demonstrating wavelet compression intuitively, but it is argued that the practical application of real-time wavelet texture decompression is to textures storing general data, such as lighting, which is not of such a homogeneous nature.

For better wavelet compression of general images, more sophisticated wavelets are typically used, for example the *Cohen-Daubechies-Feauveau* wavelet (which is part of the JPEG2000 standard[27]). However, due to the complexity of this wavelet, real-time decompression at interactive rates is not really feasible on current hardware.

Hierarchical filtering between multiple wavelet-compressed images can be efficiently performed. In addition, filtering between an image and an analytically-defined kernel can be performed (the bilinear filter kernel, for instance[28]). The only caveat here is the case where a kernel overlaps a sub-block boundary. This can be addressed by splitting the process up into multiple passes, based on the size of the smallest kernel, or alternatively by overlapping the sub-blocks slightly at a slight cost to overall compression.[29] Here, it is often better to encode the tree in a depth-first manner, rather than breadth-first.

[27] See http://en.wikipedia.org/wiki/Jpeg2000 for example.

[28] There are at least two different ways to approach bilinear filtering – one would be to perform four individual fetches using the shader in Listing A1, and apply the bilinear weights. The other would be to perform a genuine pruned depth-first tree traversal between the image and a per-pixel defined bilinear filter kernel. See section 4.5.

[29] Note this becomes less feasible the larger the filter kernel. The bilinear kernel would only require an overlap border of width one, however.

This is essentially what is being done when calculating the double-product integral for relighting – here, we are performing a filter between the image over the sphere representing the incoming light, and the image over the sphere representing the local visibility and diffuse BRDF (this is discussed in more detail in the next section).

See figure 7 for an example of wavelet-compressed textures containing spherical harmonic coefficients of order 0 and 1 (with resolution 1024^2). In this case, DC and linear spherical harmonic coefficients were stored as 16-bit floats, and wavelet compressed.

CUT-OFF	Overall pruning	DC pruning	Linear pruning
0.00008	94.0%	94.2%	93.9%
0.00016	95.2%	95.5%	95.1%
0.0004	97.9%	98.0%	97.8%

Table 1. *Pruning results for wavelet compression of spherical harmonic data.*

Table 1 shows the percentage of total nodes in the wavelet tree pruned (using the pruning method previously described) as a function of user-defined cut-off.

Here wavelets are able to provide a very high compression rate, whilst still maintaining good reconstruction quality around areas important for perceived lighting fidelity (in particular, note the soft area shadow around the base of the cuboids). Additionally, the pruning approach used here is rather clumsy, and a better algorithm would likely allow further compression.

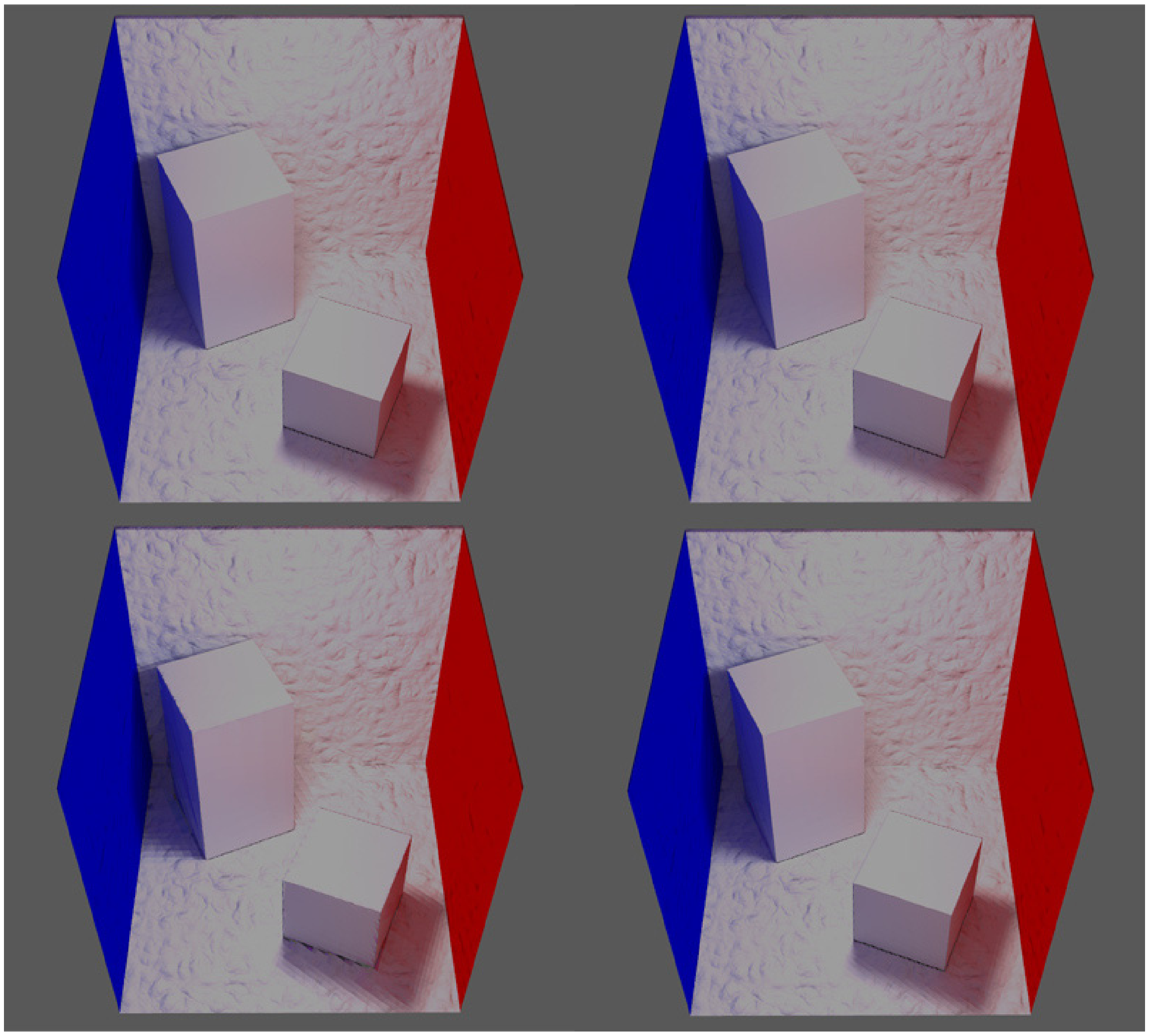


Figure 7. *Spherical harmonic lighting with wavelet-compressed coefficients. Clockwise from top-left: Reference, cut-off 0.00008, cut-off 0.00016, cut-off 0.0004.*

With spherical harmonic lighting on current hardware, one often needs to make a difficult decision about whether to include quadratic SH terms, or just use linear, since it is often the case that quadratic terms only make a noticeable difference to diffuse lighting in certain places, although they require five more coefficients to be stored per channel. A wavelet approach would help here, since the large portions of low-contrast data stored in texture channels representing the higher bands would be automatically compressed away.

Since linear spherical harmonics require twelve channels to represent a color signal, one might think that performing twelve sets of wavelet decompression in the shader would be unrealistic. However, with such high compression ratios, the typical decompression traversal would not likely descend many levels from the root.[30] In addition, representing the light in HSV format for instance would allow two of the trees representing the color information to be coarser still.

4.5 Lighting

Orthogonal wavelets can be applied to the rendering equation in a very similar way to spherical harmonics. The framework for performing double-product integration with 2D Haar wavelets was introduced in 2003 by [NRH03], and extended to triple-product integration a year later [NRH04].

Why use wavelets for lighting instead of spherical harmonics? Spherical harmonics struggle to efficiently encode spatially-compact areas of high variability. As with the Fourier series, they exhibit unpleasant ringing artifacts when trying to resolve sharp discontinuities.[31]

In general, a wavelet approach to lighting requires 2 orders of magnitude fewer coefficients [NRH03], and makes the accurate calculation of high-frequency environmental specular responses much more feasible at rates approaching real-time.

Additionally, wavelet bases with local support can in theory support staged, windowed modifications to transfer functions. For instance, consider the situation where a wall has had a hole punched in it (via a Boolean operation for instance). The hole would allow light through, which would make many surrounding transfer functions within the lightmap invalid. Although rebuilding those transfer functions in real-time might be very costly, the multi-resolution nature of wavelets can allow this cost to be prioritized, and the work spread over many frames. For instance, the highest priority work would likely be updating those transfer functions which receive direct lighting through the hole.

[30] And since one typically encounters large, coherent patches of low frequency lighting, dynamic predication is likely to work well.

[31] This can be addressed to a certain extent at the cost of feature blurring. See [SLOAN08].

Further, one would like to focus on the low-frequency wavelets first, in order to quickly carve out the approximate shape of the cast light. So only the low-frequency wavelets whose cover intersects the volume that the hole subtends onto the sphere would be initially considered.

The shape of the hole borders would then be refined, in addition to updating transfer functions with no direct view of the hole, but which need to change in order to compensate for additional bounce lighting.

This would allow worlds that are not entirely static to benefit from a full GI solution, provided that geometric changes could be controlled so as not to saturate the hardware.

For fast-moving objects, the shadowing produced would have no latency, and a fidelity that is related to surrounding lighting complexity and power of the hardware.

This would be a tricky thing to do with spherical harmonics, since the bases have global support, and could not easily be prioritized in the above fashion.

However, two significant advantages that spherical harmonics do possess (that wavelets do not) is rotational invariance and easy analytic rotation of the basis functions (at least for low-order SH). This poses a significant problem when evaluating the rendering equation for wavelet-compressed components.

For the case of the Phong BRDF[32] for instance, one would like to rotate the diffuse component of the BRDF (a cosine lobe) so that it is oriented along the normal, which might be obtained from a high-frequency normal map which has many texels overlaying a single lightmap texel. Similarly, one would like to rotate the specular component (a cosine lobe raised to a power) so that it is oriented along the reflected light direction.

In [NRH04] we see that this can be addressed by re-parameterizing the BRDF about the reflected light direction, and then generating a set of samples over all remaining variables. This dataset is then wavelet compressed using aggressive non-linear optimization.[33] When performing rendering, the closest appropriate BRDF record is selected from this compressed dataset.

We can also choose to encode the BRDF and visibility term in the local frame of the lightmap texel (as described in [MHL*06]), and rotate the lighting environment into this

[32] Here the cosine term is included in the BRDF.

[33] [NRH04] uses sampling of $(6\times64\times64)\times(6\times128\times128)$ for $\omega_r \times \omega$ for Phong (where ω_r is the reflection vector), which can handle specular powers up to 200. For 99% accuracy, only 0.1-1% of the terms are required.

local frame.[34] The lighting environment is compressed at a sampling of orientations in a similar manner to BRDF compression mentioned above. The calculation is then performed in local space, instead of world space. This is a smart idea because the lighting function is 2D, but the BRDF is 4D, so storage requirements for a compressed sampling of lighting functions will generally be significantly less than the storage requirements for a BRDF representation. However, since the visibility is locked into the same frame as the BRDF, it can't be rotated to correctly compensate for BRDF orientation around high frequency normal map directions, which leads to incorrect solutions.[35]

We will later argue that a relatively simple isotropic BRDF (such as Phong) can be treated as a separate entity, whose contribution to the integral can be naturally folded into the traversal process, and approximated via importance sampling.

See [NRH03] for a good direct comparison between spherical harmonics and wavelets for lighting.[36]

$$B(x,\omega_o) = \iint_{\Omega} L(x,\omega_i)V(x,\omega_i)\rho(x,\omega_i,\omega_o)(\omega \cdot n)d\omega_i$$

Equation 1. *The rendering equation for direct illumination.*

See equation 1 for the rendering equation. Note that it is common practice to incorporate the cosine term $\omega \cdot n$ into the BRDF ρ. The function *L* represents the incoming lighting, *V* the local binary visibility, x is the point on the surface, n is the normal, and ω_i, ω_o represent the incident and outgoing lighting direction respectively.

If the integrand is rearranged into the form *AB*, where *A* is to be varied with respect to *B*, the integration is referred to as *double product*. Similarly, if arranged into the form *ABC* where all three parameters vary with respect to each other, the integration is referred to as *triple product*.

Since non-standard 2D Haar forms an orthogonal basis set, double-product integration decomposes into a sparse set of dot product operations in exactly the same way as with spherical harmonics, since all off-diagonal terms vanish (see equation 2).

[34] [MHL*06] uses s*pherical wavelets* (see [SCHRÖDERSWELDENS95]), with a basis set that is isomorphic to 2D Haar.
[35] Since in order to get a bump-mapped response, the lighting environment is being rotated in the opposite direction so that it has the correct orientation with respect to the fixed BRDF for each normal from the normal map. You need to do this to the visibility function as well, but it is locked into the same frame as the BRDF.
[36] In particular, see figures 2 and 3 in [NRH03].

$$\iint_{\Omega} A(\omega)B(\omega)\,d\omega = \iint_{\Omega} \left(\sum_i a_i \Psi_i(\omega)\right)\left(\sum_j b_j \Psi_j(\omega)\right) d\omega$$
$$= \sum_i \sum_j a_i b_j \iint_{\Omega} \Psi_i(\omega)\Psi_j(\omega) d\omega = \sum_i \sum_j \delta_{ij} a_i b_j = A \cdot B$$

Equation 2. *Decomposition of double-product integration into a sparse dot product.*

Unfortunately, triple-product integration doesn't turn out to be quite this simple, since in general the integration between three orthogonal basis functions does not decompose to something as nice as the Kronecker delta, but a *tripling coefficient* instead. Although in general these can be unpleasant to calculate,[37] [NRH04] shows that for 2D non-standard Haar they can be obtained via the application of a simple and small set of rules.

We now consider implementing a few different approaches on hardware. Firstly, consider the case where we would like to vary the lighting *L* with respect to the rest of the integrand, and for simplicity assume that the BRDF is diffuse-only and that normals are taken from the vertices, rather than a normal map.

A UV mapping function is generated for the lightmap. For every texel within the lightmap, visibility information is generated over the sphere centered at the texel. Each record on the sphere containing binary visibility information is then weighted by the clamped inner product with the interpolated vertex normal.

This data is then projected onto a discrete decomposition of the sphere, such as a cube map, and an appropriate correction term is applied if required,[38] and then transformed into a wavelet representation. Non-linear optimization is then applied.

The above process is typically performed using a ray tracer (or photon mapper), but in this case can be performed very quickly to a reasonable approximation using rasterization into a cube map on the GPU.[39]

Incoming environmental lighting (assumed to be effectively from infinity) is then sampled onto the same sphere decomposition, for an appropriate set of values and/or orientations. In this example, we generate environment lighting sets for *Grace Cathedral* and also for a home-made red area light. Both environments are rotated one

[37] For spherical harmonics, these tripling coefficients are referred to as *Clebsch-Gordan coefficients* and are rather complicated.
[38] The cube map requires such a correction, since texels close to the corners subtend a smaller solid angle than texels close to the center of a face. See [FUJITAKANAI04].
[39] Although if bounced lighting etc. is required, this isn't really feasible. Additionally, if the spherical decomposition is not a cube map itself, the data will need to be re-sampled appropriately.

full revolution around the *X* axis, with 256 equal angular subdivisions. Each frame is compressed in the same way as the per-texel data.

Notice that the per-texel data and the environment lighting data are both defined in world space. This allows direct integration without the need to rotate. It is very important that both the environmental lighting and per-texel transfer functions are in the same space, and use the same decomposition, since we require an exact correspondence between associated wavelet bases.

When performing rendering, we firstly select the most appropriate record from the environmental lighting data set, based upon current orientation. This record contains three pruned wavelet trees, one for each color channel. Each tree is to be integrated against the transfer function wavelet tree for each texel in the lightmap.

As discussed above, the per-texel double product integration that needs to be performed decomposes to just a sum of dot product operations, where the dot product is performed between the triple $\{c_v, c_h, c_d\}$ from each node in the first tree with the triple $\{c'_v, c'_h, c'_d\}$ from the corresponding node in the second tree, if it exists.

Let *A* be the set of linearized indices of all nodes in the first tree. Let $\bar{C}_A^i$ be the triple $\{c_v, c_h, c_d\}$ at node i (appropriately area-weighted), and S_A be the scaling function coefficient of the first tree. Similarly for the second tree (replacing *A* with *B*).

$$I = S_A S_B + \sum_{\forall\, i \,\in\, A \cap B} \bar{C}_A^i \cdot \bar{C}_B^i$$

Equation 3. *Calculation of per-channel intensity.*

See equation 3 for the calculation of per-channel intensity. Note that only the intersection of both trees needs to be traversed for evaluation, since if any node in one tree does not have a corresponding node in the other, there will be no contribution.[40]

So to perform this operation in a shader for instance, both trees need to be traversed in parallel. Each node pair has their contributions added to the total (as in equation 3). However, if the node currently being considered from one tree has children but the corresponding node from the other tree does not, the evaluation function needs to be able to jump over all children of this node (and their descendants) to the next sibling or ancestor (which due to the pruning process described previously will always correspond to the node from the other tree). Traversal is then continued as before until the root is encountered.

[40] Note that this doesn't *directly* follow from equation 3, since there could be a case where a parent was pruned which had children that were not. However, our pruning process described earlier prevents this from happening (and it's actually fairly unlikely anyway).

We modify the process used in section 4 for image decompression as follows: the tree is now depth-first, and stored into a line-texture as before. The alpha channel is now used to store the tree level of the node immediately following the current node in the line texture. If the next level is greater (in other words, the next node is a child of the current node), a special *jump node* is stored in the next texel, between the current node and the child. This node contains the linear offset (in texels) from the current node to the next node on a level less than or equal to it (a sibling, parent or other ancestor).

If during traversal a disagreement is encountered concerning the next node, the jump node is used to entirely omit the descendant branch of the offending tree.

See Listing A2 in the appendix for a microcode shader which performs these steps, and figure 8 for a screenshot of a simple model (a torus over a plane) under both lighting conditions. Here, the UV mapping was generated with a modified version of UVAtlas, and the lightmap has a resolution of 128^2 texels. Each transfer function and environment lighting function was first rendered into a 32 × 32 × 6 cube map, and then compressed (one wavelet tree per cube map face). The transfer functions across the whole lightmap required 8.7MB of total storage. We recorded an average frame rate of over 250Hz was recorded[41], when using the red area light; and for Grace Cathedral just over 40Hz.

Note that this implementation can be significantly optimized – it is currently bound by texture cache stalls because of both jumping behavior, and also the fact that transfer function wavelet trees of adjacent lightmap texels are stored a long way away from each other. We discuss optimization approaches towards the end of the section, including a method of interleaving trees representing localized data. Additionally, predication behavior is quite poor since jobs of a similar size are not in general clustered together.

[41] Again, running on the Xbox 360™.

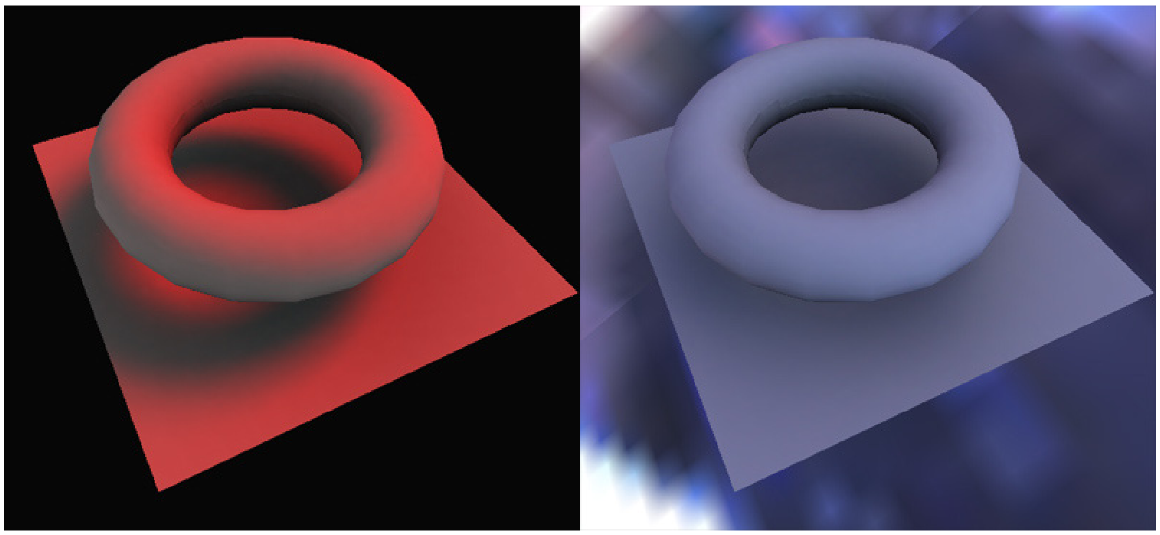


Figure 8. *Real-time GPU double product integration between (left) a red area-light and (right) Grace Cathedral lighting environment.*

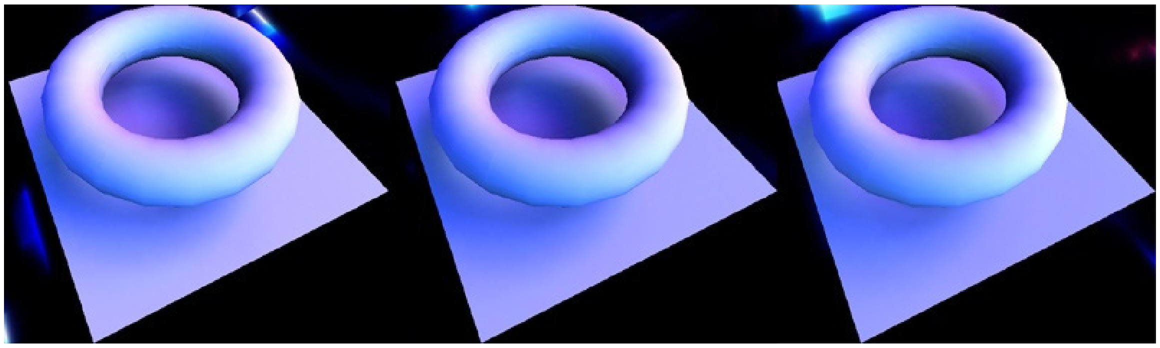


Figure 9. *Three consecutive frames of Grace Cathedral as above, but with exaggerated contrast.*

Notice that even though not entirely optimal, there is a large performance difference between grace cathedral and the much simpler red area light. This is because the red area light has an extremely sparse wavelet representation when compressed, so the intersection between this and any arbitrarily complicated transfer function is always guaranteed to be small.

Returning to figure 8, notice the quality of the torus shadow on the left – this is well beyond the resolving power of quadratic or cubic SH. However, under the smoother lighting conditions such as Grace Cathedral represented in low dynamic range, the results become much more comparable to low-order SH for diffuse lighting (which is expected, see [[RAMAMOORTHIHANRAHAN01]]).[42] See figure 9 for three consecutive frames where the contrast has been artificially increased – here, sharper shadow boundaries are clearly visible.

[42] Note that although the Grace Cathedral case looks a bit bland, it is quite convincing in motion. See associated talk.

Next, consider how one might uncouple the BRDF from the visibility function. If instead we fixed incoming light and local visibility, and chose to vary the BRDF using a user-specified high frequency normal map for diffuse (and specular) bump-mapped response, we would have another double-product integral, and could address the difficulty of orienting the BRDF about arbitrary normals and view directions by adopting the approach in [NRH04], i.e. by compressing a sampling of BRDFs.[43]

However, consider the case where we remove the cosine from the double integral, and rather than performing the integration according to equation 3, we instead perform a point evaluation of the lighting and visibility wavelet tree in a similar way to the method of image decompression discussed in the previous section. At each leaf, we multiply each of the four refined values by an approximation of the BRDF over each individual patch, and add the results to the accumulating total.

See Listing 3 for pseudo-code demonstrating this process.

```
→For each lightmap texel
  →For each transfer function wavelet tree
      →Starting from root node
        Refine current quadrant value
        Am I a leaf?
          →Yes: Multiply current refinement value against approximation of BRDF integral
             over current patch, and add this to the total.
          →No: For each child
            Does my cover intersect the BRDF kernel?
              →Yes: Descend to child
              →No: Do not descend to child
```

Listing 3. *Pseudo-code for approximating the integral between an analytically-defined isotropic cosine-power kernel and a wavelet-encoded transfer function.*

Triple product integration between two functions and the BRDF can also be approximated in a similar way using double product integration, and again multiplying the leaf values with the approximation of the BRDF integral over the patch.

How can the estimate be performed? For certain simple BRDFs, and certain spherical decompositions,[44] direct analytic integration may be possible. Even more complicated

[43] However, it might be a challenge to encode this in a cache-friendly way, especially for a noisy normal map.

[44] The cylindrical mapping is a good one here, since each patch is just a spherical polar square. Unfortunately, there is rather extreme distortion.

decompositions such as *HEALPix*[45] can permit relatively straight-forward analytic integration over each patch (although it does get a bit fiddly in the polar regions, see the appendix in [[GHB*05]]). The problem however is complicated by requiring that the BRDF be clamped to zero over the negative hemisphere with respect to the normal direction, and if the analytic integral is available in closed form, it will likely be complicated and expensive to set up and evaluate.

Alternatively, an SRBF approach could be used. Here, we would approximate each rectangular patch with a circle, and perform integration by using the inner product between the patch center and the kernel center to index into a pre-computed integral table.

A third option would be to approximate the integral by taking point samples over the patch, and averaging their values, calculated analytically.

This is a very good fit for HEALPix, which is hierarchical and has a well-defined method for *nested indexing* (see [GHB*05]). This allows easy generation of unbiased points across any HEALPix pixel, and because of the hierarchical arrangement, it would be easy to generate a quad-tree across the patch to better focus each inter-patch sampling (effectively performing importance sampling). Additionally, HEALPix has a host of other very attractive features, such as equal-area patches and low shape distortion that make it ideally suited as a spherical decomposition for wavelet lighting.[46]

To implement this approach on the GPU, one needs to have the concept of a stack to avoid the need to look backwards in the tree to recover lower level values when ascending from a child during traversal. To implement a scalar stack of fixed size 8, for instance, is a simple operation in the shader requiring only two ALU cycles for a push or pop, and 2 GPRs. Here however we require four floats per push/pop per tree, which can only be performed by using a large block of GPRs and a list of *mov* instructions. See figure 10 for three consecutive frames using this approach.

It also seems feasible to implement this on the CELL™ processor, where sections of tree can be streamed into the local store. Fast reads from the local store during traversal are likely to be better than trying to organize traversal to best utilize existing texture caching methods. Additionally, job organization for best branching/predication performance is likely to be easier.

[45] See [GHB*05] and [WanWong07].
[46] As mentioned previously, *spherical wavelets* have also been used, with similar benefits. Here the basis is triangle-shaped. See [MHL*06] and [SchröderSweldens95].

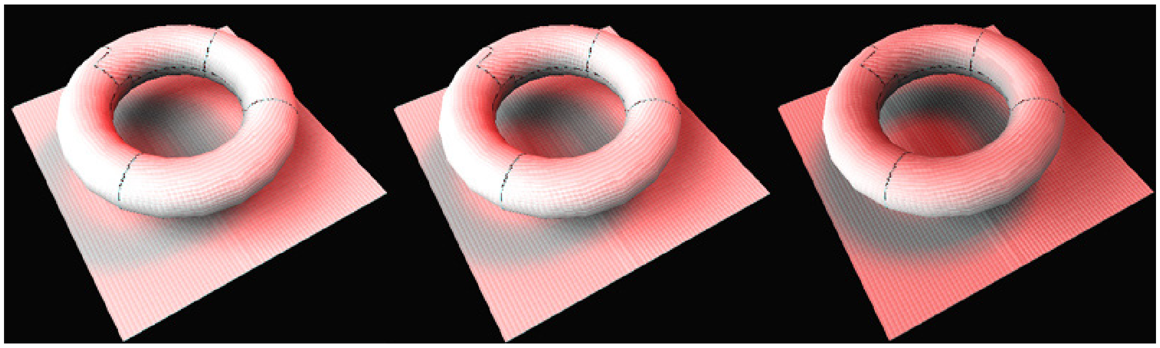


Figure 10. *Three consecutive frames of triple-product integration using point sampling for BRDF integral approximation, and a high-frequency normal map for diffuse lighting.*

4.6 Conclusion

A practical introduction to wavelets has been provided, in addition to demonstrations of real-world examples.

The author believes that methods of variable compression, and “shader tree traversal” algorithms like the ones demonstrated, are likely to play an important role on future hardware, to help better focus existing memory and bandwidth resources.

This approach also allows a generalization to occur concerning the methods we use to parameterize data. “Voronoi diagram” texture maps and 2D tangent space BSP trees are an interesting example of how we can move forward to “point cloud” type representations of data, which not only provide a more honest account of the underlying information, but also allow more ambitious methods for both its modifying and handling.

Although this increased generality does seem to come with a certain “price of entry”, it has hopefully been demonstrated that modest progress can be made even on current hardware.

4.7 Appendix

```
  xps_3_0
  config AutoSerialize=false

  dcl_texcoord0 r0.xy

  // for 1024x1024 there are 64x64 subblocks of size 16x16
  def c0, 1.0, 64.0, 0.015625, 0.0078125
  def c1, 0.00390625, 2047.0, 1024.0, 0.5
  def c2, 2.0, -1.0, 127.0, 127.5

  //
  // fetch the grid values
  //          z              = base scale at 16x16 cell
  //          (x, y)         = offset in coefficient texture to start fetching
  //

  tfetch2D r2.yzx_, r0.yx, tf1, MagFilter=point, MinFilter=point, FetchValidOnly=false

  // calculate initial quadrant dividers
  mul r1.xy, c0.y, r0.xy
  floor r1.xy, r1.xy
  + movs r1.z, c1.x
  mad r1.xy, r1.xy, c0.z, c0.w

  serialize

  // scale index to required range
  mad r2.xy, r2, c1.y, c1.w
  floor r2.xy, r2
  mad r2.x, r2.y, c1.z, r2.x


///////////////////////////////////////////////////
// to 8x8
///////////////////////////////////////////////////

  // fetch coefficient & index
  tfetch1D r4.xyzw, r2.x, tf0, MagFilter=point, MinFilter=point,
      UnnormalizedTextureCoords=true, FetchValidOnly=false  // from 8:8:8:8

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  // calculate new half-level coordinates
  mad r1.xy, r3, r1.z, r1.xy

  serialize

  // scale the index to the correct range, and round
  mad r0.z, r4.w, c2.z, c2.w

  // per-level rescaling
  mul r4.xyz, r4, c8.x
  + floors r0.z, r0.z

  // add contribution
  dp3 r4.w, r4, r3
  + sgts r0.w, r0.z
  add r2.z, r2.z, r4.w

  // half size of quarter-level difference
  + mulsc r1.z, c1.w, r1.z

  add r2.x, r2.x, r0.z

  // Abort?  Will abort if fetch points to (0, 0)
  + setp_eq r0._, r0.z

  // abortion jump
   (p0) jmp L1

  // point to selected node in texture
  mad r2.x, r2.x, r0.w, r3.w

///////////////////////////////////////////////////
// to 4x4
///////////////////////////////////////////////////

  // fetch coefficient & index
  tfetch1D r4.xyzw, r2.x, tf0, MagFilter=point, MinFilter=point,
       UnnormalizedTextureCoords=true, FetchValidOnly=false // from 8:8:8:8

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  // calculate new half-level coordinates
  mad r1.xy, r3, r1.z, r1.xy

  serialize

  // scale the index to the correct range, and round
  mad r0.z, r4.w, c2.z, c2.w

  // per-level rescaling
  mul r4.xyz, r4, c8.y
  + floors r0.z, r0.z

  // add contribution
  dp3 r4.w, r4, r3
  + sgts r0.w, r0.z
  add r2.z, r2.z, r4.w

  // half size of quarter-level difference
  + mulsc r1.z, c1.w, r1.z

  add r2.x, r2.x, r0.z

  // Abort?  Will abort if fetch points to (0, 0)
  + setp_eq r0._, r0.z

  // abortion jump
   (p0) jmp L1

  // point to selected node in texture
       mad r2.x, r2.x, r0.w, r3.w


///////////////////////////////////////////////////
// to 2x2
///////////////////////////////////////////////////

  // fetch coefficient & index
  tfetch1D r4.xyzw, r2.x, tf0, MagFilter=point, MinFilter=point,
      UnnormalizedTextureCoords=true, FetchValidOnly=false // from 8:8:8:8

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  // calculate new half-level coordinates
  mad r1.xy, r3, r1.z, r1.xy

  serialize

  // scale the index to the correct range, and round
  mad r0.z, r4.w, c2.z, c2.w

  // per-level rescaling
  mul r4.xyz, r4, c8.z
  + floors r0.z, r0.z

  // add contribution
  dp3 r4.w, r4, r3
  + sgts r0.w, r0.z
  add r2.z, r2.z, r4.w

  // half size of quarter-level difference
  + mulsc r1.z, c1.w, r1.z

  add r2.x, r2.x, r0.z

  // Abort?  Will abort if fetch points to (0, 0)
  + setp_eq r0._, r0.z

  // abortion jump
   (p0) jmp L1

  // point to selected node in texture
  mad r2.x, r2.x, r0.w, r3.w
///////////////////////////////////////////////////
// to 1x1
///////////////////////////////////////////////////

  // fetch coefficient
  tfetch1D r4.xyz_, r2.x, tf0, MagFilter=point, MinFilter=point,
      UnnormalizedTextureCoords=true, FetchValidOnly=false // from 8:8:8:8
  // don't need to fetch index

  // find out which quadrant we're in, and build the basis
  sge r3.xy, r0.xy, r1.xy
  sgt r3.zw, r3.xyxy, r3.yxyx
  dp3 r3.w, r3.xyy, r3.xyy
  + adds r3.z, r3.zw
  mad r3.xyz, r3, c2.x, c2.y

  serialize

  // per-level rescaling
  mul r4.xyz, r4, c8.w

  // add contribution
  dp3 r4.w, r4, r3
  add r2.z, r2.z, r4.w


///////////////////////////////////////////////////
        label L1
///////////////////////////////////////////////////
  alloc colors
  exece

  mov oC0.xyz, r2.z
  + movs oC0.w, c0.x
```

Listing A1. *Xbox 360™ microcode shader for texture decompression.*

```
  xps_3_0
  config AutoSerialize=false

  def c0, 16777215.0, 0.5, 1.0, 2.0
  def c1, 0.0002, 256.0, 65535.0, 0.0
  def c2, 1.386294361, 0.0, 0.0, 0.0
  def c3, 127.0, 127.5, 0.0, 0.0
  def c4, 0.8192, 0.00390625, -254.0, 4.0

  dcl_texcoord0 r0.xy
  dcl_texcoord1 r1.xyz


  // this is so that if "FetchValidOnly" says don't fetch, we're pointing at the root
  // node
  mov r1.xy, r0
  + movs r0.x, c1.w

  // fetch transport function offset for this texel
  tfetch2D r0.x1__, r1.xy, tf0, MinFilter=point, MagFilter=point,
      FetchValidOnly=true
  serialize

  // is it an invalid pixel?
  sgts r5.x, r0.x

  // rescale to unnormalized integer
  mad r0.x, r0.x, c0.x, c0.y
  floor r0.x, r0.x

  // fetch base scale from transport & environment texture
  tfetch1D r1.yx__, r0.x, tf1, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
  tfetch1D r1.__yx, r0.y, tf2, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
tfetch1D r0.___w, r0.y, tf2, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
  serialize


  // rescale and multiply base scales
  mad r1, r1, c0.y, c0.y
  mad r1.xy, r1.yw, c4.y, r1.xz
  mul r1.w, r1.x, r1.y


  // move on to root nodes
  add r0.xy, r0, c0.z


/////////////////////////////////////////////////////////////////////////////
  label L0

  // fetch current nodes
  tfetch1D r2.xyzw, r0.x, tf1, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true
  tfetch1D r3.xyzw, r0.y, tf2, MinFilter=point, MagFilter=point,
     UnnormalizedTextureCoords=true, FetchValidOnly=true

  // calculate current level scalar
  mad r0.z, r0.w, c4.z, c0.w
  mul r0.z, r0.z, c0.y
  exp r0.z, r0.z

  serialize


  add r0.y r0, c0.z

  // scale by level scalar
```

```
mul r2.xyz, r2, r0.z
mul r3.xyz, r3, r0.z

add_sat r4.x, r2.w, -r0.w
add_sat r4.y, r3.w, -r0.w
+ movs r0.w, r2.w


// multiply and add to accumulating integral value
dp3 r1.z, r2, r3
+ muls r4.z, r4.xy
add r1.w, r1.w, r1.z
+ setp_gt r4._, r4.z
subs r4.w, r4.xy

// Now we need to check if we agree on the next node: if the first tree says the next
//  node is a child, but the second says a sibling or
//  a parent, we need to jump over the child of the first tree (and vice versa).

// case 1: next node in tree 1 is child, next node in tree 2 is child
//  here we need to carry on to the child, but omit both "jump" nodes
 (p0) add r0.xy, r0, c0.z


setp_gt r4._, r4.w


// case 2: next node in tree 1 is a child, next node in tree 2 is not
//  here we need to use the "jump" node to omit the branch starting at the first child
//  for tree 1
 (p0) exec
 (p0) tfetch1D r2.xyz_, r0.x, tf1, MinFilter=point, MagFilter=point,
   UnnormalizedTextureCoords=true, FetchValidOnly=true, OffsetX=1.0
serialize
 (p0) mad r2.xy, r2, c3.x, c3.y
 (p0) floor r2.xy, r2
 (p0) mad r2.x, r2.y, c1.y, r2.x
 (p0) add r0.x, r0.x, r2.x
+ (p0) movs r0.w, r3.w

exec
setp_gt r4._, -r4.w

// case 3: next node in tree 1 is not a child, next node in tree 2 is
//  here we need to use the "jump" node to omit the branch starting at the first child
//  for tree 2
 (p0) exec
 (p0) tfetch1D r3.xyz_, r0.y, tf2, MinFilter=point, MagFilter=point,
   UnnormalizedTextureCoords=true, FetchValidOnly=true, OffsetX=1.0
serialize
 (p0) mad r3.xy, r3, c3.x, c3.y
 (p0) floor r3.xy, r3
 (p0) mad r3.x, r3.y, c1.y, r3.x
 (p0) add r0.y, r0.y, r3.x

exec


// If we're at the end node, try to jump out of the loop.  Of course, we can only do
//  this if all 64 pixels in the vector have reached
//  their end node.  So if we can't jump out we need to perform an 'idle' loop until
//  the whole vector is finished.
mul r2.w, r2.w, r3.w
mul r2.w, r2.w, r5.x

setp_eq r2._, r2.w
 (p0) jmp L1

// move onto the next node, provided we're not at the root
 (!p0) add r0.xy, r0, c0.z
```

```
  (p0) mul r0.xy, r0, c1.w

jmp L0
/////////////////////////////////////////////////////////////////////////////

label L1

alloc colors
exece

mul oC0.xyz, r1.w, c255.x
+ movs oC0.w, c0.z
```

Listing A2. *Xbox 360™ microcode shader for double product integration.*

4.8 Acknowledgements

I'd like to thank Ash Henstock for helping me make figure 7. Additionally Kieran Connell and Tom Grove for kindly permitting the use of their photograph in figure 5.

4.9 References

[Hu08] HU, YAOHUA. 2008. *Lightmap compression in Halo 3*. Presentation, Game Developer Conference, San Francisco, CA, February 2008

[GHB*05] GORSKI, K. M., HIVON, E., BANDAY, A. J., WANDELT, B. D., HANSEN, F. K., REINECKE, M., AND BARTELMANN, M. 2005.. *HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere*. The Astrophysical Journal, 622:759-771, April 2005.

[SDS95] STOLLNITZ, E. J., DEROSE, T.D. AND SALESIN, D.H. 1995. *Wavelets for computer graphics: A primer (part 1)*, IEEE Computer Graphics and Applications, Vol. 15, pp. 76-84.

[DAUBECHIES92] DAUBECHIES, I. 1992. *Ten lectures on wavelets*. SIAM publishing, 978-0898712742.

[NRH03] NG, R., RAMAMOORTHI, R. AND HANRAHAN, P. 2003. *All-frequency shadows using non-linear wavelet lighting approximation.* ACM transactions on graphics (Siggraph 2003 Proceedings), pp. 376-381, San Diego, CA.

[NRH04] NG, R., RAMAMOORTHI, R. AND HANRAHAN, P. 2004. *Triple-product wavelet integrals for all-frequency relighting.* ACM transactions on graphics (Siggraph 2004 Proceedings), pp. 477-487, Los Angeles, CA, August 2004.

[HLS07] HORMANN, K. , LÉVY, B. AND SHEFFER, A. 2007. *Mesh parameterization: Theory and practice.* ACM SIGGRAPH Course 27 course notes, San Diego, CA, August 2007.

[DCH05] DIVERDI, S., CANDUSSI, N. AND HÖLLERER, T. 2005. *Real-time rendering with wavelet-compressed multi-dimensional datasets on the GPU,* Technical Report 2005-05, UCSB.

[MHL*06] MA, W.-C., HSIAO, C.-T., LEE, K.-Y., CHUANG, Y.-Y. AND CHEN, B.-Y. 2006. *Real-time triple product relighting using spherical local-frame parameterization.* The Visual Computer, Vol. 22, No 9-11, pp. 682-692. .

[WANWONG07] WAN, L. AND WONG, T.-T. 2007. *Sphere maps with the near-equal solid angle property.* Presentation, Game Developer Conference (GDC2007), San Francisco, CA, March 2007. http://www.cse.cuhk.edu.hk/~ttwong/papers/spheremap/spheremap.html

[SCHRÖDERSWELDENS95] SCHRÖDER, P. AND SWELDENS, W. 1995. *Spherical wavelets: Efficiently representing functions on the sphere.* In Proceedings of SIGGRAPH 1995, ACM Transactions on Graphics, pp. 161-172, Los Angeles, CA, August 1995.

[SLOAN08] SLOAN, P.-P.. 2008. *Stupid spherical harmonics tricks.* Presentation, Game Developer Conference (GDC2008), San Francisco, CA, February 2008. http://www.ppsloan.org/publications/StupidSH35.pdf

[FUJITAKANAI04] FUJITA, M. AND KANAI, T. 2004. *Precomputed radiance transfer with spatially-varying lighting effects.* In Proceedings of Proceedings of the International Conference on Computer Graphics, Imaging and Visualization (CGIV 2004), pp. 101-108..

[RAMAMOORTHIHANRAHAN01] RAMAMOORTHI, R. AND HANRAHAN, P. 2001. *An efficient representation for irrandiance environment maps.* ACM transactions on graphics (Siggraph 2001 Proceedings), pp. 497 – 500, Los Angeles, CA, August 2001.