GPU-based Frequency Domain Volume Rendering

Ivan Viola* Armin Kanitsar†
Meister Eduard Gröller‡
Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria

Abstract

Frequency domain volume rendering (FVR) is a volume rendering technique with lower computational complexity as compared to other techniques. In this paper the FVR algorithm is accelerated by factor of 17 by mapping the rendering stage to the GPU. The overall hardware-accelerated pipeline is discussed and the changes according to previous work are pointed out. The three-dimensional transformation into frequency domain is done in a pre-processing step. The rendering step is computed completely on the GPU. First the projection slice is extracted. Four different interpolation schemes are used for resampling the slice from the data represented by a 3D texture. The extracted slice is transformed back into the spatial domain using the inverse Fast Fourier or Fast Hartley Transform. The rendering stage is implemented through shader programs running on programmable graphics hardware achieving highly interactive framerates.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.4.5 [Image Processing and Computer Vision]: Reconstruction—Transform Methods

Keywords: Fourier Volume Rendering, Fourier Transform, Hartley Transform, Hardware Acceleration

1 Introduction

Volume rendering are techniques for visualizing three-dimensional data sets. These techniques can be divided into several categories. One group uses an intermediate representation (i.e., a polygonal surface) for rendering. According to a user defined iso-value a surface is resampled from the data. Well known techniques are Marching Tetrahedra and Marching Cubes [Lorensen and Cline 1987]. The extracted iso-surface is rendered using traditional surface rendering techniques. In contrast to surface rendering, direct volume rendering (DVR) does not extract surfaces before rendering, but directly renders images from given three-dimensional samples. DVR techniques fall into two main classes: image-order and object-order techniques. *Ray-casting* [Levoy 1987] is a typical representative of an image-order approach, where rays are sent from the eye-point via the image plane through the volume data. On the other hand object-order techniques, e.g., *splatting* [Westover 1990], traverse the volume samples and project them onto the image plane. All these techniques are at least of computational complexity $O(N^3)$ for an $N \times N \times N$ data array.

Frequency domain volume rendering (also known as Fourier volume rendering (FVR) [Malzbender 1993]) is a volume rendering technique based on a completely different idea, i.e., the projection slice theorem [Levoy 1992]. The goal is to compute projections of the volumetric data (complexity $O(N^3)$). Projections in the spatial domain correspond to slicing in the frequency domain (complexity $O(M^2)$ for a $M \times M$ reconstructed slice). Therefore slicing is used in frequency domain volume rendering to reduce the computational complexity of projections in spatial domain.

Three-dimensional data are first transformed from the spatial domain into the frequency domain. This can be done by using a three-dimensional Fourier or Hartley Transform. The discrete one-dimensional forward and inverse Fourier Transforms are given by equations 1 and 2.

$$F[\omega] = \sum_{x=0}^{N-1} f[x] \cdot e^{-i2\pi\omega x/N} \tag{1}$$

$$f[x] = \frac{1}{N} \sum_{\omega=0}^{N-1} F[\omega] \cdot e^{i2\pi\omega x/N} \tag{2}$$

where f is the discrete function given by N samples and F is its frequency spectrum.

After the pre-processing step of complexity $O(N^3 \log N)$, slices are resampled along a plane oriented perpendicular to the viewing direction. Afterwards an

*e-mail: viola@cg.tuwien.ac.at
†e-mail: kanitsar@cg.tuwien.ac.at
‡e-mail: meister@cg.tuwien.ac.at

inverse two-dimensional transform of the resulting slice is done. The method has a computational complexity of $O(M^2 \log M)$, which is lower as compared to other volume rendering methods.

The method, however, has certain limitations: only parallel projection is possible and hidden surface removal is not included. The reason is the nature of the function that FVR computes, which is an order independent linear projection. This results in *X-ray* images. Currently ray-casting is considered as the method that produces the best image quality. Ray-casting gains performance by early-ray termination, displaying only some surfaces from the data. FVR displays the entire volumetric data set. Because of the computational complexity, FVR will gain importance when the resolution of data sets will increase.

As described earlier, the performance of FVR does not explicitly depend on the size of the input volume. The factor that directly influences the rendering performance is the number of samples contributing to the projection slice (in the previous text $M \times M$). The resolution of the projection slice should be high enough to prevent aliasing. This shows as overlapping of copies of the rendered data in the result image. Progressive refinement strategies can be realized by adjusting the projection slice resolution to achieve a desired performance. Also the resampling area can be reduced to low frequencies around the origin only. This will result in blurry preview images, but no frequency overlapping artifacts will occur. Our implementation does not include a progressive refinement mode. Highly interactive framerates are achieved even when slicing the entire data set with sufficient slice resolution.

This paper presents mapping of FVR algorithm to GPU in order to significantly accelerate the rendering performance. An overall pipeline of hardware-accelerated frequency domain volume rendering is presented. The data set is transformed into frequency domain in a pre-processing step. Then the projection slice is resampled using the following interpolation schemes: nearest neighbor interpolation, tri-linear interpolation, tri-cubic interpolation, and interpolation using windowed *sinc* with window of width four. In addition we demonstrate that current graphics hardware provides enough precision to perform FVR at high quality. Furthermore the GPU-based multi-dimensional Fast Hartley Transform [Bracewell et al. 1986; Hao and Bracewell 1987] is presented as an alternative to the wide spread Fourier Transform. The rendering results are compared according to image quality as well as performance. The performance is compared to a software implementation using the highly optimized FFTW library [Frigo and Johnson 1998].

Section 2 describes previous work related to FVR and its mapping towards GPU. The overall rendering pipeline is discussed in section 3. First, the stage performed on the CPU is presented, followed by the on-the-fly rendering stage on the GPU. The slicing in the frequency domain is discussed in sub-section 3.1. The following part, i.e., inverse transform to the spatial domain, is shown in sub-section 3.2. Afterwards we show the results in section 4 and discuss future work and conclusions in sections 5 and 6.

2 Related Work

Frequency domain volume rendering was introduced by Levoy [1992] and Malzbender [1993]. Malzbender proposes various filters for high-quality resampling in frequency domain. Totsuka and Levoy [1993] extended this work with depth cues and shading performing calculations in the frequency domain during slice extraction. Illumination models for FVR were studied in the work of Entezari et al. [2002]. They describe methods to integrate diffuse lighting into FVR. One approach is based on gamma corrected hemispherical shading and is suitable for interactive rendering of fixed light sources. Another technique uses spherical harmonic functions and allows lighting using varying light sources. These shading techniques, however, require a large amount of memory and are not well suited for visualization of large data sets.

Another approach that produces images which are similar to FVR is based on importance sampling and Monte Carlo integration [Csébfalvi and Szirmay-Kalos 2003] thus the samples are not aligned on a regular grid. This technique overcomes the limitation of parallel projection and the overall computational complexity $O(N^2)$ is better than in case of FVR.

A straightforward implementation of the Fourier transform is not suitable for high-performance FVR. The inverse two-dimensional transform must be computed at high speed to achieve interactive framerates. Therefore fast variants of the Fourier Transform are used in FVR implementations. The original idea of the Fast Fourier Transform (FFT) was introduced by Cooley and Tukey [1965]. Their algorithm decomposes the Discrete Fourier Transform (DFT) into $\log_2 N$ passes, where N is the size of the input array. Each of these passes consists of $N/2$ *butterfly* computations. Each butterfly operation takes two complex numbers a and b and computes two numbers, $a + wb$ and $a - wb$, where w is a complex number, called principal Nth root of unity [Cooley and Tukey 1965]. The complex number w corresponds to the exponential term from equations 1 and 2. Butterfly operations are based on an efficient reordering of intermediate results, which are used multiple times. After $\log_2 N$ passes the butterfly operations result into the transformed data. One of the fastest implementations available, is the FFTW library [Frigo and Johnson 1998].

The Fast Hartley Transform (FHT) was proposed by Bracewell [1986] as an alternative to FFT. The transform

produces real output for a real input, and is its own inverse. Therefore for FVR the FHT is more efficient in terms of memory consumption. The one-dimensional forward and inverse Hartley transform is described by equation 3:

$$H[\omega] = \sum_{x=0}^{N-1} h[x] \cdot \left(\cos \frac{2\pi\omega x}{N} + \sin \frac{2\pi\omega x}{N} \right) \quad (3)$$

where h is the discrete function given by N samples and H is its Hartley transform. The Multi-dimensional Hartley Transform, however, is not separable, i.e., the N-dimensional transform cannot be computed as a product of N one-dimensional transforms. Bracewell and Hao propose a solution to this problem [Bracewell et al. 1986; Hao and Bracewell 1987]. They suggest to perform N one-dimensional transformations in each orthogonal direction followed by an additional pass that *corrects* the result to correspond to the N-dimensional Hartley transform. The correction pass for 2D and 3D respectively is described by equations 4 and 5.

$$\begin{aligned} H[u,v] = \quad & \tfrac{1}{2}(T[u,v] + T[L-u,v] + \\ & T[u,M-v] - T[L-u,M-v]) \end{aligned} \quad (4)$$

$$\begin{aligned} H[u,v,w] = \quad & \tfrac{1}{2LMN}(T[L-u,v,w] + \\ & T[u,M-v,w] + T[u,v,N-w] \\ & -T[L-u,M-v,N-w]) \end{aligned} \quad (5)$$

H is the multi-dimensional transform, T is the product of one-dimensional transforms in two respectively three orthogonal directions, and L, M, and N describe the size in each particular direction.

Many approaches exist to exploit the capabilities of modern graphics accelerators for volume rendering. Fast processing and a large number of flexible features are the main reasons that make current graphics hardware attractive. Texture-based volume rendering using one 3D texture or a stack of 2D textures for volumetric data representation gained considerable interest [Cabral et al. 1994; Rezk-Salama et al. 2000]. These techniques perform very fast. They however, also compute a huge number of operations that do not contribute to the final image. A new approach was presented by Roettger et al. [2003] and Krüger and Westermann [2003a]. They propose to use front-to-back ray-casting with early ray termination. The Z-buffer is used for opacity accumulation. An early Z-test rejects fragments when the accumulated opacity reaches a certain threshold. Besides the standard volume rendering approach based on transfer function specification, also other rendering modes like MIP, contour enhancement or tone shading have been ported to GPU-based implementations [Hadwiger et al. 2003].

Flexibility of the latest graphics hardware is also used for various other general purpose computations [GPGPU 2003]. Moreland and Angel [2003] have implemented a two-dimensional FFT running on NVidia GeForceFX GPUs [NVIDIA 2003]. Their implementation is using the Cg high-level shading language [Mark et al. 2003] and is based on the *Decimation in Time* algorithm [Cooley and Tukey 1965]. Unfortunately this implementation performs slower than the software reference [Frigo and Johnson 1998], which is running on a standard CPU. Another FFT implementation was done by Hart [Engel 2003]. His implementation performs much faster than the previously mentioned implementation and runs on ATI GPUs [ATI 2003].

Recently, algorithms for numerical simulation exploit the processing power of current GPUs. Bolz et al. [2003] implemented sparse matrix conjugate gradient solver and a regular-grid multi-grid solver. Similar work was presented by Krüger and Westermann [2003b]. Hillesland et al. [2003] have turned the nonlinear optimization for image-based modeling into a streaming process accelerated on GPU.

An important aspect of FVR is interpolation, since the samples in the projection slice, in general, do not coincide with samples of the transformed input data. Current graphics hardware natively supports nearest neighbor interpolation for all texture formats. Linear interpolation is supported only for fixed-point formats. Unfortunately higher-order interpolation schemes are not natively supported at all. A general approach for GPU-based linear filtering was presented by Hadwiger et al. [2002]. Their work can be applied to arbitrary filter kernels, gaining speed-ups from various kernel properties like symmetry and separability. The filter kernel is sampled at high-resolution and stored as a texture. A particular sample that contributes to the new resampling point is convolved with a kernel *tile*. Their framework implements various higher-order reconstruction filters like cubic B-spline filters, Catmull-Rom spline filters or windowed sinc filters. As Malzbender [1993] has shown, careful filter design for reconstruction in the frequency domain is crucial for good image quality.

3 Mapping FVR on GPU

Frequency domain volume rendering can be divided into two stages. In the first stage the original scalar data is transformed from the spatial domain into the frequency domain. Before doing this, we have to rearrange the spatial data to set the origin of the data from the corner [0, 0, 0] of the data cube to the center. This *wrap-around* operation followed by a 3D FFT is usually done off-line for each data set in a pre-processing step. Although a GPU-based implementation of a three-dimensional transform is currently possible, the performance cannot com-

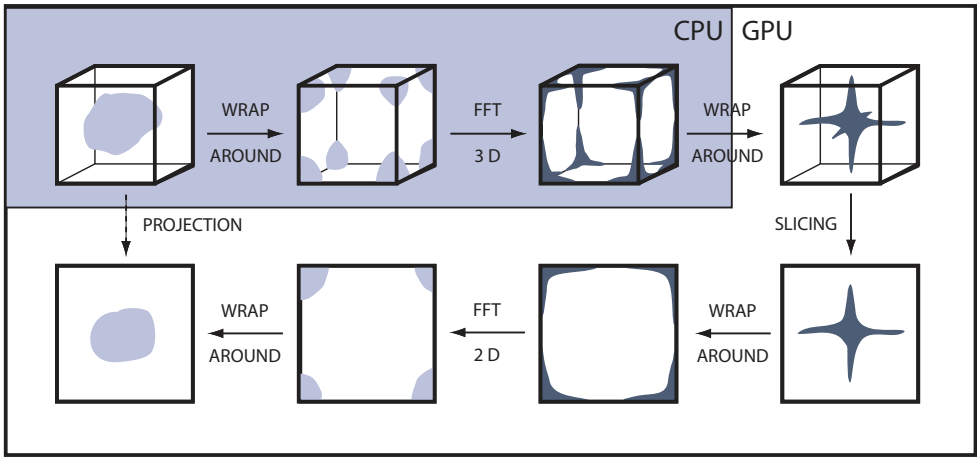


Figure 1: Frequency domain volume rendering pipeline. Instead of projecting the data in the spatial domain ($O(N^3)$), slicing in frequency domain is performed followed by inverse transform ($O(M^2 \log M)$) after an off-line pre-processing step.

pete with optimized software implementations. This is due to limitations in handling 3D textures. Another aspect is that during the transform always two copies of the data have to be present in the graphics hardware memory, i.e., one as source another as destination array. This limits the size of the rendered data set to be half of the available graphics memory resources. Since the three-dimensional transform is done only once per data set, the FFTW library [Frigo and Johnson 1998] is used to transform the data on the CPU. The transformed data is rearranged via wrap-around operation once again to set the low frequencies to the center of the data set. After this step the data is uploaded to the graphics hardware memory. All the following on-the-fly computation is done on GPU. The uploaded array of complex numbers is represented as a two-channel 32-bit floating-point 3D texture. Although FVR does not require the dimensions of the input data to be a power of two, current hardware limits dimensions of 3D textures to be a power of two. If the volume resolution is not a power of two, zero-padding in frequency domain is done. Without loss of generality we use the OpenGL API [OpenGL 2003] in our implementation. Even though implementation details are related to this API, porting the code to other APIs is straightforward.

After the data upload the algorithm proceeds to the rendering stage. This stage can also be divided into two parts, i.e., slicing in the frequency domain and the inverse Fourier transform. Slicing refers to resampling the projection slice from the 3D frequency data. Transforming the resampled frequency slice back to the spatial domain results in the projection of accumulated intensities. These two parts will now be focused on and discussed in more detail. The FVR pipeline is sketched in figure 1.

3.1 Slicing in Frequency Domain

In the first part of the rendering stage the resampling of the projection slice is performed. The slice is perpendicular to the viewing direction intersecting the frequency volume in the origin of the frequency spectrum. This is in fact the central sample of the 3D texture. The setup of this rendering part maps the 3D texture onto a *proxy* geometry, which is a single quad. Texture coordinates are set to cover the entire 3D texture under any viewing direction, i.e., slicing along the diagonal determines the maximal coverage of the 3D texture. The proxy geometry is stretched over the whole rendering target, i.e., a buffer where the results of the rendering pass are stored. When changing the viewing direction, the texture is rotated around the frequency volume origin. The setup is illustrated in figure 2.

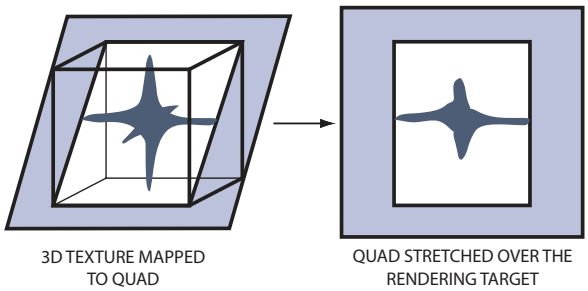


Figure 2: Setup for the resampling part shows relationship between 3D texture, proxy slice geometry and rendering target.

To avoid aliasing the resolution of the rendering target must meet the Nyquist criterion. Also the reconstruction quality of the projection slice strongly influences the fi-

nal rendering quality. Low resampling quality introduces artifacts like *ghosts* or frequencies that are not present in the original data. To avoid this, it is necessary to use higher order interpolation schemes, or at least tri-linear interpolation. The hardware does not natively support higher order interpolation schemes, so floating-point textures can be fetched only using nearest neighbor interpolation. We additionally support three other types of interpolation. The first custom filter type is tri-linear interpolation. The second interpolation scheme is tri-cubic interpolation using cardinal or BC-splines. The last scheme is windowed sinc of width four. Tri-cubic interpolation as well as windowed sinc interpolation are based on the texture-based approach of Hadwiger et al. [2002]. The following subsections describe in more detail the mapping of these custom filters onto GPU. After describing the interpolation schemes various hardware implementation issues of slicing are presented.

3.1.1 Tri-linear interpolation

Our implementation of tri-linear interpolation is based on the `LRP` instruction, where the abbreviation `LRP` refers to linear interpolation. First the eight nearest neighbors of a resampling point are fetched with nearest neighbor interpolation. The addresses of the neighbors are precomputed in the per-vertex stage for each vertex of the rendering pass. These are then stored in eight available texture-coordinate vertex-attributes. The addresses between the vertices are interpolated "for free" during the rasterization. The values of the neighbors are fetched using pre-computed addresses in the per-fragment stage from the source 3D frequency texture.

The problem is how to estimate the blending factors for all three directions. Although it is not explicitly given, they can be retrieved from the three-dimensional texture coordinates of the resampling point. These describe the offset from a corner of the 3D texture in the range $[0, 1]$ for each coordinate. Obviously the coordinates of the opposite corner are $[1.0, 1.0, 1.0]$. The multiplication of texture coordinates of a given resampling point with the original texture resolution results in coordinates where the distance between two texels is equal to 1.0. The fractional parts of these new coordinates are the blending factors we are looking for. We illustrate the blending factor estimation in figure 3.

3.1.2 Higher-order interpolation

Higher-order interpolation schemes in our implementation are using textures for storing a discretized reconstruction kernel. The kernel is represented via a high number of samples. All reconstruction filters, which are used in our implementation, are separable. This allows to store the kernel in a 1D texture instead of storing it

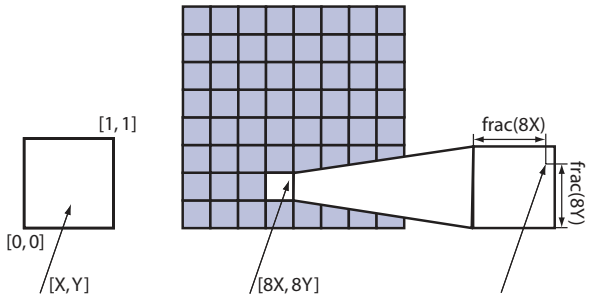


Figure 3: Blending factor estimation. The resolution of the original texture is 8×8. The texture coordinates are multiplied with the original resolution. The blending factors are equal to the fractional parts of these new coordinates.

in 3D textures and multiplying the 3D kernel *tile* on-the-fly. A tile corresponds to a part of the kernel in the interval between two integral numbers. The width of the kernel thus determines the number of tiles that cover the whole reconstruction kernel. Our implementation supports kernels of width four, so the 1D kernel is divided into four tiles similar to the method proposed by Hadwiger et al. [2002]. But instead of storing the kernel in several textures, we use a single four-channel 1D texture using each channel to store one kernel tile. This reduces the number of texture fetch instructions and increases the resampling performance. Figure 4 shows how the kernel tiles are stored in a four-channel 1D texture.

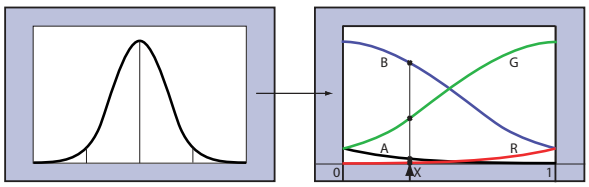


Figure 4: Filter kernel of width four and corresponding 1D floating-point RGBA texture storing the discretized kernel. Each channel stores one kernel tile. A single texture fetch at position `X` returns 4 weight values.

The filtering is divided into four passes, where each pass is computing the contribution of sixteen neighboring samples. These intermediate results are summed together resulting in the value of the resampling point. The straightforward method to sum-up intermediate results would be to set the blending operation to addition. Unfortunately current hardware does not support blending with floating-point precision. The blending is done after four intermediate passes in a separate pass in which four sub-results are summed together. After the blending pass the filtered slice is stored in a texture, ready to be processed with the inverse FFT.

3.1.3 Hardware implementation strategies

This section deals with the current state of graphics hardware. Recent interesting features are discussed, which are relevant to resampling the data from a 3D texture onto a slice.

Currently only ATI GPUs support floating-point 3D textures, so our implementation is running on these type of cards. Supporting NVidia hardware would require to store the floating-point value of every sample in a four-channel fixed-point 3D texture. The slicing part will then include an additional conversion in order to reconstruct the original floating-point value. This must be done before interpolation for every sample contributing to the resampling point on the projection slice. The rest of the pipeline remains unchanged. Additionally the implementation of texture-based higher-order interpolation is divided into four passes. This is due to limitations of the fragment program length on ATI hardware. NVidia supports much longer fragment programs, i.e., the multi-pass approach can be folded into a single pass.

The multi-pass approach discussed before, as well as FVR in general, renders an intermediate image that is used in the next pass as input. This is done by changing the rendering target to a texture and vice versa. The rendering target is any kind of buffer that can be bound as a 2D texture, which is in current OpenGL specification invisible rendering target called *Pbuffer*.

3.2 Inverse Two-Dimensional Fast Fourier Transform

The second part of the rendering stage receives the resampled projection slice as 2D input texture. The inverse two-dimensional Fourier transform transforms the data back to the spatial domain and the final result is rendered into the framebuffer. The transform used in our implementation is based on a method implemented by Hart et al. [Engel 2003].

Both transforms, i.e., forward and inverse, can exploit the separability property in the multi-dimensional case. In the case of a two-dimensional array first the *rows* are transformed as stack of independent one-dimensional arrays. These frequency arrays are then transformed *column-wise*, where each column is also handled separately. In case of the three-dimensional transform, the third dimension is handled analogously.

The two-dimensional Fourier transform is split in two almost identical parts. Each of these passes consists first of reordering the input data, also called *scrambling*. This pass prepares the data for $\log_2 N$ butterfly passes where N is the number of columns or rows respectively. Scrambling means swapping two samples, where one sample is at position `x` and the other sample is at position `y`. The relationship between `x` and `y` is that `y` is the reverse in the bit order of `x`. For example a sample at position `x=4` (bit pattern `100`) is exchanged with the sample at position `y=1` (bit pattern `001`). This reverse order function can be efficiently done using a pre-computed scramble lookup texture.

The scramble pass is followed by $\log_2 N$ butterfly passes. The butterfly passes are performed *ping-pong-wise*. One buffer is bound as the source 2D texture, another one as the rendering target. In the next pass the first buffer changes to be bound as the rendering target and the second buffer (the rendering target from the previous pass) is bound as the source texture. Each butterfly pass first performs a texture fetch into a lookup texture, which determines addresses of two samples a and b that contribute to the currently processed fragment. The value of the principal Nth root of unity w is also contained in this fetch. Afterwards, a complex multiplication and two additions are performed as described in section 2.

After performing one scramble pass and $\log_2 N$ butterfly passes in one dimension, the algorithm analogously performs the same operations to the other dimension. In case of the inverse transform the values are finally scaled down by the factor N^{-1} (see equation 2) in the last pass and the output is written to the framebuffer. The process of a two-dimensional FFT is illustrated in figure 5. The interested reader is referred to Hart's implementation [Engel 2003] for further details.

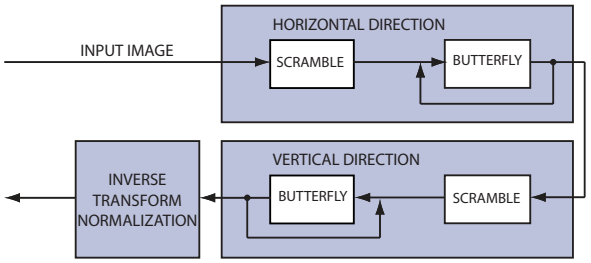


Figure 5: State diagram of GPU-based FFT.

To reduce the memory requirements by one half, the Hartley transform can be used instead of Fourier transform. The GPU-based implementation of the Fast Hartley Transform (FHT) is quite similar to the FFT. The biggest difference is the necessity of a *double butterfly*. In case of the FFT the butterfly texture stores the complex number w, addresses to values contributing on the butterfly and a sign that determines which butterfly result should be written to the current fragment. The FHT needs to store five values, therefore it is necessary to have two lookup textures instead of a single one. One possibility is to store three addresses of samples that contribute to the final butterfly result in one three-channel texture. Precomputed weights of *cos* and *sin* terms are stored in another two-channel texture. The *cos* and *sin* terms are then multiplied with the corresponding input values and summed together. More details on the FHT can be found in the referenced literature [Bracewell 1986].

In the multi-dimensional case, the last pass corrects the product of one-dimensional transforms for each direction to the multi-dimensional transform. The shader performs four texture fetches, sums them together and performs correction according to equations 4 and 5.

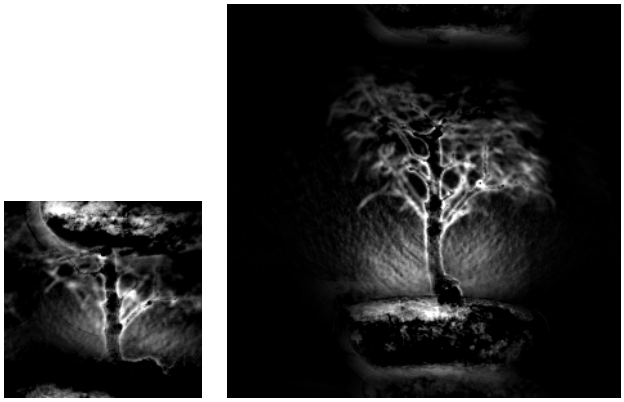
4 Results

Our implementation was tested on the ATI Radeon 9800 XT [ATI 2003] with 256 MB of memory. The test data set was the X-mas Tree [Kanitsar et al. 2002] of size 256^3. The rendering performance was tested at two different projection slice resolutions, i.e., 256^2 and 512^2 pixels. The slice was resampled using four different interpolation schemes: nearest neighbor interpolation, tri-linear interpolation, tri-cubic interpolation, and interpolation using windowed sinc. The window function was a Blackman window of width four [Theußl et al. 2000]. The framerates are shown in table 1 below. The tri-cubic and windowed sinc interpolation are using the same principle. Therefore they achieve exactly the same performance. The discretized kernel for the higher-order interpolation schemes has resolution of 64 samples per filter tile. The corresponding images are shown in figure 7. It is clearly visible that the nearest neighbor interpolation has similar performance to tri-linear interpolation, however, the rendering quality using nearest neighbor reconstruction is rather poor. The results of tri-linear reconstruction seem to be of acceptable quality. The cubic B-spline filter is an approximative filter, which can be considered as a low pass filter. The image where the projection slice was reconstructed using this filter exhibits less noise as compared to windowed sinc and other reconstruction schemes. All images clearly show copies of the dataset. This is due to the absence of zero-padding in the spatial domain [Levoy 1992]. Further datasets rendered using the GPU-based FVR approach are shown in figure 8.

Resolution	NN	TL	TC	IFFT 2D
256×256	1450	1050	180	153
512×512	500	350	45	35

Table 1: Framerates of GPU-based FVR: The first three columns show framerates of three different interpolation schemes in the slicing stage: Nearest Neighbor (NN), tri-linear (TL), and tri-cubic (TC). The fourth column shows the performance of the inverse transform. The rows describe the size of the projection slice.

The quality of the image is also strongly influenced by the resolution of the projection slice. This is shown in figure 6. This dataset has resolution 256^3, which means that the resolution of the projection slice should be at least 512×512. If this condition is not fulfilled overlapping artifacts appear.



(a) (b)

Figure 6: Quality comparison using the bonsai tree dataset (256^3). Projection slice resolution is 256×256 (a) and 512×512 (b). Low projection slice resolution results in an image distorted by overlapping copies. This effect is removed when using a sufficient resampling resolution.

The software FVR using the highly optimized FFTW library [Frigo and Johnson 1998] was running on an AMD Athlon XP 2200+ processor with 1.0 GB of RAM (2×512 DDR, 133 MHz, CL2) and VIA Apollo KT333 chipset. The software implementation is using tri-linear interpolation, a projection slice of size of 256×256 and the same test data set of size 256^3. The performance of slicing was 17 frames per second (fps). Wrap-around reordering is running at 45 fps and inverse 2D transform at 26 fps. Note that wrap-around reordering does not take additional time in the GPU implementation. Using the algorithm mapped on the GPU, a speed-up factor of approximately 17 is achieved.

5 Future work

The depth cues proposed by Totsuka and Levoy [1993] can be also integrated into our framework to improve the spatial perception. Also additional filters proposed by Malzbender [1993] will improve the quality of the resulting images.

Future work also includes mapping to other platforms, e.g., NVidia GPUs as well as a high level shading implementation using the Cg [NVIDIA 2003] respectively the OpenGL Shading Language [OpenGL 2003].

6 Summary and Conclusions

The mapping of frequency domain volume rendering onto GPU is presented. The GPU-based rendering

stage results in highly interactive framerates, achieving a speed-up factor of 17 compared to the CPU-based approach. We discussed the overall hardware-accelerated pipeline: The data set is first pre-processed on the CPU. Then the frequency dataset is uploaded to graphics hardware for the on-the-fly rendering stage. This consists of two sub-stages slicing and inverse transform. The quality of the rendered results is strongly influenced by the used interpolation scheme. Four different interpolation schemes are presented. The difference between these interpolation schemes is shown with respect to performance and quality.

The performance of frequency domain volume rendering does not explicitly depend on the data set resolution. It depends on the number of resampling points which are given by the resolution of the projection slice. The data set resolution influences the texture cache efficiency, i.e., the higher the resolution is, the higher is the number of cache misses. This can lead to slight differences in rendering performance, which is usually $\pm$ 2 fps in case of a 512×512 projection slice resolution.

Acknowledgments

The work presented in this publication has been funded by the ADAPT project (FFF-804544). ADAPT is supported by *Tiani Medgraph*, Vienna (http://www.tiani.com), and the *Forschungsförderungsfonds für die gewerbliche Wirtschaft*, Austria. See http://www.cg.tuwien.ac.at/research/vis/adapt for further information on this project.

The authors would like to thank Martin Artner, Markus Hadwiger from the VRVis Research Center for fruitful discussions on hardware-based filtering, Alois Dornhofer for the software implementation, Jason Mitchell, Evan Hart, and especially Mark Segal from the ATI corporation for providing a Radeon 9800 XT and for general support.

References

ATI, 2003. ATI web page, http://www.ati.com/, November.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *Proceedings of SIGGRAPH'03*, 917 – 924.

BRACEWELL, R. N., BUNEMAN, O., HAO, H., AND VILLASENOR, J. 1986. Fast two-dimensional Hartley transform. *Proceedings of the IEEE 74*, 1282–1282.

BRACEWELL, R. N. 1986. *The Hartley Transform*. Clarendon Pr.

BRACEWELL, R. N. 2000. *The Fourier Transform and Its Applications*. McGraw-Hill.

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of Symposium on Volume Visualization'94*, 91–98.

COOLEY, J. W., AND TUKEY, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 297–301.

CSÉBFALVI, B., AND SZIRMAY-KALOS, L. 2003. Monte carlo volume rendering. In *Proceedings of IEEE Visualization'03*, 449–456.

ENGEL, W. F., Ed. 2003. *ShaderX 2 - Shader Programming Tips & Tricks with DirextX 9*. Wordware Publishing, ch. Advanced Image Processing with DirectX 9 Pixel Shaders, 457–464.

ENTEZARI, A., SCOGGINS, R., MÖLLER, T., AND MACHIRAJU, R. 2002. Shading for fourier volume rendering. In *Proceedings of Symposium on Volume Visualization and Graphics'02*, 131–138.

FRIGO, M., AND JOHNSON, S. G. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP'98*, vol. 3, 1381–1384.

GPGPU, 2003. General Purpose Computation on GPUs web page, http://www.gpgpu.org/, November.

HADWIGER, M., VIOLA, I., THEUSSL, T., AND HAUSER, H. 2002. Fast and flexible high-quality texture filtering with tiled high-resolution filters. In *Proceedings of Workshop on Vision, Modelling, and Visualization VMV'02*, 155–162.

HADWIGER, M., BERGER, C., AND HAUSER, H. 2003. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *Proceedings of IEEE Visualization'03*, 301–308.

HAO, H., AND BRACEWELL, R. N. 1987. A three-dimensional DFT algorithm using the fast Hartley transform. *Proceedings of the IEEE 75*, 264–266.

HILLESLAND, K., MOLINOV, S., AND GRZESZCZUK, R. 2003. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. In *Proceedings of SIGGRAPH'03*, 925 – 934.

KANITSAR, A., THEUSSL, T., MROZ, L., ŠRÁMEK, M., BARTROLÍ, A. V., CSÉBFALVI, B., HLADŮVKA, J., FLEISCHMANN, D., KNAPP, M., WEGENKITTL, R., FELKEL, P., RÖTTGER, S., GUTHE, S., PURGATHOFER, W., AND GRÖLLER, M. E. 2002. Christmas Tree Case Study: Computed Tomography as a

Tool for Mastering Complex Real World Objects with Applications in Computer Graphics. In *IEEE Visualization 2002*, 489–492.

KRÜGER, J., AND WESTERMANN, R. 2003. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization'03*, 287–292.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. In *Proceedings of SIGGRAPH'03*, 908–916.

LEVOY, M. 1987. Display of surfaces from volume data. *IEEE Computer Graphics and Applications 8*, 29–37.

LEVOY, M. 1992. Volume rendering using the fourier projection-slice theorem. In *Proceedings of Graphics Interface'92*, 61–69.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of SIGGRAPH'87*, 163–169.

MALZBENDER, T. 1993. Fourier volume rendering. *ACM Transactions on Graphics 12*, 3, 233–250.

MARK, W., GLANVILLE, S., AKELEY, K., AND KILGARD, M. 2003. Cg: A system for programming graphics hardware in a C-like language. In *Proceedings of SIGGRAPH'03*, 896–907.

MORELAND, K., AND ANGEL, E. 2003. The FFT on a GPU. In *Proceedings of Eurographics / SIGGRAPH Workshop on Graphics Hardware'03*, 112–119.

NVIDIA, 2003. NVIDIA web page, http://www.nvidia.com/, November.

OPENGL, 2003. OpenGL web page, http://www.opengl.org/, November.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of Eurographics / SIGGRAPH Workshop on Graphics Hardware'00*, 109–118.

ROETTGER, S., GUTHE, S., WEISKOPF, D., AND ERTL, T. 2003. Smart hardware-accelerated volume rendering. In *Proceedings of the Symposium on Data Visualisation VisSym'03*, 231–238.

THEUSSL, T., HAUSER, H., AND GRÖLLER, M. E. 2000. Mastering windows: Improving reconstruction. In *Proceedings of Symposium on Volume Visualization'00*, 101–108.

TOTSUKA, T., AND LEVOY, M. 1993. Frequency domain volume rendering. In *Proceedings of SIGGRAPH'93*, 271–278.

WESTOVER, L. 1990. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH'90*, 367–376.

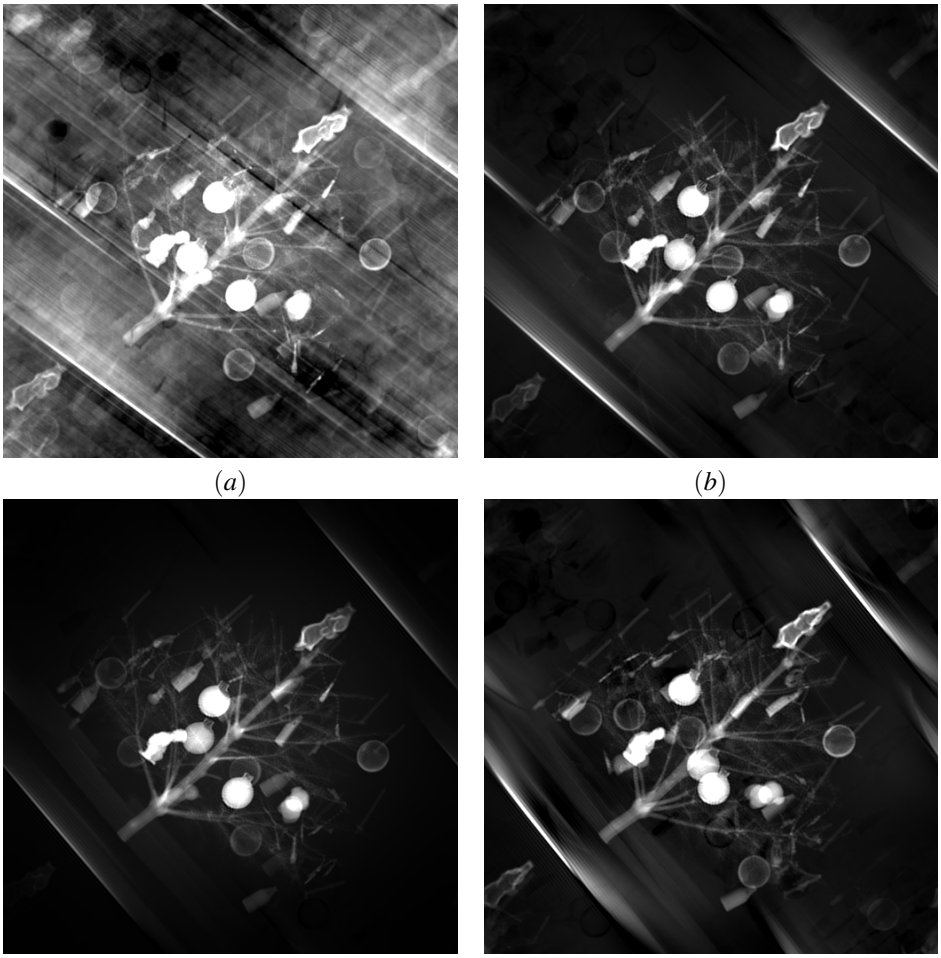


Figure 7: Result of rendering the X-mas tree test data set. The images show the rendering quality according to the used interpolation scheme. Nearest neighbor interpolation (a) exhibits noticeable artifacts, which are eliminated by tri-linear interpolation (b), respectively by higher-order interpolation schemes like tri-cubic interpolation using a cubic B-spline filter (c) or windowed sinc filter using a Blackman window of width four (d).

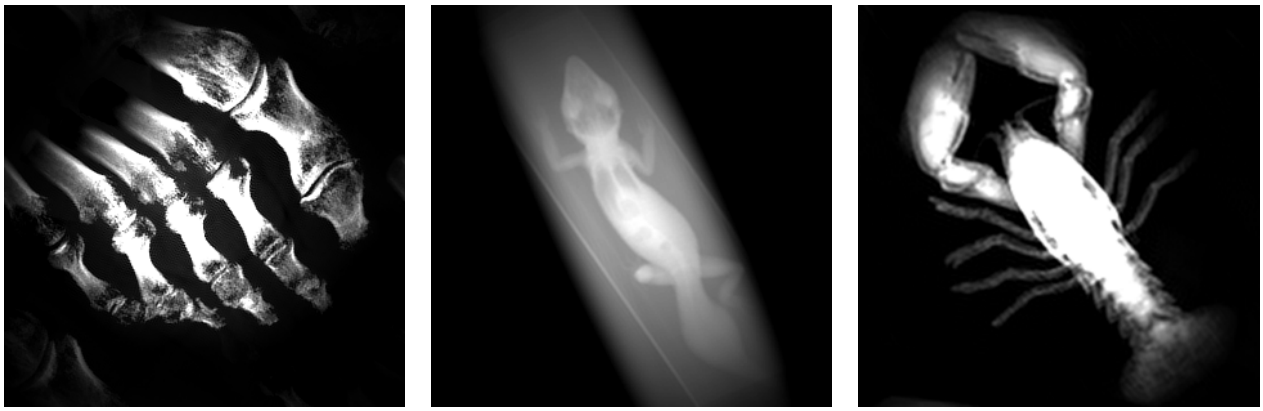


Figure 8: Other datasets rendered with projection slice resolution 512×512 using a tri-cubic B-spline reconstruction filter.