# Taking a Look Inside:
## Frequency-Domain Volume Rendering

Erin Fitzgerald
Rebecca Hildebrand
Alex Markle

18-551
Final Report
Group 3

# Table of Contents

**Background**
1. Abstract ........................................................................................1
2. Introduction....................................................................................2
3. Background.....................................................................................3
4. Why the C67 is Good for our Problem.............................................6

**Our Project**
5. Project Objectives
   5.1 Speed.........................................................................................7
   5.1 Image Quality.............................................................................7
   5.1 Usefulness..................................................................................7
6. Program Overview – Major Parts of The Program
   6.1 Data flow....................................................................................8
   6.2 FFT function...............................................................................9
   6.3 Slice function..............................................................................11
   6.4 User Interface.............................................................................14
7. Implementation on EVM
   7.1 What is on the EVM...................................................................15
   7.1 Why is there not more on the EVM?...........................................15
   7.1 Memory Allocation.....................................................................17
   7.1 Timing breakdown......................................................................18
8. Problems We Ran Into
   8.1 Hartley vs. FFT...........................................................................18
   8.1 Memory.......................................................................................20
9. Results
   9.1 Example output...........................................................................21

**Future**
10. Improvement to Project
   10.1 Speed Improvements.................................................................22
   10.1 Image Quality Improvements....................................................23

**References**
References.............................................................................................i
Code & Data Sources.............................................................................ii

Appendices
Appendix A (display.cpp) ....................................................................iii
Appendix B (slicer.c) ...........................................................................xiv
Appendix C (fftn.c) ..............................................................................xix
Appendix D (filterpc.c) .......................................................................xxxviii
Appendix E (filterevm.pc) ...................................................................xliv

# Abstract

The purpose of this paper is to describe the implementation of our project, "Taking a Look Inside: Frequency-Domain Volume Rendering". The paper will begin with some background information on Volume Rendering, including a description of Frequency-Domain Volume Rendering. It will then go on to talk about the major parts of the program, including code division and data flow. The paper will follow the code descriptions with explanations of how parts of the code have been implemented on the EVM. It will then go on to describe EVM issues that we encountered as well as other obstacles we had to overcome. The report will finish up with the final results of our project and possible improvements for the future.

## 2. Introduction

Last month, our friend Sarah went skiing for the first time. On her third run down the mountain, she took a terrible fall that caused amazing pain in her knee. After a visit to the emergency room, she had to make an appointment for an MRI to find out what damage had been done. During her appointment, Sarah had the MRI done, but unfortunately could not find out the results until the subsequent week. To help limit the patient's anxiety, it should be possible to take an MRI and view the results during the same appointment. With current technology in most hospitals, fast diagnosis is not an option and patients are made to wait to find out their results. A program that allowed a doctor to look at a data set in real time would make the diagnosis process much faster, and as a result ease the minds of patients like Sarah and make hospitals more efficient in the process. As a solution, we have implemented a program that uses frequency-domain volume rendering to display 3-D data sets. With this algorithm, "real time" volume rendering can be achieved.

## 3. Background

Volume rendering is a useful tool for producing flat projections of three-dimensional data. This is typically done using a ray casting technique. This technique calculates line integrals through the data, normal to the viewing plane, for each pixel in the rendered image. A continuing problem for the use of this method in clinical diagnoses is its high computational demands. The complexity for traditional ray-casting techniques is of $O(N^3)$. These operations are very time consuming, especially for datasets of useful size ($128^3$ or $256^3$). This makes real-time rendering virtually impossible without the use of a super-computer.

For improved usability, two-dimensional projections need to be generated rapidly, in 'real-time'. One suggested method for improving the speed of volume rendering is to perform it in the frequency domain. Frequency-domain volume rendering can be performed between one and two orders of magnitude faster than traditional ray-casting techniques, with complexities of $O(N^2 * \log N)$. This makes it an optimal choice for applications that require high rendering rates.

The basis for frequency domain volume rendering is the Fourier Projection-Slice Theorem. This theorem states that the two-

dimensional Fourier transform of a two-dimensional projection of a three-dimensional object at an angle, theta, is a two-dimensional plane passing through the origin of the three-dimensional Fourier transform of that three-dimensional object, at the same angle, theta.

To perform frequency domain volume rendering, a three-dimensional forward Fourier transfer is first performed on the data set. Then a slice is taken through the center of this transformed data, at the angle corresponding to the desired viewing angle. This slice is then run through an inverse two-dimensional Fourier transform, and the resulting data is displayed to the screen. After the complete three-dimensional transform has been completed, the only steps required to produce further renderings from other viewing angles are the re-sampling and inverse two-dimensional transforms.

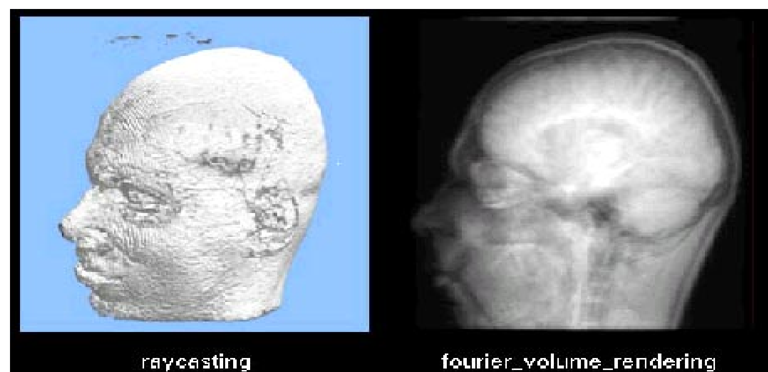



*Figure 1: Ray-casting vs. Fourier Volume Rendering*

The images produced using frequency-domain volume rendering are not identical to those produced using ray-casting, as shown in the figure on the previous page.  In frequency domain-volume rendering, all voxels (three-dimensional pixels) along the lines normal to the viewing plane contribute equally to the final pixel value, regardless of their distance from the viewer.  This eliminates the hidden surface effects that appear in images produced using ray-casting, and make frequency-domain volume rendered images look like X-ray images. (Malzbender, 1993)

# 4. Why the C67 is Good for our Problem

There are a number of advantages to using specialized hardware such as the C67 for performing real-time volume rendering. The most obvious of these is speed. As stated above, the complexity for performing the repeated interpolation and inverse transforms is very high, and for medical application time can be at a premium. The C67 can perform these computations at very high rates, especially if its multiprocessing attributes are exploited fully in the code.

The C67 is also sensible for use in this project because of its possible usefulness in the application that we are investigating: medical imaging. It would be very reasonable for a hospital to use PCs, equipped with built in DSP hardware such as the C67, which would allow doctors to render images taken through traditional means (MRI, ultrasound, etc.) “on the fly”, and to make diagnosis rapidly.

# 5. Project Objectives

## 5.1 Speed

To make this application really useful, it is important that the image can be seen in "real time". For our project, this means that the user will be able to rotate the 3-D image flawlessly, and go to any view angle directly in a very short period of time.

## 5.2 Image Quality

When a doctor is trying to make a diagnosis, it is important to have the most accurate information possible. Keeping this in mind, the image must convey the information clearly and precisely. The resolution of the image must be high enough for the viewer to distinguish fine details, and the dynamic range must be broad enough for subtle gradations to show up fully.

## 5.3 Usefulness

When designing the program we must keep the user in mind at all times. To make this application useful to a doctor, certain options must be included. The image must be able to be rotated to any angle. This angle could be entered directly or incrementally from the current view. Also, the interface must be easy to understand when the doctor is trying to manipulate the image.

# 6. Program Overview

## 6.1 Data Flow

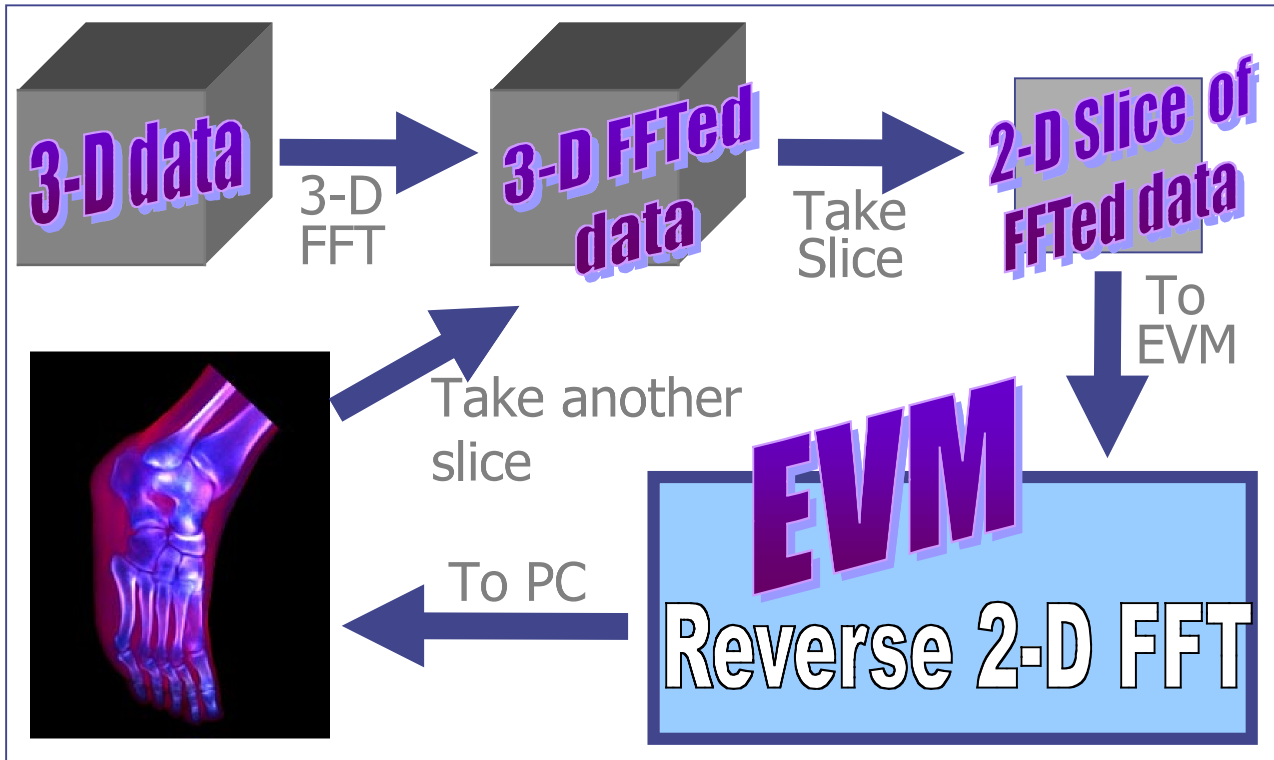



*Figure 2: Data Flow Chart*

The raw 3-D data set is first read into our program and is then sent to the FFT function, which performs a 3-D forward FFT and returns a 3-D object with frequency domain data. A slice is taken from this 3-D data using the slicer function. The slicer function also performs interpolation and resampling, and returns a 2-D object with frequency domain data. This data is then sent to the FFT function again where a reverse 2-D FFT is performed. The result of the previous steps is a volume rendered image. To see the image at a different angle, another slice is taken by the slicer function and then the same steps as previously

explained are repeated. A fast implementation of this cycle will lead to an image able to smoothly revolve as if seen in "real-time".

### 6.2 FFT function

A wide variety of FFT implementations exist in C code for distribution on the web. The code we chose to implement was presented in a previous homework assignment and was selected primarily for its versatility. Our task required both a forward 3-D FFT as well as an inverse 2-D FFT operation. It would have been more difficult to get our project to work with two different sets of code and would have made our project much larger. Another reason we decided to use the code was its stability and familiarity. We found the code to be stable and easy to understand when we used it earlier in the semester, and we became familiar with the code in the process of completing homework two. A final reason we decided to use this particular code was the fact that the FFT is computed with imaginary and real values contained in separate arrays rather than being interleaved in a single array. This attribute was useful because we only have real valued numbers in our data sets.

The FFT code we used is called in the following manner:

**int fftn (int ndim, const int dims[], REAL Re[], REAL Im[], int iSign, double scaling);**

where

- The integer value *ndim* contains the number of dimensions used (in our case 2 or 3).
- The integer array *dims* holds the dimensions of the actual data set (in our case either {256,256,128} or {256,256}).
- The arrays *Re* and *Im* hold the real and imaginary values of the data. As the raw data was all real, we inputted this information into the "Re" array, while filling the "Im" array with all zeros for the 3-D FFT.
- The integer value *iSign* determined whether the FFT was forward (1) or reverse (-1). We used both options in the course of the volume rendering procedure.
- The double value *scaling* is a normalizing constant by which the final result is divided. As all of our gray-scale voxel data fell between zero and one, scaling values down to be contained within this range was very important for the sake of image contrast and clarity.

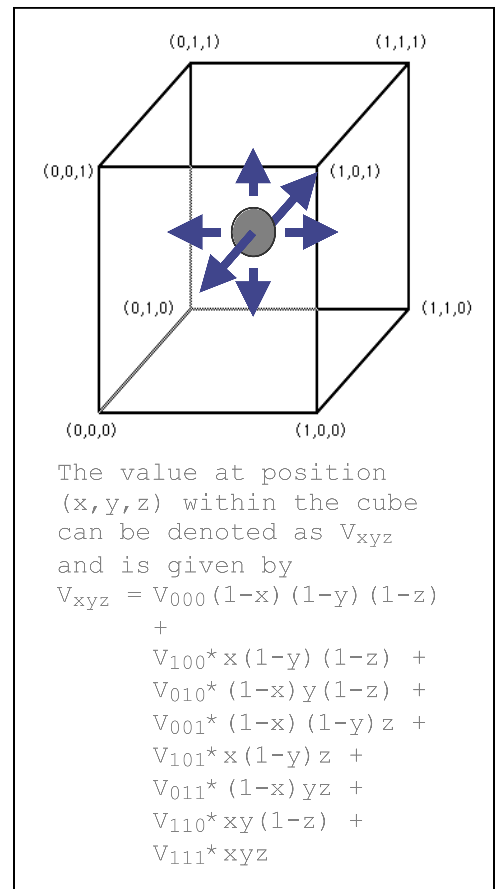
### 6.3 Slice Function

The slice algorithm developed is called in the following manner:

**double slicer(double *d, double *slice, int xsize, int ysize, int zsize, float *sliceview);**

where

- The double array *d* represents the array of data
- The double array *slice* is the sliced output of the function
- Integer values *xsize*, *ysize*, and *zsize* are the dimensions of dataset *d*
- The float array *sliceview* contains the coordinates of the vector orthogonal to the requested viewing plane




The value at position (x,y,z) within the cube can be denoted as $V_{xyz}$ and is given by

$$V_{xyz} = V_{000}(1-x)(1-y)(1-z) + V_{100}*x(1-y)(1-z) + V_{010}*(1-x)y(1-z) + V_{001}*(1-x)(1-y)z + V_{101}*x(1-y)z + V_{011}*(1-x)yz + V_{110}*xy(1-z) + V_{111}*xyz$$

Before taking an inverse 2-D FFT on the post-3-D FFT dataset, we first set out to design an algorithm to extract a two-dimensional slice from any possible angle. As only a 256x256x128 sample of the dataset is available, not all values needed for the slice will already exist during the slice construction. Therefore, this task should be divided into two parts: slice extraction and point interpolation.

Our user will define the desired slice angle by choosing the array orthogonal to that plane. According

to the Fourier Projection-Slice Theorem (see Section 3 for further discussion), any slice taken prior to the IFFT must contain the central point in the dataset (referred to from this point on as the origin) in order to return the appropriate final image. Thus, in order to extract a slice on the XY-plane around the origin, the user would input the (x,y,z) vector (0,0,1). Using a series of simple matrix algebra cross product calculations, a pair of perpendicular unit vectors on the desired plane can be found and used as the basis of determining the remaining points in the slice.

Once appropriate points in the 3-D dataset are selected to be part of the slice, pixel values not already defined through the original data and calculations must be interpolated. By definition, interpolation means to estimate the value of a function or series between two known values. According to Westenberg, "Interpolation is the most critical step in Fourier rendering, and good interpolation functions are needed to avoid artifacts such as aliasing and dishing [ a hill-shaped weighting artifact resulting in reduced intensities away from the center of the image] ." Many interpolation algorithms exist, each with its own advantages, disadvantages, and optimal uses. While algorithms such as cubic interpolation and POCS (Projection on Convex Sets) filters were considered, we selected the slightly

less precise but simple tri-linear interpolation algorithm due to the memory allocation and speed concerns these presented. This algorithm is known to be easy to implement in hardware as well as computationally very simple and fast. Tri-linear interpolation uses a weighted average of the eight surrounding density/pixel values included in the post-FFT dataset to estimate the value of the desired point. The algorithm makes many assumptions, most notably the supposition that the rate of change between any two pixels is constant. In low-resolution images this will create inaccuracies, but in the higher 256x256x128 resolution used inaccuracies should be limited.

**Additonal Considerations:** Aliasing is caused by insufficient sampling. Zero-padding separates the replicas in the spatial domain and decreases sampling distance in the frequency domain. Therefore, a way to reduce aliasing might have been to pad the data in the spatial domain with zeros before the initial 3-D FFT is taken. Our primary motivation for not implementing zero-padding again related to memory. Optimal padding would pad the dataset by at least $\sqrt{3}$, or around a factor of 2, in each dimension ($\sqrt{3}$ being the diagonal distance through a unit cube). For volume data, zero-padding by a factor of two yield a minimum memory requirement of eight times the original requirement. As memory size was an issue even for the original set, zero-padding this set in the spatial domain was not considered wise.

**6.4 User Interface**

Our project included the development of a user interface which was responsible for initializing a windowing environment on the PC, opening a window and displaying the rendered image, and allowing interaction with the user. This interface was built using OpenGL, a simple C library extension that allows graphics to be displayed on the PC.

Our interface allowed the user a number of controls over the rendering. First, the user could change the brightness and contrast of the displayed image. The user also had control over the viewing angle. Through a number of keyboard commands, the user could select from one of six pre-selected viewpoints, and could also independently rotate the view in any of the three dimensions. When entering any of these keyboard commands, the user’s desired view was rendered by first taking a new slice through the data at the selected angle, and then transferring this slice to the inverse FFT, and finally to the PC for display.

# 7. Implementation on the EVM

## 7.1 What is on the EVM?

Only one part of our program is implemented on the EVM. The EVM takes in a slice of 3-D FFTed data, performs a reverse 2-D FFT on it, and returns this information to the PC.

## 7.2 Why is there not more on the EVM?

The greatest advantage of implementing procedures on the TI 67 EVM over strictly software approaches is the inherent parallelism in the EVM's operations; operations carried out on the EVM board or DSP chip are far faster than external operations on the PC. As is common for frequency domain volume rendering tasks, the memory issues limit the EVM's usefulness. The on-chip memory is a mere 128 Kbytes total between both program and data memory, and even the on-board memory on the EVM is only 8 Mbytes, far to small to store the 256*256*128*8 bytes long dataset.

Although the (one-time) 3-D FFT operation is the most costly in terms of speed, the time-complexity of the entire frequency domain volume-rendering task is dominated by the (repeated) slicing, interpolation, and inverse 2-D FFT operations. Of these, the IFFT is computationally fairly costly. As only two

slices (one real and one imaginary) of 256x256 voxel data are needed to perform this procedure, the IFFT can be performed on the 8 MB EVM board without facing major memory issues. Additionally, the slicing algorithm deals only with vector multiplications without requiring access to the 3-D dataset, and therefore could easily be conducted on the EVM. It is the interpolation algorithm between these two procedures that presents new unavoidable issues and prevents. As discussed in Section 6.3, once the location of specific points needed in the desired slice is determined, tri-linear interpolation finds the surrounding eight points from the post-FFT 3-D dataset and estimates a voxel value weighted by the chosen point's distance from each of the surrounding points. In other words, in order to interpolate, the system must have immediate access to the dataset, which at 256*256*128*sizeof(double) is far too large to store either on-chip or on-board. Reading from and writing to external memory is very costly at around 15 cycles per access, minimized to 15 per total transfer when using DMA (Direct Memory Access) methods. For this reason it is desirable to limit the number of transfers between the hardware and the software.

After consideration these facts and as we hoped to limit the number of times data would need to be transferred between the PC

and the EVM, we opted to compute the IFFT on the EVM and keep all remaining procedures in external memory.

### 7.3 Memory allocation

**On-board memory:**

| **Variable Name** | **Description** | **Type** | **Dimension** | **Size (bytes)** |
|---|---|---|---|---|
| *frame* | (real) Post-3-D FFT graphic slice | double (8 bytes) | 256x256 | 524288 |
| *imag* | (imaginary) Post-3-D FFT graphic slice | double | 256x256 | 524288 |
| *dim* | Contains values of X_SIZE & Y_SIZE (needed for IFFT) | int (2 bytes) | 1x2 | 4 |
| | | | Total: | 1,048,580 |

**External memory:**

| **Variable Name** | **Description** | **Type** | **Dimension** | **Size (bytes)** |
|---|---|---|---|---|
| *data* | Original graphic information; also stores post-FFT real data | double (8 bytes) | 256x256x128 | 67108864 |
| *zeros* | Zero-filled array, used as imaginary input data for 3-D FFT, also stores post-FFT imaginary data | double | 256x256x128 | 67108864 |
| *dim* | Contains values of DIM1, DIM2, & DIM3 (needed for FFT) | int (2 bytes) | 1x3 | 6 |
| *slice* | 2D slice from post-FFT *data* | double | 256x256 | 524288 |
| *zeros2* | 2D slice from post-FFT *zeros* | double | 256x256 | 524288 |
| *slice2* | Absolute values of *slice*, converted to floats | float (4 bytes) | 256x256 | 262144 |
| *rawslicex, rawslicey, rawslicez, rawslicexy, rawslicexz, rawsliceyz* | *Slices of original 3D dataset taken at 6 predetermined angles prior to FFT* | float float float float float float | 256x256 256x256 256x256 256x256 256x256 256x256 | 262144*6 = 1572864 |
| | | | Total: | 1,371,011,318 |

### 7.4 Timing breakdown

| Part of Program | Software-only completion time(s) | Software-hardware completion time (s) |
|---|---|---|
| ReadFile() | 0.82 | 0.82 |
| CalcFFT() *(3-D FFT)* | 67.51 | 67.51 |
| slicer() | 7.63 | 7.63 |
| Transfers to and from EVM, 2-D IFFT | 0.33 | 6.43 |
| | | |
| Total runtime | 96.28 | 102.38 |

Analyzing the timing results, it appears that the advantages of running the inverse 2-D FFT on the EVM were overbalanced by the additional cycles needed to transfer the data to and from the EVM, resulting in a final speed decrease from the software-only implementation.

## 8. Problems Encountered

### 8.1 Hartley vs. FFT

At the start of our project we were planning on using the Hartley Transform to complete the two parts of the project that the FFT now takes care of. We came across many studies and reports that praised the Hartley Transform for its speed. "It turns out that using the Hartley transform to compute the Fourier transform is faster than computing the Fourier transform directly" (Scott, 2000). The faster speed of the Hartley is a result from the fact that the transform only uses real-valued numbers, unlike a FFT that uses both real and imaginary values. Because the Hartley Transform doesn't return imaginary results,

it can only really be used for real-world applications. This wasn't a problem for our project because our 3-D data sets are only made up of real numbers. Since there would be half the numbers needed to perform the Hartley Transform compared with the FFT, there would also be significant memory savings for our project (Theußl, 1999).

All of the advantages associated with the Hartley Transform seemed to make it a perfect fit for our project. However, we had two problems with this algorithm that led us to eventually choose an FFT implementation instead. The first problem surfaced when we came across an article by Kumar and Deo, in which the two researchers studied the performance of the 2-D and 3-D Hartley Transform on a parallel system. Their findings in essence stated that there was no clear advantage in using the Hartley Transform over the FFT when using a parallel system. "Because of complex computation graphs inherent in Hartley transforms, the inter-processor communication overheads for parallel DHT [Discrete Hartley Transform] are excessive, rendering it unsuitable for fine-grained parallelism" (Deo, Kumar 1995). Since we wanted to exploit the parallel attributes of the EVM, we felt it was not necessary to use the Hartley instead of the FFT. Our second problem with the Hartley Transform occurred when we were searching for usable code. We

found it difficult to find any working code, let alone working C code. As a result of the study and the lack of working code available, we decided to use a FFT instead of the Hartley Transform for our project.

**8.2 Memory**

Another major problem we ran into dealt with memory issues. We were dealing with huge data sets as large as 256 x 256 x 256. Storing all this information internally was impossible: the data set array taking up approximately 67MB of space while the EVM having only 8MB worth of space. We were able to get one "slice" of the data on the EVM at a time for the reverse 2-D FFT, however, the 256 x 256 slice was too large to fit on the on-chip memory (each slice equals approximately 525kB and on-chip memory equaling 64kB), so it had to be put on the off-chip EVM memory. It was not possible to use paging, as we used in lab, because all the data was needed at one time in order to complete the 2-D FFT. The transfer from the off-chip to on-chip memory and the transfer of data from the PC to the EVM cost a great deal of time when running our program.

# 9. Results

## 9.1 Example Output

Below is a sample of the graphic output upon running the FDVR code. The image to the left is a raw slice from ultrasound data, and opposite is the same view of the volume- rendered image.

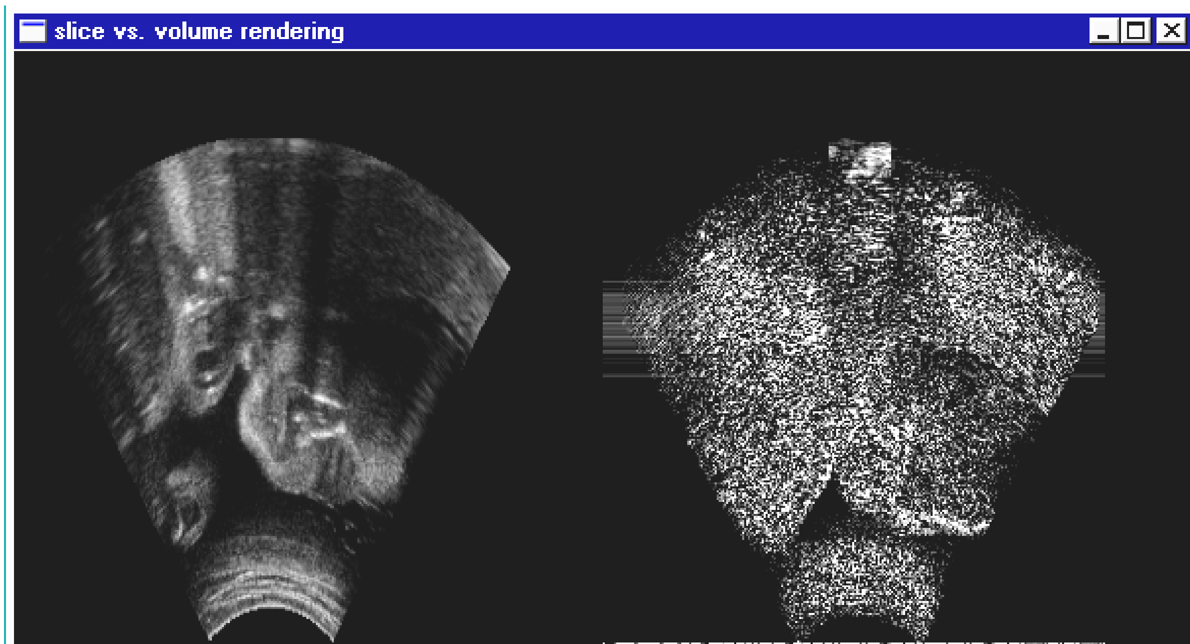



*Figure 4: Data Slice vs. Volume Rendering*

The views have many similarities as well as differences. Both images have density concentrations in similar areas and share the same basic shape, reflective of the angular sweep taken to gather the original ultrasound data. The rendered image on the right is less clear and smooth than the raw image. Additionally, a number of horizontally striped artifacts exist through the center of the rendered image. Many of these inconsistencies can be attributed to the less-precise interpolation method and issues that come into play when attempted to estimate values in

the frequency domain. Additionally, the lack of a depth-cueing mechanism which could differentiate between densities at closer and further distances contributes to the image quality. Overall however, the stipulations of the Projection-Slice Theorem held up and we were able to produce useful 2-D data at any angle and in a brief rendering time given a 3-D image dataset.

# 10. Future Project Improvements

### 10.1 Speed Improvements

Speed improvement could most reasonably come from moving a more significant portion of the rendering process over to the EVM. While our implementation only performs the final inverse two-dimensional transform on the EVM, the rendering speed for the entire application could likely be improved significantly if we were to also perform the slicing and interpolation functions on the EVM. To perform the entire operation on the EVM would not be practical, because of the memory issues resulting from the use of large datasets.

A second speed improvement that could be made would be to optimize the code to take advantage of the parallel processing capabilities of the C67. This could be done as two parts. One part would be to implement the reverse two-dimensional FFT to

take advantage of the C67's parallelism.  The second part would be to try and do the same for the slicing and interpolation algorithms.

### 10.2 Image Quality Improvements

Currently, though it is clear from looking at the rendered image that it is indeed a projection of the three-dimensional data set, the picture itself is blurred and speckled, making it difficult to discern any small details.  There are two possible improvements that could be made to our design to produce higher quality images: depth-cuing and directional shading algorithms, and more complex sampling filters.

As stated above, in frequency domain volume rendering there is no occlusion of objects that are "further" from the observer by objects that are "closer" and lie in front.  This is one of the main features that differentiates those images created by ray casting from those created through frequency domain volume rendering.  One way to try and regain that lost distance information is to run the data through a depth cueing algorithm which "weights" those pixels that are closer to the viewer to a greater degree than it weights those which are further away. The depth-cuing and directional shading can be accomplished either through spatial pre-processing, or through frequency

domain differentiation and multiplication, respectively. (Totsuka, 1993) Using these techniques will make the viewer's eye better able to interpret the three-dimensional data in a way that is meaningful for diagnoses.

A second way to improve image quality is through the use of a more complex interpolation filter. Currently we are using a simple tri-linear interpolation in our slicing algorithm. By choosing an algorithm that more completely takes into account the values of the surrounding pixels, we can better the resolution and quality of the rendered image. One option is to use the method of Projection on Convex Sets (POCS). (Nishita, 1986) This can be used with filter sizes of 3x3 and 5x5.

Finally, a number of smaller improvements could be made to the user interface. Once the rendering process had been sped up, a mouse-based viewpoint control would be very useful. This type of control would allow the user to smoothly rotate the rendered image in three dimensions in a natural way, simply be moving the mouse. Menu commands could also be added to allow the user to select and load any of a number of available datasets. Finally, a feature could be added to allow the user to save individual rendered images to the PC, and also to print them.

# References

Deo, Narsingh and Nishit Kumar. “Implementing 2-D an 3-D Discrete Hartley Transforms on a Massively Parallel SIMD Mesh Computer.” 1995.

Malzbender, T. (1993) Fourier Volume Rendering. ACM Transactions on Graphics, 12(3):233-250, July 1993

Nishita, T. (1986) Continuous Tone Representation of Three-Dimensional Objects. Computer Graphics, Volume 20, Number 4, 125-132, 1986.

Scott, Robert. “Doing Hartley Smartly”. www.embedded.com. 2000.

Theußl, Thomas., Robert F. Tobler and Eduard Gröller. “The Multi-Dimensional Hartley Transform as a Basis for Volume Rendering”. 1999.

Totsuka, T. (1993) Frequency Domain Volume Rendering. In James T. Kajiya, editor, Computer Graphics, volume 27, 271-278, August 1993

Westenburg, Michel. "Frequency Domain Volume Rendering by the Wavelet X-ray Transform", IEEE Transactions on Image Processing. Volume 9, No. 7, July 2000.

# Code and Data Sources

**FFT:**
Fortran code by:
RC Singleton, Stanford Research Institute, Sept. 1968
Translated by f2c (version 19950721).

Revisions:

| | |
|---|---|
| 26 July 95 | John Beale |
| 28 July 95 | Mark Olesen <olesen@me.queensu.ca> |
| 31 July 95 | Mark Olesen <olesen@me.queensu.ca> |
| 1 Aug 95 | Mark Olesen <olesen@me.queensu.ca> |

**TRI-LINEAR INTERPOLATION FUNCTION:**
C code from the article
"Tri-linear Interpolation"
by Steve Hill, sah@ukc.ac.uk
in "Graphics Gems IV", Academic Press, 1994

**DATA SETS:**
Paul Detmer and Jing-Ming Jong
Advanced Technology Laboratories
Bothell, WA