The Visual Computer manuscript No.
(will be inserted by the editor)

Antonio Garcia · **Han-Wei Shen**

GPU-based 3D Wavelet Reconstruction with Tileboarding

Abstract In this paper, we present a GPU-based algorithm for reconstructing 3D wavelets using fragment programs. To minimize the data transfer and fragment processing overhead, we propose a novel scheme which uses tileboards as a primary layout to organize 3D wavelet coefficients. By accessing the tileboards with correct texture coordinates, Haar and Daubechies wavelets can be evaluated by the GPU in real time. The tileboard also serves as input to the rendering programs. We demonstrate how the tileboards allow us to efficiently cull unnecessary data, and we extend our work to render large volumes with multiple resolution levels.

Keywords Tileboards · Wavelets

1 Introduction

Natural phenomena are very difficult to represent with regular geometry; therefore, they are sampled into grids with high-quality devices, such as CT scans. Grid resolutions are typically high, which results in huge demands of storage. To reduce the storage requirements, wavelets can be used for data compression and multiresolution analysis. In fact, wavelets have been widely adopted in computer graphics and visualization, especially in areas such as image processing and volume rendering where explorations of large datasets have become commonplace.

In general, wavelet compressed data need to be reconstructed before further processed by application-dependent software such as 3D volume rendering. Although software-based wavelet reconstruction can be sufficient for simple types of wavelets, such as Haar, it is expensive to compute for types that achieve higher compression ratios and better image quality, such as Daubechies. As programmable graphics processing units (GPUs) become increasingly suitable for general purpose computation, studies to reconstruct wavelets using graphics hardware have been done [7][8][21]. However, they mostly focused on 2D images, and the limitations of the available graphics hardware have not yet made hardware reconstruction competitive against the software counterpart.

With the advent of programmable graphics hardware and high-level shading languages, 3D wavelet reconstruction can now be fully realized faster and easier than before. High-precision texture formats with full floating-point pipelines and pbuffers for off-screen rendering give the necessary ingredients to implement robust fragment programs capable of carrying out the tasks for wavelet reconstruction. In this paper, we propose an interactive algorithm that performs 3D wavelets reconstruction using GPUs by mapping textures that contain wavelet coefficients and by evaluating wavelet reconstruction formulae at each fragment. Although our rendering primitives are quadrilaterals, which are 2D entities, they are the means to reconstruct a brick of voxels, which is a 3D entity. Since it is necessary to be able to access elements at different locations inside the brick, we propose the concept of tileboarding, which flattens a brick of voxels, such that each slice of the brick becomes a tile. The tileboard makes an efficient entity in terms of texture management and it proves to be highly flexible for rendering schemes that produce 3D texture coordinates [10][12]. With this approach, we depart from previous approaches that compute the reconstruction through convolution operators and the framebuffer. Furthermore, we incorporate this building block into a multiresolution hierarchy to render a wavelet octree [5].

The paper is organized as follows: section 2 reviews the previous work for wavelets and GPU programming; section 3 reviews the Haar and Daubechies wavelets; section 4 describes our GPU-based reconstruction algorithm, which includes the concept of tileboarding and its application during

Antonio Garcia
Ohio State University
395 Dreese Laboratories, 2015 Neil Av, Columbus-OH, 43210, USA
Tel.: +1-614-292-5813
Fax: +1-614-292-2911
E-mail: agarcia@cse.ohio-state.edu

Han-Wei Shen
Ohio State University
395 Dreese Laboratories, 2015 Neil Av, Columbus-OH, 43210, USA
Tel.: +1-614-292-0060
Fax: +1-614-292-2911
E-mail: hwshen@cse.ohio-state.edu

the stages of the algorithm; section 5 presents our results; and finally, section 6 summarizes with our conclusions and possibilities for future work.

2 Previous Work

Volume rendering applications typically involve large amounts of data; therefore, compression and multiresolution schemes that allow for visualization of combined levels-of-detail are attractive tools. With the support of 3D textures and its application for volume rendering [4], many visualization algorithms have been proposed, so that they avoid decompression [15][18] and fetch bricks from multiresolution octrees based on viewing metrics [11], all in an effort to accelerate rendering speed.

Wavelet theory provides a robust foundation for filter banks [19][20], which are important for multiresolution analysis. In fact, Westermann describes a multiresolution framework using wavelets for volume rendering [22]. Other authors have also exploited features of the decomposition to render wavelets fast, such as runs of zeros [9][16]. Among the different types of wavelets, the haar wavelet [1] and the Daubechies wavelet [2] are typical in studies. The first because it is very fast to calculate, and the second because it keeps details better albeit at a cost of slower reconstruction. The literature presents the reconstruction of both wavelets mostly in software. Guther et.al. [5] go one step further with wavelets and deploy them into an elegant multiresolution octree, but the reconstruction is still done in software.

Hardware implementations of wavelet transformations have mainly focused on images rather than volumes. Since wavelets can be regarded as combinations of down-sampling, up-sampling and filtering operations, Hopf et.al. proceed to solve them by applying convolution operators using OpenGL extensions [6][7][8]. As it will be explained, we organize the data in a 2D layout that allows the transformation of 3D texture coordinates into 2D counterparts [10][12] and then apply reconstruction formulae that follow the tensor product given by Muraki [14]. Wang et.al. follow Muraki's approach to transform wavelets on the GPU for RGB images [21].

With GPUs, the range of applications for computer graphics has widened, and complex operations at vertex and fragment levels are now available [3]. Shading languages complete the package to simplify much of the programming obstacles that one faced with fixed pipelines and assembly code [13][17]. In our work, we apply them to evaluate the reconstruction formulae.

3 Wavelets

Before presenting our GPU-based wavelet reconstruction algorithm, in this section we briefly overview the elements from wavelet theory that are relevant to the various stages of our process. Wavelets are defined on the basis functions that filter a set of original values (hereafter referred to as A values) into two parts: the averages or low-frequency coefficients (L values) and the details or high-frequency coefficients (H values). The process can be repeated on the L values to create a hierarchy of resolutions [20]. If the data has multiple dimensions, wavelets are applied successively on each dimension. Figure 1 shows the different kind of coefficients that result from decomposing the input in 1, 2 and 3 dimensions.

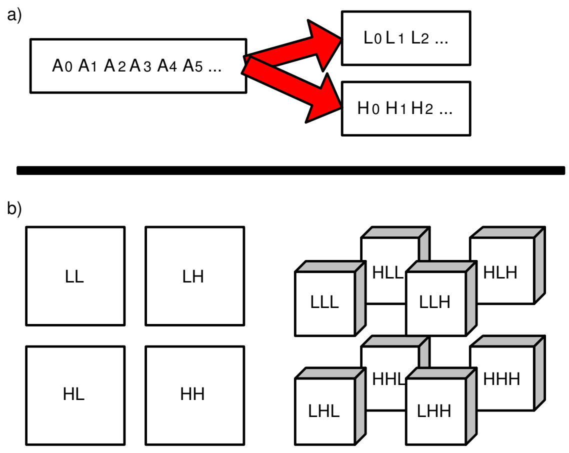


Fig. 1 Wavelet Decomposition in 1, 2 and 3 Dimensions. In the one dimensional case (a), the original A values are divided into the Low and High coefficients. In the case of multiple dimensions (b), successive transformations produce combinations of Low and High coefficients. The input for our implementations has the format of the 3D case.

Since there are many choices for the basis functions, there are many ways to reconstruct the values. We have chosen, without loss of generality, the two that are mostly used: one for its simplicity and the other for its detail compactness.

3.1 Haar

Haar wavelets are the simplest to compute, but unfortunately they suffer a larger penalty on image quality when dropping high-frequency coefficients, which happens in multiresolution environments or time critical conditions. Haar's reconstruction formulae are given in the following equations:

$$A_{2i} = \frac{L_i}{\sqrt{2}} + \frac{H_i}{\sqrt{2}} \quad (1)$$

$$A_{2i+1} = \frac{L_i}{\sqrt{2}} - \frac{H_i}{\sqrt{2}} \quad (2)$$

3.2 Daubechies

Daubechies's wavelets are more expensive to compute, but they keep better image quality. As opposed to Haar wavelets, quantizing them into integer pipelines results in too much precision loss, which is problematic for solutions that attempt to use the framebuffer as the reconstruction target. Daubechies's reconstruction formulae are the following:

$$A_{2i} = h_0 L_i + h_2 L_{i-1} + h_3 H_i + h_1 H_{i-1} \quad (3)$$

$$A_{2i+1} = h_1 L_i + h_3 L_{i-1} - h_2 H_i - h_0 H_{i-1} \quad (4)$$

where

$$h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}},\ h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}$$
$$h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}},\ h_3 = \frac{1-\sqrt{3}}{4\sqrt{2}}$$

The presence of L_{i-1} and H_{i-1} requires special handling at boundaries. This can be dealt with by zeroing, mirroring or replication [2]. In our implementations, we choose the latest, because it is known to reduce reconstruction artifacts. Finally, these formulae are part of the known Daubechies-4 (db4) wavelet. Other kinds, such as Daubechies-6 (db6) or Daubechies-12 (db12), follow the same reconstruction logic as db4, but they require more coefficients to reconstruct, which would imply more texture lookups as we will see in the following section.

The equations given in this section will be evaluated inside a fragment program, which based on texture coordinates, determines between the equations for A_{2i} and A_{2i+1}. Furthermore, the data is three-dimensional, therefore, the formulae will be evaluated three times.

4 GPU Reconstruction

The input to a reconstruction algorithm typically comes as a series of 3D bricks that represent different kinds of wavelet coefficients (L and H values). A standard approach would fetch them from disk, arrange them into a convenient layout and, by using software, reconstruct into the original data (the A values). The reconstructed volumes are later uploaded into texture memory and rendered. This software-based approach incurs high computation overhead when used together with interactive applications such as texture-based 3D volume rendering, because there is a lot of memory copying and reordering. To minimize the overhead, our algorithm moves the reconstruction stage into the GPU, where the 3D L and H coefficients are uploaded into texture memory. By rendering quadrilaterals bound to those textures with a specialized fragment program, the A values can be computed and stored into an intermediate texture that is directly accessible by the rendering fragment program without the copying and the reordering.

Assume each coefficient brick is a three-dimensional entity that has d elements along each dimension, making it a d^3 brick of voxels. As shown in figure 1, there will be 8 bricks of coefficients, LLL, LLH, LHL, LHH, HLL, HLH, HHL and HHH, coming from the wavelet transform of the original $(2d)^3$ volume. To reconstruct the A values from those bricks, a simple GPU solution is to upload into 3D textures two bricks at a time that represent the corresponding low-frequency and high-frequency coefficients in the z direction (those pairs with mismatching H and L in the left-most letter, that is, LLL and HLL, LLH and HLH, LHL and HHL, and LHH and HHH). Then, we render quadrilaterals using the same z slice from each of the two coefficient bricks as textures and combine the values together using equations 1 and 2 (for Haar), or equations 3 and 4 (for Daubechies) to produce the z-reconstructed intermediate values. The values are copied into another texture. When all z slices are processed, we have 4 new bricks of $d^2 \times 2d$. The process repeats for the y-dimension and later for the x-dimension for which the quadrilaterals must be aligned accordingly, producing a reconstructed brick of $2d \times 2d \times 2d$ original values. While straightforward, this solution has the overhead of texture copying and involves too many reconstruction passes. In a sense, it will be no different from what previous approaches have suggested [7].

To minimize the reconstruction overhead, we present a new approach that recognizes the fact that the initial 8 bricks can be compacted into 2 bricks with 4-value elements, and stored in floating-point RGBA textures, so that texture lookups are reduced. However, we use 2D textures to represent the compacted bricks and this is done by flattening the two 3D compacted RGBA bricks into two 2D RGBA tileboards, where each tile in the tileboard represents a slice in the 3D brick. The reconstruction formulae are applied to these RGBA tileboards to produce the reconstructed tileboard, which is the input for the rendering. The tileboard layout enable us to use off-screen rendering that keeps the precision and avoids texture copying and/or reordering. Even though we have a 2D tileboard, we are still able to perform 3D volume rendering with view-aligned slicing by transforming 3D texture coordinates into 2D counterparts. Figure 2 shows the whole reconstruction process. The blue boxes represent fragment programs that will be explained in the following sections.

4.1 Tileboards

A tileboard is our primary data structure inside the GPU and it represents a 3D brick that has been flattened into a 2D layout. The tileboards are implemented with 2D textures, and depending on the data sources, they are referred to as either input tileboards or output tileboards. The input tileboards hold the L and H coefficients and are implemented with the OpenGL function *glTexImage2D*. The output tileboards, on the other hand, hold the reconstructed values and are implemented with *pbuffers*[3]. The dimensions of a tileboard (T_W, T_H) and the number of tiles in each dimension (N_x, N_y) are derived from the dimensions of a brick with d^3 coefficients as follows:

$$N_x = 2^{\lceil log_2 \sqrt{d} \rceil},\ N_y = 2^{\lfloor log_2 \sqrt{d} \rfloor}$$
$$T_W = d \times N_x, \quad T_H = d \times N_y$$

Since each brick has d slices, the tileboard has d tiles and they are arranged into a board of $N_x \times N_y$ tiles, where each tile has $d \times d$ elements. We will keep in mind that the dimensions and number of tiles of a reconstructed tileboard

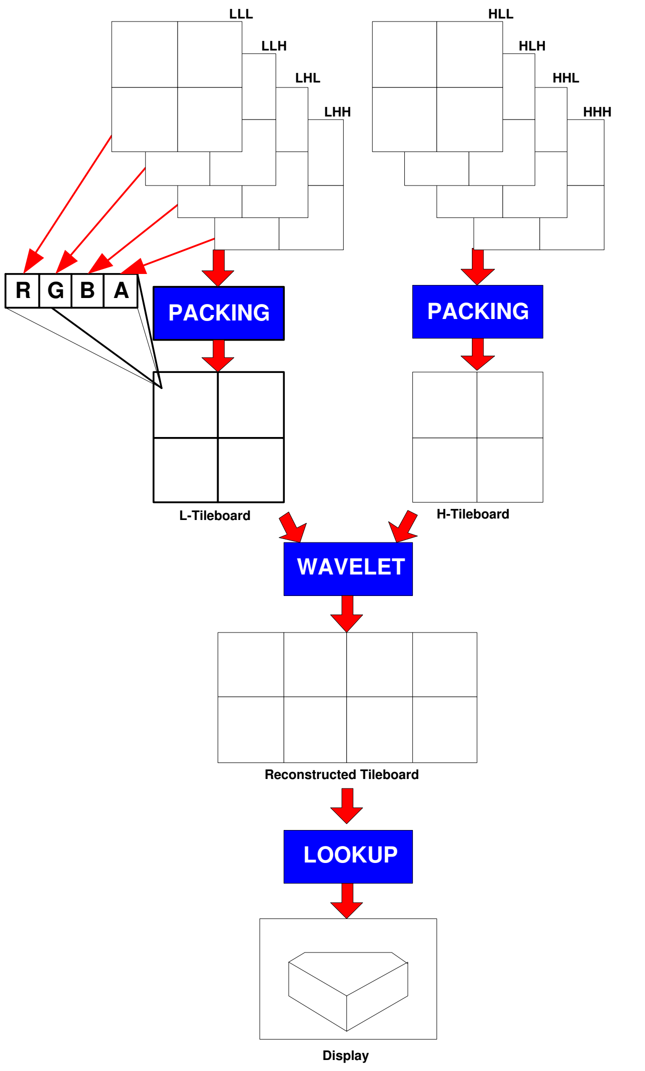


Fig. 2 An overview of the reconstruction process. Each coefficient brick, after being retrieved from disk, is uploaded into a separate tileboard; then the 8 tileboards are separated into 2 groups. Each group is packed into an RGBA tileboard such that each color channel corresponds to one of the individual tileboards of the group. The 2 new tileboards (L- and H-tileboard) serve as input for the reconstruction stage, which generates a tileboard with double the number of tiles. This tileboard serves as input for rendering. When rendering, 3D texture coordinates must be converted into 2D counterparts to access the reconstructed tileboard.

are derived from a brick with $2d^3$ voxels. An example of a reconstructed tileboard is shown in figure 3.

With such a tileboard layout, all elements from the original coefficient brick are placed in one 2D texture, hence a single texture id is sufficient to bind it with fragment programs. Also, accessing a specific element is straightforward even in the case of rendering using 3D texture coordinates. Details for texture coordinate transformations will be presented in a later section.

The tileboards provide the foundation for packing, reconstruction and rendering, which are now presented.

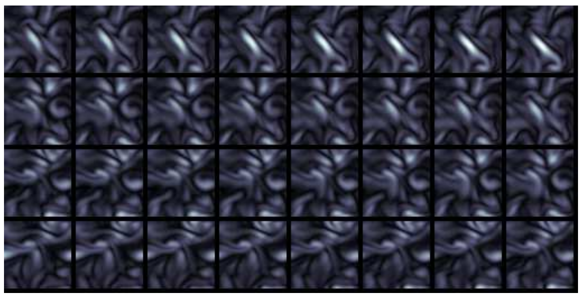


Fig. 3 A partial view of the reconstructed tileboard for the 128^3 vorticity dataset. A single tile has 128^2 reconstructed voxels, and the whole tileboard is a 2048×1024 2D texture. In order to enhance the tiles, artificial boundaries are included.

4.2 Packing

Before reconstruction, the original volume as a whole is first divided into small volume blocks of $2d \times 2d \times 2d$ voxels, each of which is then transformed using wavelets into 8 coefficient bricks of $d \times d \times d$ coefficients. The coefficient bricks will undergo some compression transformation before they are saved to disk [5].

Since multiple coefficients that are in different bricks but at the same location will be needed to evaluate the formulae presented in section 3, being able to reduce the number of texture lookups during reconstruction can effectively accelerate the calculations. In most of the graphics hardware, a single texture lookup can retrieve four color channels simultaneously with an optimized performance, so it will be beneficial if we can have four coefficients into a single RGBA texture. We exploit this property by packing our coefficient bricks into RGBA tileboards at the beginning of the reconstruction stage. Initially, we will retrieve the 3D bricks from disk and upload each one into its respective one-component tileboard. Then, using a fragment program, we look up each single-component tileboard texture and compact them into two RGBA tileboards: one tileboard will hold the L coefficients in Z, i.e., LLL, LLH, LHL, and LHH, and the other tileboard will hold the H coefficients (HLL, HLH, HHL, and HHH). In the compacted tileboard, each floating-point color channel will be used to hold one coefficient brick. For example, in the first compacted tileboard, the red channel stores the coefficients from LLL; the green channel stores the coefficients from LLH; the blue channel stores the coefficients from LHL; and the alpha channel stores the coefficients from LHH. The 2 compacted tileboards are known henceforth as the L-tileboard and the H-tileboard.

The separate uploading for each coefficient brick at the beginning of the packing process has the advantage that if one deems a given brick unnecessary, such as all coefficients being zeroes, then the uploading is eliminated. However, since the reconstruction process still needs to access those zero coefficients, a ZERO tileboard is used for binding instead of the coefficients from the discarded brick. This artificial tileboard is loaded only once with zeroes and has the same dimensions as a regular coefficient brick. Another advantage is the flexibility to incorporate methods that upload compressed textures, such as the work of Kraus et.al. [10], whose adaptive textures can easily be integrated.

With the L- and H-tileboard ready, the reconstruction stage can now proceed to evaluate the equations given in sections 3.1 and 3.2.

4.3 Reconstruction

The goal of the reconstruction stage is to transform the values stored in the L- and H-tileboard into a new tileboard that will hold the A values for rendering. A specialized fragment program will perform this task by rasterizing a quadrilateral that uses the two input tileboards to produce the new tileboard. As described in section 4.1, the output tileboard will have $2d$ tiles with each tile having a total of $2d \times 2d$ voxels, and the input tileboard has d tiles with $d \times d$ coefficients.

In our GPU program, when rasterizing the quadrilateral, each fragment will receive texture coordinates $(c_x,\ c_y)$ that correspond to a location in the output tileboard, where A values are written out; therefore, $(c_x,\ c_y)$ vary from (0, 0) to $(T_W,\ T_H)$, which are the dimensions of the output tileboard. In order to retrieve the coefficients from the input tileboards, since the size of the textures are different, it is necessary to transform the texture coordinates into the coordinate space of the input tileboards before we can reconstruct. Furthermore, it is also necessary to determine which equations to use for the reconstruction in each dimension.

First, given the texture coordinates $(c_x,\ c_y)$, we need to determine which tiles i and j this fragment corresponds to in the input and output tileboards respectively, because all calculations are performed inside their dimensions. We calculate i and j as follows

$$\begin{aligned} j &= \lfloor c_x/2d \rfloor + \lfloor c_y/2d \rfloor N_x \\ i &= \lfloor j/2 \rfloor \end{aligned}$$

where N_x is the number of tiles in the x dimension of the output tileboard. We also determine $(c_x,\ c_y)$'s relative position $(cj_x,\ cj_y)$ inside the tile j by:

$$\begin{aligned} cj_x &= c_x\%(2d) \\ cj_y &= c_y\%(2d) \end{aligned}$$

where % is the mod operator. In essence, the tiles in each tileboard are numbered linearly even though conceptually the tileboard has a two-dimensional layout. The 3D coordinates $(cj_x,\ cjy,\ j)$ decide the reconstruction formulae to apply in each dimension. If a value for a corresponding dimension is even, we use A_{2i}'s formula; otherwise, we use A_{2i+1}'s formula.

Finally, to retrieve the coefficients from the input tileboards and complete the formulae, we need to find the location $(ci_x,\ ci_y)$ that is inside tile i. We proceed to find i's beginning $(o_x,\ o_y)$ and then offset by half of $(cj_x,\ cj_y)$. The calculations proceed as follows using N_x of the input tileboards:

$$\begin{aligned} o_x &= (i\%N_x) \times d \\ o_y &= \lfloor i/N_x \rfloor \times d \\ ci_x &= o_x + \lfloor cj_x/2 \rfloor \\ ci_y &= o_y + \lfloor cj_y/2 \rfloor \end{aligned}$$

With all locations calculated, we reconstruct according to the type of wavelet. In the case of Haar, looking up both the L- and H-tileboard with the same texture coordinates $(ci_x,\ ci_y)$ fetches 8 coefficients (2 RGBA values). The j index decides between equations 1 and 2 to apply on the 2 RGBA tuples, thus reconstructing in the z-dimension and leaving a new RGBA tuple. cj_y decides the reconstruction of two more values based on this new tuple: one combining its red and blue channel, and the other combining its green and alpha channel. Finally, cj_x decides how to reconstruct the remaining two values into an A value for rendering. In the case of Daubechies, the calculations are not so straightforward, but it follows the same logic. The 8 values fetched are insufficient to evaluate equations 3 or 4 because of the L_{i-1} and H_{i-1} terms. Therefore, we need an additional coordinate, which we derive from $i-1$. In the case that $i=0$, then we derived it from $d-1$. This completes the requirements but only for the Z dimension. For the sake of simplicity, we perform a separate pass to reconstruct the XY dimension, which uses the output of the first pass. For every position in this second pass, 4 lookups with 4 values each are performed, yielding the 16 coefficients to reconstruct an A value. The 0 condition in the XY pass is enforced inside of a tile, moving to the last row or last column as needed.

4.4 Tileboard Rendering

The last stage performs the traditional 3D volume rendering algorithm [4] on a reconstructed tileboard. The volume is sliced by a set of polygons that are composited back-to-front and clipped to the boundaries of the volume. However, the tileboard is a 2D texture that represents a 3D object; therefore, a fragment program must convert the texture coordinates before lookup of the value can take place. The following does the conversion from 3D coordinates $(c3_x,\ c3_y,\ c3_z)$ to 2D coordinates $(c2_x,\ c2_y)$ using N_x of the reconstructed tileboard:

$$\begin{aligned} c2_x &= (\lfloor c3_z \rfloor \% N_x) \times 2d + c3_x \\ c2_y &= (\lfloor c3_z \rfloor / N_x) \times 2d + c3_y \end{aligned}$$

The integer part of the z-component is decomposed into the indices that indicate the row and the column of the target tile in the reconstructed tileboard. Multiplying the indices by the tile resolution ($2d$) results in the target tile's beginning. Finally, the xy-components are added to form the 2D coordinate that accesses the tileboard. This emulates the option *GL_NEAR*. To emulate *GL_LINEAR*, linear filtering is turned on for the reconstructed tileboard, and a second coordinate is calculated by using the xy-components on tile $\lfloor c3_z \rfloor + 1$. After performing both bilinear lookups, we interpolate between the retrieved values with the fractional

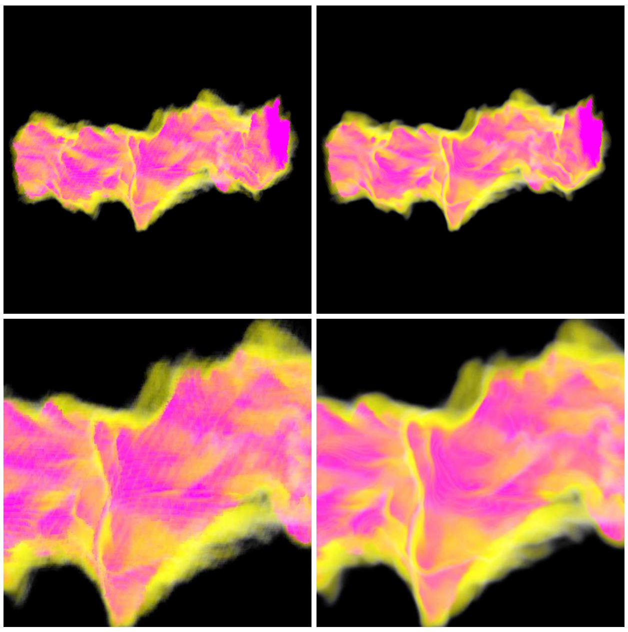


Fig. 4 Image samples of the Shockwave dataset. Left column shows the rendering of the reconstructed tileboard using nearest neighbor interpolation. Right column shows simulated trilinear interpolation.

Brick Size (voxels)	GPU Haar Total (Up/Rec) (msec)	GPU Daub Total (Up/Rec) (msec)
128^3	102.89 (44.12/15.24)	124.61 (43.89/37.34)
64^3	33.28 (5.78/13.20)	33.44 (5.64/16.48)
32^3	16.64 (0.94/0.12)	16.80 (0.93/0.12)
16^3	16.64 (0.12/0.08)	16.72 (0.10/0.08)

Table 1 Total rendering timings for single varying brick size under Haar and Daubechies GPU reconstruction using the vorticity dataset. The numbers enclosed in parenthesis are the timings of the uploading and the reconstruction stages. The window size in these testings was 256^2 pixels.

Brick Size (voxels)	256^2 (msec)	512^2 (msec)	1024^2 (msec)
128^3	43.20	121.25	419.85
64^3	16.80	64.22	210.47
32^3	16.56	63.28	209.12
16^3	16.64	61.87	210.33

Table 2 Timings of the rendering algorithm under varying window size using the vorticity dataset. The largest tileboard tested takes a performance hit that is in average twice that of the other tileboards.

part of the z-component. Figure 4 shows *GL_NEAR* and *GL_LINEAR* emulation for the shockwave dataset.

With all stages described, we present our results for both single bricks and multiresolution hierarchies in the next section.

5 Results

Our datasets, either consisting of a single brick or multiresolution hierarchies, were transformed into the wavelet coefficients with an off-line decomposition program. For single brick testing, the following 128^3 volumes were used: the **vorticity** dataset, which contains 8-bit integer voxels; and the **shockwave** dataset, which has 8-bit voxels. For multiresolution testing, the dataset was the **visible woman** dataset, but we extracted a 480^3 subvolume and converted the 12-bit values to floating-point values.

All our testings were performed on a 3.0GHz Intel Xeon processor with 3GB main memory, and an nVidia Quadro FX 3400 graphics card with 256MB video memory. The fragment programs were written using NVIDIA's Cg language [13] and, as mentioned in section 4.3, Daubechies is done in two passes: one for the Z-dimension and the other for the XY-dimensions. Finally, our software implementation followed the process described in section 3. Tileboards were implemented with 32-bit floating-point channels and NVIDIA's *GL_TEXTURE_RECTANGLE_NV*. However, reconstructed tileboards used 8-bit channels when interpolation was turned on, because floating-point interpolation is currently not supported for this card.

First, we discuss our findings for single bricks. We registered the average timings of an animation that rotates the brick with all stages turned on, which means that for every frame we uploaded, reconstructed and rendered. As expected, the Haar wavelets are faster to reconstruct than the Daubechies ones, but we also noticed that certain brick sizes made a difference for the packing and rendering stages. From table 1, we see that the uploading stage scales as expected with brick size. The reconstruction stage experiences a sudden increase when processing bricks of 16^3 and 64^3 voxels. This may be related to the parallel execution of the fragment program and how texture memory is organized. When the hardware is rendering, we see that brick sizes that are greater than 64^3 voxels experience overhead.

In table 2, we show the performance of the rendering stage alone with varying window size. The packing and reconstruction was done only for the first frame and the results were reused for the rest of the animation. We increased the window size to generate more rendering fragments, and we see that for all but the largest brick, the transformation from 3D textures coordinates to 2D counterparts is transparent to the user and scales accordingly. The 128^3 brick size makes a 2048×1024 texture that is well within the maximum texture size of 4096, but the graphics hardware makes more effort to access the texels, which may be related to cache coherency and texture memory organization. From these findings, we choose the 64^3 brick size to decompose big volumes into multiresolution hierarchies.

We decomposed the **visible woman** dataset into 64^3 overlapping bricks and generated 5 levels of detail with 585 reconstruction nodes. Since it is our intention to see the performance of the reconstruction, we rendered all nodes, disregarding viewing metrics and empty spaces. Figure 5 shows sample images at different levels of the reconstruction. Tables 3 and table 4 show the performance under Haar and

Daubechies respectively. In these tests, the time to retrieve all bricks from the disk fluctuated between 1 and 0.80 seconds. We compared against our software implementation that uses 3D textures for rendering, and we immediately see the difference. Furthermore, the packing stage shows its power when we start to discard high frequency coefficients, which in a multiresolution scenario is very likely to happen. From table 3 and 4, we see that if we systematically drop coefficient bricks, it affects the timings of the reconstruction, which includes uploading and packing. The timing of the rendering stage remains the same since the window size is kept at 512^2 pixels. The numbered lines indicate that we are reconstructing with that many bricks, 8 bricks for full reconstruction and 1 brick for no uploading. Of course, the manipulation of the bricks comes at a visual cost. Figure 6 shows the gradual loss of quality when discarding high-frequency coefficient bricks for the **vorticity** dataset. Daubechies is able to keep the features of the data better than Haar, which reconstructs most of the features with even half the coefficient bricks. However, when Haar reconstructs with just a couple of the bricks, the known blocking pattern starts to emerge.

Software counterparts cannot take advantage of the discarding of bricks since they always upload a brick to a 3D texture with the same dimensions. Moreover, in software implementations, there is an additional step for overhead: once the parent reconstructs its brick, children extract parts of it, for which a copying is performed. Our packing stage, which does the interleaving and formatting into RGBA textures, takes care of this extraction by calculating correct texture coordinates given the child's initial offsets, so no additional copying is introduced.

Finally, the speedup between software and hardware, with full 8-brick reconstruction, is 5.83 for Haar, and 6.76 for Daubechies. This makes GPU-based 3D wavelet reconstruction competitive against software counterparts and completes the findings of this work.

Type	Total (sec)	Speedup	Rendering (sec)	Haar (sec)
Software & 3D Textures	29.40	1.00	0.29	28.11
GPU				
8	5.04	5.83	0.29	3.90
7	4.60	6.39	0.27	3.46
6	4.16	7.06	0.27	2.92
5	3.87	7.59	0.28	2.68
4	3.30	8.90	0.28	2.13
3	2.80	10.50	0.29	1.66
2	2.34	12.56	0.28	1.18
1	1.90	15.47	0.27	0.75

Table 3 Total rendering, loading and reconstruction time using Haar wavelets for the visible woman dataset and for a 512^2 window size. Full GPU reconstruction achieves a 5.83 speedup with respect to the software implementation, and, by dropping coefficient bricks, the GPU gains further speed in the reconstruction stage.

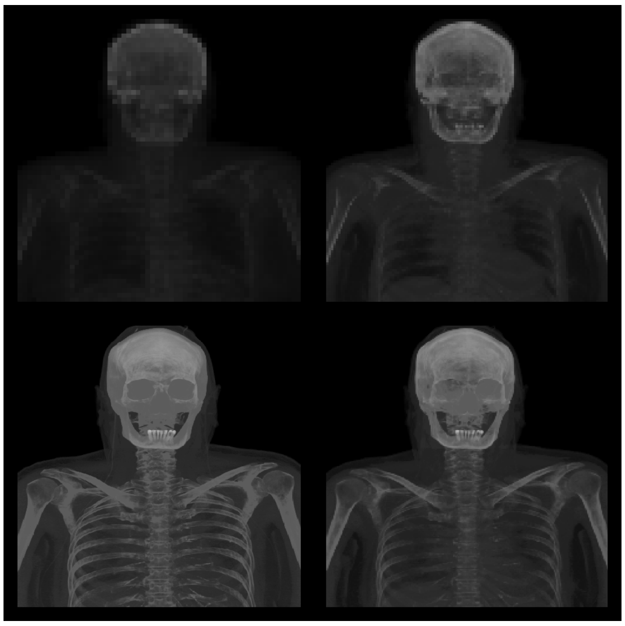


Fig. 5 A gradual reconstruction for the resolution levels of the visible woman dataset. The $480 \times 480 \times 480$ dataset is decomposed into 5 levels with 585 nodes of Haar wavelets. Starting at the upper left corner and going clockwise: level 2 (1 reconstructed node), level 3 (9 nodes), level 4 (73 nodes) and level 5 (585 nodes). As a low-resolution level is processed, the reconstructed tileboard serves as input for the next level.

Type	Total (sec)	Speedup	Rendering (sec)	Daub (sec)
Software & 3D Textures	55.31	1.00	0.29	53.79
GPU				
8	8.18	6.76	0.25	6.92
7	7.56	7.31	0.25	6.48
6	7.09	7.80	0.25	5.99
5	6.60	8.38	0.30	5.39
4	6.15	8.99	0.30	4.94
3	5.72	9.66	0.29	4.56
2	5.44	10.16	0.29	4.27
1	5.15	10.73	0.29	4.04

Table 4 Total rendering, loading and reconstruction time using Daubechies wavelets for the visible woman dataset and for a 512^2 window size. In this case, the gains are better than Haar's since software implementations are slower.

6 Conclusions & Future Work

We have devised an algorithm that can successfully utilize GPUs to reconstruct wavelet coefficients in the form of single bricks or bricks from a multiresolution hierarchy. We construct a pipeline divided into 3 stages: packing, reconstruction and rendering. All stages are accessing or rendering to tileboards that represent a 3D brick, even though, they are in fact 2D texture entities. The tileboards are directly input to our hardware-based volume renderer with fragments programs that transform 3D texture coordinates into 2D counterparts. The flexibility of the tileboard allows for rendering

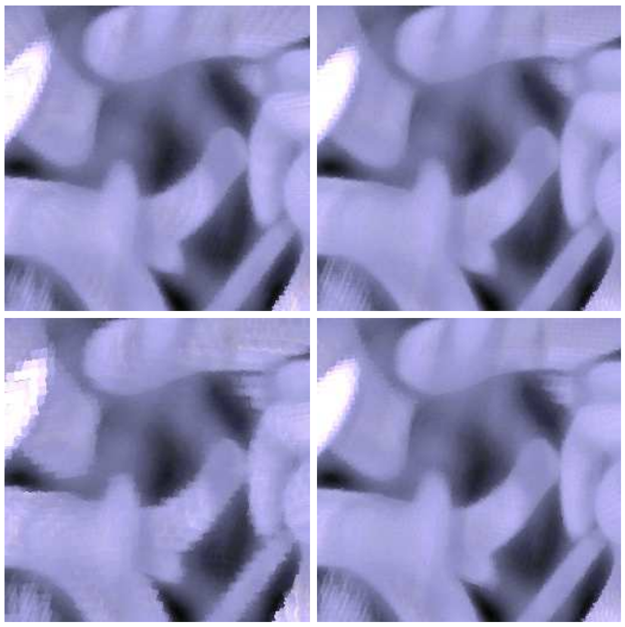


Fig. 6 A comparison between Haar (left column) and Daubechies (right column) wavelets for the vorticity dataset. Haar shows artifacts as coefficients are dropped. The images on the top have dropped the HLL, HLH, HHL and HHH coefficient bricks; while those at the bottom have dropped everything but LLL and LLH. At this point, Haar shows the traditional blocky pattern.

to textures through pbuffers, which are efficient and allow to carry out the reconstruction of wavelets.

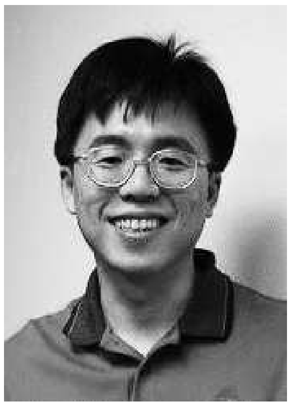
We believe that this work for 3D wavelets is unique compared to the previous approaches based on convolution. No unnecessary copying of frame buffers is done at any stage and no reordering is needed to accommodate for different dimensions.

We are able to embed our reconstruction algorithm in the context of multiresolution, and see that it provides benefits for not only reconstruction but also uploading. Although, we have not optimized our implementation, the initial results are quite encouraging. We are planning to include caching mechanisms, out-of-core algorithms and parallel distributions that can benefit the most from this experience.

References

1. Chui, C.K.: An Introduction to Wavelets. Academic Press, Inc., San Diego, CA (1992)
2. Daubechies, I.: Ten lectures on wavelets. In: SIAM. Philadelphia, PA (1992)
3. Fernando, R., Harris, M., Wloka, M., Zeller, C.: Programming graphics hardware. In: Tutorial of EUROGRAPHICS ’04 (2004)
4. Foran, J., Cabral, B., Cam, N.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In: Workshop on Volume Visualization (1994)
5. Guthe, S., Wand, M., Gonser, J., Strasser, W.: Interactive rendering of large volume data sets. In: IEEE Visualization 2002, pp. 53–60 (2002)
6. Hopf, M., Ertl, T.: Accelerating 3d convolution using graphics hardware. In: IEEE Visualization ’99, pp. 471–474 (1999)
7. Hopf, M., Ertl, T.: Hardware based wavelet transformations. In: Workshop on Vision, Modeling, and Visualization ’99, pp. 317–328 (1999)
8. Hopf, M., Ertl, T.: Hardware accelerated wavelet transformations. In: Proc. EG/IEEE TCVG Symposium on Visualization VisSym 2000, pp. 93–103 (2000)
9. Ihm, I., Park, S.: Wavelet-based 3d compression scheme for interactive visualization of very large volume data. Computer Graphics Forum **18**(1), 249–265 (1999)
10. Kraus, M., Ertl, T.: Adaptive texture maps. In: HWWS ’02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 7–15. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002)
11. Lamar, E., Hamann, B., Joy, K.: Multiresolution techniques for interactive texture-based volume visualization. In: Proceedings of Visualization ’99, pp. 355–361. IEEE Computer Society Press, Los Alamitos, CA (1999)
12. Lefohn, A., Kniss, J., Hansen, C., Whitaker, R.: Interactive deformation and visualization of level set surfaces using graphics hardware. In: IEEE Visualization 2003
13. Mark, W.R., Glanville, R.S., Akeley, K., Kilgard., M.J.: Cg: A system for programming graphics hardware in a c-like language. ACM Transactions on Graphics **22**, 896–907 (2003)
14. Muraki, S.: Approximation and rendering of volume data using wavelet transforms. In: VIS ’92: Proceedings of the 3rd conference on Visualization ’92, pp. 21–28. IEEE Computer Society Press (1992)
15. Ning, P., Hesselink, L.: Fast volume rendering of compressed data. In: Proceedings of the 4th conference on Visualization ’93, pp. 11–18 (1993)
16. Rodler, F.F.: Wavelet based 3d compression with fast random access for very large volume data. In: Proceedings of the 7th Pacific Conference on Computer Graphics and Applications, p. 108 (1999)
17. Rost, R.J.: OpenGL Shading Language. Addison-Wesley (2004)
18. Schneider, J., Westermann, R.: Compression domain volume rendering. In: Proceedings IEEE Visualization 2003 (2003)
19. Stollnitz, E.J., Derose, T.D., Salesin, D.H.: Wavelets For Computer Graphics: Theory and Applications. Morgan Kaufmann Publishers, Inc. (1996)
20. Strang, G., Nguyen, T.: Wavelets And Filter Banks, 1st edn. Wellesley-Cambridge Press (1996)
21. Wang, J., Wong, T.T., Heng, P.A., Leung, C.S.: Discrete wavelet transform on gpu. In: ACM Workshop on General Purpose Computing on Graphics Processors (2002)
22. Westermann, R.: A multiresolution framework for volume rendering. In: Proceedings of the 1994 symposium on Volume visualization, pp. 51–58. ACM Press (1994)

Antonio Garcia Antonio Garcia received the BS degree in systems engineering from Universidad Particular ”Antenor Orrego” in 1993, the MS degree in computer science from the Ohio State University in 1999. He is currently a graduate student working on his PhD dissertation at the Ohio State University. His research is focused on parallel rendering and volume rendering.



Han-Wei Shen Han-Wei Shen received the BS degree from National Taiwan University in 1988, the MS degree in computer science from the State University of New York at Stony Brook in 1992, and the PhD degree in computer science from the University of Utah in 1998. From 1996 to 1999, he was a research scientist with MRJ Technology Solutions at NASA Ames Research Center. He is currently an associative professor at The Ohio State University. His primary research interests are scientific visualization and computer graphics. In particular, his current research and publications are focused on topics in flow visualization, time-varying data visualization, isosurface extraction, volume rendering, and parallel rendering.