Fourier Volume Rendering

final version

Barthold B.A. Lichtenbelt
Hewlett Packard Laboratories
Visual Computing Department
1501 Page Mill Road
Palo Alto, CA 94304
United States
phone : +1-415-8574068
fax : +1-415-8523791
email : barthold@hpl.hp.com

Keywords: Fourier, volume rendering, graphics

November 8, 1995

Abstract

In this report a direct volume rendering technique called *Fourier Volume Rendering* (FVR) is explored in depth. FVR computes projection of a three dimensional dataset. Conventional volume rendering methods have a complexity of $O(N^3)$. FVR has a complexity of $O(N^2 logN)$ and thus can be computed much faster.

The mathematical theory behind FVR is explained first. Then FVR itself is discussed, and an example is given. The reason why the Hartley transform was chosen instead of the Fourier transform is explained. Finally a possible real time implemention in hardware is discussed. The last chapter handles about a completely different subject. It is about coronary artery extraction out of a three dimensional dataset of the heart obtained by MRI.

Images computed using FVR look like X-ray pictures. FVR is a fast method for computing projections. A hardware implementation mainly concerns a resampling chip that can resample a plane out of a three dimensional dataset in 150 ns. This chip should use a resampling filter of size $5 \times 5 \times 5$, which needs 64 voxels to compute one sample point.

List of Abbreviations

1-D	One Dimensional
2-D	Two Dimensional
3-D	Three Dimensional
CAT	Computer Aided Tomography
DFT	Discrete Fourier Transform
DHT	Discrete Hartley Transform
FFT	Fast Fourier Transform
FHT	Fast Hartley Transform
FVR	Fourier Volume Rendering
MIP	Maximum Intensity Projection
MRI	Magnetic Resonance Imaging
RAM	Random Access Memory
VLSI	Very Large Scale Integration

List of Figures

Contents

1 Introduction

Volume visualization is the term used for all possible ways to represent a three dimensional dataset on a two dimensional plane, e.g. a computer display. It is concerned with the representation, manipulation, and rendering of volumetric data. Volume visualization algorithms can be divided into two main categories: *Surface rendering methods* and *volume rendering methods*.

Volume rendering is the term used for the conversion of three dimensional datasets into two dimensional images, without use of an intermediate geometrical description of the three dimensional dataset, as used in traditional computer graphics. Traditional computer graphics represent three dimensional objects as geometric surfaces and edges, approximated by polygon and lines. This is called *surface rendering*. A popular surface rendering algorithm is Marching Cubes, first published in [10].

Volume rendering is a direct display of the voxels in the three dimensional dataset. It does not assume any existing structure in the dataset, as surface rendering methods do. Any part of the dataset, including interior parts, normally not visible, can be viewed with volume rendering. It does not require substantional preprocessing, although relevant components in the dataset should be identified during a process called *classification*. See also [8].

Surface rendering assumes some structure in the dataset to be viewed. Thus one is rendering a representation of this dataset instead of the data itself. By trying to find some structure in the dataset, errors might be introduced. A surface is either present or it is not. This binary classification scheme might display a surface where there is none, or vice versa. For e.g. medical applications this is not desirable. Doctors want to view the raw data itself, and not some representation. This is why direct volume rendering methods often are used in medical imaging.

Volume rendering methods can be further divided into a *screen space* approach, and an *object space* approach. For the screen space approach a ray is cast into the three dimensional dataset, and samples along the ray are taken at evenly spaced intervals. These samples are composited into a pixel value by applying some function to those samples. See again [8]. In the object space approach the three dimensional dataset is traversed either front to back or back to front, and each sample is used to compose the final pixel value.

A problem with volume rendering in general is its complexity of $O(N^3)$ for a $N \times N \times N$ dataset, since the whole three dimensional dataset has to be traversed. Some smart algorithms use the fact that a typical dataset consist of a lot of empty space, or stops processing a cast ray when the maximum pixel value has been reached before the ray exits the dataset. The drawback is that those algorithms depend on the structure of the three dimensional dataset, and their complexity still remains $O(N^3)$. Typical rendering times on a fast graphics workstation using the above optimizations for a 256x256x225 dataset range from 1-4 seconds, see [7]. In Section 4.10 more benchmarks are given.

This report discusses one certain volume rendering algorithm, called *Fourier Volume Rendering*, first published in [12]. This algorithm reduces the complexity to $O(N^2 logN)$. Fourier Volume Rendering computes projections of the three dimensional dataset. This is done by first transforming the dataset into the frequency domain by either a 3-D Fast Fourier Transform (FFT) or the Fast Hartley Transform (FHT). See Chapter 2. Then projection images can be quickly generated at any viewing angle by resampling along a plane perpendicular to the viewing direction, and taking the inverse 2-D transform of this resampled plane. The (computing intensive) 3-D FFT has to be evaluated only once. After that is done only the resampling step and the inverse 2-D FFT have to be performed to generate a projection.

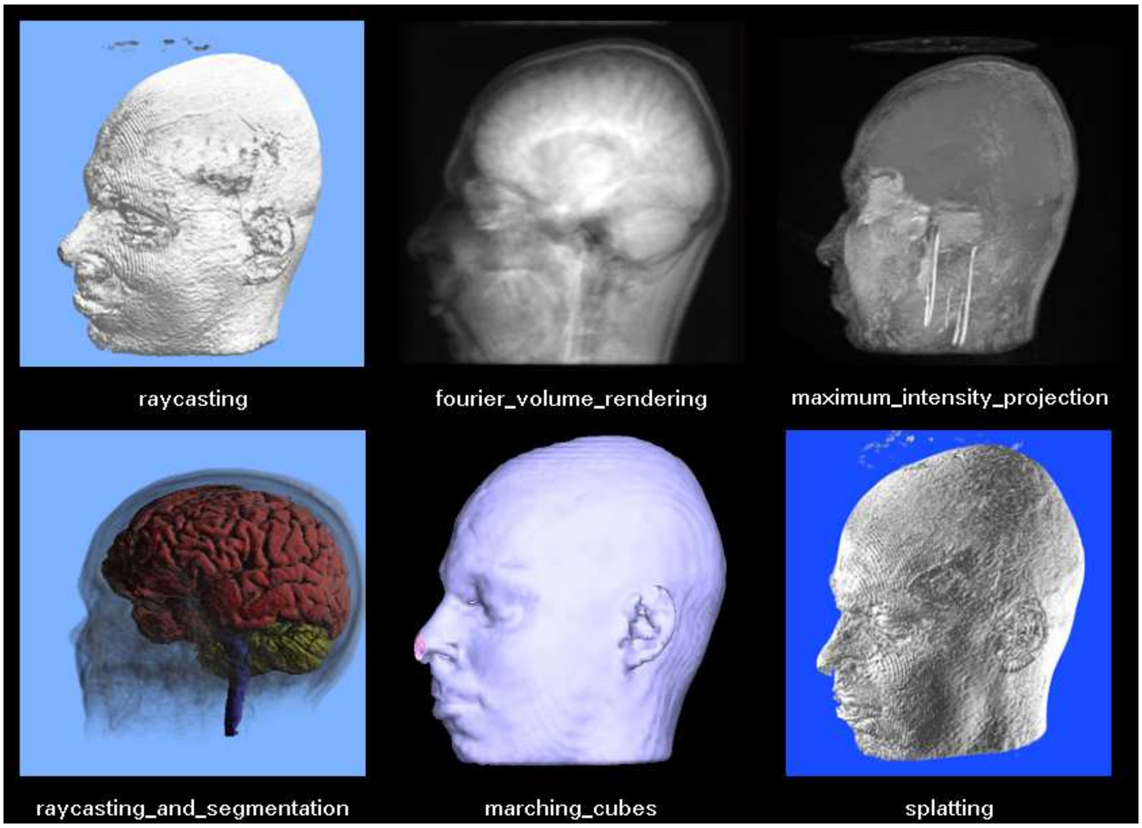

Figure 1.1: *Five different volume visualization methods. The lower left image was segmented by hand. The lower middle image is the only image rendered with a surface rendering technique.*

For more information about volume visualization see the excellent tutorial [6], which provides many hours of reading pleasure. To get an idea of what volume visualization is, see Figure 1.1.

Raycasting is the name of the method briefly mentioned before. A ray is cast through the dataset, and samples are taken along the ray. Those samples are composited into a pixel value and projected on the screen. Classification of the dataset is important. It determines what will be viewed. In Figure 1.1 upper left image the dataset was classified so that the skin is visible. In the lower left image the data was classified so that the interior of the skull is made visible. On top of this different parts of the brains were segmented by hand. Note that the lower left image was obtained from a different dataset. The other five images were all obtained using the same dataset.

The rest of this report will discuss Fourier Volume Rendering, of which an example is shown in the middle upper image.

Maximum Intensity Projection is very similar to raycasting. Now only the maximum sample value found on one ray is converted into a pixel value and displayed to the screen.

The Marching Cubes algorithm is a surface rendering method. It first divides the skin into small polygons which then are rendered and displayed to the screen.

The lower right image was obtained by using another direct volume rendering method called splatting. It is an object space front to back algorithm. It is called splatting because it can be seen as throwing a voxel at the screen. The contribution to the final image will be high at the center of the impact and will drop off further away from it. Splatting was first published in [20].

2 Theory of Fourier Volume Rendering

This chapter discusses the Fourier and Hartley transforms and their properties, as far as they are relevant to understand Fourier Volume Rendering (FVR), which is discussed in the next chapter. This is by no means meant to be a complete discussion of Fourier and Hartley transforms. See [1], [18], [4], [2] and [16] for an in depth discussion.

A physical process can be described either in the time domain or in the frequency domain. In the time domain the process is described as some quantity h as a function of the time, e.g. $h(t)$. In the frequency domain the same process is described as its amplitude H as a function of the frequency f, e.g. $H(f)$. H in general will be complex, indicating phase too.

2.1 The Fourier transform

Fourier's theorem states that it is possible to form a function $h(t)$ as a summation of a series of sine and cosine terms of increasing frequency. The Fourier transform of the function $h(t)$ is written $H(f)$ and describes the amount of each frequency term that must be added together to make $h(t)$.

2.1.1 The continuous Fourier transform

$h(t)$ and $H(f)$ can be thought of as two different representations of the same function. One goes back and forth between these representations by means of the Fourier transform equations:

$$H(f) = \int_{-\infty}^{\infty} h(t)e^{2\pi ift}dt \tag{2.1}$$

$$h(t) = \int_{-\infty}^{\infty} H(f)e^{-2\pi ift}df \tag{2.2}$$

If t is measured in seconds then f is measured in cycles per second, or Hertz. Likewise, if t is measured in meters then f is measured in cycles per meters. The equations 2.1 and

2.2 are linear equations. Thus the Fourier transform is a *linear* operation. That means that e.g. the transform of the sum of two functions is equal to the sum of the transforms of each function.

A shorter notation for the pair 2.1 and 2.2 is:

$$h(t) \Leftrightarrow H(f) \tag{2.3}$$

Some basic knowledge by the reader of the Fourier transform is assumed. The properties of the Fourier transform used in this report are described in the subsequent sections. For more information about the Fourier transform see [16], [18], [4] and [1].

One important property used in Fourier Volume Rendering is the *Convolution Theorem*. it states that:

$$g * h \Leftrightarrow G(f)H(f) \tag{2.4}$$

or in words, the Fourier transform of the convolution of two functions g and f described in the time domain is equal to the product of their individual Fourier transforms $G(f)$ and $H(f)$.

2.1.2 Fourier transform of discretely sampled data

Often the continuous function $h(t)$ has to be processed by a computer. In that case $h(t)$ is being sampled. In most cases that will be done at evenly spaced intervals in time. let Δ denote the time interval between consecutive samples, and n the number of samples. The sequence of sampled values then is:

$$h_n = h(n\Delta) \quad n = \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \tag{2.5}$$

$(1/\Delta)$ is called the sampling rate. Related to the sampling rate is the *Nyquist critical frequency* defined as:

$$f_c \equiv \frac{1}{2\Delta} \tag{2.6}$$

The Nyquist critical frequency is important for two reasons. The first reason is known as *the sampling theorem*. This states that if a continuous function $h(t)$ happens to be bandwidth limited to frequencies smaller in magnitude than f_c, i.e. $H(f) = 0 \; \forall \mid f \mid \geq f_c$, then the function $h(t)$ is completely determined by its samples h_n as follows:

$$h(t) = \Delta \sum_{n=-\infty}^{\infty} h_n \frac{\sin[2\pi f_c(t - n\Delta)]}{\pi(t - n\Delta)} \tag{2.7}$$

This is quite a remarkable theorem. Fairly often one is dealing with a bandwidth limited signal. E.g. the signal might have passed an amplifier with a known finite frequency response. If this is the case it is possible to record the entire information content of the signal $h(t)$ by sampling at a rate $(1/\Delta)$ equal to twice the maximum frequency f_m passed by the amplifier. This can be seen by realizing that:

$$2f_m = \frac{1}{\Delta} \tag{2.8}$$

$$f_m = \frac{1}{2\Delta} = f_c \tag{2.9}$$

When the signal $h(t)$ is not bandwidth limited to f_c and it is sampled at the sampling rate $1/\Delta$, the (undesired) effect of *aliasing* comes into focus. This is the second reason why the Nyquist critical frequency is important. Aliasing means that all of the power spectral density that lies outside of the frequency range $-f_c < f < f_c$ is moved into that range. More about aliasing is explained after the next section. See also [16] page 113-115.

2.1.3 The discrete Fourier transform

Suppose $h(t)$ is sampled with N samples. Lets call these N points h_k.

$$h_k \equiv h(t_k), \quad t_k \equiv k\Delta, \quad k = 0, 1, 2, \ldots, N-1 \tag{2.10}$$

Now the question is what is the Fourier transform of h_k? Or differently phrased, can the Fourier transform of $h(t)$ using h_k be estimated? For simplicity it is assumed that N is even. Now try to estimate $H(f)$ at the following discrete points f_n in the range $-f_c$ to f_c.

$$f_n = \frac{n}{N\Delta}, \quad n = -\frac{N}{2}, \ldots, \frac{N}{2} \tag{2.11}$$

Note that the extreme values of n exactly correspond to the absolute value of the Nyquist critical frequency. By using 2.1, 2.10 and 2.11 the following is found:

$$H(f_n) = \int_{-\infty}^{\infty} h(t)e^{2\pi i f_n t}dt \approx \sum_{k=0}^{N-1} \Delta h_k e^{2\pi i f_n t_k} = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{2.12}$$

where the integral in 2.12 is approximated by a discrete sum. The final summation in 2.12 is called the *Discrete Fourier transform* (DFT) of h_k. It is called H_n:

$$H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{2.13}$$

Note that H_n is independent of Δ, the sampling interval. So in short the relation between the continuous time Fourier transform and its discrete counterpart is given by:

$$H(f_n) \approx \Delta H_n \tag{2.14}$$

Note that in Equation 2.11 n ranges from $-N/2$ to $N/2$. This is not a problem, because 2.13 is periodic in n with period N. This means that $H_n = H_{N+n}$ and that $H_{-n} = H_{N-n}$. This can be seen by considering that:

$$\begin{aligned} H_{-n} &= \sum_{k=0}^{N-1} h_k e^{-2\pi i k n/N} && (2.15) \\ H_{N-n} &= \sum_{k=0}^{N-1} h_k e^{2\pi i k (N-n)/N} = \sum_{k=0}^{N-1} h_k e^{-2\pi i k n/N} && (2.16) \end{aligned}$$

and because

$$e^{2\pi i k} = 1 \quad k = 0, 1, \ldots, N-1 \tag{2.17}$$

it follows that

$$H_{-n} = H_{N-n} \tag{2.18}$$

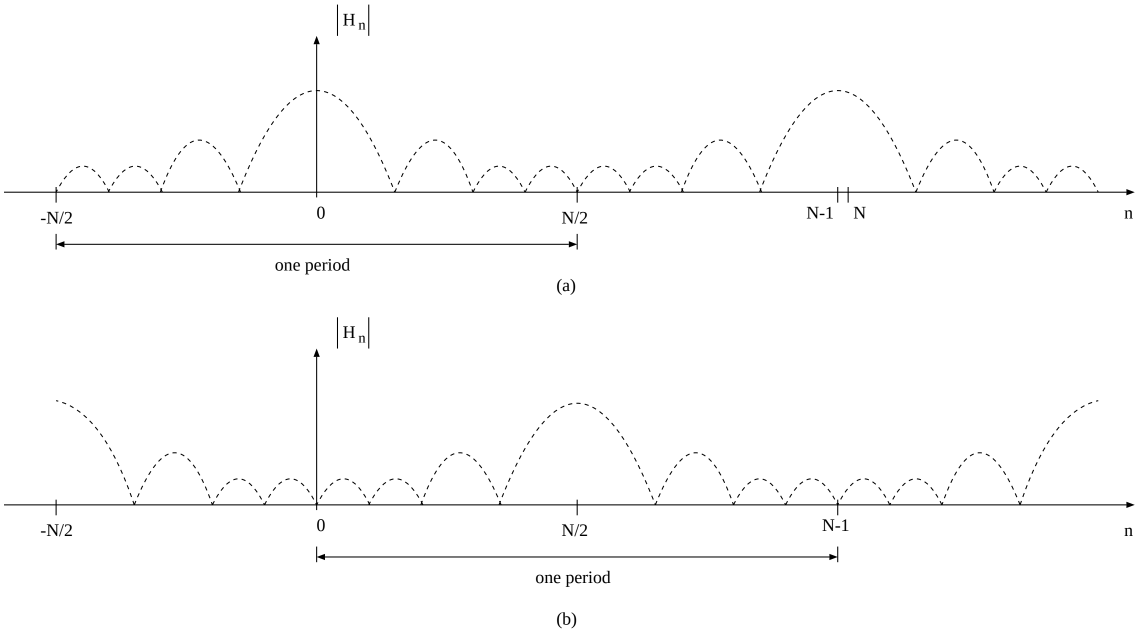

Figure 2.1: *Periodicity of the discrete Fourier transform. (a) Showing back to back half periods in the interval* $[0, N-1]$*. (b) Shifted spectrum showing a full period in the same interval.*

Because 2.13 is periodic in n with period N, n can as well range from 0 to $N-1$ instead of from $-N/2$ to $N/2$. See Figure 2.1(a). This is equivalent to shifting the window with which H_n is looked at with half a period. This is useful because now k in 2.10 as well as n in 2.13 vary exactly over the same range, and the mapping from the N numbers h_k to the N numbers in H_n is manifest. There is one pitfall though. Because of the shift with half a period, the frequencies corresponding to n in 2.13 no longer range from $-f_c$ to f_c. The frequency $f = 0$ now corresponds to $n = 0$. Positive frequencies in the range $0 < f < f_c$ correspond to $0 < n < N/2$, while negative frequencies $-f_c < f < 0$ correspond to $N/2 < n < N$. The value $n = N/2$ corresponds to both $f = f_c$ and $f = -f_c$. See Figure 2.1(a). This is not very convenient. To overcome this and display a full period, it is necessary to move the origin of the transform to the point $n = N/2$, as is shown in Figure 2.1(b). This can be done by multiplying h_k by $(-1)^k$ prior to taking the transform. (This is the frequency shift property, see [18] page 112 and [4] page 95)

The discrete Fourier transform of a real function is hermitian. This means that the real part of the transform is even and the imaginary part is odd. There is a redundancy of a factor of two in the DFT:

$$H^*_{N-n} = \sum_{k=0}^{N-1} h_k e^{-2\pi i k(N-n)/N} = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} e^{-2\pi i k} = H_n \tag{2.19}$$

Thus by storing only $N/2$ complex numbers the DFT of N real numbers is completely defined.

The *Inverse Discrete Fourier transform* is defined as:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi ikn/N} \tag{2.20}$$

2.1.4 Discrete convolution

In formula 2.4 the convolution theorem for the continuous case was defined. Here the *discrete convolution theorem* will be defined. If a signal s_j is periodic with period N, so it is completely determined by its values $s_0, \ldots, s_{N-1}$, then its discrete convolution with a function r_k of finite duration N has the following Fourier transform relationship:

$$s_j * r_j = \sum_{k=-N/2+1}^{N/2} s_{j-k} r_k \Leftrightarrow S_n R_n \tag{2.21}$$

S_n and R_n are the discrete Fourier transforms of the values s_j $(j = 0, \ldots, N-1)$ and r_k $(k = 0, \ldots, N-1)$ conform 2.13. Note that these values of j and k are the same as in Equation 2.21, where they run from $-N/2+1, \ldots, N/2$, but shifted as described in Paragraph 2.1.3.

The discrete convolution theorem assumes that the two signals are periodic and that those two signals are of the same length. That is often not the case. Real data is often not periodic, and the constraint that the two functions must be of the same length is not a convenient one. The latter case is easily overcome. If r_k is of length M smaller than N, simply extend r_k to length N by padding it with zeros. I.e. set $r_k = 0$ for $M/2 \leq k \leq N/2$ and for $-N/2 \leq k \leq -M/2+1$.

To be able to use the discrete convolution theorem on non periodic signals as well, the following preprocessing to the data s_j has to be done. Pad the data s_j with a number of zeros on one end equal to the maximum positive duration or negative duration of the function r_k, whichever is larger. (For a symmetric function r_k of duration M, one needs to pad s_j with $M/2$ zeros.) Combining this operation with the padding of r_k will effectively make it possible to use the discrete convolution theorem for non periodic data as well. In [16] page 541 this is explained in more detail.

2.1.5 Sampling and aliasing

Let the signal $h(t)$ have the Fourier transform $H(f)$. $h(t)$ is being sampled with period Δ. The resulting discrete signal is $g(k)$.

$$h(t) \Leftrightarrow H(f) \tag{2.22}$$

$$g(k) = \Delta h(k\Delta) \Leftrightarrow G(e^{jf\Delta}) \tag{2.23}$$

G is the frequency spectrum of the sampled $h(t)$. It can be shown that G is of the following form (see [18] page 113 and 114):

$$G(e^{jf\Delta}) = \sum_{n=-\infty}^{\infty} H(jf + jnf_0) \tag{2.24}$$

f_0 is the sampling rate $1/\Delta$. Equation 2.24 and Figure 2.2 show the consequence of sampling. Sampling can be seen as multiplying the original continuous function with a series of delta functions. This is shown in Figure 2.2(a). The frequency equivalent of sampling in time is shown in Figure 2.2(b). The frequency transform of the original continuous function is convolved with the transform of the series of delta functions (which again is a series of delta functions). The result is that the original frequency spectrum $H(f)$ is repeated infinitely. The frequency axis is cut into strips of length f_0, which then are superimposed to form G. Note that $H(f)$ can be reconstructed from $G(f)$ by multiplying $G(f)$ with a block function of width f_0.

If a signal, which is not bandwidth limited to the Nyquist frequency f_c, is being sampled, aliasing will occur. Aliasing means that power outside the range $-f_c < f < f_c$ is moved into that range. This is not desirable. Or in other words, if f is not bandwidth limited to $f_c = (f_0/2)$ then frequencies larger than f_c will be folded back into the range $-f_c < f < f_c$. One solution for this problem is to low pass filter the original signal $h(t)$ before the actual sampling takes place.

Aliasing also occurs when the sampling of $h(t)$ is not done quickly enough, although $H(f)$ is bandwidth limited. The $H(f)$s that form $G(f)$ will in this case partly overlap each other. See Figure 2.2(c) and 2.2(d) for an illustration of this effect. $H(f)$ no longer can be reconstructed from $G(f)$.

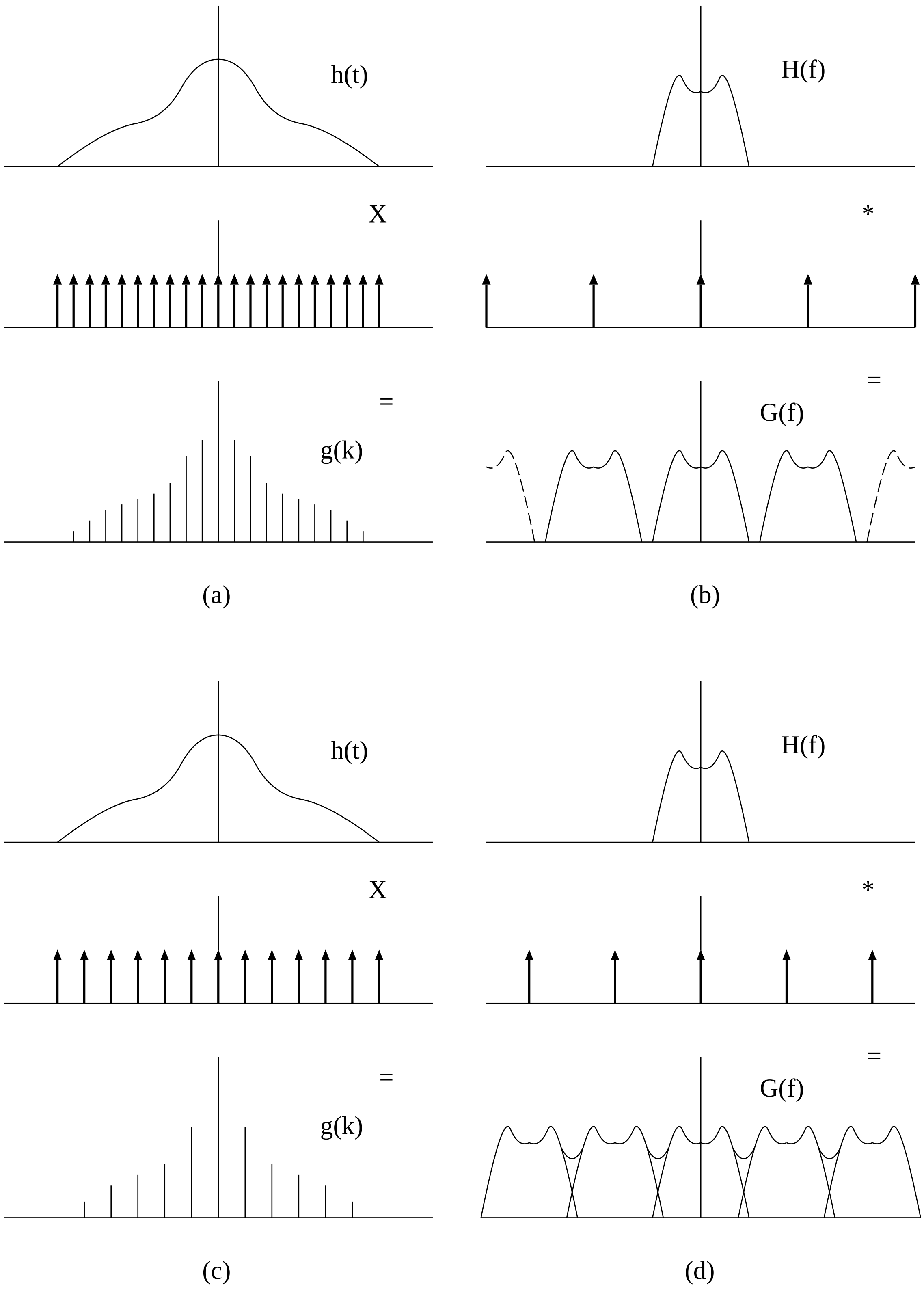

Figure 2.2: *Sampling a continuous function. (a) Sampling in time. (b) The frequency consequence of sampling.(c) Sampling in time, but the sampling is too slow. (d) The frequency result of sampling at a too slow rate. The copies of $H(f)$ overlap each other to form $G(f)$.*

2.1.6 Scaling

If h_k has a discrete Fourier transform H_n then $h_{\alpha k}$ has a discrete Fourier transform $H_{\frac{n}{\alpha}}$. This can be seen by considering that:

$$\sum_{n=0}^{N-1} h_{\alpha k} e^{2\pi ikn/N} = \sum_{n=0}^{N-1} h_{\alpha k} e^{2\pi i\alpha kn/\alpha N} = H_{\frac{n}{\alpha}}. \tag{2.25}$$

This means that compression of the time (or spatial) scale means expansion of the frequency scale and vice versa.

2.1.7 The 2-D and 3-D Fourier transform

The Fourier transform can be easily extended to the 2-D or 3-D case. In the 2-D case the discrete Fourier transform is:

$$H_{u,v} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h_{x,y} e^{2\pi i(ux/M+vy/N)} \tag{2.26}$$

for $u = 0, 1, 2, \ldots, M-1$ and $v = 0, 1, 2, \ldots, N-1$. The inverse transform is:

$$h_{x,y} = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H_{u,v} e^{-2\pi i(ux/M+vy/N)} \tag{2.27}$$

for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$. The Equations 2.26 and 2.27 are the most general case. Most of the time square images will be dealt with, i.e. $M = N$. This is assumed for the rest of this report.

The 2-D and 3-D transform can be performed separately in each direction. For a 2-D image for example, it would be possible to perform a 1-D Fourier transform on each horizontal scan line of the image, storing the intermediate result, and then perform another 1-D Fourier transform on the vertical scan-lines of the intermediate stored result.

The discrete Fourier transform in Equations 2.13 and 2.26 and the inverse in equations 2.20 and 2.27 are periodic with period N. For the 2-D case this means that:

$$H_{u,v} = H_{u+N,v} = H_{u,v+N} = H_{u+N,v+N} \tag{2.28}$$
$$h_{x,y} = h_{x+N,y} = h_{x,y+N} = h_{x+N,y+N} \tag{2.29}$$

This can be seen by substituting $(u + N)$ and $(v + N)$ in Equation 2.26, and the variables $(x + N)$ and $(y + N)$ in Equation 2.27.

The same observations as for the 1-D discrete Fourier transform in section 2.1.3 holds for the 2-D case. Again, to display one full period in the frequency domain just multiply $h_{x,y}$ with $(-1)^{x+y}$ prior to taking the 2-D transform. See also Figure 2.1 and [4] page 95. u and v then can run from 0 to $N - 1$, without mixing up the order in which the frequencies are traversed. See also [4] pages 95-97 and [18] pages 105-106.

The 2-D and 3-D DFT of a real function are hermitian too, just like the 1-D DFT. That means for the 2-D case that:

$$H^*_{-u,-v} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h_{x,y} e^{-2\pi i(-ux/M - vy/N)} = H_{u,v} \tag{2.30}$$

Which is the same as stating that $H_{-u,-v} = H_{N-u,N-v} = H^*_{u,v}$. Something similar goes for the inverse DFT:

$$h^*_{-x,-y} = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H_{u,v} e^{2\pi i(-ux/M - vy/N)} = h_{x,y} \tag{2.31}$$

Or, $h_{-x,-y} = h_{N-x,N-y} = h^*_{x,y}$.

2.2 Fourier Projection Slice Theorem

A projection is a mathematical operation, to be compared with taking a X-ray picture of a three dimensional object. The resulting X-ray picture is a two dimensional image which holds information about the density of the three dimensional object. See Figure 2.3.

The brick in the figure has three holes in it. If the holes where not to extend to the walls of the brick, those holes would not be visible for a human observer. By making a projection of the brick something can be said about its inner structure. The intensity of the projection is less at those places where the X-rays go through the holes. This idea is used in Fourier Volume Rendering.

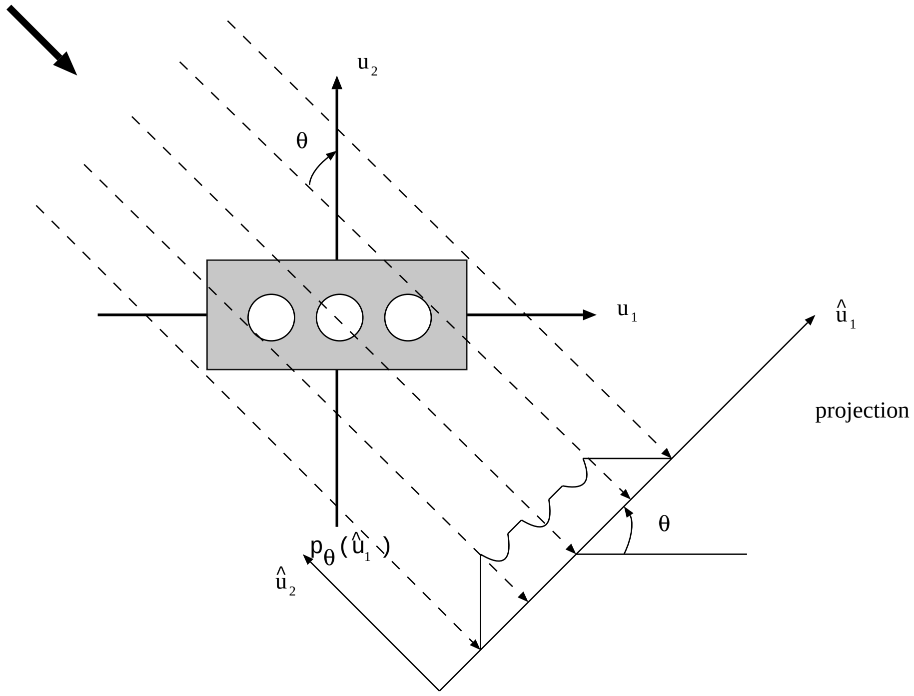

Figure 2.3: *Projection of a 2-D object*

2.2.1 The slice theorem

For simplicity reasons the slice theorem will be discussed for the one-dimensional and two-dimensional cases. Mathematically it is of course possible to consider projections of any dimension.

In words the Fourier projection slice theorem tells us that *the 1-D Fourier transform of a 1-D projection of a 2-D object at an angle θ, is a 1-D line across the 2-D Fourier transform of that 2-D object, at the same angle θ.*

Let the brick in Figure 2.3 be described by the unknown density function $x(u_1, u_2)$. By defining:

$$\hat{u}_1 = u_1 \cos\theta + u_2 \sin\theta \tag{2.32}$$

$$\hat{u}_2 = -u_1 \sin\theta + u_2 \cos\theta \tag{2.33}$$

the projection of $x(u_1, u_2)$ at angle θ is (see also Figure 2.3):

$$p_\theta(\hat{u}_1) = \int_{-\infty}^{\infty} x(u_1, u_2) d\hat{u}_2 = \int_{-\infty}^{\infty} x(\hat{u}_1 \cos\theta - \hat{u}_2 \sin\theta, \hat{u}_1 \sin\theta + \hat{u}_2 \cos\theta) d\hat{u}_2 \tag{2.34}$$

If enough of these projections are made at different angles θ, it is possible to use reconstruction techniques to compute the 2-D density function $x(u_1, u_2)$. One way to achieve this is through the use of the Fourier Slice Theorem.

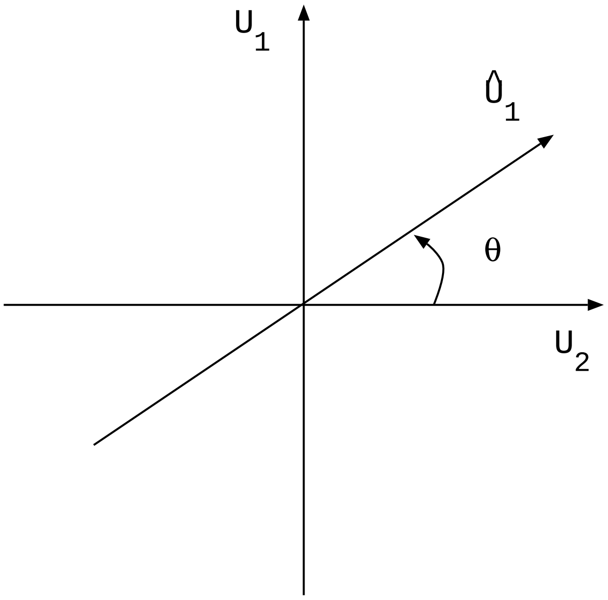

Figure 2.4: *The frequency plane* (U_1, U_2)

If $x(u_1, u_2)$ has a 2-D Fourier transform $X(U_1, U_2)$, the 1-D Fourier transform of $p_\theta(\hat{u}_1)$ will exist. This transform is denoted by $P_\theta(\hat{U}_1)$:

$$\begin{aligned} P_\theta(\hat{U}_1) &= \int_{-\infty}^{\infty} p_\theta(\hat{u}_1) e^{2\pi i \hat{U}_1 \hat{u}_1} d\hat{u}_1 & (2.35) \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x(\hat{u}_1 \cos\theta - \hat{u}_2 \sin\theta, \hat{u}_1 \sin\theta + \hat{u}_2 \cos\theta) e^{2\pi i \hat{U}_1 \hat{u}_1} d\hat{u}_2 d\hat{u}_1 & (2.36) \end{aligned}$$

Or by using the normal coordinate system (u_1, u_2):

$$P_\theta(\hat{U}_1) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x(u_1, u_2) e^{2\pi i \hat{U}_1 (u_1 \cos\theta + u_2 \sin\theta)} du_1 du_2 \tag{2.37}$$

or

$$P_\theta(\hat{U}_1) = X(\hat{U}_1 \cos\theta, \hat{U}_1 \sin\theta) \tag{2.38}$$

Equation 2.38 is known as the Fourier Projection Slice Theorem. It says that the Fourier transform of the projection taken at an angle θ is the 2-D Fourier transform of the unknown $x(u_1, u_2)$, i.e. $X(U_1, U_2)$, evaluated along a line passing through the origin of the (U_1, U_2) plane and making an angle θ with the U_1 axis. See also Figure 2.4.

From Equation 2.38 it is seen that knowledge of several projections of an object provides knowledge of the 2-D Fourier transform along lines in the Fourier plane. The problem of reconstructing the wanted $x(u_1, u_2)$ (or its Fourier transform $X(U_1, U_2)$) is thus equivalent to interpolating the whole 2-D Fourier transform from these 1-D samples. See also [2] pages 363-383 and [9] pages 42-49. This technique for example is used in Computer Aided Tomographic (CAT) scanners and is called tomographic reconstruction.

2.3 The Hartley transform

In 1942 R.V.L. Hartley presented a transform very similar to the Fourier transform. Given a real function $h(t)$ its Hartley transform will be real too. The Hartley transform equations are:

$$H(f) = \int_{-\infty}^{\infty} h(t)(\cos ft + \sin ft)dt \tag{2.39}$$

$$h(t) = \int_{-\infty}^{\infty} H(f)(\cos ft + \sin ft)df \tag{2.40}$$

Compare this to the Fourier transform:

$$S(f) = \int_{-\infty}^{\infty} s(t)(\cos ft + i \sin ft)dt \tag{2.41}$$

$$s(t) = \int_{-\infty}^{\infty} S(f)(\cos ft - i \sin ft)df \tag{2.42}$$

Let $H(f) = E(f) + O(f)$, where $E(f)$ and $O(f)$ are the even and odd parts of $H(f)$ respectively:

$$E(f) = \frac{H(f)+H(-f)}{2} = \int_{-\infty}^{\infty} h(t)\cos ftdt \tag{2.43}$$
$$O(f) = \frac{H(f)-H(-f)}{2} = \int_{-\infty}^{\infty} h(t)\sin ftdt \tag{2.44}$$

This is an interesting result. Given the Hartley transform $H(f)$ the Fourier transform $S(f)$ can be formed by adding $E(f)+iO(f)$.

$$S(f) = E(f) + iO(f) \tag{2.45}$$

The reverse is true too. Given the Fourier transform $S(f)$ $H(f)$ can be obtained by:

$$H(f) = Re[S(f)] + Im[S(f)] \tag{2.46}$$

For the reverse transforms something very similar can be done. Let $h(t) = e(t) + o(t)$, where $e(t)$ and $o(t)$ are the even and odd parts of $h(t)$ respectively:

$$e(t) = \frac{h(t)+h(-t)}{2} = \int_{-\infty}^{\infty} H(f)\cos ftdf \tag{2.47}$$
$$o(t) = \frac{h(t)-h(-t)}{2} = \int_{-\infty}^{\infty} H(f)\sin ftdf \tag{2.48}$$

Given the inverse Hartley transform $h(t)$ the inverse Fourier transform $s(t)$ can be readily formed by:

$$s(t) = e(t) - io(t) \tag{2.49}$$

Again, the reverse is true too. Given the inverse Fourier transform $f(t)$ the inverse Hartley transform can be obtained by:

$$h(t) = Re[s(t)] - Im[s(t)] \tag{2.50}$$

2.3.1 Hartley Projection Slice Theorem

The Slice theorem from holds for the Hartley transform as well. That will be proved here. Refer to section 2.2.1 also.

The Hartley transform of the projection $p_\theta(\hat{u}_1)$ is:

$$
\begin{aligned}
P_\theta(\hat{U}_1) &= \int_{-\infty}^{\infty} p_\theta(\hat{u}_1) cas(2\pi \hat{U}_1 \hat{u}_1) d\hat{u}_1 && (2.51)\\
&= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x(\hat{u}_1\cos\theta - \hat{u}_2\sin\theta, \hat{u}_1\sin\theta + \hat{u}_2\cos\theta)\cdot \\
&\quad cas(2\pi\hat{U}_1\hat{u}_1) d\hat{u}_2 d\hat{u}_1 && (2.52)
\end{aligned}
$$

Where $cas(x) = \sin(x) + \cos(x)$, an abbreviation adopted from Hartley. Equation 2.34 is used to get to this result. Or by using the normal coordinate system (u_1, u_2):

$$P_\theta(\hat{U}_1) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x(u_1, u_2) cas(2\pi\hat{U}_1(u_1\cos\theta + u_2\sin\theta)) du_1 du_2 \tag{2.53}$$

or

$$P_\theta(\hat{U}_1) = H(\hat{U}_1\cos\theta, \hat{U}_1\sin\theta) \tag{2.54}$$

Where $H(U_1, U_2)$ the Hartley transform of $x(u_1, u_2)$ is. Equation 2.54 is the Hartley Projection Slice Theorem.

2.3.2 The discrete Hartley transform

The discrete Hartley transform is defined by:

$$H_n = \sum_{k=0}^{N-1} h_k cas(2\pi nk/N) \tag{2.55}$$

The inverse discrete Hartley transform is almost the same:

$$h_k = \frac{1}{N}\sum_{n=0}^{N-1} H_n cas(2\pi nk/N) \tag{2.56}$$

As for the continuous case an even and odd part can be defined, and the discrete Fourier transform formed out of those:

$$
\begin{aligned}
H_n &= O_n + E_n && (2.57)\\
E_n &= \frac{H_n + H_{N-n}}{2} && (2.58)\\
O_n &= \frac{H_n - H_{N-n}}{2} && (2.59)
\end{aligned}
$$

These relations can be verified by substituting in Equation 2.55. The discrete Fourier transform is then given by:

$$S_n = E_n + iO_n \qquad (2.60)$$

2.3.3 Properties of the discrete Hartley transform

The discrete Hartley transform (DHT) is used in the actual software FVR implementation. Only the DHT properties important for the implementation are discussed in this section. These are:

- periodicity
- shifting
- convolution theorem
- scaling

The following relationship will be used to prove some of the theorems:

$$
\begin{aligned}
cas(a+b) &= \cos(a+b) + \sin(a+b) && (2.61)\\
&= \cos a \cos b - \sin a \sin b + \sin a \cos b + \cos a \sin b && (2.62)\\
&= \{\cos a + \sin a\} \cos b + \{\cos a - \sin a\} \sin b && (2.63)\\
&= cas(a) \cos b + \{\cos a + \sin -a\} \sin b && (2.64)\\
&= cas(a) \cos b + cas(-a) \sin b && (2.65)
\end{aligned}
$$

The DHT is periodic in N:

$$H_{N+n} = \sum_{k=0}^{N-1} h_k cas(2\pi(N+n)k/N) \tag{2.66}$$
$$= \sum_{k=0}^{N-1} h_k cas(2\pi k + 2\pi nk/N) \tag{2.67}$$
$$= \sum_{k=0}^{N-1} h_k cas(2\pi nk/N) = H_n \tag{2.68}$$

The last step was derived by using Equation 2.65.

If h_k has a DHT H_n then h_{k-a} has a DHT of $H_n \cos(2\pi na/N) + H_{-n} \sin(2\pi na/N)$. Proof:

$$DHT\ of\ h_{k-a} = \sum_{k=0}^{N-1} h_{k-a} cas(2\pi nk/N) \tag{2.69}$$
$$= \sum_{k=-a}^{N-1-a} h_k cas(2\pi n(k+a)/N) \tag{2.70}$$
$$= \sum_{k=-a}^{N-1-a} h_k cas(2\pi nk/N)\cos(2\pi na/N) + \sum_{k=-a}^{N-1-a} h_k cas(-2\pi nk/N)\sin(2\pi na/N) \tag{2.71}$$
$$= \sum_{k=0}^{N-1} h_{k-a} cas(2\pi n(k-a)/N)\cos(2\pi na/N) + \sum_{k=0}^{N-1} h_{k-a} cas(-2\pi n(k-a)/N)\sin(2\pi na/N) \tag{2.72}$$
$$= H_n \cos(2\pi na/N) + H_{-n}\sin(2\pi na/N) \tag{2.73}$$

This is the shifting theorem. This theorem is needed to prove the convolution thereom. If f_k is the convolution of g_k with h_k, i.e.:

$$f_k \equiv g_k * h_k = \sum_{l=0}^{N-1} g_l h_{k-l} \tag{2.74}$$

then $F_n = H_n E_n + H_{-n} O_n$. F_n is the DHT of f_k, E_n and O_n are the even and odd part of the DHT of g_k and H_n is the DHT of h_k. Proof:

$$\begin{aligned}
DHT\ of\ f_k &= \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} g_l h_{k-l} cas(2\pi nk/N) && (2.75)\\
&= \sum_{l=0}^{N-1} g_l \sum_{k=0}^{N-1} h_{k-l} cas(2\pi nk/N) && (2.76)\\
&= \sum_{l=0}^{N-1} g_l \{H_n \cos(2\pi nl/N) + H_{-n} \sin(2\pi nl/N)\} && (2.77)\\
&= H_n \sum_{l=0}^{N-1} g_l \cos(2\pi nl/N) + H_{-n} \sum_{l=0}^{N-1} g_l \sin(2\pi nl/N) && (2.78)\\
&= H_n E_n + H_{-n} O_n && (2.79)
\end{aligned}$$

This is not quite the same as the convolution theorem for the Fourier transform. But in a lot of applications, and also in FVR, at least one function used in the convolution is even. In that case O_n is zero and the convolution theorem simplifies to $F_n = H_n G_n$, which is of the same form as its Fourier transform counterpart. For image processing in two dimensions exactly the same procedure is available.

If h_k has a discrete Hartley transform H_n then $h_{\alpha k}$ has a discrete Hartley transform $H_{\frac{n}{\alpha}}$. This can be seen by:

$$H_{\frac{n}{\alpha}} = \sum_{k=0}^{N-1} h_x cas(2\pi xn/\alpha N) \quad x = \alpha k \tag{2.80}$$

This means that compression of the time (or spatial) scale means expansion of the frequency scale and vice versa.

2.3.4 The 2-D and 3-D discrete Hartley transform

The discrete Hartley transform can be easily extended to the 2-D or 3-D case, similar to the discrete Fourier transform. The 2-D DHT of an image of size $M \times N$ results in an array of the same size of real numbers. The 2-D DHT is:

$$H_{u,v} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h_{x,y} cas[2\pi(ux/M + vy/N)] \tag{2.81}$$

for $u = 0, 1, 2, \ldots, M-1$ and $v = 0, 1, 2, \ldots, N-1$. The inverse transform is:

$$h_{x,y} = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H_{u,v} cas[2\pi(ux/M + vy/N)] \tag{2.82}$$

for $x = 0, 1, 2, \ldots, M-1$ and $y = 0, 1, 2, \ldots, N-1$.

For the 2-D DHT similar relationships as for the 1-D DHT and DFT can be derived. One relation will be given here, as it is needed for FVR. Let $h_{x,y} = e_{x,y} + o_{x,y}$, where $e_{x,y}$ and $o_{x,y}$ are the even and odd parts of $h_{x,y}$ respectively.

$$e_{x,y} = \frac{h_{x,y} + h_{-x,-y}}{2} = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H_{u,v} cos[2\pi(ux/M + vy/N)] \tag{2.83}$$

$$o_{x,y} = \frac{h_{x,y} - h_{-x,-y}}{2} = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H_{u,v} sin[2\pi(ux/M + vy/N)] \tag{2.84}$$

Then the 2-D inverse DFT is given by:

$$s_{x,y} = e_{x,y} - io_{x,y} \tag{2.85}$$

Similarly the inverse DHT is given by:

$$h_{x,y} = Re[s_{x,y}] - Im[s_{x,y}] \tag{2.86}$$

Note that the 2-D DHT is not separable in two 1-D DHTs, as can be done for the DFT. A 2-D DFT is performed by calling a succession of one dimensional DFTs. First one transforms all the rows using the kernel $e^{i2\pi ux/M}$ and then transforms column by column using $e^{i2\pi vy/N}$. The results amount to transforming with the product kernel $e^{i2\pi(ux/M+vy/N)}$, which is the 2-D DFT kernel, see Equation 2.26. However the kernel $cas[2\pi(ux/M+vy/N)]$ is not separable in a product of factors. In [17] and [14] an elegant solution to this problem is given. Refer to [1] Chapter 19 and Chapter 20 for more information on the DHT and for a software implementation of the Fast Hartley transform.

2.4 Conclusions

The Fourier Slice Theorem tells us that the Fourier transform of a projection of an object equals to a slice out of the Fourier transform of that object itself. The Slice Theorem also holds for the Hartley transform.

Transforming an array of real values using the Hartley transform results in an array of real numbers, transforming the same array using the Fourier transform in general results in a set of complex numbers. This property of the Hartley transform is very useful when dealing with real input data, as is the case for Fourier Volume Rendering.

For each Fourier transform theorem there is an equivalent Hartley one, but it is not always of the same form.

3 Processing images in frequency space

Sometimes it is easier, or less computing intensive, to perform certain filtering operation on the frequency transform of an image, instead of on the spatial image itself. This chapter discusses some aspects of operations in the 2-D frequency domain, and will try to make the reader more comfortable with interpretating 2-D frequency transformed images.

3.1 Spatial frequency

In a 1-D signal (e.g. audio) high frequencies occur when the signal is fluctuating fast, and low frequencies occur when it is fluctuating slowly. In the 2-D case (e.g. an image) there is an equivalent, called *spatial frequency*. high spatial frequencies occur when the difference in pixel-value of two neighboring pixels is large, and low spatial frequencies occur when the difference is small. So e.g. an edge in an image will consist of high frequencies because there is a large difference between pixel values on opposite sides of the edge (compare this to a 1-D step-function which contains high frequencies).

3.2 Interpretation of frequency transformed images

A frequency transform in general results in a complex number, consisting of a magnitude and a phase. This is difficult to display. In frequency domain images the phase information is not used, only the magnitude is displayed. Sometimes the square of the magnitude is displayed instead, which is also called the image power spectrum.

Figure 3.1 shows four images with a perfectly sinusoidal variation in brightness. The first two vary in orientation, while the third one varies in orientation as well as in spacing (frequency). The fourth one is the super-imposition of the first three. Below each image their frequency transforms are shown.

The frequency transforms rotate with the spatial images in the top row in Figure 3.1. Since a perfect sinus only has two delta functions in the frequency domain (which is the same as saying is consists of one frequency component), the frequency transforms for the first three images show only two dots, one for the positive and one for the negative component. The further the dots are away from the origin in the middle, the higher the frequency. The origin denotes the DC level, or 0 Hz. It is easiest to describe the frequency

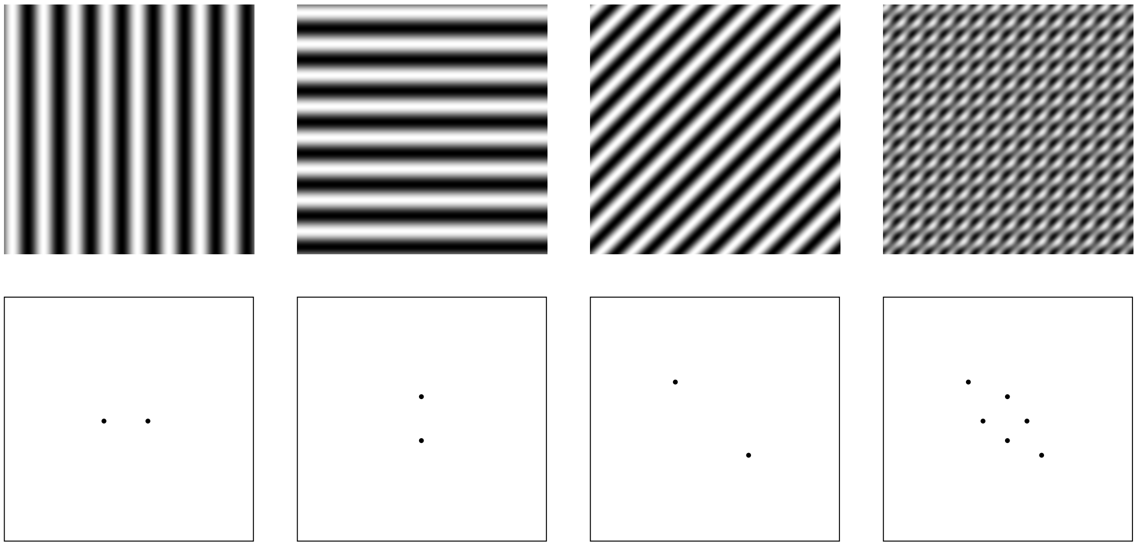

Figure 3.1: *Three sinusoidal patterns*

plots in polar coordinates. The frequency increases with r, the radius around the origin, and the orientation depends on the angle θ. The brightness of the dots denote the relative magnitudes of each frequency component in an image.

3.3 Spatial convolution in the frequency domain

Convolution is one of the more common operations performed on an image. A small kernel of numbers is multiplied by each pixel the kernel covers, the results summed and that result placed in the original pixel location. Then the kernel is moved one pixel and the process is repeated again. This goes on till all pixels in an image are processed.

Even if the kernel is not that big, it still involves quite a few operations. Increasing the kernel size eventually reaches a point where it is computationally more economical to perform the convolution in the frequency domain. The time it takes to do the inverse transform from frequency space to the spatial domain is more than balanced by the speed with which the convolution can be carried out in the frequency domain.

In Chapter 2 the relationship between convolution in the spatial domain and the equivalent operation in the frequency domain is given.

$$g_k * h_k \Leftrightarrow G_n H_n$$

This can be easily expanded to the 2-D case:

$$g_{x,y} * h_{x,y} = \sum_{a=0}^{N-1} \sum_{b=0}^{N-1} h_{a,b} \cdot g_{x-a,y-b} \Leftrightarrow G_{u,v} H_{u,v} \tag{3.1}$$

where a, b are dummy variables for the summation ranging over the entire image. Thus, convolution in the spatial domain is equivalent to multiplication in the frequency domain. In order to do this, the frequency transform of the convolution kernel needs to be computed. The kernel will almost always be smaller in size than the spatial image, so it has to be padded with zeroes to the size of the image before the discrete frequency transform is calculated. This is to ensure that the frequency transform of both the kernel and the image are of the same size.

Since discrete images are used, and a discrete frequency transform of those images, this frequency transform really represents a continuous 2-D spatial function which is repeated over and over in each axis. This means that the left edge of the spatial image is contiguous with the right, and the top edge is contiguous with the bottom. This is important when a closer look is taken at what happens at the edges of the spatial image when pixels closer to the edge than half the kernel size are being convolved. In this case the kernel will partially go over the edge of the image. This means that applying a convolution by multiplying in the frequency domain is equivalent to using the pixels of the next copy of the image in the spatial domain. This usually will produce some artifacts.

Since the Fourier and Hartley transforms are linear transformations, the above method only works for linear filters when those transforms are used.

4 Fourier Volume Rendering

The projection slice theorem can be used to visualize 3-D datasets. The Fourier projection slice theorem also holds in higher dimensions. For the 3-D case it can be stated as follows: *The 2-D Fourier transform of a 2-D projection of a 3-D object $x(u_1, u_2, u_3)$ at an angle θ, is a 2-D plane passing through the origin of the 3-D Fourier transform of that 3-D object, at the same angle θ.*

4.1 Theory and example

Once the 3-D spatial function $x(u_1, u_2, u_3)$ is being transformed by a Fourier transformation, it is possible to compute projections at arbitrary angles quickly by taking 2-D slices out of that 3-D transform and doing an inverse 2-D transform of the slice. This is called Fourier Volume Rendering (FVR), a term adopted from Malzbender in [12]. He, and an independent group of researchers, see [3], came up with the idea of Fourier Volume Rendering at about the same time.

A big advantage of FVR is that the (computing intensive) 3-D Fourier transform only has to be computed once. After that is done, only 2-D planes have to be extracted out of this 3-D dataset, and these planes 2-D inverse transformed, to generate images at any viewing angle. The complexity of FVR is O($N^2 logN$), which is the complexity of the inverse Fourier transform.

The steps to be taken for Fourier Volume Rendering are as follows:

- Calculate the 3-D Fourier transform $X(U_1, U_2, U_3)$ of the function $x(u_1, u_2, u_3)$.
- Make a cut-plane $P_\theta(\hat{U}_2, \hat{U}_3)$ out of this transform perpendicular to the viewing direction. This involves resampling in the frequency domain.
- Calculate the 2-D inverse Fourier transform $p_\theta(\hat{u}_2, \hat{u}_3)$ of the cut-plane and display it to the screen.

If the viewing angle alters only the last two steps have to be re-evaluated.

Graphically this is shown in Figures 4.1, 4.2, 4.3 and 4.4. It starts with the 3-D spatial function, Figure 4.1.

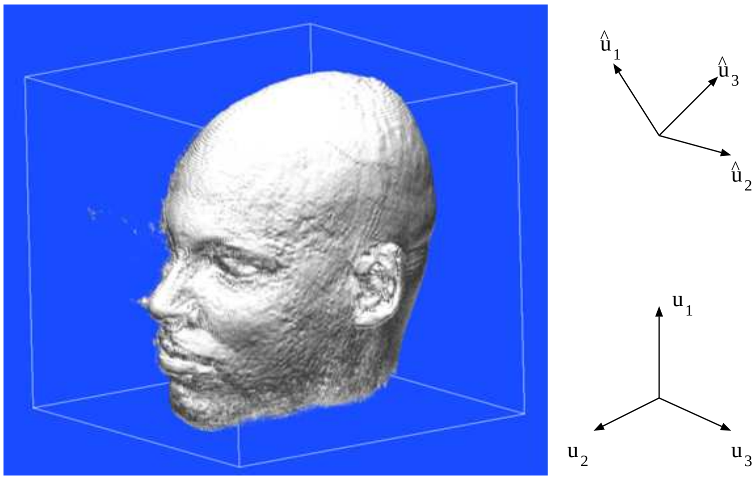

Figure 4.1: *The 3-D spatial function $x(u_1, u_2, u_3)$. The viewing direction is along the $\hat{u}_1$ axis.*

Then the 3-D forward Fourier transform of it is taken, Figure 4.2:

$$X(U_1, U_2, U_3) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x(u_1, u_2, u_3) e^{2\pi i (u_1 U_1 + u_2 U_2 + u_3 U_3)} du_1 du_2 du_3 \tag{4.1}$$

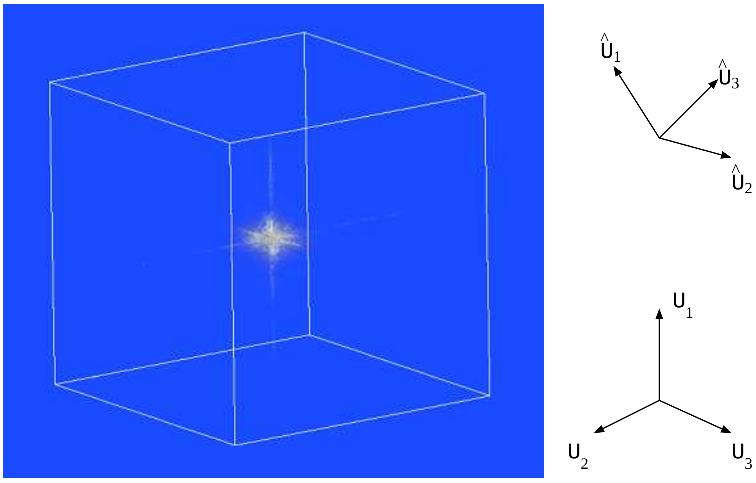

Figure 4.2: *The 3-D frequency representation $X(U_1, U_2, U_3)$.*

Then a parallel projection of $x(u_1, u_2, u_3)$ can be computed along an arbitrarily chosen viewing direction $\hat{u}_1$ by making a cut-plane $P_\theta(\hat{U}_2, \hat{U}_3)$ through the origin of $X(U_1, U_2, U_3)$ perpendicular to the viewing direction. This is shown in Figure 4.3.

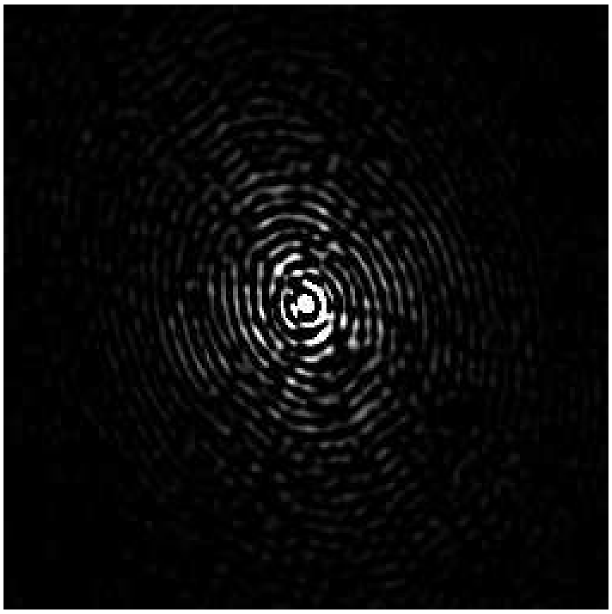

Figure 4.3: *The 2-D frequency cut-plane* $P_\theta(\hat{U}_2, \hat{U}_3)$

The inverse 2-D Fourier transform of $P_\theta(\hat{U}_2, \hat{U}_3)$ results in the projection:

$$p_\theta(\hat{u}_2, \hat{u}_3) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} P_\theta(\hat{U}_2, \hat{U}_3) e^{-2\pi i(\hat{u}_2\hat{U}_2 + \hat{u}_3\hat{U}_3)} d\hat{U}_2 d\hat{U}_3 \tag{4.2}$$

This is shown in Figure 4.4.

In [5] use of the Wavelet transform, instead of the Fourier transform, in the projection slice theorem is discussed. The authors of [5] conclude that there is no practical use for a Wavelet transform in FVR.

4.2 Depth cueing

Because the projection slice theorem calculates a line integral over the function $x(u_1, u_2, u_3)$

$$p_\theta(\hat{u}_2, \hat{u}_3) = \int_{-\infty}^{\infty} x(\hat{u}_1, \hat{u}_2, \hat{u}_3) d\hat{u}_1 \tag{4.3}$$

control over the transparency of the resulting image is not possible. This means that i.e. hidden surface effects are not present. Images produced with Fourier Volume Rendering will look like X-ray pictures.

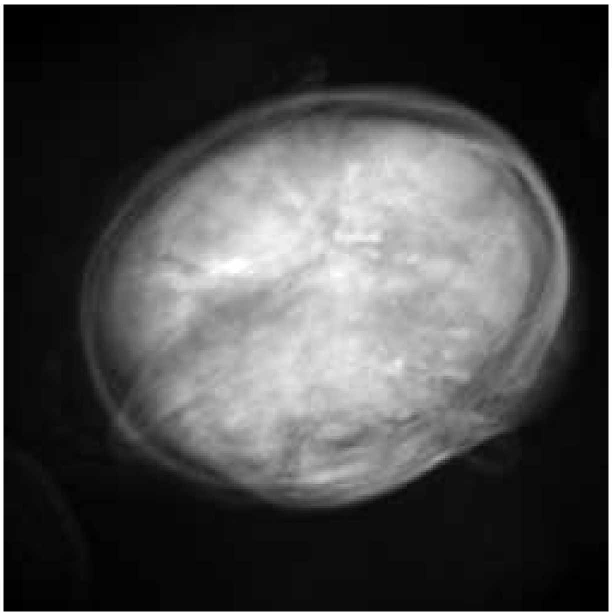

Figure 4.4: *The 2-D spatial projection* $p_\theta(\hat{u}_2, \hat{u}_3)$

Depth cueing is relatively easy to implement in Fourier Volume Rendering. Depth cueing means that voxel intensities are weighted according to their distance to the observer. let $\mathbf{u}$ be a vector of three elements; $u = [u_1, u_2, u_3]$. If $d(\mathbf{u})$ is a depth cueing function for a certain observer viewer position, then $x(\mathbf{u}) \cdot d(\mathbf{u})$ is the depth cued dataset. This depth cued dataset can then be rendered with FVR. The disadvantage of this approach is that for each viewing angle the depth cued dataset has to be recomputed, and an expensive forward 3-D Fourier transform has to be done, effectively countering the advantages of FVR. There is a better way to implement depth cueing, which operates in the frequency domain.

Let $\mathcal{F}(.)$ be the Fourier transform. FVR (without depth cueing) can be written as follows:

$$
\begin{aligned}
x(\mathbf{u}) \cdot h(\mathbf{u}) \quad &\Leftrightarrow \quad \mathcal{F}\{x(\mathbf{u}) \cdot h(\mathbf{u})\} && (4.4)\\
&\Leftrightarrow \quad \mathcal{F}\{x(\mathbf{u})\} * H(\mathbf{U}) && (4.5)\\
&\Leftrightarrow \quad X(\mathbf{U}) * H(\mathbf{U}) && (4.6)
\end{aligned}
$$

$H(\mathbf{U})$ is the reconstruction filter that extracts the 2-D plane out of the 3-D Fourier transform. Now depth-cueing can be incorporated, $x(\mathbf{u}) \cdot d(\mathbf{u})$:

$$
\begin{aligned}
x(\mathbf{u}) \cdot d(\mathbf{u}) \cdot h(\mathbf{u}) &\Leftrightarrow \mathcal{F}\{x(\mathbf{u}) \cdot d(\mathbf{u}) \cdot h(\mathbf{u})\} && (4.7)\\
&\Leftrightarrow \mathcal{F}\{x(\mathbf{u}) \cdot d(\mathbf{u})\} * H(\mathbf{U}) && (4.8)\\
&\Leftrightarrow \{X(\mathbf{U}) * D(\mathbf{U})\} * H(\mathbf{U}) && (4.9)\\
&\Leftrightarrow \{X(\mathbf{U})\} * \{D(\mathbf{U}) * H(\mathbf{U})\} && (4.10)\\
&\Leftrightarrow \mathcal{F}\{x(\mathbf{u})\} * G(\mathbf{U}) && (4.11)\\
&\Leftrightarrow X(\mathbf{U}) * G(\mathbf{U}) && (4.12)
\end{aligned}
$$

where $G(\mathbf{U}) = D(\mathbf{U}) * H(\mathbf{U})$. Thus by using the new reconstruction filter G depth cueing is implemented. Note that G is dependent on the viewing angle, and thus has to be recomputed every time the viewing angle is changed. This is not a real problem, since the filter H is small and D can be a simple linear function. Note that $d(\mathbf{u})$ can be any function. Equation 4.12 is a 2-D operation, and the depth cueing operates entirely in the frequency domain. [19] shows that images rendered using this method don't look very different from images rendered without any depth cueing though. In [19] a method also is presented to implement a simple shading model for use with FVR, using the same techniques.

4.3 Resampling in the spatial domain

FVR requires resampling or interpolation of the 2-D plane in the frequency domain. Normally resampling is done in the spatial domain, and its effects evaluated by looking at the frequency response of the resampling function. An ideal interpolation and resampling function is the $sinc$ function:

$$sinc(x) = \frac{sin(x)}{x} \qquad (4.13)$$

Its transform is a rectangle with width and height of one. Unfortunately this is not a practical usable function, since it has an infinite extent. Therefore approximations of the $sinc$ are used, and inadvertably artifacts are introduced by resampling with these approximations of the $sinc$.

In general resampling works as follows. The origin of the resampling function is positioned at the sample point to be resampled, and the left and right neighboring pixel values are multiplied with the resampling function value at those pixel points, and added together to form the resampled value. This is equivalent to convolving the dataset with the resampling function. A couple of things have to be taken into account when resampling an image.

- A discrete image represents an infinite continuous periodic function in the frequency domain.
- The other way around also holds, a set of discrete frequencies, as in FVR, represents a spatial continuous function which is infinitely periodic.
- The ideal interpolation function is a $sinc$. Its transform is a block function with height and width of one.
- A spatially limited function has an infinite extent in the frequency domain, and vice versa (This is the Uncertainty Theorem).
- Resampling in one domain means convolving the data with the interpolation function in that domain. In the other domain this means multiplication with the transform of the resampling function in that other domain.

The first two items are a result of sampling a continuous function. Any continuous function that is being sampled is made infinite periodic in its transfer domain by that sampling. If the function also is not bandwidth limited to the Nyquist frequency the sampling introduces aliasing too.

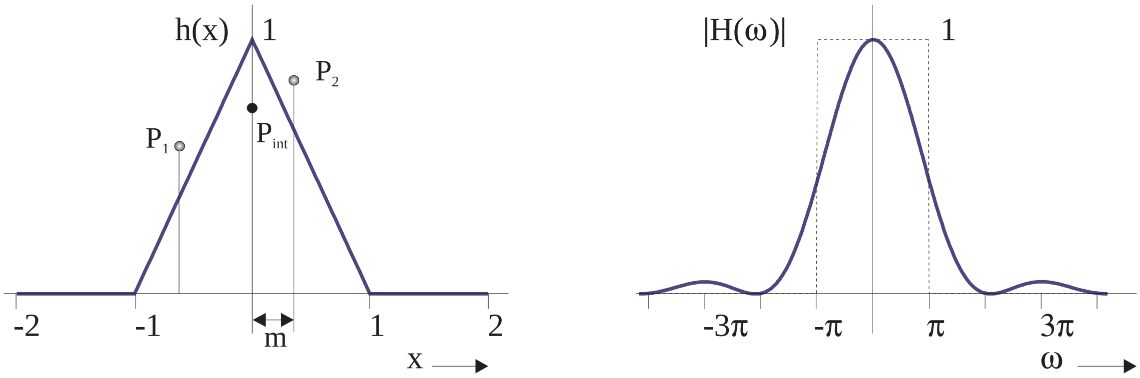

Figure 4.5: *Linear interpolation filter. (a) Spatial domain. P_1 and P_2 are known grid point values. P_{int} is estimated using P_1 and P_2. (b) Frequency consequence of linear interpolation in the spatial domain.*

A commonly used interpolation method is linear interpolation. See Figure 4.5. From this figure it is seen that the frequency response is not zero outside the pass band, denoted with the dotted lines in Figure 4.5(b). Or differently phrased, the frequency response is not an ideal low pass filter. An ideal low pass filter will filter out all replicas in the frequency domain. Since this is not the case, aliasing will occur due to non perfect interpolation. An important design goal is to minimize the energy of the frequency response outside the pass band. Several different interpolation function have been proposed, see e.g. [11] and [15]. For FVR a 3-D resampling method is used. The resampling filters used in FVR are separable. That means that 3-D resampling can be realized by 1-D resampling in respect to each coordinate axis. This is the reason only the 1-D case is shown in Figure 4.5.

4.4 Resampling in the frequency domain

Resampling in the frequency domain reverts the effects described earlier. The ideal filter is still the $sinc$, since it will filter out all spatial copies that will occur due to the fact that a set of discrete frequency points represents a continuous spatial periodic function. This can be seen as recovering the original non-periodic spatial function by multiplying with a cube in the spatial domain. This in effect suppresses all spatial periodic copies except one. This amounts to convolving with the $sinc$ in the frequency domain.

In FVR a plane out of the 3-D frequency transform is resampled. This frequency plane represents the projection of the spatial infinite periodic copies of the original dataset.

Because of the reasons mentioned earlier, the $sinc$ function is not a feasible filter to use for the resampling of this plane. A tri-linear filter could be used to interpolate sample points in the frequency domain. This will introduce aliasing in the spatial domain, which will show up in the interpolated image as copies of the dataset are folded back into it. This is because there is still some amount of energy outside the passband in (now) the spatial domain. Thus this allows the periodic copies of the spatial dataset to remain (at a weaker intensity though), and a projection of these copies will overlap each other. This is a 3-D aliasing phenomena. See also Figure 4.5 but now Figure 4.5(a) is the frequency filter and Figure 4.5(b) is the spatial consequence of using that filter.

Another point to take into account is that the resampling should be done quickly enough to prevent spatial copies of the periodic projections overlapping each other, and introducing aliasing. This effect is called 2-D aliasing. The widest one copy in the spatial domain gets is $\sqrt(3)$ times N. N is the length of one side of the 3-D spatial dataset. The factor of $\sqrt(3)$ is the diagonal through a cube of size $1 \times 1 \times 1$. By using the scaling property of the Fourier transform it is found that the resampling has to be done at at least a spacing of $s/\sqrt(3)$ in the frequency domain. s is the spacing of the 3-D frequency transform in each axis. This will ensure that no overlap will occur. It is of course possible to resample at a higher rate.

Figure 4.6: *Aliasing. Left: tri-linear interpolation, middle: POCS* $3 \times 3 \times 3$ *right: POCS* $5 \times 5 \times 5$

Resampling in the frequency domain has to be done very accurately. In [12] several different filters are discussed. If a tri-linear interpolating function is used then the energy in the passband is more than enough to introduce very noticeable artifacts in FVR. See Figure 4.6 left image. Parts of the skull are aliased back into the image, e.g. the back of the skull is aliased right into the front of the nose.

In [12] a far better resampling filter is designed, using a method called Projection on Convex Sets (POCS). In short POCS works as follows. POCS allows constraints in both the spatial and frequency domain to be optimized. The resampling filter that has to be designed has two constraints. First it should be of a limited size and box shape in the spatial domain, to filter out the periodic copies of the dataset. Second, the filter should be small in the frequency domain, because that limits computing time and complexity. Unfortunately these two constraints can never be met exactly, since the Uncertainty Theorem states that a spatially limited function has an infinite extent in the frequency domain, and vice versa. But those constraints can be approximated by using POCS. In [12] Malzbender starts with a Hamming windowed $sinc$ in the frequency domain. Then he transforms it to the spatial domain, yielding a transform of an infinite extent. Now he applies the spatial domain constraint by chopping off the tails outside the first periodic copy. This results in a space limited filter. He transforms this (truncated) filter back to the frequency domain. Then he applies the frequency domain constraint by chopping off the tails again. This completes one iteration. This process can be repeated till one is satisfied with the resultant filter. Malzbender designed several filters, including one of an extent of $3 \times 3 \times 3$ frequency samples, and one of $5 \times 5 \times 5$ frequency samples. The former has the same extent as the tri-linear interpolation function, but is much better, see Figure 4.6 left and middle image. The result of using a $5 \times 5 \times 5$ POCS filter is shown in the same figure, right image. This filter performs even better than the previous two, but takes more time to evaluate, and is more complex to implement in hardware. See also Section 4.9. See the appendix for a table of the used POCS filters.

4.5 Spatial premultiplication

Any practical interpolation filter is an approximation of a box function at its best. They are not of a constant amplitude in the pass band, see e.g. Figure 4.5. If the tri-linear interpolation function is used in FVR then the outer regions of the dataset will be attenuated. See Figure 4.7 for an example of this effect. The right image has been compensated for this. It can be compensated for by premultiplying the spatial dataset (before it is 3-D transformed) by the inverse of the resampling filter. This is shown in Figure 4.8(a). Note that the stop band energy is somewhat increased in Figure 4.8(c). This is due to the fact that all periodic copies of the spatial dataset are premultiplied with the function in Figure 4.8(a), which can be seen as multiplying the original resampling filter in Figure 4.5(b) with an infinitely periodic version of the spatial premultiplication function, as shown in Figure 4.8(b).

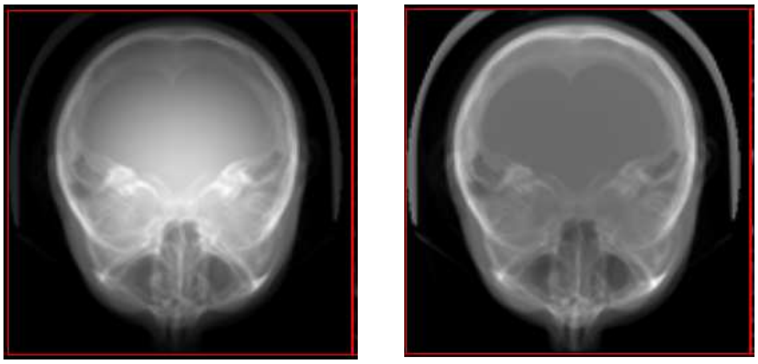

Figure 4.7: *Left: Attenuation of outer regions. Right: Compensated for by using spatial premultiplication*

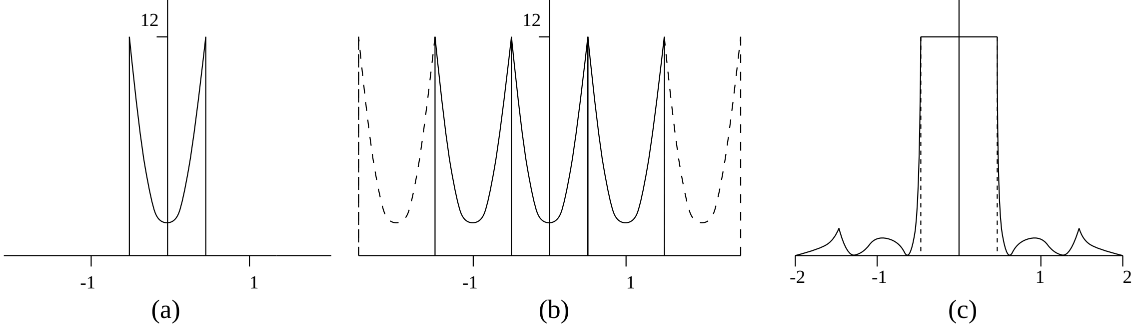

Figure 4.8: *(a) Spatial premultiplication function for the linear interpolation function. (b) Spatial premultiplication can be seen as multiplying the transform of the original resampling function with an infinitely periodic version of the function in (a). (c) Resultant spatial function.*

4.6 Zero padding of the 3D spatial dataset

One can think of FVR as making one projection of all the periodic copies of the spatial dataset, albeit at a lower amplitude outside the first periodic copy. Resampling in the frequency domain introduces aliasing artifacts, because the resampling filter is not perfect. There still is energy in the stop band of the resampling filter. This error can be reduced by designing a good filter, and by zero padding the original dataset. This will ensure that part of the energy outside the passband of the resampling filter is used to project those zeros, which obviously will not add up to any artifacts. Note that most of the energy in the stop band is right after the passband, or right at the edge of the first periodic copy of the dataset. See e.g. Figure 4.5(b). By zeropadding the image all this energy will be nullified.

4.7 Maximum intensity projection with FVR

A different volume rendering technique is called Maximum Intensity Projection (MIP). Normally this is done in the spatial domain, by casting rays through the 3-D dataset and displaying the maximum value found on the ray to the screen. The complexity is of $O(N^3)$. This is a time consuming operation since the complete 3-D dataset has to be traversed. Physicians often like to look at MIP images because those images look like X-ray images, and the blood vessels show up very clearly in MIP images.

If the original 3-D spatial dataset is preprocessed the following way, MIP images can be approximated by using FVR. Let v be the value of one voxel of the 3-D dataset. Then create a new 3-D dataset by replacing all voxel values by:

$$v = b^v \tag{4.14}$$

b is some small number, e.g. 2. The idea behind this is to emphasize the high voxel values in the original dataset, by replacing them with much bigger numbers than the low voxel values. Then if a projection of this dataset is made it will look like a MIP image because the maximum voxel value will be so much bigger than the other voxel values on the projection ray. Of course the ${}^b log$ of the resulting projection has to be taken before displaying it to the screen.

Unfortunately the resulting image is full of artifacts. See Figure 4.9.

For the FVR rendered image in Figure 4.9 a $5 \times 5 \times 5$ POCS filter was used, and b was set to 1.1 in Equation 4.14. As this figure shows, artifacts are everywhere. This is due to

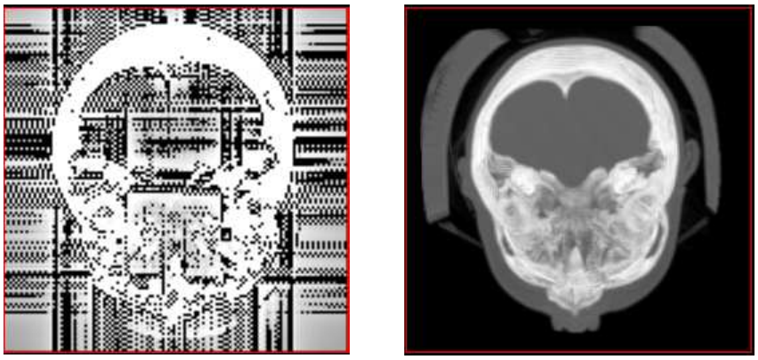

Figure 4.9: *MIP projection. Left: Image rendered by FVR. Right: Same image rendered by a conventional MIP ray caster*

a fundamental problem. Since the log of the projection has to be taken, the resampling filter is compressed with that same log. This is a non-linear compression, and this means that the tails of the resampling filter (i.e. the stop band) will be amplified compared to the passband. In other words, the signal to noise ratio of the resampling filter gets worse by taking the log of the projection.

Thus a far better resampling filter is needed to overcome this problem. If such a filter is used, the advantage of doing a MIP projection by using FVR will probably be nullified. This new filter will be of a bigger extent than the one used now, and resampling a plane in the frequency domain will take considerably longer to compute. Possibly even longer than rendering the MIP image with a conventional ray caster.

4.8 Software implementation of FVR

This section will briefly outline the software program *hvr* used to generate all FVR images in this report.

The discrete Hartley transform is used in the program, since it is a real transform if the input data is real (and that is the case), and the fast Hartley transform (FHT) is as fast or faster than the FFT. Because the 3-D FHT will result in real data no separate imaginary and real part have to be taken into account during the resampling stage. The DHT of N real numbers results in a transform of N real numbers. The DFT of the same N numbers in general results in N complex numbers, which are $2N$ real numbers. This may seem strange, but since the input values are real, S_0 and $S_{N/2}$ don't have an imaginary part and half the remaining values of S_n are complex conjugates of the other half $S^*_{N-n} = \sum_{k=0}^{N-1} s_k e^{i2\pi-(N-n)k/N} = S_n$. This is the hermitian property of the Fourier transform. Thus the positive frequency half is sufficient to determine the DFT completely, which are $N/2$

complex numbers or N real ones, the same number as in the Hartley case.

4.8.1 Features

The hvr program has the following features:

- The input is the 3-D forward discrete Hartley transformed dataset in floats, which has to be premultiplied with the inverse of one of three resampling filters, tri-linear, POCS $3 \times 3 \times 3$ or POCS $5 \times 5 \times 5$, and optionally zero padded.
- Displays the 2-D resampled Hartley plane and the resulting projection in 8 bits.
- User interface is either by keyboard or by the nine dial knob-box.
- Image can be rotated over x,y and z axis.
- Zooming in and out by compressing or expanding the frequency plane.
- Auto scaling of brightness.
- Screen image can be resized by resizing the hvr window.
- MIP projection support.

The program was written in C, and uses Motif and starbase calls to handle its graphics.

4.9 Hardware implementation considerations

A block diagram of a possible hardware realization of the FVR algorithm is shown in Figure 4.10. The memory block holds the 3-D Hartley (or Fourier) transform of the dataset being rendered. The resampling block does the interpolation and resampling of a 2-D plane out of the 3-D transform. It puts its output in the buffer. The Inverse FHT/FFT block performs the inverse discrete Hartley (or Fourier) transform in place. The resulting image can be output to screen directly or to a frame buffer, which is not necessary a part of a FVR hardware realization.

The building blocks will be discussed in more detail in separate sections.

To build a hardware FVR rendering engine the resampling hardware and memory addressing logic needs to be designed. The rest of the hardware needed can be put together

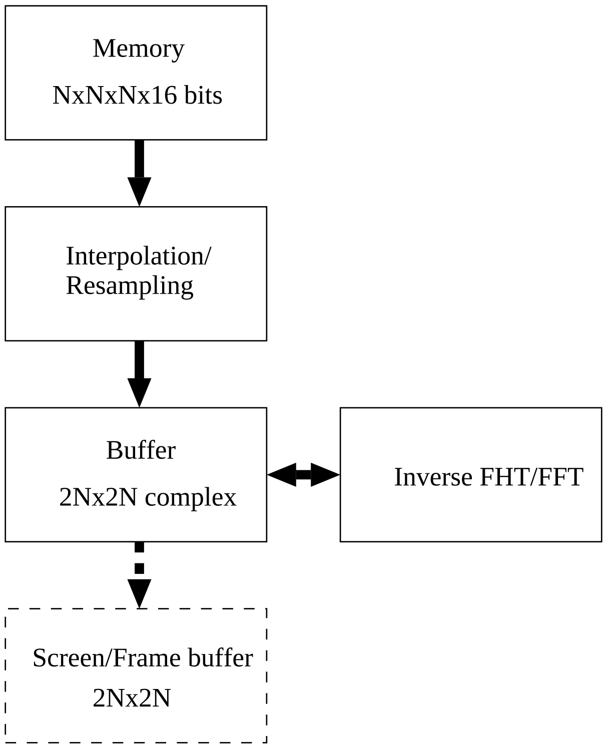

Figure 4.10: *Block diagram of a hardware FVR engine.*

using commercially available chips, like (S)(D)RAMs and FFT chips. The final version of the FVR rendering engine depends largely on the application needs. It might e.g. be feasible to design a PC board, where the PC will do the forward 3-D transform in software, and supply the viewing parameters to the board which will do the resampling and inverse transform.

For now the following specifications will be assumed. These will be reconsidered later.

- Dataset size is 256^3 16 bits wide (32 Mb) This is the 3-D Hartley or Fourier transformed data.
- Frame rate is 25 frames/sec.
- Screen resolution is 512×512 pixels.

4.9.1 The interpolation unit and memory access

Using the specifications given in the previous section the following calculations can be made. 25 frames per second means that there is 40 ms to resample one plane. One interpolation step in this plane should be done within $0.04 \times 1/(512 \times 512) \approx 150$ ns. For todays technology 150 ns is a lot of time. This means that the resampling option does not need to be parallelized. One interpolation unit and memory addressing logic is all that it takes for the resampling hardware.

Often a tri-linear interpolation is used when a hardware implementation is build. Tri-linear interpolation is easy to implement in hardware and very fast. See e.g. [13]. Unfortunately the software simulations clearly showed that tri-linear interpolation is not good enough. A hardware implementation of a POCS $5 \times 5 \times 5$ filter is preferred. This can be realized by using a lookup table. The POCS function values can be precomputed for a certain desired precision and those stored in a lookup table (EPROM o.i.d.). A $3 \times 3 \times 3$ filter, like the tri-linear or the $3 \times 3 \times 3$ POCS filter, needs 8 surrounding voxels to compute one sample point. A $5 \times 5 \times 5$ filter, like the tri-cubic spline or $5 \times 5 \times 5$ POCS filter, needs 64 surrounding voxels to compute one sample point.

First the 8 voxel case will be considered. It is assumed that expensive static RAM is not to be used, but instead the cheap standard DRAMs with a 100 ns access time. Since 8 voxels have to be fetched within 150 ns, 8 RAM banks instead of one can be used, and each of the 8 voxels stored in a different bank. The 8 banks can be accessed in parallel and the voxels fetched within 150 ns. To ensure all voxels are in a different bank the address of each voxel is divided in odd and even rows, odd and even columns and odd and even depth, and this information is used to divide the voxels over the 8 RAM banks. See also Figure 4.11. The cost for this is some extra addressing hardware.

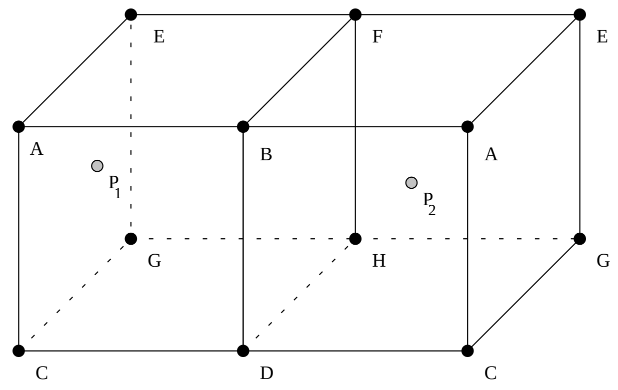

Figure 4.11: *Fetching 8 voxels (A-H) in parallel*

The 8 RAM banks in Figure 4.11 are labeled A to H. P_1 and P_2 are two points to be sampled. Since sampling takes place on a voxel sub-cube level, the addresses of P_1 and P_2 in general will be fractional. The truncated address of P_1 can be used to fetch all its 8 surrounding voxels, but the truncated address of P_2 cannot be used to fetch all its 8 surrounding voxels, since 4 of those 8 banks (RAM banks B,D,F and H) were addressed by the truncated address of P_1, which is a different address than the truncated address of P_2. Some extra addressing logic is needed to distinguish between the different cases. This logic is very simple though, since the situation depends on if the truncated address is odd or even in any one of the three directions.

In the 64 voxel 64 voxels have to be fetched from memory at once. A solution would be to use the same scheme for the RAM banks as in the 8 voxel case. Thus using 64 RAM banks and fetching all voxels in parallel. This imposes a problem on the width of the data path. At 16 bits for each voxel, fetching 64 voxels at once means a 1024 bits wide data path, and an interpolation chip with at least that many pins. This is not feasible with the current VLSI technology, so the voxels have to be time multiplexed and clocked into the interpolation chip in smaller quantities. This can be done by clocking the RAM banks at a 4 times higher speed. It is then possible to reduce the number of RAM banks by the same factor of 4. This still means a 256 bits wide data path. It would be desirable to reduce that even further. This can be achieved by caching voxels in the interpolation chip, and by using the fact that there always will be 4 resampling points within the space of a sub-cube of 8 voxels. This means that the same set of 64 voxels will be used for interpolating 4 resampling points. The proof for this is as follows:

Given is a $n \times n \times n$ cubed dataset. A plane out of this dataset will be resampled a a rate of $2n$ in x and y. The step size between the resampling points depends on the orientation of the plane to be resampled out of the dataset. If the resampling plane is oriented parallel to one of the main axis, the step size will be minimal. The length of one side of the dataset is $n - 1$. The number of steps to take in one direction is $2n - 1$ in this case resulting in $2n$

sample points. The step size thus is $(n-1)/(2n-1)$. See also Figure 4.12(a).

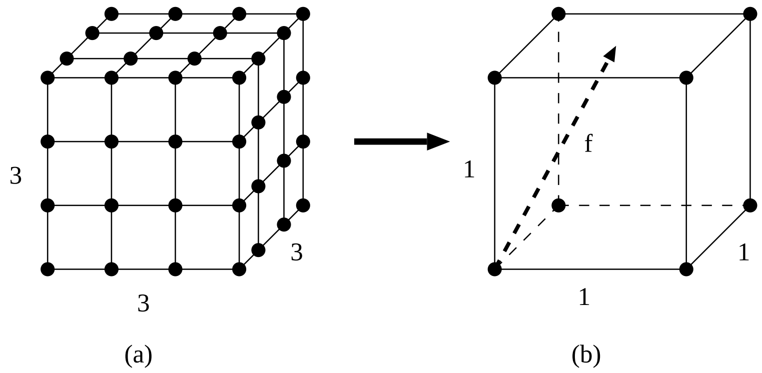

Figure 4.12: *(a) A $n = 4 \times 4 \times 4$ cube has side lengths of 3. (b) f is the length of a ray through one voxel sub cube*

If the resampling plane is oriented along one of the diagonals of the cube, the step size will be maximal. It will be a factor of $\sqrt{3}$ times bigger than for the minimal case. For an arbritary orientation of the resampling plane the step size will be:

$$s = f \cdot \frac{n-1}{2n-1} \quad n > 1 \quad 1 \leq f \leq \sqrt{3} \tag{4.15}$$

Where f is the length of the part of the ray which goes through one voxel sub cube. If it can be proven that $s < f/2$ then there always will be 4 resampling points within one voxel cube since the step size is the same in both directions. See also Figure 4.12(b).

$$s = f \cdot \frac{n-1}{2n-1} < \frac{f}{2} \tag{4.16}$$

$$\frac{n-1}{2n-1} < \frac{1}{2} \tag{4.17}$$

$$n - 1 < n - \frac{1}{2} \quad q.e.d. \ \Box \tag{4.18}$$

Since the same 64 voxels will be used 4 times it is possible to reduce the number of RAM banks by another factor of 4, while clocking them at the same speed. This results in 4 RAM banks clocked at $150/4 = 37.5$ ns. The interpolator needs two voxel cache banks of 64 voxels each. While one cache bank is used to calculate the 4 resampling points the other bank will be filled with the new set of 64 voxels. This means a total of $2 \times 64 \times 2 = 256$ bytes of on chip cache is needed.

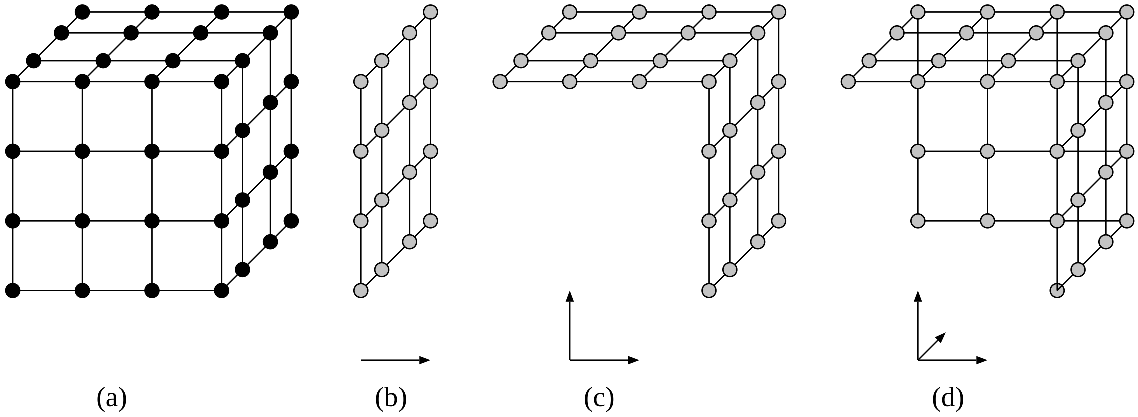

Figure 4.13: *Voxels that have to be fetched from memory if a resampling step in x, xy or xyz is taken and a voxel sub cube boundary is crossed. (a) A* $4 \times 4 \times 4$ *set of voxels. (b) 16 new voxels have to be fetched from memory when a step in x only is taken. (c) 28 new voxels have to be fetched when a step of one voxel sub cube in x and y is taken. (c) 40 new voxels have to be fetched when a step in all three directions is taken.*

$2 \times 64 = 128$ different voxels are not needed to calculate 8 consecutive sample points. Suppose 64 voxels where fetched to calculate a sample point. If an adjacent voxel sub cube boundary is crossed to calculate the next sample point, a minimum of 16 voxels and a maximum of 40 voxels have to be fetched from memory, instead of all 64. The remaining voxels are already present in the voxel cache. See Figure 4.13. By taking this into account it will be possible to clock the RAM banks at an even lower speed but the addressing logic will become more complex. This approach is not pursued further.

Because the resampling plane can have any arbritary orientation, points of the plane can fall out of the 3-D dataset. It is assumed that those points have a value of zero (the hvr program does this too). Since points outside the 3-D dataset represents frequencies higher than the ones in the dataset, and since those frequencies components are not known, it is legitimate to assume that they are zero.

Currently the fastest available SDRAMs can supply data at a rate of about 30 ns. The fastest static RAM can supply data at a rate of about 10 ns.

4.9.2 The inverse discrete Hartley or Fourier transform

The interpolator stores its output in the buffer. The last step to take before it can be displayed to the screen is the inverse discrete Hartley or Fourier transform of the data in the buffer.

Using the Hartley transform for the software implementation had numerous advantages, for a hardware implementation that might not be the case.

Hartley transform chips are not commercially available. But a wide range of Fast Fourier transform (FFT) chips are. These chips also can perform an inverse FFT (IFFT). The Hartley transform can be easily obtained by subtracting the real and imaginary output of the IFFT. See Section 2.3.4.

A hardware implementation using the Hartley transform and FFT chips and an implementation using the Fourier transform and FFT chips are discussed next.

4.9.2.1 2-D inverse Hartley transform using a FFT chip

The fastest FFT chips available typically perform a FFT or IFFT on 1024 complex points in about 100μsec. For example the PDSP16510/A stand alone FFT processor made by GEC Plessey. If the chip can handle two 512 point complex arrays in the same amount of time one row of 512 complex points can be inverse transformed in 50μsec. A 2-D FFT can be split into 1-D transforms. First transform all rows of one image, and then transform all columns. The input data to the FFT chip is real. (The original 3-D spatial dataset is real, and the forward 3-D Hartley transform results in a real 3-D dataset in the frequency domain.) Since a FFT requires a real and an imaginary part, the input array for the real and imaginary part can be filled with a set of two 512 point arrays each. Thus it is possible to perform the IFFT on 4 rows in 100μsec. The PDSP16510/A has support for this on chip. See also [16] pages 414-417. This approach can be taken for the rows only, since the resulting transform of the rows is complex. (A FFT of a array of real numbers is in general complex). This means that 2 columns can be transformed in 100μsec. But the FFT of a real signal was computed (for the rows) so the hermitian property can be taken into account, and only half of the columns have to be transformed. See also Figure 4.14. This results in $512/4 \times 100\mu + 512/4 \times 100\mu = 25,6$msec for one 512×512 image, or 39 frames/sec. The time to ungarble the 4 arrays of 512 points into 4 separate transforms is not taken into account, so it might not be possible to transform 4 rows in 100μsec, but in a slightly longer time. The resulting frame rate is much higher than the required 25 frames/sec.

After the IFFT is performed 512×512 subtractions have to be performed. This cannot be done by performing an IFFT on one row, subtracting the imaginary part from the real part, storing this row and continuing with the next row. The Hartley transform is not separable like the Fourier transform. This means that first all rows have to be Fourier transformed, then all columns and then the subtraction can take place. Alternatively, after all rows and one column is Fourier transformed the subtraction can be done on that column, saving some time by not waiting till all 512 columns are Fourier transformed.

4.9.2.2 2-D inverse Fourier transform using a FFT chip

Another option is not to use the Hartley transform at all, but storing the 3-D discrete Fourier transform, and resampling this 3-D dataset. The advantage of doing this is that no subtractions have to be performed after the 2-D IFFT step is completed. The disadvantage is that the 3-D transform will have an imaginary part not equal to zero. Thus to save memory, the hermitian property has to be taken into account. See also Figure 4.14.

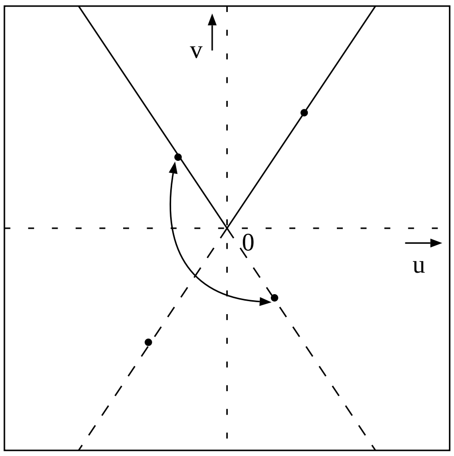

Figure 4.14: *Hermitian property of the FFT for a 2-D dataset. The points in the upper half are point symmetrical with their complex conjugates in the lower half. The origin is in the middle.*

The hermitian property means that one half of the 3-D Fourier dataset is the complex conjugate of the other half. Another way to look at this is that a point and its complex conjugate are point symmetrical through the origin. This means only e.g. the upper half of the dataset has to be resampled to get all the information needed. Note that this does not mean that less points are resampled compared to the Hartley case, since the Fourier dataset is complex. The steps to take are the following:

1. Take $N \times 2N$ steps through one half of the 3-D Fourier dataset (of course only this half will be stored in memory) and fetch the 64 voxels needed to interpolate the real and the 64 voxels needed for the imaginary part out of memory.
2. Resample the real value for each step.
3. Resample the imaginary value for each step.
4. Store this complex resampled number in the buffer.
5. Repeat this till $N \times 2N$ steps are taken (and thus $2N \times 2N$ points resampled). This results in a buffer of size $N \times 2N$ with complex numbers.

6. Perform the inverse 2-D FFT.

For the The 2-D IFFT the hermitian property again can be used. (The 2-D IFFT is split into two 1-D IFFTs, first transform all the rows and then all the columns). The 1-D IFFT for the rows has to be done on N rows only to generate $2N$ complex inverse transformed rows. But this property cannot be used on the columns. If that would be the case it would mean that in the resulting spatial image, the projection, columns would have a relationship to each other, which cannot be true (a projection of an arbitrary object does not have to have any symmetry at all). Thus a 1-D IFFT has to be done on all $2N$ columns. This means the buffer should hold $2N \times 2N$ complex numbers. Using the same scheme as for the Hartley transform (storing two 512 complex arrays in one 1024 complex array and performing the transform on those two at the same time) the transform will take about $N/2 \times 100\mu + 2N/2 \times 100\mu = 38, 4$msec, or 26 frames/sec, slower than in the Hartley case. The result of the 2-D IFFT is a $2N \times 2N$ plane of real numbers, which can be output to a frame buffer or screen. The numbers should be real because a projection of an object of real data is real. But because of resampling artifacts there will be a small imaginary part too.

The number of memory accesses and number of resampling points is the same as in the Hartley case. The same memory fetching scheme can be used, since the same number of resampling points are calculated.

4.9.2.3 Conclusions

Using the Fourier transform instead of the Hartley transform for a hardware implementation of FVR will save some hardware. No more subtractions have to be performed to get the DHT out of the DFT. The resulting frame rate for the Hartley case is 50% higher than in the Fourier case. 39 frames/sec as opposed to 26 frames/sec. The 26 frames/sec is close to the specifications of 25 frames/sec, but the time needed to split the resulting arrays into the individual transforms is not taken into account, so the actual frame rate will be less than those 26 frames/sec.

Although for the software implementation the discrete Hartley transform has advantages over the discrete Fourier transform, the lack of fast Hartley transform VLSI chips make using the DFT a logic choice.

The size of the buffer in Figure 4.10 is $2 \times 512 \times 512 \times 16$ bits $= 1$ Mb. Note that the buffer has to be a double buffer. While one part is being filled by the interpolator, the other part is being transformed by the IFFT block. Thus the buffer has to be 2 Mb in size.

A FFT chip that can handle 1024 points in 100μsec, like the PDSP16510/A, costs about $1100. It might be more cost effective to use two or maybe 4 slower chips, and have them work on the transform in parallel. More research into available FFT chips has to be done. The numbers given in this section are estimates, but they do show that a 2-D real time FFT is possible.

4.9.3 Specifications reconsideration

The FFT chips typically expect 16 bits wide input data. This is the main reason why 16 bits was chosen in the design specifications. Furthermore, the data path should not get too wide, as explained earlier. The width of the input data also affects the pin count of the interpolator. Certainly if a $5 \times 5 \times 5$ filter will be used, more bits will have a big impact on the design of the interpolator.

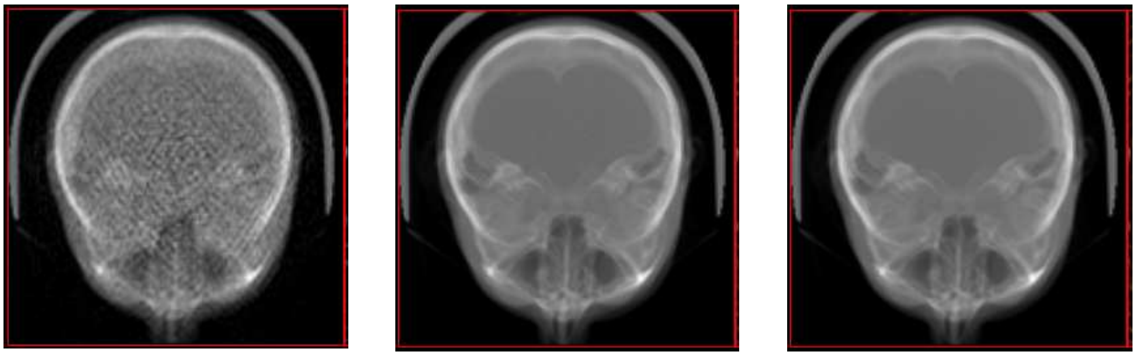

Figure 4.15: *Results of storing the 3-D forward Hartley transform in 8, 12 and 16 bits*

Some software simulations were done where the 3-D forward Hartley transform was stored into 8 to 16 bits integers. The FVR software program (used to generate all images in this report) normally uses 32 bit floats for this. The reducing was done by replacing each value by:

$$h = round(\frac{h}{h_{max}} * 2^{bits}) \tag{4.19}$$

h_{max} is the maximum value in the 3-D forward Hartley transform. See Figure 4.15 for results of storing the transform in 8,12 and 16 bits. Note that the inverse Hartley transform still was performed in 32 bits floats, as was the resampling. This visual test shows that 16 bits are enough to store the forward 3-D Hartley transform in. The PDSP16510/A FFT processor uses 19 bits internally to compensate for loss of precision while performing the FFT.

Without parallelizing the interpolation operation it doesn't look feasible to go to a 512^3 cubed dataset, and a 1024×1024 output image. This means that all operations (memory fetches, interpolation steps, IFHT) have to be done four times as quickly as for the 256^3 case. If designed properly this means that four interpolation chips and 4 FFT chips are needed. Furthermore a different memory fetching scheme probably is needed. Fetching e.g. 64 voxels using the same proposed memory scheme means clocking the memory at $37.5/4 \cong 9$ ns. Only the fastest and very expensive static RAM might be able to meet this speed requirement. For a 512^3 cubed dataset one needs $512 \times 512 \times 512 \times 2 = 256$ Mb of memory!

4.10 Rendering times

This section discusses the times it took to render the images in Figure 1.1. All timings are performed on a HP 9000/735 running at 99 MHz with a CRX48Z graphics accelerator and 144 Mb of RAM. All timings are given for the same dataset of $256 \times 256 \times 256$ voxels, 8 bits per voxel.

The 3-D forward Hartley transform takes about 5 and a half minutes. Calculating a projection using FVR and linear interpolation takes about 9 seconds. The 2-D inverse Hartley transform takes about 4 seconds out of those 9. Calculating a projection of the same dataset using a POCS $3 \times 3 \times 3$ filter takes about 30 seconds. This is much longer due to the fact that the POCS filter implementations were not optimized for speed. E.g. only the 1-D positive half of the filter is stored in a lookup table, instead of the full 3-D filter. Calculating a projection using a POCS $5 \times 5 \times 5$ filter takes about 100 seconds. These timings show that the resampling step in FVR takes the most time. Optimizing the resampling filter for speed will certainly reduce the rendering time.

Splatting the lower right image in Figure 1.1 takes about 15 seconds, but the gradient information was precomputed which took about 75 seconds.

Surface rendering the lower middle image with marching cubes takes about 3 seconds for about 50000 polygons.

Raycasting the upper left image in Figure 1.1 takes about 25 minutes, and computing the MIP image at the upper right takes about 8 minutes. In each case the resulting image was 256×256 pixels in size. Computing the raycasted image is about three times slower than computing the MIP image because the dataset is first classified and colorized into three bytes for each voxel, one byte each for red, green and blue. This results in 3 datasets of size $256 \times 256 \times 256$ which have to be traversed. The MIP image was computed using one byte for each voxel. These implementations do not perform any optimization, so the complete dataset is traversed for each ray. Each ray has 512 sample points.

The marching cubes implementation and the splatting implementation benefit from the

CRX48Z graphics accelerator. The other algorithms do a direct pixel write to the screen, and thus do not use the accelerator.

4.11 Conclusions and recommendations

Fourier Volume Rendering generates projections of a 3-D dataset. The resulting images look like X-ray pictures. In [12] is shown that FVR typically is a factor 100 to 1000 times faster than conventional ray casting. This is supported by the timings given in Section 4.10.

Control of the opacity of the dataset is a problem with FVR. Linear depth cueing and shading can be implemented relatively easily.

Interpolation in the frequency domain has to be done very accurately. A $5 \times 5 \times 5$ POCS filter seems necessary to minimize aliasing artifacts. A filter of this size will make a hardware design a lot more difficult, due to the fact that 64 voxels are needed for this filter.

There are two different aliasing mechanisms to take into account when resampling a plane out of the 3-D frequency transform. 3-D aliasing will occur due to the fact that an imperfect resampling filter is used. 2-D aliasing will occur when the 2-D plane is not resampled quickly enough out of the 3-D frequency dataset.

A more in depth study of filters designed with the POCS method should be made. E.g. a comparison between a filter designed with the POCS method and the commonly used cubic spline filters could be done.

An attempt was made to implement Maximum Intensity Projection by using FVR. The resulting images unfortunately show too much artifacts.

A hardware implementation of FVR largely means designing an interpolation chip, which preferably can use a filter of size $5 \times 5 \times 5$. This is not a trivial task.

A memory fetching scheme to fetch the necessary 64 voxels quickly enough has been proposed.

The inverse discrete Hartley or Fourier transform can be realized by using commercially available FFT processors. For a hardware implementation it makes more sense to use the Fourier transform instead of the Hartley transform, since it saves extra hardware to do the additions after the 2-D IFFT is done.

The artifacts shown when doing MIP projections with FVR have to researched further. Maybe it is possible to come up with a better filter that will overcome this problem, or another way to approximate MIP images with FVR.

A Images

In this appendix a high-resolution and bigger version of some of the images in this report are reprinted.

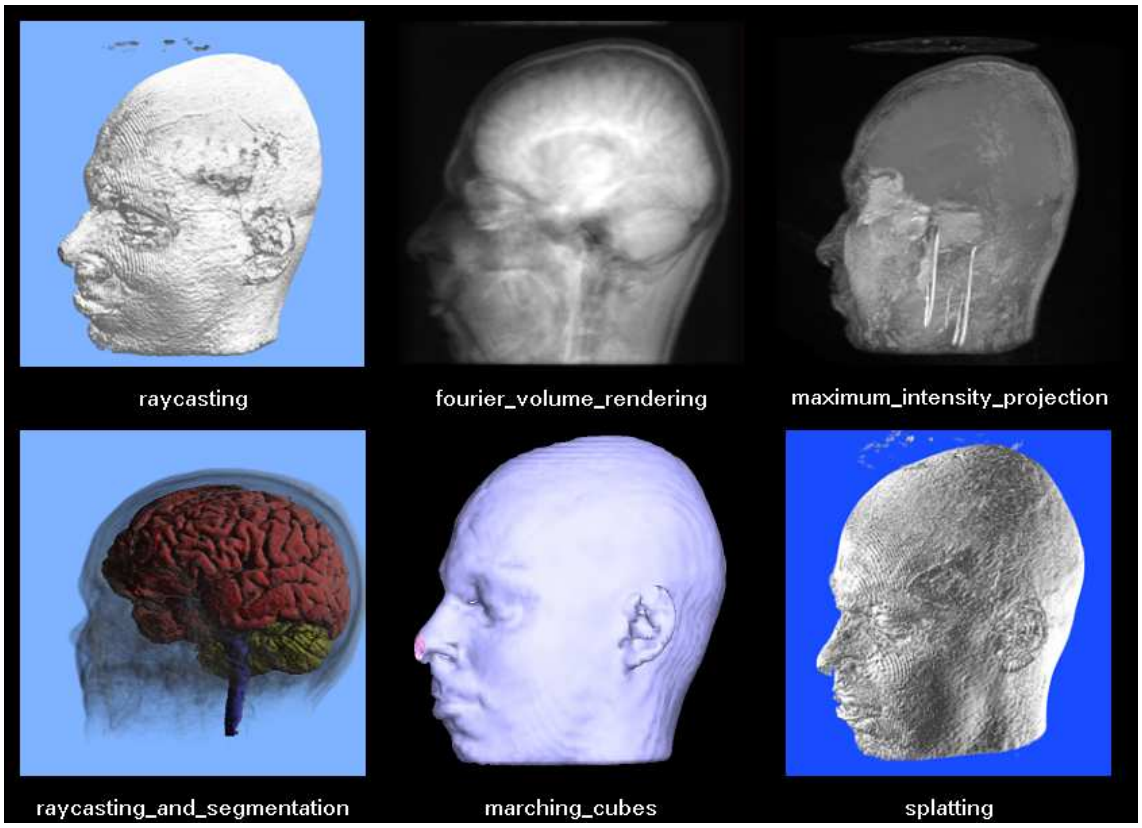

Figure 1.1:*Five different volume visualization methods. The lower left image was segmented by hand. The lower middle image is the only image rendered with a surface rendering technique.*

Figure 4.6(a):*Aliasing. trilinear interpolation*

Figure 4.6(b):*Aliasing. POCS* $3 \times 3 \times 3$

Figure 4.6(c):*Aliasing*. *POCS* $5 \times 5 \times 5$

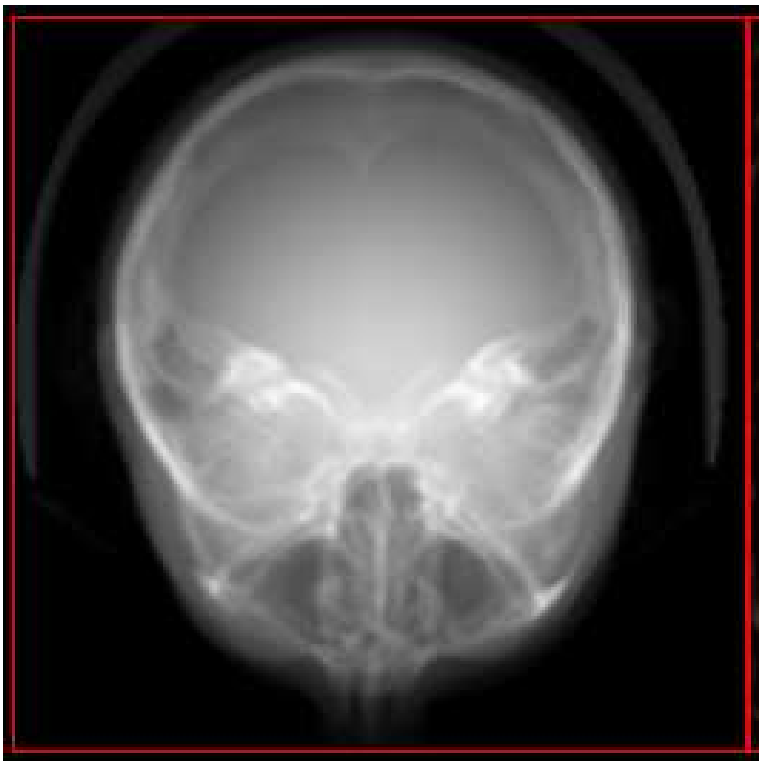

Figure 4.7(a):*Attenuation of outer regions*

Figure 4.7(b):*Compensated for by using spatial premultiplication*

Figure 4.9(a):*MIP projection. Image rendered by FVR*

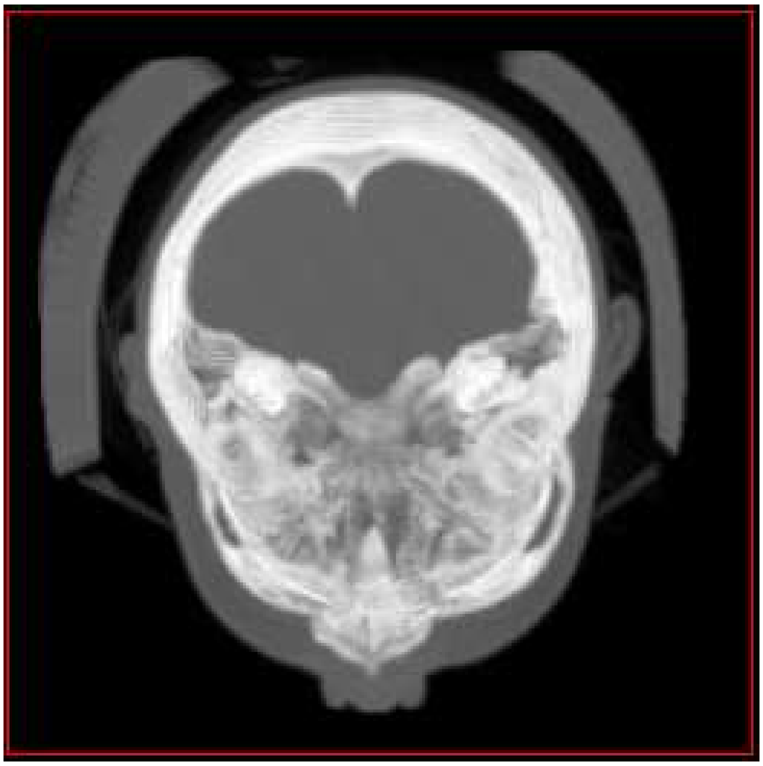

Figure 4.9(b):*MIP projection. Same image rendered by a conventionel MIP raycaster*

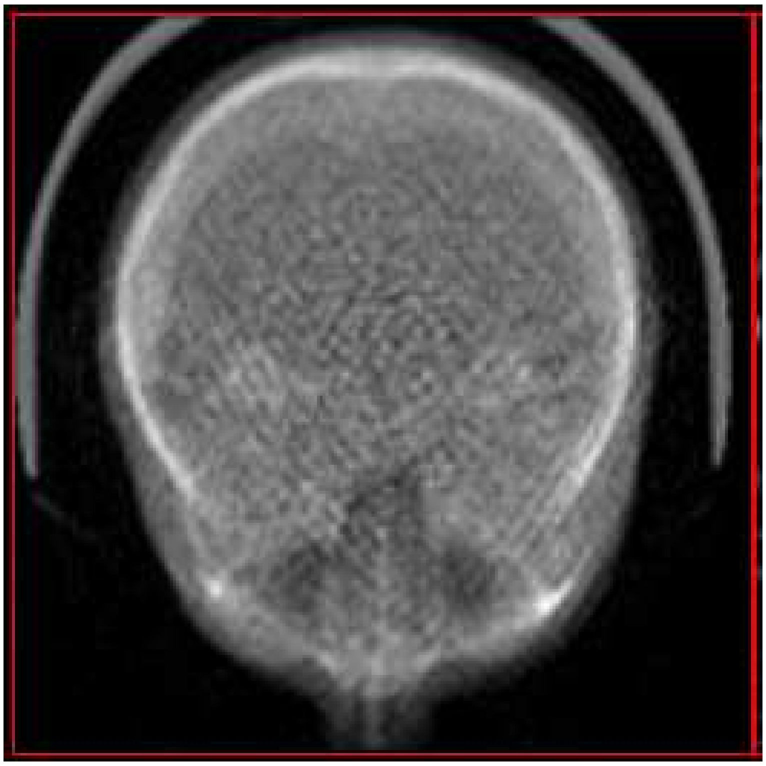

Figure 4.15(a):*Results of storing the 3-D forward Hartley transform in 8 bits*

Figure 4.15(b):*Results of storing the 3-D forward Hartley transform in 12 bits*

Figure 4.15(c):*Results of storing the 3-D forward Hartley transform in 16 bits*

Bibliography

[1] Ronald N. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, 1986.

[2] Dan E. Dudgeon and Russell M. Mersereau. *Multidimensional Digital Signal Processing*. Prentice-Hall, 1984.

[3] S. Dunne, S. Napel, and B. Rutt. Fast reprojection of volume data. In *Proceedings of the First Conference on Visualization in Biomedical Computing*, pages 11–18, 1990.

[4] Rafael C. Gonzales and Richard. E. Woods. *Digital Image Processing*. Addison Wesley, 1992.

[5] Taosong He and Arie Kaufman. Non-existence of the wavelet slice-projection theorem. Technical report, State University of New York at Stony Brook, October 1994.

[6] A. Kaufman. *Volume Visualization*. IEEE Computer Society Press, 1991.

[7] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, pages 451–458, July 1994.

[8] M.S. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, pages 29–37, May 1988.

[9] Jae S. Lim. *Two-Dimensional Signal and Image Processing*. Prentice-Hall, 1990.

[10] W.E. Lorensen and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[11] Einar Maeland. On the comparison of interpolation methods. *IEEE Transactions on Medical Imaging*, 7(3):213–217, September 1988.

[12] T. Malzbender. Fourier Volume Rendering. *ACM Transaction on Graphics*, 12(3):233–250, July 1993.

[13] C.H. Slump M.J. Bentum, M.M. Samson. A multi-asic real-time implementation of the two dimensional affine transform with a bilinear interpolation scheme. *submitted to the Journal of VLSI Signal Processing*, June 1993.

[14] C.H. Paik and M.D. Fox. Fast hartley transforms for image processing. *IEEE Transactions on Medical Imaging*, 7(2):233–250, June 1988.

[15] J. Anthony Parker, Robert V. Kenyon, and Donald E. Troxel. Comparison of interpolating methods for image resampling. *IEEE Transactions on Medical Imaging*, MI-2(1):31–39, March 1983.

[16] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C, the Art of Scientific Computing*. Press Syndicate of the Univeristy of Cambridge, 1989.

[17] H.Hao J.Villasenor R.N. Bracewell, O.Bunemn. Fast two-dimensional hartley transform. In *Proceedings of the IEEE*, volume 74, pages 1282–1283, September 1986.

[18] Richard A. Roberts and Clifford T. Mullis. *Digital Signal Processing*. Addison Wesley, 1987.

[19] Takashi Totsuka and M. Levoy. Frequency domain volume rendering. *Computer Graphics*, pages 271–278, August 1993.

[20] Lee Westover. Interactive volume rendering. In *Chapell Hill Workshop on Volume Visualization*, pages 9–16, May 1989.